Industrial Robot Programming

Nilsson, Klas

1996

Document Version:
Publisher's PDF, also known as Version of record

Preface

Machines that perform hard or boring work have interested me ever since I was a boy and had to help my parents on our farm. I was more interested in mathematics physics, and machines. During the end -70's, I got acquainted with computers and control. That opened a new and very interesting world for me. When I finished my studies for a master's degree in mechanical engineering, I was still very interested in helpful (intelligent) machines. It was therefore an easy choice where to apply for a job: ASEA Robotics.

When I came to ASEA (later ABB) in 1982, Robotics was a new and progressive division which was managed more like a small company. At that time, the demands on profitability were not severe. The primary goal was to make good robot systems. Market share and company size therefore grew rapidly. For me it was six years of stimulating control engineering together with very good friends/colleagues.

The control engineering work ranged from tuning of joint servos to overall system design. When more than thousand, instead of a few hundreds, of robots were manufactured, profitability and cost efficiency for present major applications were getting increasingly important. That was of course a correct policy, but customers with good but unforeseen ideas about how to use robots, too often could not accomplish the control. To start with I was most interested in feedback control theory, but later it became clear to me that the structure of the control system was of key importance for the development of intelligent machines. Published research results were, however, not quite useful because important industrial aspects were overlooked. To do research in this direction, we moved back to Lund where I got the opportunity to do a PhD at the Department of Automatic Control.

Research within automatic control almost always treats well defined problems well suited for formal methods. It has therefore not been easy to tackle a problem that does not fit into this pattern. I hope control researchers do not get too disappointed when they do not find their favorite equations in this thesis. My work has been problem oriented. The subject and the developed solutions are closely related to computer science and production engineering, but the interplay between robot programming and feedback control is of key importance.

It is now a great pleasure for me to complete this thesis, and I very much hope that ideas presented here will contribute to systems that better can perform work that is unfriendly to humans.

Acknowledgments

This work was carried out in a friendly and inspiring atmosphere at the Department of Automatic Control in Lund, Sweden. Professor Karl Johan Åström has been a constant source of inspiration and information. It is mainly due to him that I have had the opportunity to work in a research environment where one can get support and advice when needed, but also freedom to continue along my own ideas (even at times when nothing seems to work). I am also grateful to all other people at the department. Without mentioning all names, this includes professors, technical staff, our charming and helpful secretaries, PhD student friends, and master thesis students that I have been guiding.

During the first stage of my work (until my licentiate thesis was completed), Lars Nielsen was my closest supervisor. He helped me turn my ideas into problems definitions and solutions, and he encouraged me to pursue this direction of research. He and Ola Dahl were very helpful and it was a pleasure to work with them.

Rolf Johansson has given me invaluable guidance during the final part of my work. We have also applied for funds and managed robot research projects together. I really appreciate his comments on this thesis, and I hope we can collaborate somehow also in the future. Thanks also to Björn Wittenmark and Karl-Erik Årzén for valuable comment on the manuscript. The hardware interfaces to our ABB Robots were built by Rolf Braun, who also made the detailed design according to my (coarse) descriptions. I do not know the number of weeks we put into this, but I am very grateful for all his help. Many thanks also to Anders Robertsson for his efforts in the robot lab.

The financial support from NUTEK, which made this work and the development of the experimental platform possible, is gratefully acknowledged. I am also grateful for support and comments from my colleagues at ABB Robotics. The ABB/LTH robot interface could hardly have been developed without the access to information within ABB. Among others, Håkan Brantmark, Torgny Brogårdh, Peter Eriksson, Anders Holmer, Åke Madesäter, Mats Myhr, Einar Myklebust, Ulf-Göran Norefors, Ingemar Reyier, Lars Östlund, and all my friends within the Motion Control Group have helped me to keep in touch with the industrial reality.

It has been nice to use our properly working computer system managed by Leif Andersson, who is always helpful and interested in arranging things for the user's best. The real-time kernel and communication software I have been using is mainly due to Anders Blomdell. He has also been a great source of information concerning practical aspects of software design and hardware-related programming. He also convinced me

that his implementation of dynamically linked *actions* (Chapter 7) usually is more appropriate than the function-based solution that I made. Present and former members of the CACE project has also given valuable hints about software. This includes Dag Brück who often helped me to to use and misuse C++. Contributions from the guest researchers Albert-Jan Baerveldt and Marcel Schoppers have also been helpful.

Due to the multidisciplinary topic of robot control and programming, interaction with other departments here in Lund has been of great value. At the Department of Production and Materials Engineering, Gunnar Bolmsjö, Krister Brink, Per Hedenborn, Hamid Nasri, Georgio Nicoleris, Magnus Olsson, and Björn Ågren have given valuable comments. Some of these persons also gave me access to the robot programming system IGRIP.

At the Department of Computer Science, I have had the pleasure to interact with Elizabeth Bjarnason, Görel Hedin, Roger Henriksson, and Boris Magnusson. Their comments about object-oriented software, and their efforts put into the Applab system, have been valuable both for development of principles and for prototyping language aspects of robot programming.

The always enthusiastic Gustaf Olsson, head of the Department of Industrial Electrical Engineering and Automation, has also been a source of information. At that department, an interesting development of ultrasonic sensing for industrial robots is made by Gunnar Lindstedt, who also programmed the some of the circuits in the robot interface. Our aim to utilize his ultrasonic system for robot control appears to be promising.

The computer vision group at the Department of Mathematics, currently including the researchers Anders Heyden, Gunnar Sparr, and Kalle Åström, has contributed with some vision facilities within our laboratory. I look forward to connecting new vision algorithms to improve the abilities of our (currently blind) robots.

Finally, with some bad conscience for all evenings and weekends that have been spent at work, I want to thank my beloved wife Rosel and our lovely children for their love, patience and support.

About the thesis

Chapter 1 of this thesis contains a short introduction in general terms. Chapter 2 is also introductory, but it gives a broader manufacturing systems perspective as well as a more specific description of the authors view of the problems. Conclusions are given in Chapter 9, whereas Chapters 3 to 8 present the contributions which also have been published according to next page.

The software architecture proposed in Chapter 4 and two of the application examples presented in Chapter 8 have been presented in

> K. NILSSON AND L. NIELSEN. "An architecture for application oriented robot programming." In *IEEE International Conference on Robotics and Automation*, Nice, France, 1992.

Part of Chapter 5 treats software techniques for special purpose hardware. This was presented in

> K. NILSSON. "Object oriented DSP programming." In *Proceedings of The Fourth International Conference on Signal Processing Applications & Technology*, DSP Associates, Santa Clara, CA, 1993.

which was judged as one of the best papers, and therefore also published in a condensed form in

> K. NILSSON. "DSPs moving up to object-oriented programs." *Electronic Engineering Times*, September, 1993.

Real-time and control aspects of such hardware are also mentioned in Chapter 5 and presented in

> K. NILSSON. "Software for embedded DSPs." In *Proceedings from The American Control Conference*, 1994. Invited Paper.

The ideas about how open embedded control (Chapter 7) can be utilized in robot programming were presented in

> K. NILSSON AND L. NIELSEN. "On the programming and control of industrial robots." In *International Workshop on Mechatronical Computer Systems for Perception and Action*, Halmstad, Sweden, 1993.

whereas the thorough description of the software technique, as presented in Chapter 7, hopefully will appear in

> K. NILSSON, A. BLOMDELL, AND O. LAURIN. "Open embedded control." Submitted to: *Real-Time Systems – The international journal of time critical computing*, 1996.

Apart from the control structure part of Chapter 6 which would benefit from a more thorough control analysis, remaining parts of the thesis should be ready to be written and presented as scientific papers. Most important is Chapter 3 about end-user robot programming, but also the robot reconfiguration part of Chapter 5 and the control engineering part of Chapter 6 contain novel ideas. These contributions have not yet been submitted for publication.

Contents

1

Introduction

Making machines programmable has been very beneficial in industrial production systems. The programmability is normally achieved by controlling the equipment from a computer which also provides a user interface for operation, configuration, and programming. Typical examples are NC machines, industrial robots, fixtures, transporting equipment, etc. The use of computer control to achieve flexibility implies that software issues for embedded control systems are central for the applicability and utilization of the equipment.

Industrial robots are distinguished from other types of machinery mainly on the basis of their programmability and adaptability to different tasks. Robots are therefore probably the most demanding type of equipment concerning the software and control aspects. This thesis treats software issues for embedded robot control systems, with the aim to improve applicability of industrial robots and hopefully also for other types of manufacturing equipment. 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.

Improved performance has been the primary goal in the still very active research field of robot control. Despite many advanced algorithms that have been developed [59, 193, 109, 45, 136], only relatively simple solutions have been successfully used in real industrial products and applications [59, 136]. There are several reasons for this. Problems concerning the numerical properties, the computing efficiency, and the need to tailor text-book algorithms for practical use is well known [140, 29, 71, 163]. However, less attention has been paid to the interplay between the algorithm and the system, particularly considering real-time implementation and industrial aspects of end-user programming.

To simplify robot programming, a common approach is to increase the level of abstraction. In today's standard applications this is done to a limited extent to reduce the (expensive) programming of the robot [181]. A typical research aim is fully automatic robot programming or task level programming, see for instance [59] and any robot conference. However, also in this case very few research results have been applied in practice. The complexity of the problem, especially when confronted with industrial constraints, is the main reason for this.

The development towards improved performance and ease of use will be further pursued in this thesis, considering that superior manufacturing practices require appropriate possibilities for man-machine interaction. Motivated by the presence of **feedback** in manufacturing systems, the approach taken is influenced by principles of automatic control. As for many other types of dynamic systems, robustness and performance can be improved by introducing new (local) feedback loops. Two types of feedback are of special interest:

- The factory floor operator observing the actual production result should be able to adjust robot programs in an appropriate way. The purpose is improved factory floor operation.
- New (application specific) control loops should be possible to introduce in the motion control system, considering typical industrial demands. The purpose is to improve applicability and/or performance.

To achieve the main goals, a number of related problems concerning design and implementation of embedded control systems have been encountered.

- Structuring and implementation principles for control algorithms.
- Real-time programming of open, layered, and embedded systems.
- Control engineering of distributed embedded systems.

These problems are also major topics, and they are likely to be useful also for other types of embedded control systems.

This thesis takes a problem oriented approach. It includes a discussion of real industrial problems. Solutions to these problems are the major topics, but the problem formulations are in some cases (like the industrial application examples) contributions in themselves. There is also emphasis on a software architecture, called the Open Robot Control (ORC) architecture. The purpose of a software architecture is usually to organize the software in such a way that the implementors of the system can cope with the complexity and reuse of software [81, 188, 186, 13, 133, 168, 70]. The design of ORC [150], on the other hand, is based on *user views* (related to use cases [96]). The primary purpose is to support different types of programmers and operators by providing suitable layers of programming.

Outline of the thesis

Chapter 2 about manufacturing systems will give a more extensive introduction and put the aim of the research into a broader perspective. Robot programming on a standard user level is then treated in Chapter 3. A fundamental idea is to view robot programming as a control problem. The process output is the production result and that is influenced by modifications of the robot programs. The proposed solution includes a revised handling of world models [135, 81, 59] (related to the blocks world within artificial intelligence [166]). Furthermore, the proposed representation of robot programs uses syntax trees in an extended way as compared to compiler technology [12]. This exemplifies how control system technology can gain from computer science and software technology, which has been a source of inspiration for technical solutions throughout this work.

Relations to so called intelligent robots and task level programming is treated in Chapter 4. A basic idea is that high level planning systems should deal with robust unit operations created by experienced robot programmers. This means that low level effects such as friction and tolerances are taken care of in these unit operations. That simplifies the high level planning problem.

An experimental platform has been developed to verify the proposed solutions. Chapter 5 presents the platform. It is built around commercially available robots. The original control systems have been replaced by new open controllers. Implementation of embedded controllers on multiprocessor hardware, with severe demands on efficiency, require special solutions which are presented in Chapter 6. Making use of those principles, design and implementation of robot motion control are then treated in Chapter 7. The proposed control implementation is layered and open.

Examples in Chapter 8 show some demanding applications which can be better solved using alternative low-level motion primitives. It should be possible for an advanced user to introduce such new primitives in the embedded control system. How to do this, and how the applications can be solved, are major topics in Chapter 8. Some conclusions and a summary of the contributions are finally presented in Chapter 9.

2

Preliminaries

Problems within manufacturing and production systems span a wide range from servo control of individual machine tool motors and up to overall control of large scale production facilities. Solving such problems often requires competence from quite different fields ranging from feedback control theory to management and personnel policy. Depending on background and circumstances, engineers consider different problems and solutions important and relevant.

This thesis treats problems related to programming and control of very flexible production devices, such as industrial robots. The purpose of this chapter is to give an industrial perspective, and to describe some important problems in control and software technology with relevance to manufacturing systems. We try to merge two points of view; manufacturing systems in general, and control system aspects that are fundamental to the thesis.

2.1 The importance of manufacturing systems

Meerkov [128] clearly described the importance of manufacturing. One of the conclusions is that "To live well the nation should manufacture well". The background is that wealth could be either grown, mined, or manufactured. Growing and mining can not alone provide sufficient wealth for industrialized west world economies. That implies that manufacturing is important.

Experiences from last decades show that manufacturing has been much more successful in Japan than in Europe and Northern America. This is sometimes explained by "lack of automation" outside Japan. Several facts indicates, however, that this is not the explanation. For example, the "Lowest in automation Japanese plant is 70% more efficient than the most automated plant in the world". Still, many facts show that automa-

tion is very important, but case studies show that it has to be done in a proper way, and to a suitable extent.

In addition to a suitable degree of automation, products should be designed for production in general and for automation in particular. Even if the product design is very important for the productivity, it can be copied more easily than production systems involving humans with certain skills and cultural background. Trying to explain the Japanese success, this instead indicates the importance of good *factory practices*.

The term "factory practices" means rules and habits of manufacturing process operation and control. A striking example is the electronics industry. The three most successful new products of the last 20 years are the video cassette recorder, the fax, and the CD-player. They were all invented and designed in the West. Today, except for one Philips factory in Austria, all are produced in the Far East. The explanation is superior manufacturing practices.

Another term for factory practices is control of manufacturing processes. Control is here in the sense of process operation, which indicates the need for research on the large scale aspects of manufacturing, using system-theoretical and other approaches. There is, however, also another interpretation which is the basis for this thesis. There are a number of demands from the production system on the manufacturing equipment, concerning flexibility, efficiency, etc. Furthermore, in order to achieve superior factory practices, it is necessary that workers, operators, and engineers can interact with the equipment in an efficient way.

2.2 Industrial robots

An industrial robot as such is basically not dedicated for a particular task or application (even if some types of robots are preferably used in certain applications). That distinguishes robots from other types of machinery. Originally, however, the individual joints of a robot were commanded and controlled as for any other multi-axes servo-controlled machine. This means that motions were specified numerically by sequences of simple motion commands. Interpretation of these commands results in calls to move-procedures provided by an interface to the servo control algorithm, which controls the physical system via sensors and actuators, as shown in Figure 2.1. This means that rudimentary robot control is similar to standard servo control, and is easily incorporated into any of today's programmable control systems.

To make robots more useful, the development during the last 20 years has resulted in more sophisticated specification of motions, both in terms

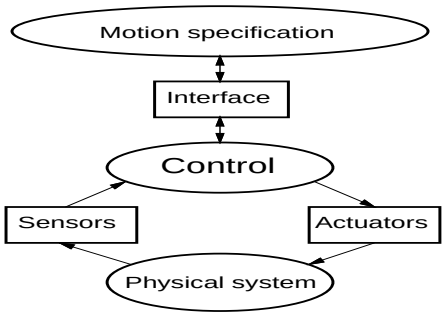

Figure 2.1 Basic components of motion control.

of how motions and computations are specified/programmed, and in terms of the tools used for the programming. For example, the abstraction level and ease of use of the system were increased by having a kinematic model of the robot built into the control system, and special programming tools and languages were developed to aid the robot programmer [41]. Robot programming was still manipulator oriented, i.e. the manipulator motions were specified rather than the task to be performed.

A recent trend has been to include more knowledge about the physical system into the controller. For the control algorithms, this means that dynamic models are utilized in order to improve performance. Performance demands come from required utilization of the relatively expensive mechanical part of the system. The interface (see Figure 2.1) models the physical environment on some level, just like a reference signal to a simple control loop can be viewed as a model of the controlled output. The development for the interface and for the motion description has aimed at an increased level of abstraction to make robots easier to use.

One example is motion commands specified as relations to the objects being manipulated [81, 181]. Another example is application specific task-level programming, allowing the programmer to specify motions (or on-line adjustments of motions) in terms of production data that he/she is used to, like arc-welding parameters [49] etc.

Clearly, the development of advanced robot control systems has made them quite different from, and in practice incompatible with, PLC systems and NC machine controllers.

2.3 Large-scale versus small-scale production

Hierarchical decomposition into smaller subsystems are used in large production facilities to make the plant more manageable. Such decomposition

should be made in such a way that the subsystems are as independently operational as possible to provide robustness, but at the same time total cost efficiency must be achieved. The aim for independent subsystems leads to large-sized buffers for materials and components, and to machines that are not shared between different products. Aiming at cost efficiency typically means the opposite. To make proper trade-offs between these contradictory demands is called "production planning and management". This is not explicitly treated here, but we will see how the manufacturing organization influences the desired properties of the local equipment.

Figure 2.2 shows an example of a hierarchical organization for a larger manufacturing system. Such a facility has (not shown in the figure) central engineering departments for product design, production planning, and production operation. Computer networks are widely used to connect machines, work cells, and the central engineering facilities. Now, consider the local robot equipment. It should be clear that a powerful host computer interface for both programming and supervision is essential to achieve Computer Integrated Manufacturing (CIM). On the other hand, perhaps a too centralized approach is not the best solution.

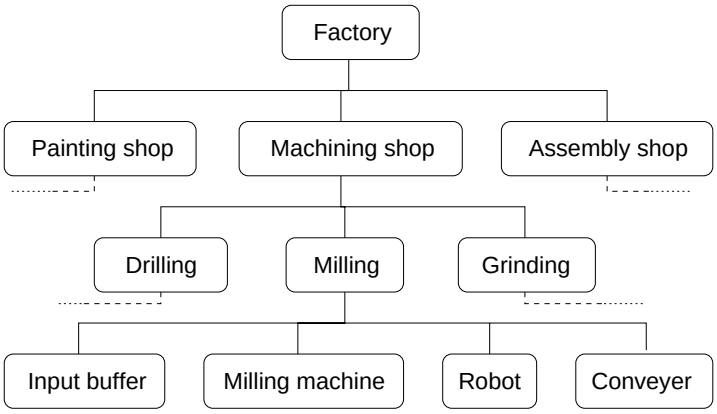

Figure 2.2 Hierarchical levels in a large scale production facility.

If we look at a sample small mechanical workshop, there will be other demands on the (local) equipment. The production planning and operation is probably not computerized, and a machine such as a robot is typically used as a stand-alone system which is programmed and operated directly on the workshop floor. Furthermore, the local operator has a good overview of the production, and he/she knows how to adjust the equipment to obtain the desired production result. This means that there will be a very short turn-around time from a detected production problem until it

is taken care of. Such an approach has turned out to be very beneficial also in larger production facilities. Figure 2.3 exemplifies that, but the situation shown in the picture is also typical for smaller workshops.

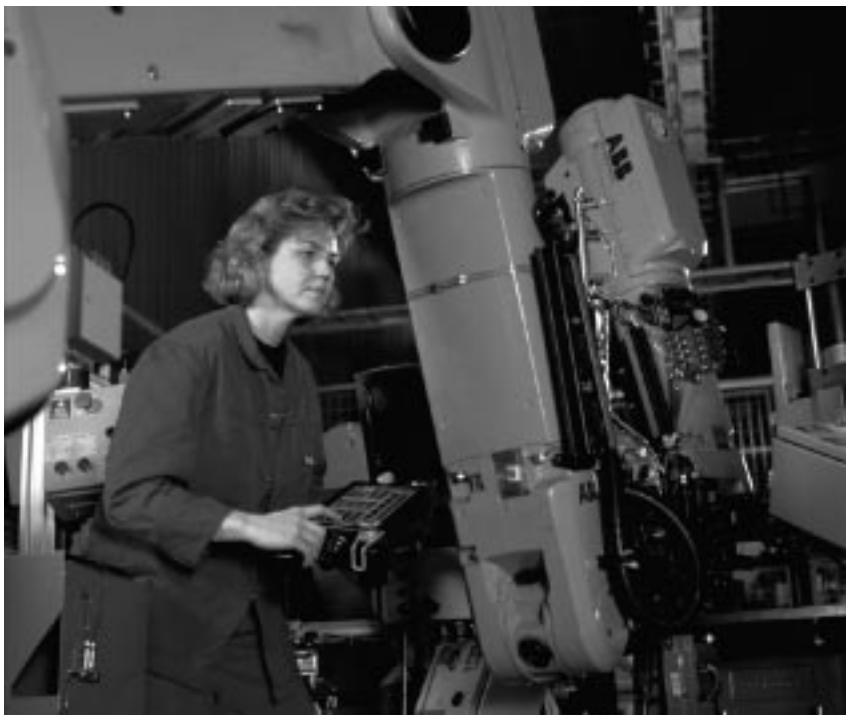

Figure 2.3 Robot operator/programmer at Volvo, Sweden, doing on-line changes of robot motions for gear-box assembly, using ABB Irb-3000 robot systems. (With permission from ABB and Volvo.)

In the small workshop case, it turns out that the preferable robot operator interface is quite different from the one mentioned above for central engineering in the large-scale case. It is of course desirable to combine the benefits of small-scale and large-scale production systems. For instance, rapid adjustment of the production process should not deteriorate the consistency between the local robot program and the central engineering databases. It should be clear to the reader that such a combination puts some not easily combined requirements on the robot system. One reason for this is the differences in preferable programming methodology depending on application, type of user, etc. A new approach based on multiple representations of robot programs will therefore be developed in the next chapter.

2.4 Robot programs

So far we have only dealt with (different types of) *end-user programming* of robots, but what does robot programs look like? To clarify that for the reader not exposed to robot programming, a few comments will be given in this section. Since a robot program contains the specifications how the robot should move, procedures for motion control will of course be used (i.e., the 'interface' in Figure 2.1 will be accessed). Such procedures have been implemented as part of the *system programming*, typically done by the robot manufacturer. A high level of abstraction is often preferable in end-user programming, but a low level of abstraction is used here for clarity. Abstract actions will be converted to concrete ones anyway.

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. Early languages used in simple applications resembled BASIC. To deal with more complex situations, more structured (Pascal-like) languages were introduced. The first such language appears to be AL [135], in which 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 motion instructions) is a reserved identifier and parameters are specified with predefined attributes belonging to the `MOVE` instruction. For example, a grinding motion may be expressed as (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 [41, 118, 6] is therefore used.

Sensor inputs affect robot motions in four different ways [118]:

1. Initiating and terminating motions.
2. Choosing among alternative actions.
3. Obtaining the identity and position of objects and features of objects.
4. Complying to external constraints.

The first three of these are simpler to handle because they map well

on conventional programming of computers and process control systems. The fourth type of sensory interaction is used when motions are continuously adjusted based on sensor inputs. Opposed to Items 1 to 3 which more have to do with event handling and reactive systems [76], Item 4 requires feedback control [29, 193] Typical examples are combined position/force control using a force/torque sensor mounted on the robot wrist, or seam-tracking during arc-welding using a laser scanner. A straightforward solution would of course be to specify the application-specific motion control strategy directly on the end-user level. However, there are efficiency, complexity/simplicity, and safety reasons for not doing so. Instead, implementation of such sensor based motion control strategies used to be completely done by the system programmer [118]. Therefore, such applications put special demands on the control system.

2.5 Special applications

Contemporary systems usually allow customer specific sensors to be installed, either direct via standard IO ports, or by installing a device driver for more advanced sensors. So called open control systems [150] may even allow replacement of specific control modules [70], but is that enough? The following example shows that the answer is no.

Industrial example – Deburring of castings

Figure 2.4 shows a typical example of a casting. Consider the upper circular edge in the figure, i.e., it should be circular but remaining burrs from the sand-casting process make it look different. Those burrs have to be removed, either by machining or manually using a grinding tool. Figure 2.5 shows the profile in more detail.

Removal of the burrs, so called deburring or cleaning of the casting, is a task that is preferably performed by flexible machines such as industrial robots. That is because the task is monotonous, and there are unhealthy vibrations and air pollution. However, automatic deburring is feasible only in simple cases today. Specific problems are to make the robot recognize where additional grinding is required and, if so, to program suitable deburring motions. More specifically, we can think of the following strategies:

- The burrs can be **cut off**. Such a process is, however, often too slow and the position of the profile to clean must be known rather precisely. That is typically not the case due to casting tolerances.

Figure 2.4 Valve housing from Flygt, Sweden, with burrs remaining from the sand casting. The height of the piece is approximately 170 mm.

- A grinding tool can be moved with position control along a nominal profile, and with force control in a direction normal to the path. The force control entails a compliant motion [193] which avoids problems with casting tolerances, but exceptional places with much material may remain, for example due to the large bulge shown in Figure 2.5.
- Sensors like cameras and laser scanners can be used to overcome the problems with the two previous alternatives. Still, there are problems with sensor technology and feature detection, cost, complexity, and with programming of the grinding strategy.

The first two alternatives make use of the third type of sensing according to the previous section. Considering needs for production speed and efficiency, type 4 sensing and 'continuous' (in a practical sense) feedback control during the grinding process would be attractive. That is to adjust grinding speed etc. during operation to obtain optimal productivity and quality. Vision systems are hard to use because the grinding equipment will be part of the picture and it will partly hide the work piece. Special purpose sensing is therefore needed.

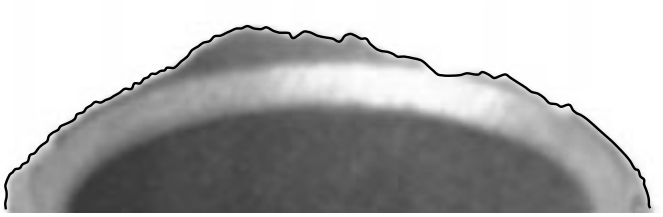

Figure 2.5 The profile of an edge on a casting. The contour to be grinded is marked with a solid line.

External sensors are sometimes necessary, but they have drawbacks since they cost, fail, complicate the installation, etc. A basic idea in this work 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. In this case the position error in the force control direction would be very useful since that will contain a recording of the actual contour **after** grinding.

Assuming that we can solve the sensing problem so we get information about (remaining) burr size etc., we could utilize knowledge about the grinding process [137] to plan and compute suitable grinding motions. But how should that be achieved, and how should it appear to the ordinary robot programmer?

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. Finding an appropriate blend between efficiency/safety on one hand and flexibility on the other, as well as real-time software solutions to achieve it, is a non-trivial problem that will be tackled in this thesis.

A benchmark/toy problem

Some characteristics have been identified in special applications like the deburring one above.

- Some **initial control** strategy is required before the desired movement can start. For example, a grinder tool must be started and a certain initial contact force between the grinder and the work piece must be achieved.
- A **sequence** of initial control actions may be needed. The robot must first be properly positioned, etc.
- Normal operation entails cyclic execution of a feedback **control algorithms**, which can be derived from dynamic models and/or from heuristics. Control of path speed depending on burr size is one example.
- There are some online supervision of control states to detect if the process enters a prohibited region. For instance, a too slow path speed (depending on exceptional burr size) may cause overheating of the tool or of the work-piece. Special control must then be switched in to handle the exception.
- Control of **mechanical systems** is typically **non-linear**, but can often be controlled locally using (piecewise) **linear** controllers.
- Special control may be required to gracefully finish the operation. A smooth edge may be required also where grinding finishes.
- High sampling frequencies are sometimes required, and the real-time demands on the control may be severe.
- The special application specific control is preferably encapsulated in a new customized statement, like the `GRINDMOVE` above.

It is also characteristic that a lot of equipment and installation is required to run a real industrial application. That makes setups in different robot laboratories difficult and expensive. Still, it would be valuable to have a benchmark problem that can easily be setup, and that could be a test how well a system can be used for 'special applications'. Such a suitable benchmark problem has been found to be *control of an inverted pendulum held by a robot hand*.

Control of a pendulum [27, 192] illustrates that many different control principles have to be used to accomplish a given task. Specific control problems include

- Initial control to a well defined initial state to prepare for special swing-up strategies.

- Swinging up from a well defined initial state, possibly via several control modes.
- Control of the pendulum in upright position using feedback control algorithms. Normal operation is close to the equilibrium.
- Online supervision, typically model based, to detect if external forces have brought the pendulum away from normal operation, i.e., far away from the equilibrium.

Thus, it is complicated enough to capture the aspects of industrial applications and yet so simple that experiments can be set up with a reasonable effort. The control can also be tested initially using pendulums that are readily available in many laboratories.

Note the difficulty imposed by manufacturing constraints. It is not only the specific pendulum control including its mode switches etc. It should be achievable in an embedded robot control system using industrially useful programming principles. That means simple and restricted end-user programming using for instance a `PENDULUM` statement, while the pendulum control should be implemented by a control engineer without using inside information about the built-in motion control system.

2.6 About this research

This chapter has so far motivated a closer look at end-user programming and integration of principles that may improve manufacturing practices. Use of techniques available from the field of computer science is probably a good idea. That also applies to the problem of finding a proper architecture for robot control systems. Another important topic is support for special applications, which implies a need for *intermediate level programming* in which the experienced user can implement new low-level features. That will then require use of principles from real-time systems and control theory, as in the second half of this thesis. Finally, a powerful experimental platform based on industrially used robots with state-of-the-art mechanics and motors needs to be developed.

Specific robots and user interfaces for industrial applications should be developed within the robotics industry in close collaboration with sales, customer support, and with the robot users. It is therefore not covered here. There are of also many interesting and challenging theoretical problems within robotics and control, but that is subject to extensive research in many universities. **This research is an attempt to bridge the gap between standard industrial usage of robots and research results from programming and control.**

Misconceptions

Tackling problems that do not have an established definition can create some confusion. Depending on the reader's background and previous experiences (e.g., from some existing system), different parts of the solutions presented can lead to some confusion concerning approach, importance, novelty, etc. Based on reactions to viewpoints presented in technical discussions and earlier work [144], the following remarks are made in order to point at some standard misconceptions.

The purpose of a software architecture Software architectures have received much attention within robotics research [56, 13, 127]. One reason is that when robot controllers (e.g., for space applications) become more and more complex, abstractions and software structures are introduced to cope with complexity. In other words, the purpose of the architecture is to make the implementation of the system feasible.

In industrial robotics the software is complex and various functions must be tightly coupled to achieve efficiency. However, the implementation complexity is not worse than it can be handled by proper software engineering methods, like an object-oriented design. But the variety of user interactions in flexible manufacturing systems indicates that user views of the system should be the basis for the architecture. This is a completely different approach that should not be confused with implementation architectures.

"We can do that in our system" When suggesting a new embedded control system, there will always be alternative ways to do it. Take, for instance, some advanced process control systems. First, such a system can of course be used to control a robot, but will the desired performance, cost efficiency, programmability, and flexibility be achieved? Secondly, when doing servo control using process control systems, will use of a specific system and its special language etc. be appropriate for interfacing to stand-alone servos? In conclusion, almost anything feasible can be implemented in any system, but specific application demands as considered in this thesis are typically not taken into consideration.

"Layered systems are not useful without a detailed specification" Layered systems are perhaps most common within computer communication. Refer for example to the OSI model [90], and to the even more specific MAP standard [156]. Within computer and telecommunication applications, it is crucial that the specification of the layers is complete in all its details. It must be possible to interleave component and layers from different vendors, and the layers reflect the implementation.

In this work, the layers reflect **user views** of the system. Detailed internal interfaces could of course also be developed, but that needs to be made in collaboration with major vendors and/or standardization organizations. Otherwise, the industrial impact would be too small. On the other hand, we claim that earlier and ongoing standardization of robot interfaces on high [130], intermediate [212, 91], and low levels [169] are not appropriate. Just like this work is not devoted to some new programming language, it does not depend on a detailed standard. It is usage of the principles proposed that yields the benefits. Standards come with maturity!

"A robot controller is just another PLC block" Process control systems and PLCs (Programmable Logic Controllers) often control motions, usually via dedicated servo controllers containing the drive electronics and the low level feedback control. Process controllers are typically programmed by combining and connecting PLC blocks into a block diagram defining the control program. A servo controller can then be encapsulated in such a block. What is then a robot controller? In simple and less demanding cases, a robot control system is just a multi-axes programmable servo controller. For the reader with experience mainly from such applications, it could be hard to understand why robot control should be such a big issue; it is just another PLC-block. However, investigating demanding use of industrial robots, the needs for motion descriptions, operator interactions, nonlinear and variable structure control, and computing efficiency clearly show that robot control requires its own control techniques. It is still desirable to combine process control (for manufacturing) and advanced robot control, but that is outside the scope of this thesis.

"We already have an open system" A system that is *open* allows the user or system manager to change or add certain internal components of the system. In practice, systems are a mixture of open and closed parts [70]. The open parts can also be open in many ways. As an example, consider a robot control system with a replaceable trajectory generator. Such a system can be claimed to be open. However, the interfaces to the software component (the trajectory generator in this case) could be so rigid that only the algorithm can be replaced. In other words, structural changes involving, for example, new types of interaction with the servo control are often not possible. Therefore, an open system should be reviewed concerning the type of changes possible, and what degree of flexibility that implies. Still, one should keep in mind that there can be safety and proprietary reasons to keep parts of the system closed.

"Our customers have not required that feature" Industrial development has to focus on customer requirements. Sometimes, however, new features that let customers explore new possibilities simply have to be offered. Proposal of new features for application or customer support engineers, sometimes results in a comment that "our customers have not required that feature".

This research is inspired by real industrial problems, but specific solutions can very well be questioned in the light of short term requirements. The reader should in those cases, however, not forget the fundamental long-term benefits.

"The problem is on-line teach-in programming" Different ways to program robots are preferable for different situations/applications. If we use the robot system for the programming, we call it on-line programming. Combined with definition of coordinates by manually commanding/moving the robot, we have *on-line teach-in programming* (OLTP). Such programming has turned out to be very useful in many (or even most) industrial applications. Within robotics research and for some advanced applications, use of OLTP is often considered to be a problem because the off-line systems can not fully cope with on-line changes of the programs, and because the robot control system does not provide sufficient support for advanced applications. This is how it happens to be, but do not confuse the possible benefits of OLTP with the disadvantages of today's systems. This topic will be returned to in the next chapter.

"A new type of motion is just another procedure" Principles for incorporation of application features have not received the same attention as other software aspects in robotics like high-level planning, or low-level explicit joint control. It has been neglected by statements of the type "just implement a procedure" or "implement another robot function". On the other hand, robot manufacturers spend major efforts in designing and implementing such robot functions. Even so, it is well known that it may be difficult or impossible to slightly modify a function, to change an application feature, or to include a new type of sensor in existing systems. The reason is of course that the software is complex 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 represent a major effort. Applications mature over time and it is natural that more and more special features need to be implemented. If this can be done efficiently using the principles proposed in this thesis, then the implications for production speed and efficiency are obvious.

Concluding remark

Throughout the thesis we take issue with the misconceptions quoted. In conclusion, it is believed that more generally applicable robots is a key issue in providing flexible components for manufacturing systems, even if it is realized that control of specific machines is not the most important aspect of efficient manufacturing. The approach taken here, however, is related to the manufacturing practice aspect in the sense that the equipment should be designed in such a way that man-machine interactions supporting superior manufacturing practices should be allowed. The key motivation for this research is that the problems have been deficiently observed/solved elsewhere. Note that there is usually no contradiction between principles presented here and other established research approaches; **one aim here is to ease practical use of available and future research results by proposing a suitable framework that also considers typical industrial demands, and by providing an experimental platform for verification of these ideas.**

3

End-User Programming

An industrial robot is a programmable industrial manipulator. A robot program is expressed in some kind of robot programming language (RPL), as described in Section 2.4. In this chapter we assume availability of a control system providing a set of motion primitives, including those possibly required in special applications according to Section 2.5. In other words, we specifically focus on possible improvements of end-user programming.

Proposals how to deal with this problem can be found in (almost) any robotics conference. A common approach is to increase the level of abstraction aiming at so called task-level programming [112, 106, 59] (an early reference appears to be [203]). Some approaches include the physical layout in the design of robot tasks [155, 52]. In any case, an embedded control system is used for feedback control of manipulator motions. Such robot controllers [118, 59, 46] also provide manipulator-level programming and operator interaction [164]. These features are carefully designed to meet the requirements of standard applications, but has until now not been fully exploited when combined with high-level (fully computerized) methods of programming. So, instead of trying to find the ultimate solution (like special languages, databases, etc.), **the aim is to better combine and integrate such methods that are promising or successfully used for industrial applications**.

A small introductory example to be used throughout this chapter will be defined in Section 3.1, followed by some comments on different ways to do the programming today in Section 3.2. With the presented example and the interpretation of available methods in mind, the following problems will be tackled:

- In Section 3.3, which first describes the approach towards integrated robot programming, the misconception concerning on-line programming mentioned on Page 17 is treated.
- The desire to combine the benefits of small-scale and large-scale pro-

duction as explained in the previous chapter will then in Section 3.4 be interpreted as a feedback control problem.

- In Section 3.5 we look at internal states of robot programs. Needs to maintain some world model data also in on-line programming reveal a context sensitivity problem that until now have been overlooked.
- A solution to the context sensitivity problem will then be proposed in Section 3.6. This is a key issue to accomplish the desired integrated end-user programming.
- An implementation of a full prototype is described in Section 3.7.

3.1 Small introductory example

To focus interest on some important robot programming aspects, a simplified robot welding task will now be defined. The standard "peg-in-hole" problem [67, 59] is another such test-case for assembly applications. The following welding application does, however, better represent such applications that are of primary interest in this thesis.

Consider an arc-welding task. For simplicity we study only a single weld-joint. The task is to weld a piece of flat iron bar on a base plate mounted on a fixture. See Figure 3.1. Before welding is to be started, the robot is supposed to move the weld-gun to a location `clean` where cleaning of the tool takes place. Welding should start on a location `start_pose` and proceed along a straight line to location `end_pose`, also on the base plate. After welding is completed, the robot should move to a position `home` where it waits until next task starts.

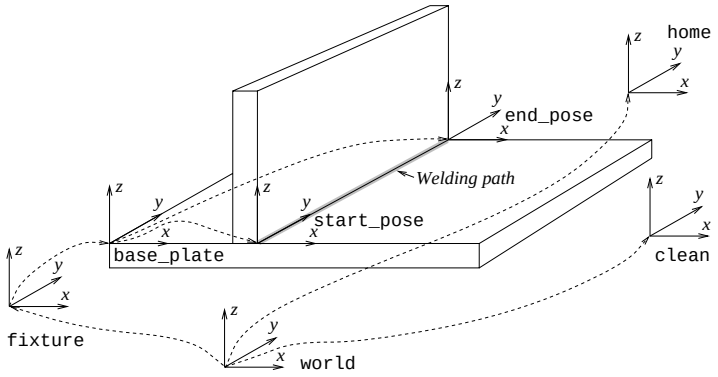

Figure 3.1 A simplified robot welding task.

Note that the locations `clean` and `home` are expressed relative to a fixed world coordinate system, whereas the locations `start_pose` and `end_pose` should be expressed relative to a frame `base_plate` defining the location of the base plate on the fixture. That is to permit an alternative mounting on the fixture without changing the welding sequence.

3.2 Review and classification of current approaches

Many preferences by robot programmers for one or the other robot programming method stem from the way different methods are combined in systems today. 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 [41] and [59] have such preferences, but some factory floor aspects (mostly to the favor of on-line programming) deserve some comments. This section reviews current robot programming concepts in four different ways to prepare for a treatment of some basic underlying problems.

On-line or off-line programming

Let *use of mechanical robot* be the basis for our classification. *Off-line* programming means that the mechanical robot and other production equipment is not occupied during programming, which instead takes place on a host computer. *On-line* programming means that the physical robot is occupied during the programming. These two alternatives for *where* the programming takes place are standard [41, 59] and used in different situations.

Off-line programming has the advantage that a production cell can be designed, programmed, and its operation may be simulated before the cell is actually built. The result of the simulation can be which type of robot that should be used, or how the equipment in the cell should be arranged. To this purpose, advanced modeling and 3D graphics are used. As an example, Figure 3.2 shows the above application example modeled in an industrially widely used off-line system. Several advanced off-line systems provide a general purpose RPL, and code generators for specific robot controllers. A uniform style of programming for robots of different brands can thereby be achieved [59]. However, each off-line system has its own 'general purpose' language. Hence, independency of robot systems instead results in dependency on the off-line system.

One problem in off-line programming is that generation of programs for embedded controllers often can not be done in a way that preserves the program and data structure of the robot program. This is due to limitations of the RPL used by the embedded controller. Translating programs written in SIL [184] to ARLA [4] for example, is comparable to translating Lisp to Basic, which clearly is hard to do if the structure of the program should be preserved. Other problems are that off-line programming tends to be unsuitably abstract for certain programming situations, and the robot program may be inaccurate as a (approximate) computer representation of the robot and its environment is used.

The computer representation of the robot and its environment is called a *world model*. An example of a world model for the above example is shown in Figure 3.3. Each object in the world model contains frames (as

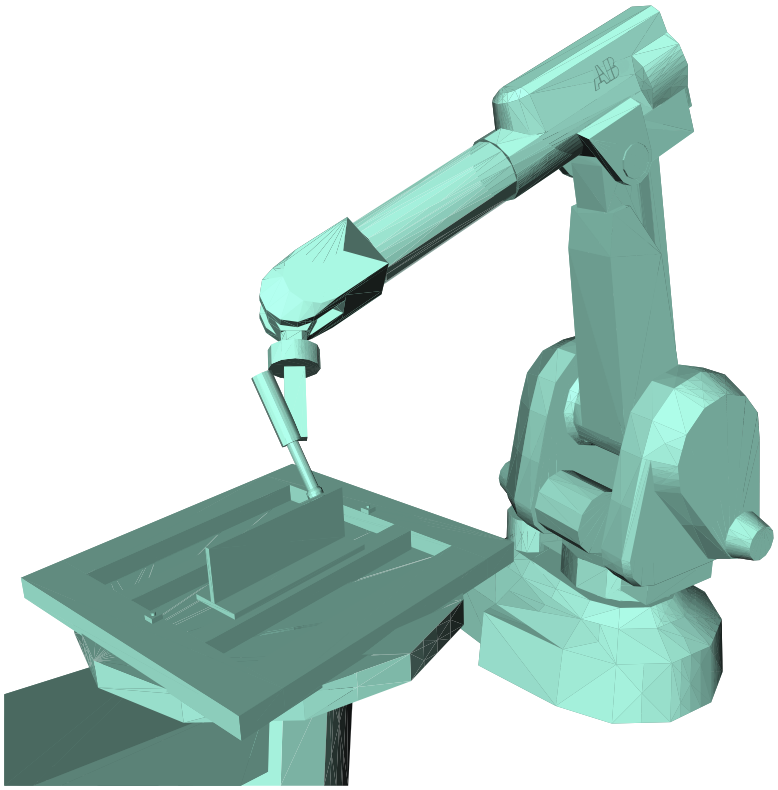

Figure 3.2 The simple welding application, using an ABB IRB-2000 robot, modeled in IGRIP [65].

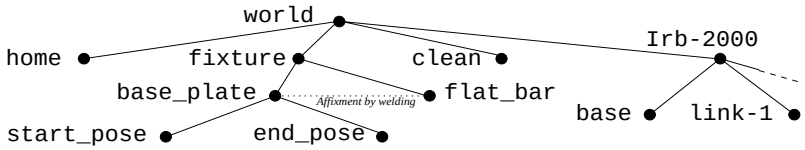

Figure 3.3 World model objects for the simplified example.

coordinate systems are designated within robotics) defining the position, orientation, and geometry of the corresponding physical object. Other attributes may describe the graphical view of the object in the case when a graphical user interface is used. Structured objects containing a number of sub-objects may also contain kinematics and affixments. Affixments are used for temporary rigid connection between objects, like when a simulated robot picks up a simulated workpiece [59].

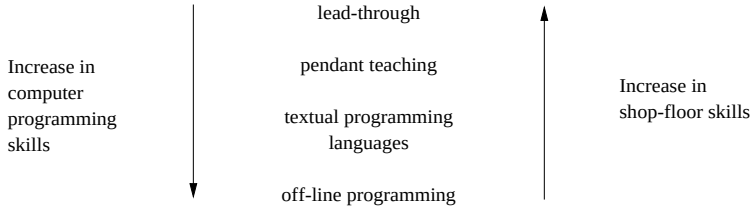

Figure 3.4 Computer programming versus shop-floor skills, considering human factors in robot programming [164].

On-line programming does occupy the production equipment, but an advantage is that it is tangible, i.e., abstract world modeling and computer simulation are not needed. The programmed motions will also be accurate since locations and frames can be defined via teach-in referring to the true frames. On the other hand, complex motions along mathematically well defined paths (for example on airplane wings, turbine shovels, etc.) can be very hard to program by teach-in. Such motions are better directly described based on data from CAD systems.

Generally, a tradeoff between simplicity and programmability appears to be necessary for a specific programming environment. On-line programming systems naturally tends to focus on simplicity, whereas off-line systems tends to focus on programmability. See Figure 3.4. The conclusion is that both on-line and off-line programming are needed in various situations. Thus, we want a system combining the two.

Level of "physical abstraction"

The way we describe physical operations (like robot motions), or the way we refer to physical objects, will now be used to classify the level of programming. Let us introduce the term *physical abstraction* for robot program abstraction based on (a model of) the physical properties of operations and objects. The reason to introduce this term is to avoid the usual confusion in the literature with the degree of abstraction for the programming language itself, which is called programming abstraction below. Concerning the physical abstraction, we have what is usually called the degree of abstraction according to the following levels [41, 59, 118]:

Task-level programming is the highest level and it is a research field of its own. The goal is to allow the programmer to specify what to do in a declarative manner, and the system figures out how to perform the task. This level, though not a main subject in this thesis, will be commented in Chapter 4.

Object-level programming also utilizes a model of the environment, i.e., the world model, but normally not including all obstacles etc. Motions are programmed by specifying relations between objects, but the planning is mainly done by the human operator. Implementation of object-level programming can be done on top of a database [181].

Manipulator-level programming focuses on the definition of 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 and constraints in joint space [60] can also be dealt with more easily.

As mentioned, the typical research approach to robot programming has been to increase the level of robot programming to make robots easier to use. However, the following example (modified from [43]) indicates that this is not necessarily true. A task-level program for the small example above may look like:

```
ARCWELD flat_bar ON base_plate
```

which implies that the system should figure out how to place the `flat_bar` on the `base_plate`. This looks very simple, but the required knowledge about the objects, rules how to do the welding, planning of motions, etc. hinders use of such programming in two ways: Such systems and planning algorithms have not been fully developed yet, and in most applications it is easier for the programmer to do the planning and enter an explicit program then it is to enter all data required by the planner. A more explicit, but still on an object level, program would be

```
PLACE flat_bar ON base_plate SUCH THAT
    flat_bar.side IS PERPENDICULAR AND
    flat_bar.end CONTACTS base_plate.weld_line
ARCWELD flat_bar.end AND base_plate ALONG CONTACT
```

which perhaps still looks like a simpler and more elegant way of describing the task than teach-in programming would be. It should be clear, however, that also for a quite simple task a quite extensive and accurate description is required.

Object-level programming supported by graphical programming tools [188, 181] have turned out to be quite useful in circuit board assembly applications. In such applications, component data are available in data bases and the planar geometry of the task maps well on the (also planar) computer screen. On the other hand, in many other applications, manipulator-level programming is more natural for the operator without extensive programming knowledge; a sequence of motion commands directly reflects the way a human worker would perform the task.

The conclusion is that all the different levels of physical abstraction are suitable in different programming situations. That implies that the programming system should be layered in such a way that higher levels of abstraction can be added on top of the basic manipulator-level programming.

Level of programming abstraction

Robot programs consist of statements for motions and for information processing. Either motion statements are built into the language, or they are achieved by use of a robot software library. In the former case we have a special manipulator language, e.g., languages like ARLA [4], AL [135], or AML [204].

It has turned out that major parts of typical robot programs consist of information processing, i.e., robot programs resemble computer programs. This implies that robot programming includes all the aspects and problems of computer programming, plus some additional ones. This is also the reason why recently introduced robot programming languages either belongs to existing computer programming languages (with a robot library), or are a new general purpose programming languages with some special robot programming support (types, syntax, etc.). Examples of the former are RCCL [81], PASRO [41], and HAL [134], whereas Karel [72] and RAPID [6] are examples of the latter.

In conclusion, the similarities to computer programs indicate that the same abstractions and paradigms (abstract data types, object orientation, etc.) should be used for robot programs. However, the difference is that

robot programming should be possible to do in a way that supports "superior manufacturing practices" as mentioned on Page 5. Thus, the level of programming abstraction should be selected depending on application and type of production system. This is typically related to the level of physical abstraction, but does not need to be.

User interface

Robot programming and operation can be performed via a user interface including 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 RPL.
- A graphical user interface, possibly with 3D visualization.

A well designed system may use all three alternatives if appropriate. Fancy graphics is not always the best alternative. Use of the physical world can, for instance, be more beneficial than a graphical view of it in some cases. One such case is when on-line programming using a hand-held terminal is preferred.

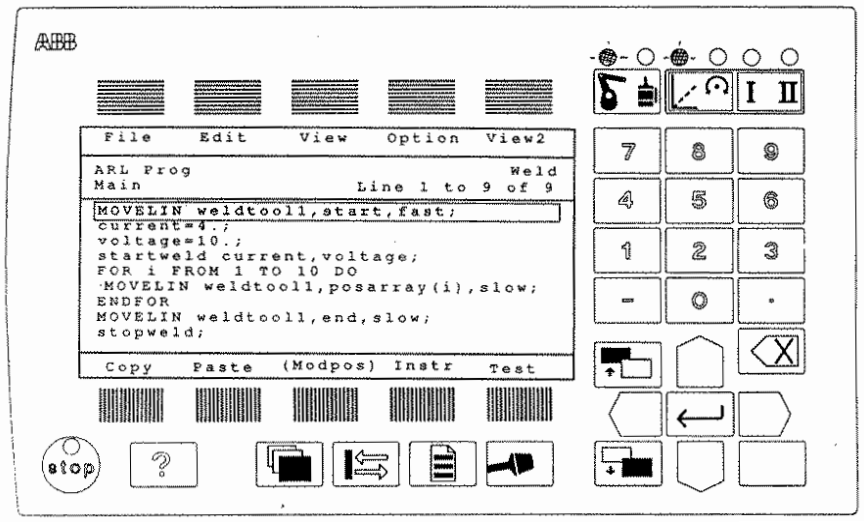

Figure 3.5 Panel layout of ABB hand-held robot programming terminal (first S4 version). Joy-stick, emergency stop button, and 'dead-mans-handle' located beside the panel are not shown.

Hand-held terminals are popular in robot programming because they allow the programmer to move freely during the programming and stay close to the pieces of interest. Practically all robot systems are equipped with some kind of teach-pendant, allowing the operator to manually move the robot and request storage of the current coordinates. When a teach-pendant provides a complete robot operation and programming environment, we call it a hand-held terminal. The first, and most successful system on the market, providing a complete programming environment via a hand-held terminal was ARLA [4] developed at ABB Robotics. Both ARLA and its recently introduced successor RAPID [6] robot programming/operation interface can be characterized by the following:

- Programming can be completely 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. Access to most features are via function buttons, or via pull-down menus in the new system.
- Programming and editing is performed in a syntax-based style using the same type of interface as for manual operation, i.e., all instructions and attributes are easily selected in the menus. Only syntactically correct programs can be written this way.
- Special application support can be defined in a way uniform to the standard interface, i.e., using menus etc. Such functions may even affect the motions during program execution. For example, a spot-weld position may be adjusted without interrupting the robot motion.

In essence, it is a careful design of the user interface in combination with a suitable programming language and style of operation that makes this type of robot programming preferable in a wide range of applications. A picture of the new programming unit is shown in Figure 3.5. The previous version can be seen in use in Figure 2.3 (p. 8).

Future designs of such a terminal could provide a pen- or pocket-computer like interface, voice input, 6 DOF joystick [94], wireless communication with the robot controller, built-in gyro for maintaining joystick coordinates relative to the world coordinates, force feedback to the joystick for programming of force controlled motions. New types of hand-controllers developed for telerobot control [164] may also be useful in the future.

Computer-based interfaces are preferably combined with off-line programming. Availability of 3D graphical models of the robot and its environment then makes programming much easier [59] (pp. 418–419). Several such system are available on a commercial basis, e.g., CimStation [183] and IGRIP [65]. It would of course be desirable to combine such systems and the more on-line oriented systems mentioned earlier. That will be returned to, but an interesting (already solved) special case when on-line and off-line programming can use the same kind of user interface can be found in many circuit-board assembly applications. The "planar world" in these applications can be well mapped onto a computer screen that can be located close to the equipment, and powerful systems like the SMALL system [188] and the AIM system [181] have emerged.

3.3 An approach to integrated programming

Software support for demanding robot applications, requiring dextrous motions and on-line tuning to deal with the manufacturing process and its uncertainties, are the focus in this thesis. Appropriate choices of programming principles and tools are probably most crucial in such applications. A development towards off-line task-level abstract programming only, utilizing modern computer graphics, can be questioned in the light of the following quote [200]:

> *The use of the best available interface techniques does not assure the production of a good interface; a good menu system cannot make up for a poor task analysis. Success in interface design comes when an interface properly addresses the semantics of its users' tasks and domains.*

In other words, programming principles should be decided according to the production situation, not the other way around. This motivates further work in the following directions:

1. Dedicated program packages for specific applications should be developed in close interaction with the application developers and industrial users.
2. Design of robot control systems and robot programming principles supporting application specific programming.
3. Planning and scheduling of manufacturing activities should be done in such a way that hands-on adjustments of the task are taken into consideration.

Whereas Item 2 merits our further attention, Item 1 is best solved in an industrial environment in close interaction with customer support. It is

therefore not treated here. Though planning and scheduling of manufacturing have been subject to extensive research, the need to actually deal with on-line changes has only more recently been considered [58].

Benefits of on-line programming

Advanced robot programmers often consider on-line programming to be a problem. One reason is that when their favorite off-line system is used to create and down-load programs to the embedded controller, on-line modifications then make the programs differ from the version maintained in the central database. (Compare with the large-scale system on Page 7.) The usual research approach is not to use (or even allow) on-line modifications. The approach here will be the opposite.

Another problem with on-line programming of advanced applications has been that the programming language/tools (available on the hand-held terminal) have been too limited. Use of a very simple teach-pendant is probably the reason for statements like the following about a sample application including palletizing ([59], p. 396): "It should be clear that the definition of such a process through 'teach by showing' techniques is probably not feasibly". However, such an application can be very well solved by available built-in functions in the ARLA system [4] from ABB Robotics. The new RAPID system [6] even (potentially at least) allows the advanced user to introduce certain application specific functions in a way that supports on-line usage [46].

A third type of problem in on-line programming is due to limited availability of computing and control tools in embedded systems. Although not explicitly found in the literature, it is a straight forward step to extend the networking principles used in manufacturing to also allow use of host computer software from the embedded system. This has also been implemented in our lab [191] where we used Matlab [124] as a compute server for the embedded system.

In conclusion, we should not neglect, as typically done in university-based research, the benefits of on-line programming. Note, however, that this is not to say that on-line programming suits all situations. In a car production line, for instance, off-line programming is very useful as a way to make the production stop as short time as possible when reprogramming for new car models is to be done.

3.4 Local operation entails local feedback

We are now ready to analyze the misconception "The problem is on-line teach-in programming" from Page 17. Consider a large-scale production

facility with a central engineering department. Let us study the programming and operation of an industrial robot in that production system. Assume that we allow/support interaction with the robot controller in two different ways:

1. New or modified robot programs can be down-loaded from the central engineering computer system.
2. Robot programs can be edited by the local operator on the factory floor.

Feedback control: The flow of information (physical signals, data, or issued operations) in the described situation is viewed in a block-diagram style in Figure 3.6. As shown in the figure, our two types of interaction with the controller result in a cascade control structure; an inner loop for the local operation and an outer loop for the overall control.

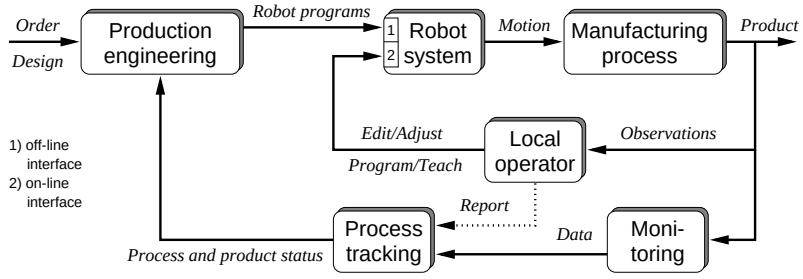

Figure 3.6 Local feedback by local manufacturing operation.

In control engineering, the principle of local feedback is very well known. It is a way of taking care of process variations locally, avoiding effects on the total more complex system that would be harder to control. Local feedback has been of great use ever since Black [40] invented it as a way to make electronic amplifiers insensitive to component variations. That was in the beginning of this century. Successful cascaded feedback typically relies on the inner loop being faster. What about that in this case?

The local operator is typically able to adjust the robot program in a matter of minutes. Operator feedback is applied in the sense that the operator observes the controlled process while adjusting it. The feedback in the outer loop, however, is considerably slower. A problem is typically first detected on the shop floor and reported to the central control. Adjustment then either uses observations from the local operator, or from some type of supervision system. Clearly, the outer loop is anyhow slower than the

inner one. This means that the structure necessary for successful cascade control appears to be fulfilled. But what are the benefits of each loop?

The outer loop is required for management and control of the complete facility. The "production engineering" also includes off-line programming when major changes are imposed from the *Order* and *Design* input, see Figure 3.6. This is for instance the case in car production as mentioned above. In control terms, set-point changes need to be formed in a way that the operation of the systems does not deteriorate.

The inner loop is required for robustness and performance during every-day operation. Efficiency as practically obtained in small-scale production is the key motivation for this; it is desirable to have the benefits of small-scale production also in larger facilities. Specifically, the *Report* path in Figure 3.6 should not need to be used for changes already expressed in the robot program. Instead, such information should be propagated to the "Production engineering" (backwards) via the *Robot programs* path. In control terms, this is outer loop tracking due to inner loop saturation or manual operation.

Flexible manufacturing systems (FMS): As shown in several case-studies [24], modular locally manageable cells improve robustness and flexibility. More specifically, trying to achieve flexibility and efficiency via computer integration and scheduling does often not work very well in practice as expressed[1] in [24] (p. 170). This clearly motivates an approach that supports locally operational units.

The most feasible way to implement the production feedback to the central manufacturing engineering, and to maintain consistency between the on-line and off-line versions of the robot programs, would be automatic retrieval of the modifications done on the factory floor. Manufacturing practice today is, however, less productive. Statements like ([24], p. 156) local operation "allows an operator at the machine tool to edit" the program, and the improved program should only be saved so it "allows an authorized operator to create new production versions". Furthermore, when the structure of robot programs is changed on-line, the changes cannot in general be mapped back to the engineering workstation [59, 183]. This circumstance means that local feedback today actually makes overall manufacturing control harder.

1 "FMSs promise 50% reductions in lead and manufacturing times using only a handful of machines and a few operators. Yet flexible manufacturing systems are far from being irresistible. Many manufacturers, having long considered FMS technology, are now opting for something smaller, simpler, and cheaper. They are buying flexible manufacturing cells and even stand-alone CNC machine tools".

3.5 Internal states and external reality

Transfer of robot programs between on-line and off-line systems may look like an ordinary software translation task. There are known problems with too restrictive RPLs as explained on Page 22 (see also [59], pp. 431-433), but let us assume that the on-line language is powerful enough.

A robot program created in an off-line programming system makes use of the world model, which is an abstraction of the ambient work cell. In on-line programming, however, the real physical world is there, which makes some simplifications possible. There are, on the other hand, cases when data have to be kept and maintained in a way that reflects the external reality. Such a principle of organization, known as the *internal model principle* in control theory [221], therefore appears relevant also for robot programming. The purpose of this section is to present an (until now overlooked) on-line programming aspect which requires special treatment in integration of on-line and off-line programming.

Object views in world modeling

State of the art off-line programming entails object-level programming using a world model as described earlier. To facilitate transfer of robot programs later, the attributes of the object in the worlds model will now be classified. The term *object view* will be used to refer to a group of attributes (or methods) that capture a certain aspect of the object. This is in conformity with views as defined in process control [199, 1], control engineering [142], and software engineering [44]. It is of course preferable to have support for views in the programming language. The SIL [184] RPL provides this feature in connection with multiple inheritance. In languages without such support, views can be maintained by management of object attributes.

Object-level robot programming can be viewed as object-oriented [126] support concerning the physical properties of the objects within the work cell. 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? Modeled in a off-line programming system, we can think of the following object views:

Physical attributes represent physical properties that should be possible to refer to when specifying robot motions. This of course includes the geometry of the physical object, but also the dynamic properties like mass and inertia.

Graphical attributes are needed only for the graphical presentation during off-line programming.

State attributes represent the state of the physical task. This view is defined for on-line programming purposes.

The state (called soft in the original presentation [144]) view and its use in the sequel is believed to be new. A related but different classification of the physical and graphical attributes has been proposed by others (introduction to Chapter 8 in [112]), but for off-line and planning purposes.

As an example, consider a pallet being used in an assembly task (as in Chapter 12 in [59]). Considering the pallet object and the objects for the pieces fitting into the pallet, an off-line world model today would have attributes like:

1. Physical attributes would typically be 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. Graphical attributes for nice 3D animation.
3. State attributes would not be present.

Instead of state attributes, the state of the application (the occupied slots of the pallet) would be implicitly defined by spatial relations (affixments etc.) between objects in the world model [112] (p. 451). An off-line robot program can then directly access the world model and compute the state of the application whenever needed to determine what move to perform. Alternatively, robot program variables are declared and used for bookkeeping of the current state of the task (as in the palletizing examples in [59]).

Having the state of the assembly spread out in variables makes it difficult to transfer and maintain object properties during on-line programming, which is needed for later retrieval to the off-line system. Recall that the overall manufacturing control (i.e., the outer loop in Figure 3.6) needs feedback from performed on-line changes.

The most successful software technique to keep related data together is object-oriented programming. Object-oriented off-line programming [112, 184, 188] is straight forward. Software objects can encapsulate or refer to world model objects. It is, however, not clear that object-oriented on-line programming suits a typical production engineer or robot operator. The manipulated objects are physically available and sequences of operations are well expressed in an ordinary imperative language. But even if object orientation is appropriate for off-line programming and planning systems, and it might be useful also for on-line programming, we do not want to resort to a certain paradigm for successful robot programming. Instead, such states of the assembly that are used to determine motions

should be stored in the proposed state attributes in the world model. It is then up to the robot programming interface what paradigm that should be supported for the end-user programming.

Specification of motions

The general problem of motion specification includes formalisms for rigid body motions, representations suited for calibration of models according to measured geometries, combination of force/position specifications, and handling of compliance. See for example [112, 176, 177, 59] and references therein. Even if ongoing research may result in other ways to describe a desired motion, there are some fundamental aspects of transferring motion specifications between on-line and off-line programming environments. Let us approach the problem from the off-line programming side.

For simplicity, let us assume that motions are defined by simply specifying the via points and the end point of the motions, i.e., specification of velocity profiles etc. are neglected for clarity of the discussion. Frames are represented by 4×4 Denavit-Hartenberg transformation matrices as standard [64, 59]. Their use by motion instructions form what is called *position equations*, for instance in RCCL [81]. A position equation for the above example can look like

```
ARM * TOOL = fixture * base_plate * start_pose
```

where the `ARM` for example includes the kinematics of the IRB-2000 robot and `TOOL` models the weld tool used. In RCCL and in powerful off-line programming systems, the system keeps track of the factors (objects) in the equation, and the equation is solved with respect to the free variables. Typically, this means that the `ARM` transform is computed, and the desired joint angles are then computed as the solution to the inverse kinematic problem for the manipulator used.

In an on-line programming environment, the `ARM` transform is known by the robot system. The `TOOL` data is entered by the user according to the tools actually used. When the programmer using teach-in programming defines a position, he/she defines the right-hand side of the position equation. It may also be possible for the user to define some intermediate frame. For instance, the `fixture` frame may be specified using teach-in. Successive positions can then be defined relative to that frame.

World model simplifications

When we transfer a desired robot pose from the above off-line description, to an on-line system, it is in the simplest case only two frames of the equation that are required 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. After down-loading to the robot controller, these two frames will simply and concretely describe the geometry of the tool and a position to which the robot should move. Use of the introduced object views may facilitate the extraction of the geometries in that only access to the physical view of the world model objects is required. Instead of single frames, frame sequences modeling a path on the object can be retained for on-line use. For objects with spatial attributes which depend on time, path coordinate, or sensor signals, those dependencies must of course be maintained.

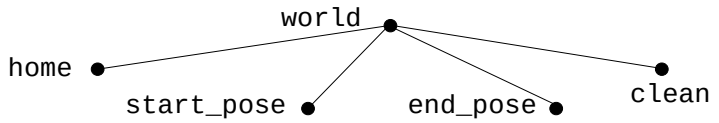

Figure 3.7 Simplified world model for on-line use. Intermediate frames in Figure 3.3 have been eliminated.

As an extension for advanced on-line programming, the entire physical and state views of the off-line world model could be transferred to the embedded system which would only imply a minor increase of required memory. 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.

Translation to an on-line system [59, 65, 183, 41] means that the robot program expressed in the language of the off-line system is translated to some other so called native language of the on-line system. Thus, each equation will define another goal frame that will be used in the translation to the on-line system. In the on-line system, these frames will define positions just as if programming would have been carried out by teach-in. Intermediate frames without explicit access from the robot motions can be eliminated. Instead, the eliminated frames are represented by the real physical environment. The resulting simplified world model for our small example is shown in Figure 3.7. This is supported by systems today [65].

Recall from Section 3.4, however, that we also need to translate the (changed) program back to the off-line representation to achieve the desired property of local feedback. That is not possible today. To prepare for a solution, let us consider the syntax tree [12] for our example task. For simplicity of the discussion, we will only consider the motions, i.e., that the control of the welding equipment is omitted. Figure 3.8 shows

an abstract syntax tree for the sequence of four motions. Each motion statement will refer to the world model as shown in the figure.

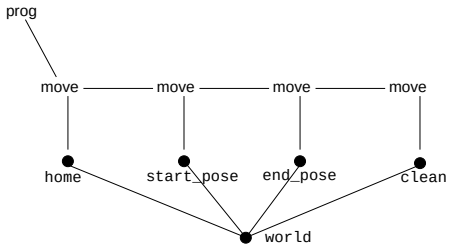

Figure 3.8 Abstract syntax tree (upper part) and simplified world model (lower part) for the four motions program. The world model is turned up-side-down as compared to Figure 3.7 for better visualization of connections.

In the case that the robot program is first created on-line, the ordinary robot programmer will define a sequence of positions 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. Uploading of coordinates is possible today but, as mentioned, not upload of complete programs. The major reason is that today's on-line languages are too limited (as already explained). But even with a richer language like RAPID [6], translation of robot programs includes some special aspects due to differences in preferred programming style (different requirements for the inner versus outer loop in Figure 3.6). The following treatment of intermediate frames will illustrate the problem.

On-line frames

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, the location where to weld in the example in Section 3.1 is preferably defined relative to the location of the base plate. Then, if the mounting on the fixture is changed, only the frame describing the base-plate location needs to be changed. Programming of the welding would then include the definition of 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. Most robot systems provide such a feature. Having the robot controller to store this structural information actually means that we keep an intermediate frame of the world model to support work-piece relocation, i.e., to support flexibility. We will call such an intermediate frame an *on-line frame* in the sequel.

When a robot program is created off-line, after defining the actual world model, the programming system in practice imposes robot motions to be specified relative to the manipulated physical objects. See for instance [65] for a further description of the methodology. Furthermore, robot motions will refer to the frames actually defining (the model of) the environment. Therefore, moving the base plate in the example above will automatically update the description of the robot motions.

In the on-line programming case, however, the on-line frames do influence the robot motions but an incorrect on-line frame may be hard to discover before the motion is performed. To prevent damaging or dangerous motions, it is therefore crucial that the on-line frame is properly maintained according to the states of the physical objects. Note that even if this aspect of on-line programming is possible to simulate in a system like CimStation [183] (using a world model feature called proxies [185]), we are here concerned with the actual transfer of program to embedded systems. The following example further explains the problem.

Context sensitivity

The term context, within computer science, denotes a set of bindings to the environment or surrounding scope [38]. A problem within robot programming is that robot program execution changes the state of the environment (for instance when two pieces are welded together), and the influences on the environment are sometimes hard to reflect or know in the robot program (the success of an operation may depend on unknown disturbances). We therefore have a context sensitivity problem to deal with [59] (p. 409).

It is of course important, in both computer and robot programming, to provide good methods to cope with context sensitivity. Sometimes we may need external sensors to detect if an operation has been successful or not. That is a standard and explicit way to handle uncertainties in the environment. Here, we instead look at such context sensitivity problems that are due to operator interactions. This aspect of the problem has, as far as known, not been treated elsewhere.

Assume that the flat bar in our small example turns out not to be quite straight. Also assume that the operator decides to adjust the welding path by inserting a via-point between the `start_pose` and the `end_pose`. We will now compare different ways to modify the program:

In the off-line programming case we modify the model of the flat bar. We then insert an intermediate position in the weld path. The intermediate position will refer to the updated flat-bar object. If the `base_plate` is relocated on the fixture, the complete (simulated) weld

path is relocated by the system.

In the on-line teach-in programming case we first locate where in the program the additional statement is to be inserted. That can be done in two ways:

1. We can load and run the existing program, stop it when the robot approaches the welding task to be modified, and then issue stepwise execution (typically without welding) until the robot has reached position `start_pose`. That move statement will then be the active one, and an additional 'via-move' can be inserted after the current statement.
2. Instead of letting the robot execute the program until we are at the right place, we may use the on-line editor to make the same statement as item **1** to the current statement. This is quicker if the operator knows the program.

The difference between these two cases is that in case **1** we activate the on-line frame by running the program, while in case **2** we manually have to do that activation since the activation statement was not executed. **If the programmer forgets to do the manual activation**, or in any of the cases forgets to refer to the on-line frame in the move-statement inserted after the current one, there will be a severe problem. The inserted via-point will be expressed in the wrong coordinate system, **the result will be a completely wrong (and possibly also damaging or hazardous) motion**. This will show up either directly or after relocation of the `base_plate`. Analysis of the problem in general, and of special cases in special systems, could go on for another page but is not very interesting. The conclusion is that systems more (see [144] referring to [4]) or less (as in [6]) unnecessarily require that the programmer does the right thing.

From this simple example we can see that although on-line programming is in principle a tangible and user-friendly method, we may encounter problems when we want to utilize structural information about the environment. In other words, when an otherwise superior on-line programming method makes assumptions about 'object' properties, there will be problems if the state of the software objects have not been properly maintained during manual operation.

In computer programming, such problems are avoided by the sequential execution model, scope rules, and the absence of externally maintained states. The problem here is that we manually have to maintain variables (or object attributes if an object-oriented framework would be used) corresponding to states in the real physical environment. This can be called "physical context sensitivity" due to operations on physical objects. An-

other such case can be definition of frames describing the tool; when the robot changes tool, the frame describing it must also be changed.

Note that even if the "internal model versus external reality" problem as such is mentioned in the literature [59], we are here concerned with how the on-line programming language and system can support the programmer, and how such support can be translated to/from the off-line programming system.

Further examples of physical context sensitivity are software controlled setup of external equipment for welding, grinding or gluing. This implies that it is not enough with fixed specific support for on-line frames in the robot system. We need a mechanism that permits the developer of application packages to introduce application specific scopes in the language. This must be part of the syntax so that the syntax-based editing gives the ordinary user the desired support.

3.6 Integrating on-line and off-line programming

The following solution to the problem defined in the previous section is inspired by syntax-based programming tools. One example of such a programming tool is the ABB interface described in Section 3.2. That system is built on a predefined programming language and operator interface. Another more dynamic, but experimental, system has been developed in the Mjølner project [107]. That system has been used here for prototype implementations [152]. The key idea is to map objects and accesses thereof to blocks and scopes of the on-line programming language.

Figure 3.8 shows the simplest situation when the on-line program refers to positions without any on-line frames. The program according to that figure can either be written on line using teach-in programming, or it my be down-loaded from the off-line system utilizing the simplified world model in Figure 3.7. The same program, but using an on-line frame, is visualized to the left in Figure 3.9. Then, we want to insert the intermediate move statement. Figure 3.9 (right) shows the desired situation after insertion. So far everything is correct. However, inserting the intermediate position in the on-line system easily results a robot program according to Figure 3.10. This is due to the deficient handling of the context sensitivity problem as described in the previous section. The figure clearly shows that the inserted location refers to some earlier defined `old_base` instead of the correct `base_plate`, but this is hard to expose to the programmer. In some existing systems, the on-line frame is activated by a separate statement in the program [144]. The programming fault can then only be seen indirectly on coordinates attached to the motion statements. A better ap-

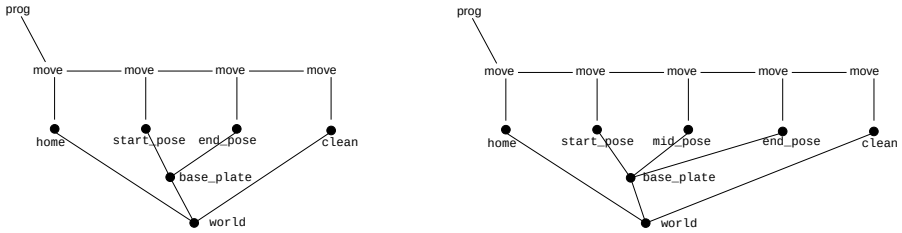

Figure 3.9 Positions referring to an on-line frame `fixture` (left), and a correctly inserted intermediate via point (right).

proach is always to display the frame list in each move statement. That approach does, however, not scale very well in complexity. Hierarchical use of on-line frames, or longer motion sequences using the same on-line frame, clutters the program in a way that is undesirable for hand-held terminals.

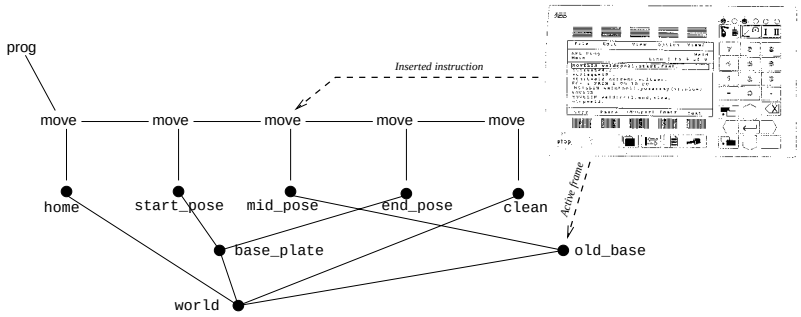

Figure 3.10 Inserted move instruction referring to an incorrect on-line frame. To the upper right, the teach pendant shown in Figure 3.5 illustrates manual operation. The `old_base` is the most recently used on-line frame, which may be from an other task in an other part of the working range of the robot.

The following claim is fundamental here: The location of the base plate is in reality part of the physical scope for the motion to be performed. Expressing the welding path in a robot program should then utilize scope rules as in computer programming using standard imperative languages. The same applies to control of welding equipment etc., which entails that the description of the welding path should be expressed in a surrounding scope including the actual on-line frame. This implies that the concept of on-line frames should be part of the syntax for the RPL, which provides new possibilities to support the robot programmer by syntax-sensitive editing.

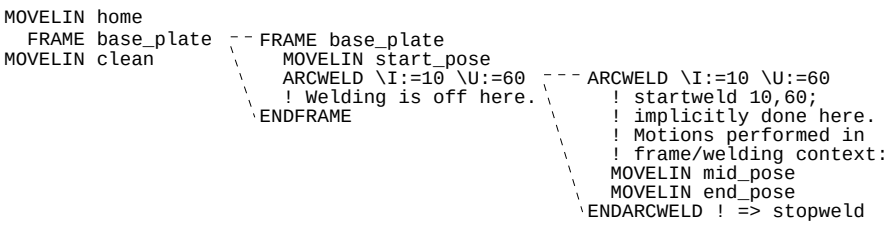

Figure 3.11 Utilization of RPL extensions in syntax-based editing. When a block is opened the system ensures proper initialization of system settings reflecting the physical context. Deactivation, like turning of the welding before moving to some other task, can in this way also be ensured by the system.

Another approach would be always to keep the geometrical description in a separate data structure. That would correspond to maintaining an explicit world model (typically much reduced as compared to the off-line version) in the control system. Editing that world model would, however, put the same demands on the system as when we give the motion coordinates directly in the motion statement.

Considering also the control of the welding, we need to specify a current and a voltage. Start and stop of the welding have to be expressed in the program together with the welding motions. Using separate statements/procedures for start and stop of the welding, as indicated in Figure 3.5, again resorts to proper management of the physical context (in this case the state of the welding equipment). If we extend the syntax and semantics of the RAPID language, our small example may look as shown in Figure 3.11 after zooming and modification.

Note that when the frame block is opened, the on-line frame associated with the `base_plate` object is automatically activated. That is achieved by semantic rules for this syntactic production. Of course the programmer can still explicitly use some other frame if desirable, but the default behavior (obtained by the non-expert robot programmer) does not result in an unexpected/undesired motion as before.

3.7 Implementation

Having realized that the robot programming languages used in on-line and off-line programming in many cases need to be different, it is apparent that automatic translation between these languages is needed. It now remains to show how the desired bijective mapping between the representations can be achieved. A full prototype is currently under evaluation. The implementation represents a considerable effort, which was possible

by initiating and guiding the work presented in [152].

Two major industrial products were selected to represent the off-line and on-line programming systems. The off-line system is the IGRIP [65] from Deneb, Inc. The off-line model for our example task was shown in Figure 3.2 (p. 22). The recently released S4 control system from ABB Robotics was selected as target (on-line) system. The RAPID language interpreter of that system was not available for experimental purposes, so we implemented our own RAPID to C compiler. By utilizing the architecture that is presented in Chapter 4 together with the robot system presented in Chapter 5, and using software techniques presented in Chapter 7, we emulated the front end of the ABB system. Thus, the program transformations involved are those shown in Figure 3.14. (An on-line connection between IGRIP and the experimental robot controller was also developed, but that is for other purposes as will be described in Chapter 8 about advanced applications.)

The structure of the IGRIP system is shown in Figure 3.12. The input/output marked UNIX files in the figure is used to import and export robot programs. The programs were written in the internal language GSL. Programs were, however, exported as Karel [72] programs. Karel is very similar to GSL but the format of the Karel files were more suitable for our translation purposes.

Translation of Karel to RAPID, and RAPID to Karel, was achieved by using the Application Language Laboratory (APPLAB)[190] software.

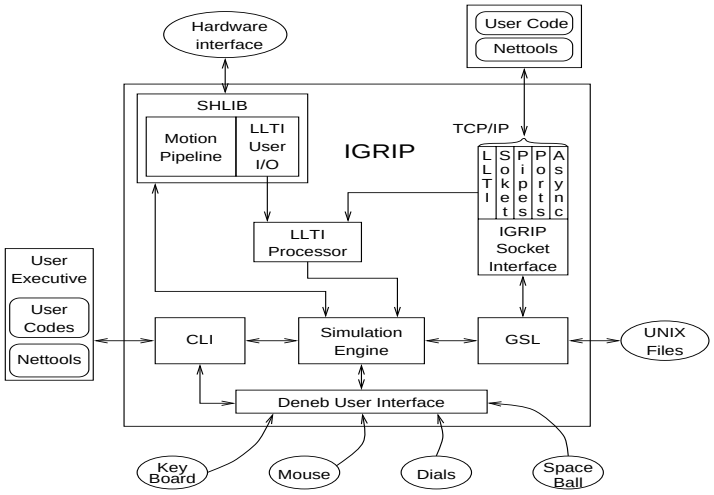

Figure 3.12 Software modules and interfaces of the IGRIP system.

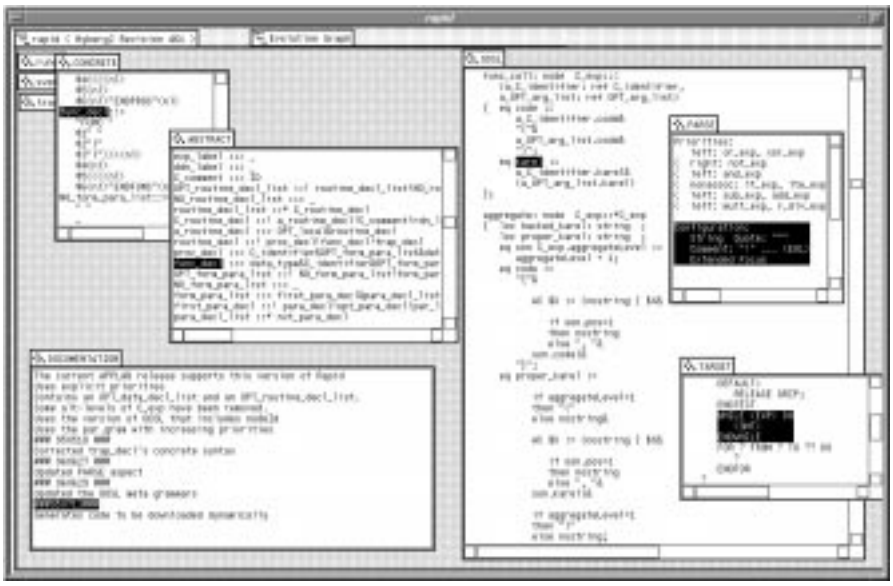

Figure 3.13 User interface of the Application Language Laboratory [190] used for prototyping robot programming extensions.

A very important feature in APPLAB is the interactive and incremental definition of grammars and its automatic generation of syntax based editors for the specified language [107, 82]. To achieve that, the tool uses a special internal data structure, but that also makes it more difficult to parse programs expressed in ordinary text form. An important extension which was developed in connection with this implementation is a dynamic parser [39] which was used for the parsing of robot program into the language laboratory. Target language output is a special case of

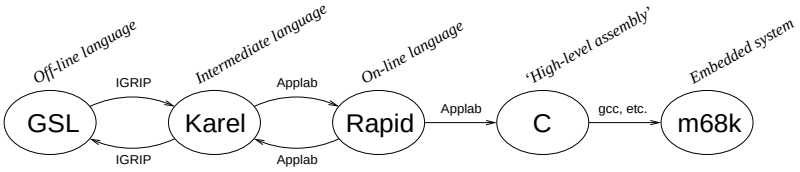

Figure 3.14 Implemented translations of robot programs. The C and m68k representations emulate the ABB system (according to ABB specifications or according to the proposed extensions depending on Applab grammars). The gcc, etc. denotes the cross compiler and our tools for dynamic binding (explained in Chapter 7).

code generation. Figure 3.13 shows the user interface of APPLAB when developing the grammars.

There are of course many semantic details that need to be worked out before the system can be used industrially used, but the approach to integrated robot programming as presented here appears to be quite promising.

Operator safety

Trying to integrate on-line and off-line programming for more efficient manufacturing control is clearly being more complicated due to the proposed management of context sensitivity in the on-line system. It is therefore natural to question the dedicated support for on-line operation, but the following indicates that this is important

An unexpected/undesired erroneous motion can of course result in damage of equipment. But more importantly, unintended motions can be hazardous to the robot programmer. According to the rules, the robot programmer are not allowed to enter the robot work-space when the robot is in operation. In practice, however, "*insufficient planning of the running work process resulted in workers having to interfere while the installation was in operation, e.g. in order to readjust some robot component, program the machine or remove remains of cast metal from the workpiece*" [164] (p. 250).

There are of course many types of possible incident types, and many other aspects may be more common/important. This is on the other hand hard to know because "*only a few original reports on accidents, critical incidents and abnormal stoppage cases have been reported up to now. Authors often rely on second-hand information and on the reinterpretation of available statistics*" [164] (p. 249). Nevertheless, wrong motion has been noticed as a primary risk: "*In order to execute and check exact adjustments, the programming as well as a trial run is usually undertaken in the direct vicinity of the robot. Thus possible dangers in the course of this activity are: the programmer might enter a wrong direction, thereby causing the robot to move towards him/her;*" [164] (p. 245). These observations, and the trend towards more advanced applications (more complex and nested contexts), therefore motivate the proposed support for shop-floor operation/programming.

3.8 Summary

A review of different robot programming methods shows that a variety of methods are beneficial in different programming situations. As to support

manufacturing practices, it should be possible to provide the end-user programmer with proper programming tools, programming language, level of abstraction, and use of the robot. The crucial point is then how these techniques can be integrated. After presentation of a small example and a review of techniques for robot programming, the following contributions formed a new approach to end-user programming:

- Some benefits, like easy-to-use features for quick adjustment of the manufacturing process, of on-line programming were pointed out in Section 3.3. These benefits are fundamental to the approach taken, but they have until now been deficiently observed in academic research.
- An interpretation of robot programming and operation in control terms, as done in Section 3.4, further motivated on-line operation as a way to introduce local feedback. The formulation of the programming process in terms of feedback control appears to be new despite its similarities with other interactive uses of computers. The control interpretation clarifies the need for back-propagation of local changes of robot programs (that is, automatic translation both ways between the on-line to the off-line system is highly desirable).
- To facilitate translation of robot programs, three object views were suggested for world-model objects (Section 3.5). A new way to illustrate robot programs, with the syntax tree connected to the world-model tree was also introduced. A key observation, not found in the literature, is that context sensitivity in the on-line programming case is related to states that today have to be manually maintained by the operator.
- A new way to manage the context sensitivity in on-line programming was then proposed in Section 3.6. The key idea is to make use of scope rules also for the (application specific) system states. Utilizing the block structure of the programming language implies that tailoring the system to a certain application may require extension of the syntax of the on-line language. That makes it possible for the syntax-based editor to maintain important system states.
- The "Application Language Laboratory" software tool applied to implement the proposed ideas forms a system with unique robot programming properties; "meta grammars" and "incremental semantic analysis" make it possible to incrementally during robot programming extend the syntax of the language as required in previous item.

In conclusion, a new approach to end-user programming has been proposed.

4

Architectures

The term architecture may generally denote structure or style of structure with various—and sometimes not very precise—meaning. For the purpose of industrial robot control systems, and for this thesis, an architecture denotes **the concepts and techniques that characterize the structure of the system**. Then, there is the question about what should be considered to be the characteristic properties. The approach adopted here is to base that on the usage of the system, rather than on the internal design.

The purpose of defining a system architecture normally is to support understanding and implementation by coping with the complexity of the system. That also improves modularity and reuse of software components. Here, aiming at improved manufacturing support, application aspects will be the main objective. This is approached in the following way:

- Taking a broader intelligent machines perspective, a review and classification of architectures found in the literature will be presented in Section 4.1, where also a concept of *user views* will be introduced.
- Section 4.2 treats architectures and abstractions used for so called intelligent robots. That is to draw some conclusions about the special case when robots are operating in more well defined environments, like in the manufacturing case.
- Finally, in this chapter we propose an architecture well suited for programming of industrial robots.

From an intelligent systems perspective, overall control of fully intelligent or autonomous robots is outside the scope of this thesis. Instead, the aims are to find proper abstractions for industrial robot systems, to make such systems fit into more complex autonomous systems, and to facilitate autonomous robot operation for well defined applications [98, 49].

4.1 Role of Software Architectures

There is no way the designer of the system can foresee all application demands. For the motion control in particular, we realize that the motion control properties must be possible to modify in advanced applications. Secondly, to provide sufficient scope for task-level programming it is desirable to preclude unnecessary interference or obstruction deriving from software architectures. This is a problem that has been overlooked within robotics research. One reason is that research systems can be completely open and they do not have to cope with a lot of special (customer) requirements. Development of specific control functions have therefore been neglected by statements like "just implement another procedure", a misconception that has already been commented on Page 17.

From an industrial perspective, development towards more advanced industrial equipment should start from cost efficient, basic, but open, control systems. Robot systems for instance, can then gradually increase in complexity wherever appropriate. This approach minimizes the technical risks and allows machine development to go along with the (sometimes slower) evolution of manufacturing practices. Solving the problems of high-volume applications first gives experience and pay-off at the same time.

Such a basis for the architecture appears to reflect one of the fundamental problems of control system design. The "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)" [13] is one of the most well known architectures for robots. This candidate for robot software architecture has also been directly supported by some experimental control systems [196].

The NASA architecture for telerobots

The general structure of the architecture is shown in Figure 4.1. Each of the six layers has a horizontal partitioning into sensory processing, world modeling, and task decomposition. On the lowest level for instance [69], 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 second level, called Primitive, equips the motion control with functions like gain scheduling, adaptivity, i.e., functions that run less frequently than the servo level algorithms. The frequency hierarchy is the dominant type of abstraction in NASREM. The E-Move level handles pieces of robot programs for elementary operations. One such operation can be "pick bolt from pallet", assuming that the robot is properly

positioned somewhere close to the target objects. In an industrial situation, composition of such (parameterized) unit operations are also used for flexible manual programming of assembly tasks [95].

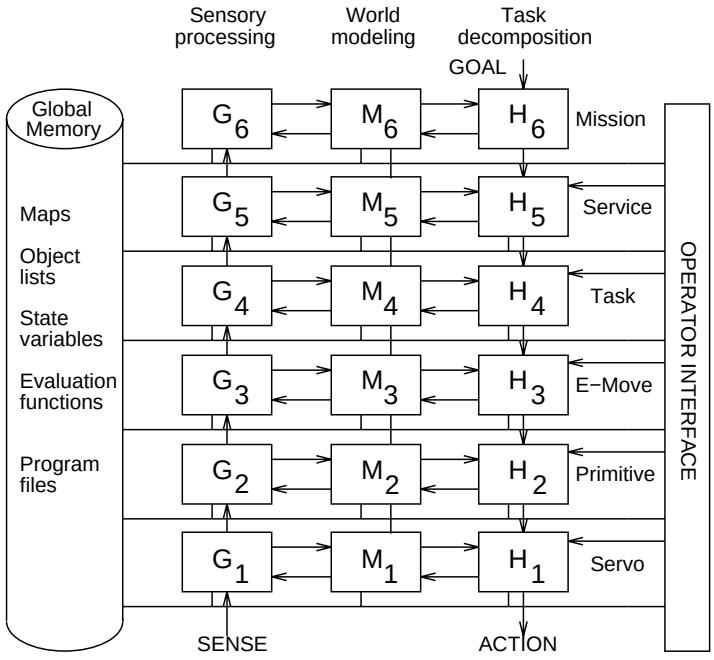

Figure 4.1 The NASREM control system architecture for telerobots

Application of the NASREM architecture has been suggested both to autonomous and industrial robots [56, 127]. However, both directions can be criticized, indicating that NASREM is not the solution to our problems. A closer look from two points of view—robots acting in *uncontrolled environments* and *intelligent industrial robots*—reveals a number of problems. Such analysis, however, requires some basic distinctions as to the functionality and to this end we present some fundamental concepts.

Fundamental concepts

The following list is an attempt to classify different architectures.

Hardware: While the computing power of microprocessors still was the major limiting factor for advanced robot control, the main purpose of an architecture was to define a hardware structure that provided the required real-time computing power. Descriptions of several such

architectures have been published [209, 20, 19, 206, 113]. The hardware structure is still important. Experimental systems should be modular in such a way that a variety of interfaces, sensors, special computers, etc. should be possible to easily add or replace. This is of course desirable also for industrial systems, but low cost (at least for the basic system without options) is usually even more important. Chapter 5 contains an example of an appropriate experimental hardware configuration for the purposes of this research.

Control: The manipulator dynamic control problem as such, especially considering integrated position/force control, is a challenging problem that has inspired to a lot of research efforts [59, 193, 109, 136, 100]. A scheme that solves the control problem can be seen as an architecture. It typically involves several interacting control modules. From a control theoretical point of view, other aspects of the system design are often considered to be just a matter of implementation. More experiment-oriented research, on the other hand, often defines the control and the physical architecture jointly [209, 15, 165]. While others explicitly talk about a control architecture [110], we will use terms like control structure and block diagrams.

Task specification: Systems and methods designed for convenient description of the task that the robot should perform, is another source for control system architectures. This is the topic of end-user robot programming (sometimes related to principles used in process control systems for the cell and factory control levels). Solutions range from explicit manipulator programming [188] and up to systems where the system automatically generates or links robot programs. No general purpose system for automatic robot programming has yet been developed. Systems have been designed, however, for supporting some particular aspects of task-level programming [210, 117], or for specific applications [181, 115, 95, 49], or using special algorithmic concepts [67, 92, 170]. An appropriate control system architecture should of course be compliant with such approaches. They may turn out to be useful in future robot programming systems. In this work, a software layer for task-level programming will be specified concerning its relations to other parts of the system, but selection of specific methods are subject to other ongoing and future research projects.

Abstractions: Robots operating in an unstructured and mainly unknown environment must make excessive use of external sensors, and they must perform dynamic world modeling and real-time planning. Such systems are more complex than the control systems for robots in manufacturing. Two questions deserve further investigation:

- It would be preferable to be able to use standard industrial robots as components/modules in more complex autonomous systems. What kind of demands does this put on the industrial robot controller?
- Can principles from intelligent robot control systems be beneficial also for industrial manufacturing robots?

These questions are treated below in Section 4.2.

User views: From the perspectives of various categories of programmers that need to configure or program industrial robot systems, several programming situations may be identified. When solving a specific application problem, we may need to modify the system in several ways (control laws, operator interfaces, etc.) requiring different types of competence. Assume that we have one user type for each type of required competence, each user type viewing the control system in a certain way. Unless the system is carefully designed, any particular one such view will be unnecessarily complex (involving a variety of computers, programming environments, special restrictions on use of software interfaces, etc.).

If we instead base the architecture on properly selected user views, it is more likely that programming can be done more conveniently. This approach differs from before in the way that the external view, rather than the internal implementation, is the primary matter. A view may map well onto internal modules based on some of the above principles, but it need not to be the case. If we focus on the software design for one specific view, the use-case driven approach by Jacobson [96] would be the object oriented design method corresponding to the user views of the system. Looking at objects, rather than on the entire system, related view-based solutions have been proposed in process control [199] and in computer aided control engineering [142].

In conclusion, control system design can be viewed in several (relevant and equally important) ways. Motivated by the wide variety of users/engineers of manufacturing systems, and by the importance of manufacturing practices, this thesis tries to forward the concept of *user views* as the fundamental principle for industrial robot control systems.

A specific implementation will, of course, be based on a certain hardware and control architecture, and means of task specification for the end-user will be provided. Chapters 3, 5, and 6 each present examples of such implementations (without calling it architectures).

Defining appropriate user views for manufacturing operation will, of course, not solve complexity problems encountered for so called intelligent

robots working in uncontrolled environments. For such robots, the author acknowledges that architectures according to the abstractions item above are more appropriate. The aim is, however, that robots for manufacturing should be useful as modules in fully autonomous systems. Furthermore, suitable abstractions are important also for task-level robot programming and autonomous operation in some manufacturing situations. The following treatment is therefore of interest.

4.2 Intelligent robots

We will now study how different forms of abstractions have been used to define architectures for so called intelligent robots. We will then see how these principles can be used for more conventional industrial robots, and how industrial robots can fit into so called intelligent systems.

Abstraction beats complexity

The purpose of abstractions is to cope with complexity. When a complex system is divided into smaller manageable parts, those parts can be given a new simplified interface, and aspects of the internal behavior is omitted or simplified in some sense. We find this principle in organizations, industrial production (Figure 2.2 on Page 7), control theory (cascade control for instance), and in software engineering (abstract data types etc.). The abstraction then allows more powerful hierarchies to be built, using more abstract interfaces on higher levels of the hierarchy.

An interesting question about any control system architecture is what type of abstractions or hierarchies it is built upon. Consider a multilayered real-time control system for fully autonomous robots. Such a system will contain both hard real-time software and artificial intelligence (AI) related features (planning). The detailed implementation of such a system would of course make use of data abstraction and other software paradigms, but some kind of high level abstraction is needed to build hierarchies to cope with the complexity [127]. Several such hierarchies have been proposed by different researchers as presented by Schoppers [173] according to the following:

Frequency hierarchies [13, 202] are based on the standard 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 [111] are closely related to the data abstraction ideas—e.g., in object-oriented programming. A lower software layer provides an abstract machine for the adjacent higher layer.

Representational abstraction hierarchies [13, 202] is normally used by AI people as the method of building an abstraction by suppressing or ignoring information.

Deresolution hierarchies [131, 119] 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 [111, 13] 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 [50] 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 [108, 202] are designed so that higher levels manage behaviors of longer duration. Note that higher-level computations associated to actions over a longer period of time may require updating and re-computation more frequently than lower levels do.

An attempt to a uniform approach for the design of software architectures is **behavior abstraction** [173] which is not about the software modules themselves. Instead, the definition's focus is on the behavior being generated, i.e., the effects the modules have on the hardware being controlled. The idea to let the architecture specify external properties instead of the internal design is in common with the approach in this work. Topics about behaviors and behavior control, however, are subject to prolonged discussions [56] and outside the scope of this thesis. Let us therefore continue with aspects related to industrial robots.

Uncontrolled environments

As mentioned earlier, 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 the motions according to environmental changes. A 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. An architecture should of course aid in the development of such systems.

From an AI point of view, an upper layer plans and sends down detailed actions to the next lower layer, keeping the abstract plan in the higher layer. Unless some more sophisticated management of the interplay between the layers is introduced, the lower layer cannot cope with changes of goal or environment and will be stuck dealing with the originally expected situation. Such a modification appears to be very hard to combine with hard real-time requirements. A more promising approach would be to also supply more abstract actions and replanning functions along with the detailed orders, a problem which is still a research subject. Though the selection of suitable abstractions and associated architectures is still a topic for ongoing research [222, 180, 80] and discussions [127], a promising prototype implementation of high level concepts for robots in space is underway [175, 174].

From an embedded control point of view, when it comes to full implementations of autonomous robot control, it is of course very desirable to use "off-the-shelf" manipulator control systems already widely used in industrial applications. That would decrease the development effort and/or increase reliability. To meet the special demands like special interfaces to the planning levels, a very open manipulator controller would, however, be required. Development of intelligent industrial robots can hopefully lead to this. Another reason would be to support a bottom-up approach (build the system upon principles that have turned out to really work).

The top-down approach dominating the intelligent robot research seems to be a major problem. It appears to the author that there are too many possible solutions that are subject to investigation prior to implementation (if that is ever done). A bottom-up approach would therefore be more appropriate, as pointed out also by Harmon [79]. That means using open and flexible industrial robots (tailored to the high-level requirements) as modules for full implementation of specific intelligent robot control tasks. Experiences from many such prototypes may likely influence the definition of a suitable architecture for fully autonomous robots.

Intelligent industrial robots

A standpoint declared in this thesis is that open control systems provide robot systems with flexibility and wider scope of application. Now we turn to the question whether the NASREM architecture is suitable for such flexibility.

Figure 4.1 shows that there is a global database and an operator interface connected to all levels of the system. This is good in the sense that information from lower levels can be useful on higher levels, and the operator may need to change or monitor the low level algorithm during

for example new advanced robot tasks. However, our desire to have a lean and efficient system rather means the opposite—i.e., that control signals, sensor values, parameters, etc. should be kept as locally as possible in the system [34]. The standpoint in this work has been that data flows between the different software layers, required for certain tasks, should be created on demand. This has been hard to achieve in embedded systems without changing the software on the lower level.

Another observation in NASREM is that sensor signals only enter the system at its lowest level. This reflects the frequency hierarchy; the most frequent use of sensor signals is in the lowest layer where the highest sampling frequencies are used. However, sensor-based functions on higher levels may very well need fast enactment, as the deburring and welding applications in Chapter 8 will show. Furthermore, sensors used only on higher levels leads to unnecessarily high data flow through lower layers.

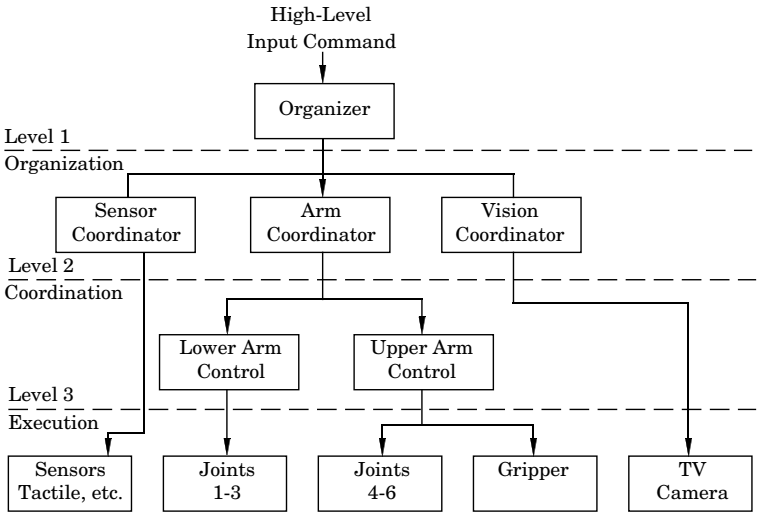

Figure 4.2 Structure of hierarchical intelligent robot control system (from [210] referring to [171]).

Looking for something more dedicated to industrial robots, an architecture that is often referred to is exemplified in Figure 4.2. As with NASREM, this is a way of organizing the control system internally. However, for the industrial robot controllers, an object-oriented design and implementation would be sufficient to cope with the internal complexity. A structure according to the figure can then very well be used, but should that be specified by the architecture? As mentioned above, the proposed architecture will be based on user views supporting convenient factory

floor operation. Techniques allowing the system to be open will also be introduced, still maintaining the desired efficiency. Concerning the link to high level planning, the following approach is believed to be well suited for industrial operation.

Hand crafted agents

Considering the issue how an 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 [123, 77]. The system plans and executes assemblies in a special artificial block-type 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 [77] 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 tractablely 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 this quote is shared. However, the main interest here is not planning or activity orchestration, but rather the executive agent and the supporting software layers 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 control approaches such as planning.

4.3 The proposed Open Robot Control architecture

We are now ready to present the new Open Robot Control (ORC) architecture, which may be characterized by the following key properties:

- The layers are more dedicated to specific programming cases requiring a certain type of competence.
- The motion control has been split up for control engineering reasons.
- The intermediate level has a specific layer for application specific motion control, admitting more general and advanced control features than other systems do.
- The system programming level of other systems is mainly covered by the executive layer in ORC. That layer also serves as an holder of the robot programming language, which is fixed in other systems.
- There are both an on-line and an off-line programming layer. These are uniquely integrated on an equal level basis as explained in Chapter 3.

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.
Off−line programming	Programming without use of robot	Robot programmer with computer experience		
On−line 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 control	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		
Motor control	Control suitable for impl. in distributed hardware	Servo control engineer		

Figure 4.3 Users and properties of software layers/views in the ORC architecture.

- Task-level features are today usually implemented on top of off-line systems. Such features also defines a higher-level user interface or programming environment. Therefore, task-level programming has it's own software layer in ORC.

The layers and typical users are shown in Figure 4.3 whereas the hierarchy of layers is shown in Figure 4.4. Please relate the on-line and off-line layers in that figure with the robot system interfaces in Figure 3.6 on Page 30.

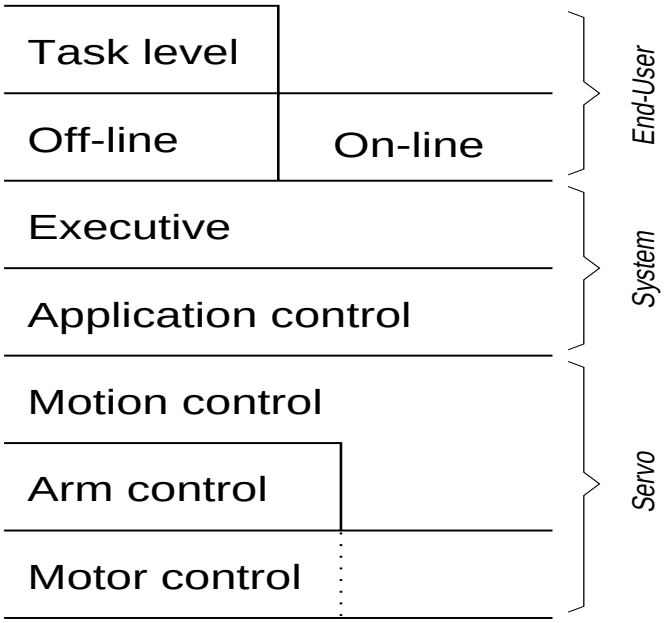

Figure 4.4 The Open Robot Control (ORC) architecture.

System level programming

From computer systems, we are used to system level programming and configuration. That includes writing new device drivers and installing them into the system [139]. Development of certain software libraries and tools can also be considered as system programming from a user point of view.

Robot programming shares the properties of computer programming. Additional aspects have to do with the control of the physical world. Let us define the meaning of system programming in three stages:

1. Implementation of (robot and process independent) libraries, and conventional implementation of drivers for IO devices and sensors.
2. Implementation of robot specific libraries, robot programming tools, and robot programming languages.
3. Implementation of application specific motion control as mentioned above.

We will regard the type **1** as just a matter of programming appearing on any level of the system. Just like the selection of (computer) programming language, it is not critical for the design of the system, and it is therefore not further treated.

From an engineering point of view, we note that tailoring the motion control (type **3**) requires control engineering competence, while the (type **2**) application support does not. It is therefore reasonable and appropriate to define two different layers for these two types of programming. The lower layer for application specific motion control is called the *application layer*, and the upper layer for tailoring of the programming interface is called the *executive* layer [150]. These two intermediate programming layers form the system level in Figure 4.4.

End-user programming

The aim that standard industrial robots should be possible to use as components for so called intelligent robot control implies that task-level programming principles should be put on top of explicit robot programming tools. Task-level programming facilities can of course be accessible directly from an on-line programming tool, but its software should rely on off-line programming. The reason is that off-line and task-level programming have the same need for abstract world modeling. On-line programming on the other hand, deserved a separate approach as described in Section 3.5.

Another conclusion from Chapters 2 and 3 was that robot programs needed to be represented in different ways in the on-line and off-line cases. Furthermore, factory floor and engineering department programming should be integrated on an equal level basis. This implies the need for transformation of robot programs according to Chapter 3, a need related to different programming views and manufacturing practices and, thus, to the proposed architecture.

Figure 4.5 shows these adjacent software views. Adding task-level programming on top of off-line programming results in the upper part of Figure 4.4. Note that even if task-level and off-line programming are based on the same tool (IGRIP [65] in this case), our architecture specifies that the task-level features should expose a uniform view to the user.

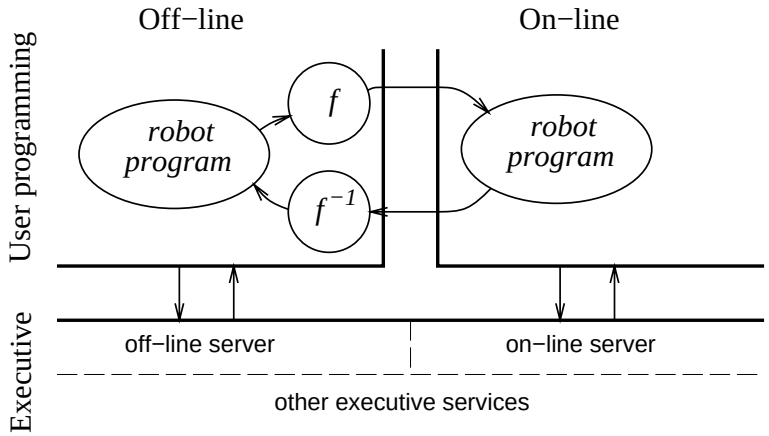

Figure 4.5 Integration of on-line and off-line programming required transformation of the robot programs according to Chapter 3. These transformations are denoted f and f^{-1} here.

Servo level programming

On the level of servo programming, a persistent feature is that software and hardware modules exhibit a very close relationship. Hardware and software design, at least for the motor control, are done as integrated activities to facilitate price/performance optimization. The software therefore naturally takes on a structure that reflects the hardware structure, whereas programming is performed by the implementors of the system. This implies that the user views of the architecture maps well onto an object-oriented design of the motion control system, which in turn reflects the structure of the physical objects. This is shown in Figure 4.6. Having realized the meaning of that figure, it is straight forward to define the servo control part of the architecture, that is, the three lower layers in Figure 4.4.

Discussion

As pointed out several times, application specific motion control is very important to provide flexibility and wider scope of application of industrial robots systems. Industrial manipulators are characterized by a strong interplay between user level commands, which often look robot independent, a good (but not perfect) world model, motion control services, and external signals from, say, a welding or grinding tool. This interplay is crucial to obtain flexibility and performance, but also to avoid the cost of otherwise necessary external sensors.

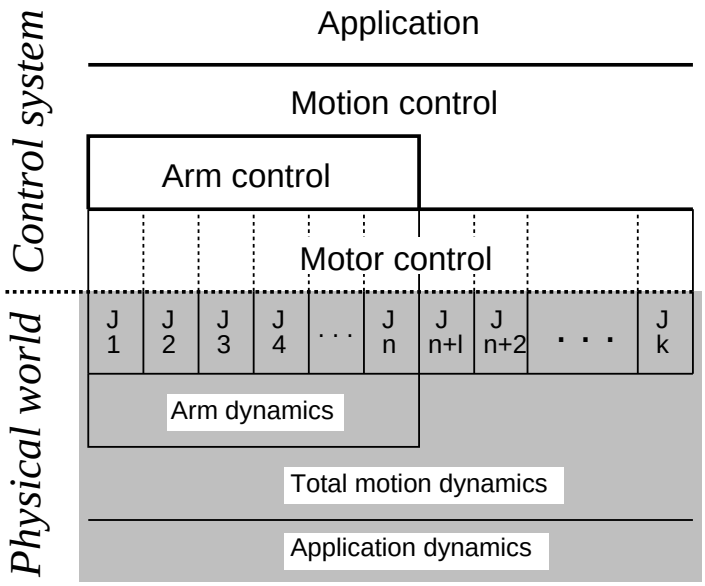

Figure 4.6 The structure of the control layers mirrors the physical environment having the corresponding dynamic properties. For a manipulator with n joints, the Motor control for joints J_1 to J_n (controlling the joint drives $\overset{J}{1}$ to $\overset{J}{n}$) is handled by the Arm control. Joints J_{n+1} to J_k control external (application or task specific) devices, which is handled directly from the Motion control. Thus, multivariable control laws are kept within the Arm control.

Robot operations, or executive agents, are preferably expressed on a simple end-user programming level of the system. However, as examples in Chapter 8 show, performance and efficiency demands sometimes make it necessary to modify the motion control system.

The motion control system provides a set of robot functions. Seen from the outside it consists of data and procedures. Programmers often regards it as a set of device drivers. One could 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. We can then consider the robot as an abstract data type [215], and it can even be nicely incorporated in an object-oriented framework [133]. There are proprietary and safety reasons for having a closed system, and they are also easier to implement than open systems are. This explains why several of the currently available robot control systems seem to have such a closed structure. This is a major reason

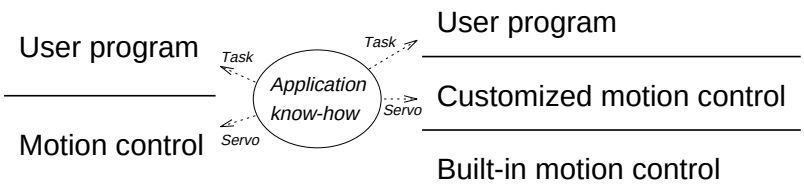

Figure 4.7 Application know-how gets expressed in robot programs. Presently (left) however, it also affects the built-in motion control. The proposed system (right) contains a software layer for application specific (customized) motion control.

for the difficulties to slightly modify the motion control, to include a new sensor etc. This leads to a situation when it is no longer possible for the robot manufacturer to do the required modifications. Thus, the flexibility or performance unnecessarily gets limited. It is therefore central for the purposes of this thesis, that application know-how can be added on top of the built-in motion control as shown in Figure 4.7. Note that application specific motion control only in simple cases can be achieved by changing available control parameters. More often, new control strategies need to be added. This puts special demands on the implementation (which will be returned to).

4.4 Software paradigms

Robot control systems are typically heterogeneous, both in terms of hardware and required real-time properties. Concerning the (internal) implementation of the system, it was mentioned above that object orientation could be useful, but other software paradigms may also be appropriate. A review of some possibly useful techniques for the implementation of the proposed architecture is presented in this section. A complete investigation would be a topic of its own, but some alternative paradigms have been studied. Note that the end-user programming (within the top three layers in Figure 4.3) is not the subject in this section. Instead, we are concerned with programming within, and implementation of, the other (lower) layers.

Functional programming There are several potential benefits (like formal verification and easy mapping to parallel hardware) of using functional programming [38, 8]. We may then use stream-oriented programming [8] to implement low-level data-processing/control blocks. Using functional programming for implementation of the stream operations

would limit 'side-effects' and increase reliability, which is sometimes hard when traditional programming methods are used.

A particularly interesting development in this direction, including both the functional real-time programming language H and special purpose hardware, has been done by Carlstedt Research [208]. That approach would, for instance, be well suited for implementation and programming of the computer nodes in our system (Figure 5.4, p. 74). For instance, our data streams are time stamped, which fits the IO model in the H language. On the other hand, as far as known to the author, no functional approach has yet fully proven its applicability to embedded control problems of realistic size and complexity.

Imperative languages For almost all programming in industry, imperative languages are used, i.e., it is explicitly stated in the program (statement by statement in order as written) the operations that should be performed by the computer. Assignments to variable/memory (and for instance in the C [104] language also pointer arithmetics and bit manipulations) is almost arbitrarily permitted, thereby giving the programmer (too?) much freedom. The opportunity to tune the software to hardware and for efficiency, without resorting on advanced compilers, are major reasons for the success of the C language, and later also C++ [198]. With the aim to implement industrially relevant robot control systems, such practical aspects have been considered also in this work (at some expense of programming style/elegance). As for any language or paradigm, we need appropriate abstractions to cope with software complexity [8].

Compared to traditional structured procedural programming, object orientation offers a powerful way of packaging data together with related functions manipulating that data. The object-oriented paradigm [126] is based on objects having internal attributes/states that can be manipulated by methods that are available via a type/class declaration. This means that object orientation resorts on side effects, at least in a functional programming sense. So even if the data being manipulated is well encapsulated within classes and objects, we can in a way consider object orientation as the opposite of functional programming. One could therefore suspect that object-oriented implementation of complex systems leads to unpredictable behaviors and complicated faults [217], particularly if we use active objects (i.e., objects with internal threads of execution [93]). The problem is, however, not the object oriented paradigm. Instead of rejecting object orientation, we need a proper separation between the description of data flows in one mode of operation, and the transitions between different such modes.

Declarative and formal methods Declarative languages, such as Prolog [57], have been designed to make the programmers task easier by admitting programs to specify *what* should be done (instead of *how* to do it as when imperative languages are used). The problem is that a declarative language well suited for one type of problem, like logical relations between objects in the Prolog case, in practice turns out to be less appropriate for other applications. An interesting approach for embedded systems is the Erlang language, which also shares some properties of functional languages [25]. The simplicity of Erlang, in combination with built-in support for concurrency (among other features), makes it very powerful for systems containing a large (and dynamically changing) number of concurrent processes, but we found it to less suitable for motion control systems.

Another declarative approach is the synchronous one [36, 76], which offers a uniform approach and some formal verification tools concerning the logical and temporal properties of the software. It is being applied to robot control [186]. Synchronous languages provide a programming interface for (correct) specification of software interconnections. However, each software component has to be implemented in a traditional way, for instance using the C language. Thus, this is not a complete solution, but it may in the future, for instance, be useful for safe programming within the application layer.

A pragmatic approach There are no general agreement on the criterias for a "good" paradigm. We may strive for code reuse, efficiency, maintainability, ensured correctness, or for programs that are easy to understand. The most appropriate language or paradigm may depend on system level, application, actual hardware, price/performance demands, etc. In fact, a blend of techniques is believed to be appropriate.

Use of techniques that still are in a research stage, like the synchronous and functional approaches, would put the implementation efforts of this project at risk. A well structured and widely accepted paradigm is object orientation (for analysis, design, and programming). At a hardware related level of the system, however, it was not thoroughly investigated at the time when implementation of the lowest part or the ORC architecture was carried out. Therefore, this was part of the research as will be described in the final part of Section 5.3 (p. 83). For other parts of the system, my attitude has been to use or combine whatever paradigm that suits the actual situation. The notion of function objects [198], for instance, is a way of combining functional (or data flow) programming with object-oriented programming. The ORC architecture does not prescribe any particular software paradigm.

4.5 Summary

Abstractions are used to cope with complexity, a property which applies to mathematics, to computer science, and also to robot control systems. The aim of this chapter was to find abstractions suitable for industrial robots used in manufacturing. For such robots, complexity issues deal with the programming/operation of the robot rather than the internal design. The reason is that robots in manufacturing operates in fairly well known environments, so there are less demands on knowledge representation, automatic world modeling, reasoning, planning, etc. Instead, the most important thing is to support manufacturing practices by having well defined interfaces for different programming situations. However, to permit more complex situations in the future, the control system should be structured in such a way that 'machine intelligence' can be added on top of the traditional functionality. This approach and the new architecture result from the following contributions:

- In Section 4.1, the fundamental concept of available architectures were reviewed, and the new user-view concept was introduced.
- Abstractions used for so called intelligent robots were studied in Section 4.2. Experiences from real experiments indicated that low level effects should be explicitly dealt with by the application expert programmer. Thus, it is a good solution to have a low-level (on-line) programming interface, and to add higher (planning) levels on top of that.
- Some typical programming situations were identified in Section 4.3, and the ORC architecture was proposed based on those situations. From bottom to top, the proposed architecture supports modular implementation of the built-in motion control (three layers), implementation of application specific motion control, tailoring of embedded system primitives, and end-user programming (three layers). Our desire to have appropriate programming interfaces without imposing any global database or hierarchical data-flow structure, which was observed as a problem with existing architectures, is believed to be achieved with the proposed ORC.

A major contribution of the thesis is the combination/integration of the overall architecture (as defined in this chapter), and technical solutions (presented in other chapters) making the ORC architecture possible and efficient. Requirements on These ideas can of course also be applied to other types of machines. To have a unified approach is particularly useful when designing complicated manufacturing systems.

5

Experimental Platform

Experimental verification is very important in applied research. It is to advantage for robotics research that robots are feasible to have in an ordinary laboratory, and many different control subjects can be put in a robotics context. Several different robot systems have therefore been built for experimental purposes within research and teaching. However, available systems were all found to belong to the following categories:

- The experimental system is built on top of an industrially available system [133]. This is a way of reducing the necessary engineering effort, but the limitations of the original system remain.
- The control computer hardware has in several cases been replaced by external computers allowing complete replacement of the control software [206, 197, 37]. That is, however, done for older types of robots, typically with interfaces to DC-motor drives and angular encoders.
- Complete robot systems have been designed and built, sometimes including special mechanical solutions [88], but more often just with simplified mechanics (to reduce cost and/or to make very specific experiments possible) like direct drive robots.

In applied robot control research, it is in most cases important to use robot manipulators that are relevant for industrial use. Simplified or special purpose research robots do not have realistic dynamic properties. The mechanical design done by major robot manufacturers has been worked out considering many application, quality, and maintenance demands. Thus, control of such robots are industrially more relevant. As an example, direct drive (rigid) robots are (today) appropriate only in special applications.

The experimental robot control system developed is unique in the sense that it is based on modern robots commonly used in the industry, still maintaining important safety functions of the original system, but allowing the researcher full access to control and programming functions.

A complete framework for reconfiguration of ABB industrial robots will first be presented. The framework supports reuse of hardware modules for recent (S3 and S4) versions of ABB system. Specific designs for the two systems in our laboratory will then be described, followed by some computing hardware considerations. Finally in this chapter, a remark about industrial aspects of experimental control is given.

5.1 Experimental control using ABB robots

Reconfiguration of robots has been limited to those from ABB Robotics. These robots are widely used in industry, and detailed information was available to the author. The main goal of the reconfiguration is to make possible experiments in control, where the experiments can be done on several levels ranging from basic servo experiments to overall programming of the robot functions.

Control system generations

The first ASEA/ABB robot was introduced on the market in 1974. The control system was called S1. Two types of mechanical robots were available, the Irb-6 and the Irb-60. The next system, called S2, was introduced in 1982, together with the spot welding robot Irb-90 and later also the assembly robot Irb-1000. These types of robots are not available with control systems later than S2.

The control system S3 [3] was released in 1986 together with the still quite modern Irb-2000. S3 systems from the years 1986 to 1991 having the measurement electronics placed on in the control cabinet are called S3a below. Later S3 systems having the measurement electronics on the robot are called S3b. Systems S3b and later all have digital-to-analog conversions taking place in the robot and serial communication with the control cabinet. That is to reduce cost and increase reliability.

During 1994, the S4 control system was introduced. S4 systems from 1995 and 1996 having the same type of IO interface and drive units as in S3b are called S4a. The next generation still under development is the S4b, which will have a quite different IO and drive system interface. That future system has already been studied, and a new version of the robot interfaces is planned. A wide range of mechanical robots are available with the S3 and S4 systems.

Hardware modules

In order to fill a general need for realistic but affordable (used) robots, any generation of ABB robot system should be possible to use. (S1 and

S2 systems are not quite relevant for servo control purposes, but may be useful for evaluation of other control solutions also requiring access to the low level interface.) It is therefore important to have modular hardware and software allowing reuse of modules for different systems and research purposes. Looking at the hardware, there is the hardware from the original ABB systems, and the added hardware modules which are called LTH modules in the sequel. The ABB hardware can be divided into modules according to the following (abbreviations are given within parenthesis for use in the table below):

Robot (Irb): The mechanical robot, including its motors and sensors, is regarded as one unit. Circuit boards mounted on or inside the robot may, however, be subject to replacement if they comprise a subsystem that is replaced. So far that applies to measurement electronics.

Measurement electronics (AME): All ABB robots are equipped with resolvers for measuring the angular motor positions (opposed to the angular encoders used on most other brands). The resolvers, and the tachometers in older systems, are not replaced whereas the electronics providing digital sensor values is.

Drive electronics (ADE): The power electronics for driving the motors forms one module which is never replaced.

Control cabinet (ACC): The cabinet including main power transformer, rectifier, cabling, etc. is here called control cabinet. It also contains systems supervision hardware. The drive electronics is (so far) located in the cabinet, but that is a separate module.

Computing hardware (ACH): The microprocessors, mounted on one or several boards, of the original system form the computing hardware. The reconfiguration means that it is removed, disconnected, or connected differently in the control cabinet.

It is the AME and ACH modules that are subject to replacement. The others are still used. The following LTH modules have been designed to replace or interface the ABB modules above:

Measurement electronics (LME): Completely new measurement electronics was developed because the original one was too involved with other system functions. Our solution is slightly more expensive than the cost optimized ABB one. On the other hand, we get a lower noise level.

System adapter (LSA): Additional interface needed for S3a systems to use the LME of S3b systems. This module has not been completed yet. It is therefore unclear what noise level there will be for position signals of S3a systems equipped with tachometers.

Control cable (LCC): A new version of the control cable between the robot and the control cabinet is required for S3b and S4a systems. That is because the original system uses asynchronous communication, while our communication is synchronous.

Embedded computer (LEC): We use VME-based board computer systems placed beside and connected to the ABB control cabinet. Other types of computer hardware could just as well be used; the interfaces fit ordinary off-the-shelf IO boards.

Drive system interface (LDS): This is to get access to drive current references internally in the ABB controller. Early systems (S1 and S2) have DC motors which means that the current reference can be used as a torque signal. Later systems (S3 and later) are equipped with AC motors, requiring torque/commutation control to be performed in software.

ABB bus interface (LAB): Modern control systems have less of fixed logic (relays etc.) for supervision and maneuvering purposes. Instead, a safety/system board takes care of this. That board is important to use (to protect the robot from damage) also for experimental control, but it is accessed via the internal ABB IO bus. A special LTH/ABB bus interface was therefore developed. This also gives that advantage that the ABB IO boards and customer IO connections can be accessed from our external controller.

The LME module replaces the AME one in the cabinet or on the robot depending on system generation. A serial communication interface has also been developed, but it is here viewed as a part of the LME module.

Hardware configurations

In S4 systems, which have an internal VME-bus, there is a possibility to make use of the ABB motion control software. Excluding that possibility, the possible hardware configurations are shown in Table 1.

Most effort has been put into the Irb-2000/S3b system that is available in our laboratory. That system, and the additional computer hardware we are using, will be described in more detail below. In addition to our modifications, an S3a system only requires the quite simple LSA module which is only connections to the measurement electronics that is placed in the control cabinet instead of on the robot. The S4a[2] system

2 The reconfiguration of S4a according to this chapter, including a second revision of our S3 interfaces, is currently being completed by Norberto Pires at Coimbra University in Portugal.

Sys.	Remove	Disconnect	Add	Comment
S1				See [143]
S2		AME, ACH	LME2, LDS2	See [47]
S3a	AME, ACH		LME3, LDS3, LSA, LAB	Not tested
S3b	AME, ACH		LME3, LDS3, LCC, LAB	Sec. 5.2
S4a	AME, ACH		LME3, LDS4, LCC, LAB	(Coimbra)
S4b	AME, ACH		LME3, LDS4b, LAB4	Planned

Table 1 Overview of reconfigurations of ABB robots. A number x appended to an added module name denotes a specific version of the hardware first developed for the ABB controller Sx.

interface is also quite similar to the S3b system. Thus, the next section almost covers newer systems too. The systems we have built are based on mechanical robots of type Irb-6 and Irb-2000 respectively, but it should be possible to do the same also for other robots using the same type of control system.

Reconfiguration of S1 systems

Due to its limited complexity and lack of supervision functions, S1 systems are quite simple to open up. One sensor interface is described in [120]. The resolvers are approximately the same as in the later S2 systems. It could therefore be a good idea to use the resolver interface from [47] to improve the quality of the sensor signals. That might, however require some additional wires between the resolvers and the Resolver-to-Digital Converter (RDC).

The analog speed control of the original system can still be used simply by using the drive unit input as a control signal. Direct access to the torque can be achieved by cutting the speed control loop and using the current reference as control signal as was done by the author in an earlier work [143].

Reconfiguring an Irb-6/S2 robot system

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. That is to get an industrially relevant system, and it is no limitation for our research purposes. The documentation and drawings of the modifications included in [47], together with manuals available from ABB Robotics, form a complete description on how to accomplish the reconfiguration.

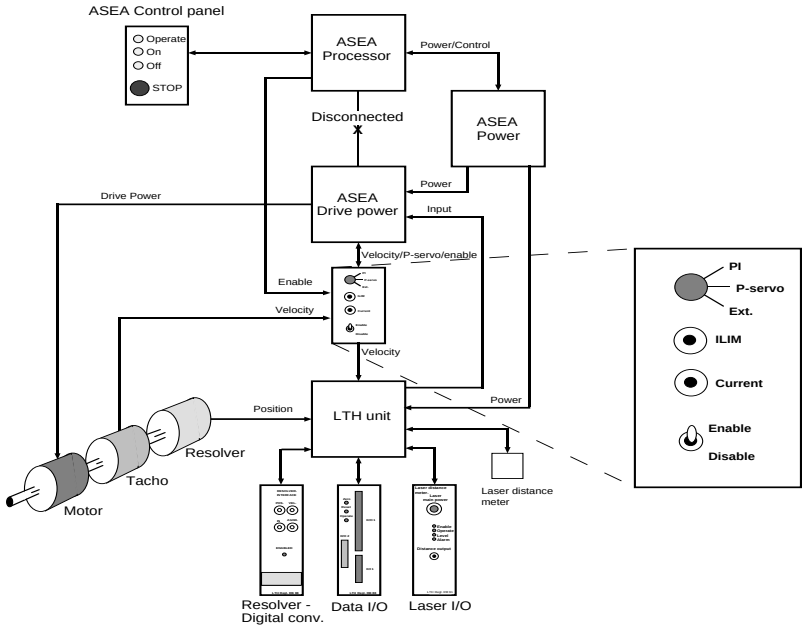

Figure 5.1 Overview of S2 modifications. There is one Resolver-to-Digital converter (RDC) module for each joint, but only one axis is shown for clarity. The analog control signal inputs are also located at the RDC modules, where analog signals for joint speed and position are available, mainly for test 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 5.1. The original control computer is retained 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 (LME2), 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 from hardware based on existing commercial chips for resolver-to-digital (R/D) conversion [14]. The solution principle can be seen as part of Figure 5.1, which also shows the solution principle for driving the motors (LDS2). 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 5.1.

5.2 An experimental Irb-2000 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, a more modern industrial robot with six degrees of freedom, like the Irb-2000, is a better testbed. The added computers are also included in this presentation to give the reader a better understanding of the complete system.

System overview

Both the Irb-6 and the Irb-2000 robot are controlled from VME-based embedded computers. Sun workstations are used for software development and control engineering, as well as for robot operator interaction. Figure 5.2 shows the Irb-2000 part of the laboratory (Cameras and a video interface mounted in one of the workstations are not shown for clarity).

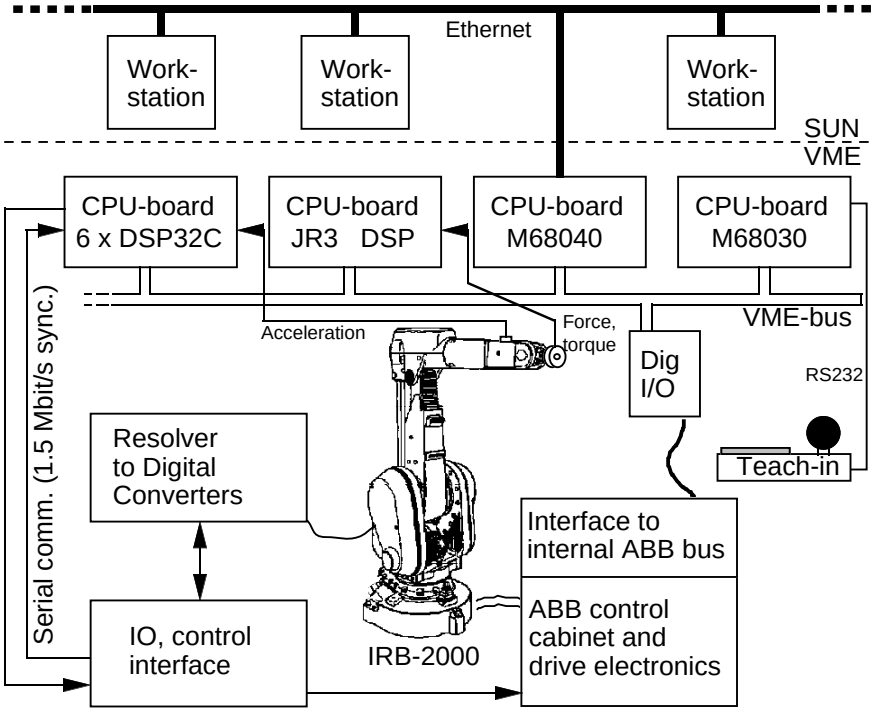

Figure 5.2 The experimental Irb-2000 system.

Signals from the internal sensors of the robot to the VME system go via the sensor interface to the DSP board connected to the VME bus.

The master computer in the VME computer is based on a M68040 microprocessor. Supervision and safety functions are implemented on a M68030 board, well separated from the rest of the system to prevent damage of the robot. Digital Signal Processors (DSP) are used for low-level control and filtering of sensor signals. Sensors requiring very high data bandwidths are connected directly to the DSP boards. An additionally DSP board belongs to the force-torque sensor [101]. A six DOF joystick [94] can be connected to a serial port of the M68030 supervision computer for data transfer to memory accessible from the VME bus.

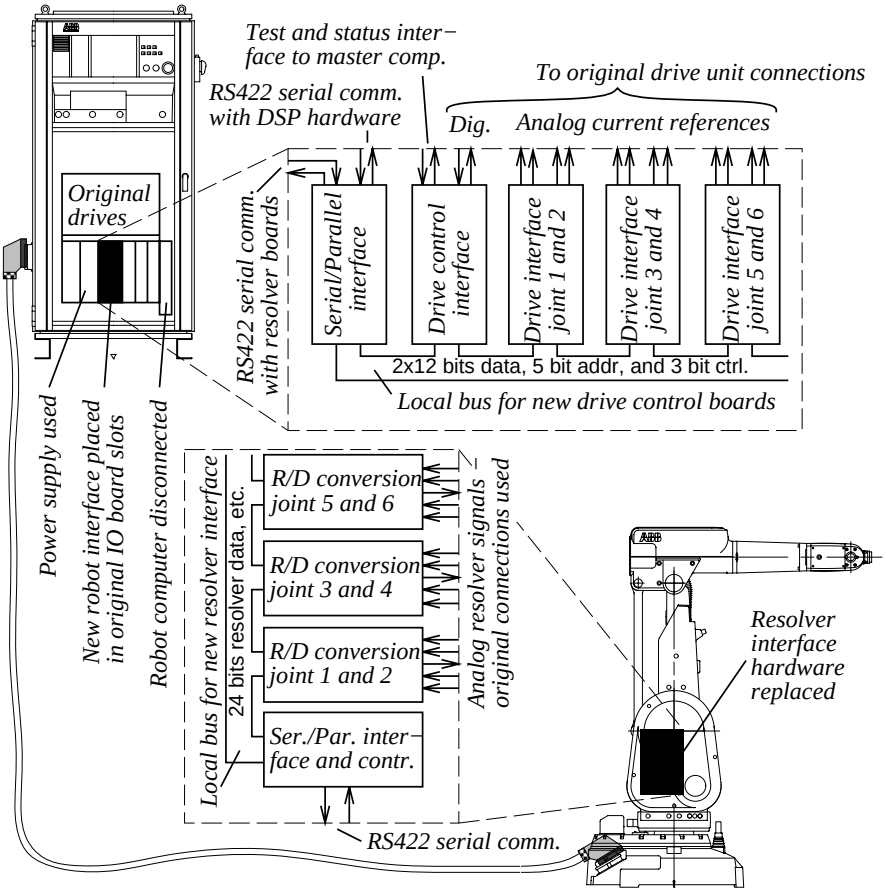

Figure 5.3 Placement of hardware added to the Irb-2000 system.

Basic design

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 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 (preferably 2 kHz). Otherwise the torque control will not be good enough, particularly for high speeds. This means that the requirements on computing power are quite severe. Our solution is to use Digital Signal Processors (DSP) as described below.

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 8 revolution counting bits in hardware. Both hardware and software are prepared for a resolution of 16 bits per motor turn. Thus, joint angles are provided as absolute 24 bit values. Using commercially available R/D converters [14] 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 hardly possible with optical encoders or with other types of resolver measurement principles (a higher sampling frequency and the roll-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 sampling period will then, however, be quite crucial. The next generation of the sensor interface will therefore also provide the speed value from the RDC chip. The R/D conversion hardware is located on the robot which reduces the required length of the wires for analog signals to a minimum.

The current references for each motor are output from the computer 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 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.

Communication with sensors and actuators

Reduced cost for cables, reduced noise level, and increased reliability are advantages obtained by placing sensors, actuators, and necessary electronics close to the motors on the mechanical robot. That was experienced more than 10 years ago using optical connections [153], and this approach is also used in the S3 control system, although not optical. It is also be-

lieved to permit more modular (and thereby more flexible and reusable) machine designs within mechatronics in general [207]. Such a modular approach, however, has not had much industrial impact yet.

The serial communication line of the original ABB S3 system was for our purposes too dedicated and involved with the rest of the hardware. It was therefore realized that new communication hardware and software needed to be developed. Such an interface between sensors, actuators, and control modules was designed and built with two purposes in mind:

1. To connect the Irb-2000 robot and its reconfigured control system in a simple and efficient way allowing sampling rates of 4 kHz for the motion control.
2. To serve as a testbed for research within decentralized control and real-time systems.

This has so far been successful. An early description of the design was given in [144]. The system has been used in a case study on distributed real-time control [207], and it has been referred to [220] as a platform for verification of timing problems in decentralized real-time control [219].

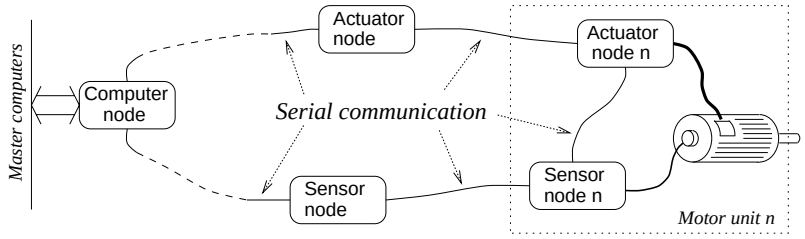

Figure 5.4 The used principle for connecting the joint servos of the robot to the control system. A sensor (input) and actuator (output) node sharing the same address can be viewed as a *motor node* from the control side.

Using physically distributed, i.e., decentralized, hardware means that the control system (including its sensors and actuators) contains a number of so called nodes. A node may constitute a sensor, an actuator, or a processor. A node can, of course, physically include several such elements, but they should conceptually be separated. Several types of so called field buses [156] are today used in process control. Specific designs are, however, often done for specific machines. Our type of bus, which we named the SA-bus (Sensor Actuator bus), is one such example. This bus has three types of nodes as shown in Figure 5.4. More specifically, some properties/features of the modified system are:

- The robot system communicates via synchronous serial communication. A bit transfer rate of 1.5 Mbit/s makes it possible to transfer 32

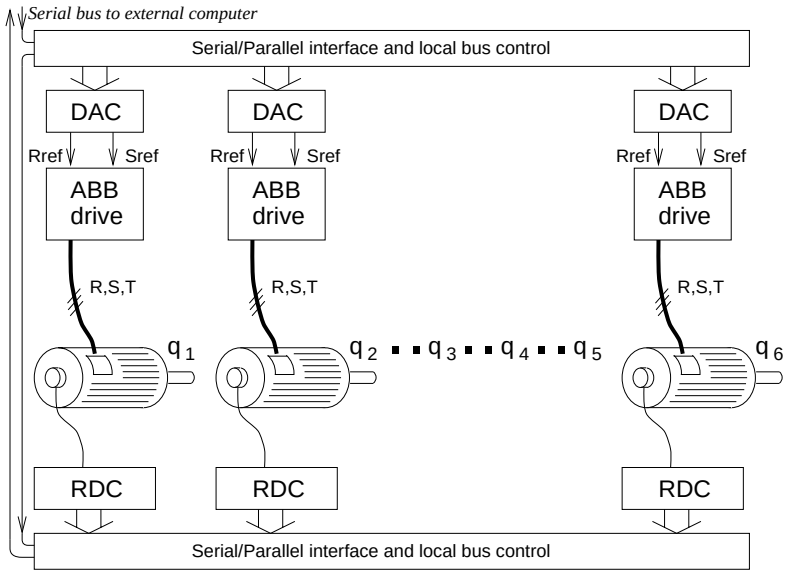

Figure 5.5 Servo hardware in the modified S3 system. Please compare with Figure 5.3 to see the physical placement.

bits of data with a sampling frequency of 8 kHz for each joint, if six joints are used. The communication protocol fits directly to the serial ports of most DSP types.

- 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.
- Our SA-bus protocol specifies that data are 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 the return communication line.
- Computer nodes do not have addresses themselves. They are just processing elements connected in a data-flow oriented [8] style. The motor node number is attached as an address to each data message. Time stamping is implicitly achieved by using synchronous communication in combination with explicit time delay messages after temporary faults.

- 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 planned version for S4b systems will (according to the new ABB design) use the serial bus instead.

Even if the nodes are conceptually separated seen from the computer side, they were actually built in clusters as already shown in Figure 5.3. That was to simplify the hardware considering the design of the original system. This means that the physical LME and LDS modules mentioned on Page 69 relate to the six motor units as shown in Figure 5.5. Note, however, that the motor control conceptually agrees with the *motor control* layer of the ORC architecture on Page 57.

System supervision

Development of software for the supervisory computer board (M68030 in Figure 5.2) was initially done for two reasons: 1) An overheated motor should cause the robot to stop for sure. 2) Pushing the RUN button on the cabinet should turn on the main power to the drives. The mechanical design of the robot should normally permit end-stops to be hit with maximum torque, at least a few times. Supervision speed and position limits etc. was therefore postponed to some later stage of the project. Both needs, 1 and 2, required access to the ABB safety board (called system board in later systems) via an internal ABB bus. An Interface to internal ABB bus was designed and built as shown in Figure 5.2. That board, together with an available digital I/O board and flat cables, formed an VME/ABB bus connection.

An interface to the ABB supervisory control of the drive units was also included. That interface is used to reset the drives at startup and to get information about errors (like unacceptable current references, which for example occurs when control gains are set too high or the excitation during system identification is too persistent). This means that the interface board physically replaces the ABB robot computer.

Supervision functions may not be disabled by any experimental control software. For example, increasing the order of the control algorithm which perhaps run with maximum priority, may not cause a motor burn-up when motions are too demanding. That could be the case if the control activity, instead of the supervision functions, occupies the CPU. This is avoided by using a separate CPU-board for the supervision. Furthermore, the software process with the lowest priority must trigger a mono flip-flop at least each 20 ms. Otherwise, the drive power is shut off. That detects if the supervision system is stopped.

Supervision of control states like joint positions and velocities (as done in the original system) is also necessary. Unfortunately, these functions were not implemented to start with because we were short of time and eager to do control experiments. These supervision functions were implemented first after an accident during the experiments [129]. Servo control of a single joint was done with a sampling frequency of 24 kHz. The torque control then works for much faster commutation (motor speed) than possible in the original system. The robot was therefore able to move past an extra mechanical stop, past the standard mechanical stop, and continuing outside the working range damaging cables inside the robot. The final extensions also included supervision of the cyclic execution of the control tasks. After that, the system works very well.

The software modules shown in Figure 5.6 form three layers. At the lowest level, the physical connections to signals in the control cabinet are encapsulated. The mid levels contain the actual supervision, which reflect internal details of the ABB hardware. Proper error recovery further increased the complexity of this part. The top layers contain speed, position, torque, sampling, and jam supervision. The latter means detection of high torque without motion. Additional functions can easily be added.

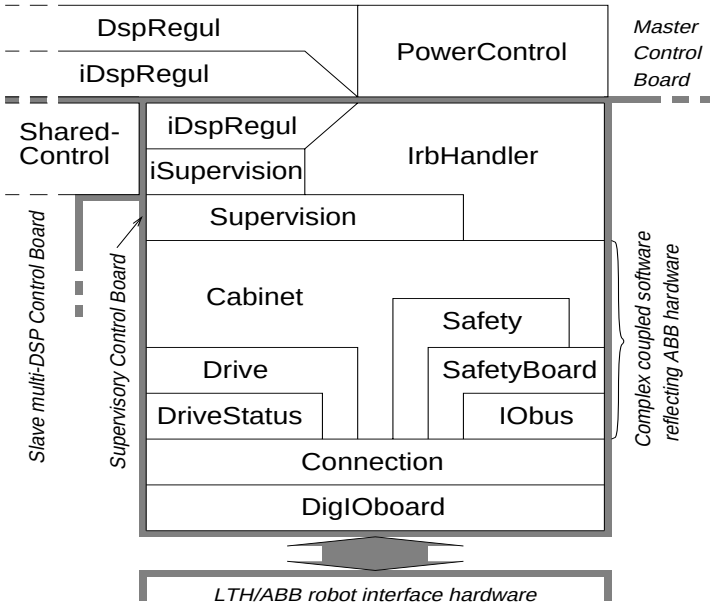

Figure 5.6 Supervision software modules. These contain totally 15 software processes performing the supervision and operation of the system.

Opposed to the layers of the ORC architecture (Page 57), these layers reflect the implementation (just like the OSI-layers does for a network connection). In other words, some functionality provided might be viewed as a system service available in a user view, but the layered implementation of the supervision system is only exposed on a system programming level.

5.3 Low level control – Signal processing

Robot motion control can be divided into servo control of individual motors and control of the complete arm. The motor control including drive electronics and measurement system can be implemented close to the motor, or distributed elsewhere in the system as described in the previous section. The arm control is typically a multivariable controller that is implemented on top of the motor control. Required sampling periods, i.e., cyclic execution rates, for a typical industrial manipulator are for example (according to the author's experience)

- 20 Hz (for trajectory generation,
- 100 Hz for position control using an inner velocity control,
- 500-2000 Hz for velocity control (depending on friction etc.),
- 2000 Hz for torque control using inner phase-current control,

where the first two items applies to the multivariable part, and the latter two applies to the motor control. Considering that there are six joints to control, and that a timely response on each sampling instant is important for the performance of the robot, we see that the real-time implementation for the low-level motor control has to be carefully designed.

DSP hardware

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, which are the standard computations in signal processing and control algorithms. This was the most promising type of processor for control implementation at the time when hardware was selected for this project. The processor DSP32C from AT&T [31] was selected mainly for the reasons of simple low level programming (C-like assembly language and data directed programming [114]), compatible performance, and because a VME board and required software tools was provided by a single source (AT&T).

It would be natural to use DSPs dedicated to each motor/joint control. But the connections with the robot, and the desire to efficiently use

commercially available hardware, resulted in use of the same processing elements for all joints. The VME board with six floating point DSPs of type DSP32C is shown in more detail in Figure 5.7. Four of these DSPs are involved, in a serial fashion, with the motor control. The other two are available for additional sensors and actuators that need to be handled by the motor control (see Figure 6.2 and Figure 6.2 on Page 90).

Real-time control

Academic research often tries to solve the computing power requirements for robot control by massive parallelism [35]. Special compilers and run-time systems should then take care of the multi-processing. That is the aim also in many other fields, but this type of parallel processing has not had much impact on systems that are actually being used. Successful industrial robot control systems, for instance, use several but few processors. These are typically of different types and connected in a cost and execution efficient way.

In our system, microprocessors of type Motorola 680x0 are used for high level control, while the DSPs are used for the low level control and filtering. A real-time kernel developed within the department is used for the Motorola 680x0 boards [17]. No specific concurrency model is prescribed,

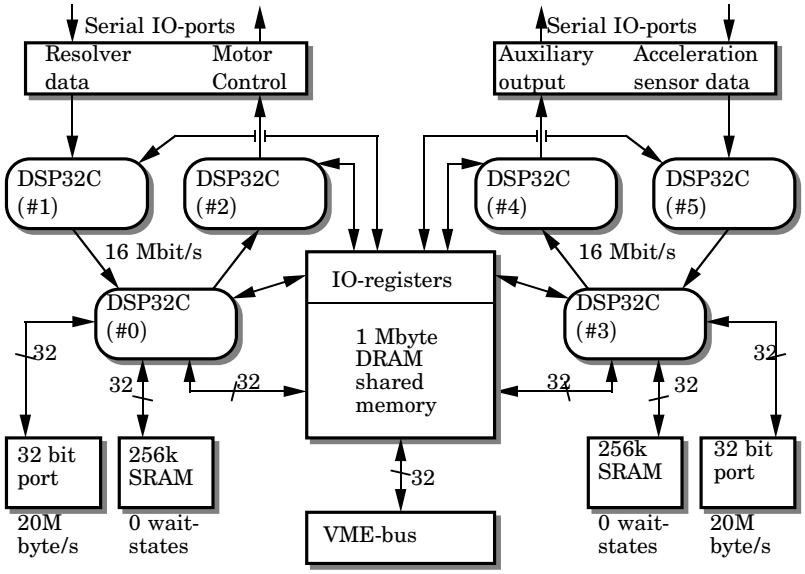

Figure 5.7 The VME board with six DSPs.

and real-time primitives are built on top of coroutine and interrupt routine facilities. Sampling rates currently supported range up to 1 kHz for the M680x0 boards. Programming languages currently supported are Modula-2, C, and C++. Even if this part of the real-time environment is rather standard, it is public domain in terms of source code, etc. That has allowed implementation of new real-time software techniques [122] in connection with this project [145, 121].

Looking at the special control hardware like the DSPs, it is quite clear that more dedicated real-time solutions are needed. We know from the description above that we have to maintain a data flow of control signals with a rate of $6{\cdot}8 = 48$ kHz data packets per second. For each DSP32C this implies that there are 260 CPU cycles available to process each sample. In fact the main DSPs (#0 and #3 in Figure 5.7) are presently only run at a 24 kHz rate, which gives us 520 CPU cycles per sample. (When fully utilized, each cycle consists of one floating point multiplication, one floating point addition, two pointer modifications, and memory accesses.) It is natural to connect the control algorithm to interrupts, and to let the 'background' process handle online parameter updates, supervision, and logging of control signals.

Context switching The real-time demands for DSPs are contradictory, particularly for floating point DSPs because of their highly pipelined architecture. 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, are 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 make a context switch including the floating point registers. The only shadow register accessible is that for the program counter, so the only possibility is to modify the code that will be executed after returning from the interrupt, in such a way that the pipeline is emptied before the context switch. There are two problems with such an approach; 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.

A review of other hardware solutions for context switching has also been done. On one hand, it is desirable to have a quick interrupt facility to serve short interrupts without breaking the pipeline of the DSP. This means that the floating point context (registers and pipeline status) needs

to be saved in shadow registers. On the other hand, preemptive context switches are typically required in demanding control applications. That would then require that shadow registers could be saved in memory. Examining the four major brands of floating point DSPs, one sees that none of them have an interrupt architecture very suitable for control applications [148].

Motorola (DSP96002) has adopted a concept of fast interrupts that do not perform any stack operations, and long interrupts that do. A long interrupt can be formed from a fast interrupt. However, the fast interrupt is too limited to be used for operations such as a clock interrupt routine. Analog Devices (ADSP-21020) has a special status stack, but the floating point pipeline is aborted when servicing an interrupt (as with Motorola's long interrupt). Texas Instruments (TMS320C30) also aborts the FP pipeline, and only uses the ordinary stack for saving the context. AT&T (DSP32C) has an interesting quick interrupt facility, in which the entire floating point context is saved in one instruction cycle. However, that context can not be accessed; it can only be restored as mentioned above. That means that the only way (that the author has found) to do a preemptive context switch is to replace the next instruction of the interrupted program with a call to a context switch subroutine. That is, however, not a good solution since it would then not be allowed to interrupt code that has been optimized utilizing latency effects.

The DSP32C works well in our application because the background process is only preempted by interrupt routines. In more general control applications, a combination of features from the different vendors would form the most powerful hardware. Considering the software part, it would be desirable to have means for limiting the size of the context for frequently executed low-level control tasks. A DSP real-time kernel with such a feature is commercially available [213], but the fast processes must be coded in assembly language since there is no support from today's high level programming languages.

Concurrency We now have a process interface via the SA-bus, a number of processing elements, and an interrupt driven execution model. Interrupts are caused by availability of sampled data on input ports. The control actions will then be synchronized with the sampling. Interrupts either trigger control actions, while others are needed just to handle data transfers to/from other processors.

The real-time schedule of control actions is defined statically by tables containing joint numbers and execution control data. Schedules are loaded from the VME bus via IO-ports (Figure 5.7) and DMA into the on-chip memory of the DSP. The schedule can be changed on line if carefully done.

Static scheduling is a research topic of its own [194, 195, 54]. Here, it has so far been done by hand.

To conclude the presentation of the hardware, we will now follow the motor control loop around the hardware nodes. We start in DSP #2 (see Figure 5.7) with current references available for the next scheduled joint.

1. Upon interrupt for output buffer (current references to the SA bus) empty, DSP #2 looks up the next joint number in the static schedule. Most recent output data (phase-current references in this case) are looked up and an output message is composed and sent.
2. The actuator node for the addressed joint catches the data, and the sensor node later replaces the data with the sensor value. These actions are done without delaying the message.
3. A new sensor value causes an interrupt on DSP #1. The interrupt routine unpacks the message data (extracting angle data and joint number), performs some error checking, and stores a valid sample in a buffer. The background process for this DSP reads the buffer and performs digital filtering from 8 kHz to a possible lower sampling rate used in the control. The filtered value is sent to DSP #0 when the serial output buffer is empty.
4. The new sample interrupts DSP #0, which then gets the joint number from the input message, looks up sensor and actuator (software) objects, and converts the data according to those objects. For AC motor control this means that the data stream is split up into two paths:
   - The sensor value is passed to the actuator object which computes the commutation angle of the rotor of the motor. An actuator output message is then prepared (in the actuator object) to permit rapid response when a new torque reference gets available.
   - The sensor value is also converted to floating point and SI units. It is then put in a 'shared array' in the DRAM memory. An interrupt in then generated to DSP #3.

   The background process for DSP #0 performs parameter updating and data logging for master computers (M68040) on the VME bus.
5. External sensors for motor control, in our case an accelerometer is available, are read by DSP #4 and sent to DSP #3. The serial input port of DSP #3 is configured to transfer data into the internal memory by DMA. The overhead for the program is therefore only due to cycle stealing, which is less than $0.1\mu s$ per sample.
6. The background process for DSP #3 does the same as in DSP #0, but for the motion control instead of for the actuator control. Sensor

values are available in the DRAM for the motion regulator which is triggered by the interrupt from DSP #0. The computed control signal is put into the DRAM. In addition, a state vector is exported via the DRAM to the M68040 master control board. The master also supplies references and slow control signal corrections because integral action and windup protection are done by the master. A DSP implementation of this part would waste computing power since logics and mode switches do not fit well into the highly pipelined architecture.

7. The control signal (torque reference) is then passed to the actuator object that one or a few samples earlier (according to the static schedule) computed the commutation angle. That object limits the control signal and composes a torque request message that is put in the serial output buffer. This was done at the end of the interrupt service for the sensor data.
8. The torque request message is transferred via the serial connection and via DMA into a buffer in the on-chip memory of DSP #2. The background process then unpacks the joint number, the commutation angle, an the requested torque. A function that computes the current references is then called. (This function is written in assembly language, all others are written in C++ as described below.) The references are put in an array for the output interrupt routine. Thus, we are back to item 1, and the loop is closed.

This makes more than 10 parallel processes totally. Absence of debugging tools, and presence of both hardware and compiler faults, made the implementation a very time consuming task. An industrial implementation would involve hardware and software development using professional tools. So why implement the control this way?

Even if a commercially available board is used, it provides several hardware features that are configured and controlled from the software. That, in a way, corresponds to the custom made circuits that interface the CPUs in modern embedded controllers. With a background in assembly programming of such systems, it was as part of this research to investigate how high level software techniques could simplify programming of this type of embedded systems.

Low-level object-oriented programming Data processing blocks (or filters) are naturally represented by software objects. Control often also includes sequences, mode changes, etc., but such control is better put on top of pure functional blocks (as most often done in process control systems). We can therefore reason about the control in terms of block diagrams.

In an object-oriented framework, we may have methods (member functions in C++ notation) that implements the signal processing of the (control) block. Such control objects are sometimes called function objects [198]. Additionally, there may be other member functions for mode changes etc. For the implementation, we want to use available compilers and a language that allows efficient manipulation of the hardware.

In practice, the only language available for (almost) all new types of hardware, including our DSPs, is the C language. It is of course desirable to use the same programming language for DSP programming as for programming of the rest of the system, where C++ [198] was available. The C++ compiler *Cfront* [30] used on the Sun workstations was therefore 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++, making it possible to connect C++ functions to hardware interrupts for example.

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 [53]. Therefore, the C++ compiler was extended to permit easy and structured interfacing of inline assembly code. This feature has also been useful for implementation of new real-time primitives.

The developed real-time primitives, and several features for hardware utilization, have been encapsulated in C++ classes providing a high-level interface. Hidden for the ordinary system programmer, the member functions accessing the hardware features were implemented as assembly macros. Call of such a member function results in inlining of the code on an assembly level, allowing the inlining to depend on actual allocation of the data. For example, if data is already loaded into a register due to the assembly code from the C compiler, that register can be used directly. This feature in C++ relies on extensions of the underlying AT&T C compiler [32]. Using this concept, superior efficiency was achieved in combination with object-oriented programming. Part of the software handling object interactions was, however, implemented in a procedural (C) style (but still written in C++) for efficiency reasons also observed by others [211].

Another attractive property of object-oriented programming is that hardware dependencies could be nicely encapsulated in classes. More specifically, we have multiple CPUs that differs in type, representation of floating point numbers, address spaces, sizes of pointers, byte ordering, and they have different real-time properties. A drawback of C++ compared with most other languages is that more programming skill is required since it is a very rich language. However, many of the features – operator overloading, conversions, class dependent memory allocation etc. – were found to be very useful.

The employment of a high level software technique for very low-level programming is a contribution [147, 146] related to this thesis, but a further description is left out here for brevity.[3]

5.4 Concluding remark

Robot systems are normally used as intended by the robot system designer. That gives reliability, and permits full support for repair and maintenance. If customers would start to rebuild some robot systems for special applications according to our drawings, and then use the system for full production, could that cause problems? It is quite understandable if the robot manufacturer believes that the answer is yes, being worried about reputation and customer support. We claim, however, that the answer is no.

Our experimental system can very well be used in real industrial applications, but only for experimental purposes. The reasons are:

- The warranty for new robots would hardly apply.
- The original system provides a lot of special features, developed according to specific requirements from the customers. One example is automatic restart of robot programs (and motions) after a power failure. Many man-years are put into development of such features. They will not be supported for the experimental system.
- Supervision and self-test functions are much more extensive in the original system. Even if the system can be changed back to its original shape (to run the ABB self-test software) in less than an hour, that is too long for most production systems.
- There are less support for repair and maintenance, as mentioned above.

A system like the one described in this chapter will therefore not be useful for full-scale every-day production. It can, on the other hand, be used in real applications for rapid prototyping and evaluation purposes. That should be of interest also to the robot manufacturer. Thus, an R&D version of a robot control system would fill a need not only in academic research within control and programming, but also in industrial feasibility studies for possible new robot applications.

3 More details and examples are given in [147]. That paper was judged as one of "the best technical presentations" (Page 2 in [146]), which resulted in publication of a condensed version[146], reaching a large engineering community.

6

Motion Control Structure

The primary purpose of the motion control system is to read signals from internal sensors of the robot system, to compute control signals in such a way that position and force control as requested from higher levels of the system are achieved, and to output the control signals to the hardware. Superior performance and robustness of the control is important for productivity and utilization of the equipment, and possibly also to allow less expensive mechanical design in the future. The ongoing research within control theory to improve on these points is therefore very important.

A perhaps even more important problem today, given all the algorithms that have been proposed so far, is to facilitate use and evaluation of available control solutions on realistic robot systems. The hardware platform presented in Chapter 5 constitutes a system well suited for this purpose. Implementation of the control will be treated in this chapter. Still focusing on the robot programming subject of this thesis, now on a system/control level, the topics are software engineering and control engineering, and how that can be supported by a proper design of the system. More specifically, we want to find answers to the following questions:

- What is the rationale of the control part of the ORC architecture concerning management of control complexity, hardware structure, and system development?
- Given the basic design of the architecture, what functions should the layer interfaces contain, and what functionality is needed internally in the software layers?
- Our approach to split up the built-in motion control in the layers, will that make it hard to apply modern control techniques?
- Advanced planning/optimization of motions/trajectories often needs to be performed in advance (in an off-line manner). On the other hand, motions depending on sensor values must be computed in real time. Therefore, such motions cannot be optimized in systems today.

Could the management of motion commands and sensors be designed in such a way that the distinction today between plan-time and run-time can be relaxed?

- Is a fixed definition of control interfaces (in terms of libraries or control blocks as in process control systems) appropriate for our purposes (flexibility and efficiency), or do we need techniques to "get inside" control modules?
- Each layer in the proposed ORC architecture defines a view for programming/engineering. Considering the low-level motion control, how can industrially relevant hardware (like the experimental platform in Chapter 5) and engineering tools be utilized to obtain both real-time computing efficiency and flexible control engineering?

These issues will now be treated in one section each (in order as mentioned). The problems are approached in two stages. The first (descriptive) stage contains a basic design (Section 6.1) in accordance with the ORC architecture, and a specification (Section 6.2) of the functionality required in each software layer. This first part presents relevant requirements on the robot system. Section 6.3 then states two open issues associated with the specified structure. These issues/problems are then treated in one section each. Finally, an implementation of the motor control layer is presented.

6.1 Basic design

A coarse modularization of the application independent part of the motion control system was made in Section 4.3. That was done as part of the definition of the complete architecture. The proposed ORC architecture is based on user views. On the servo level of the system, a need to support three types of users was identified as shown in Figure 4.3 (p. 56). Thus, we have three different software layers named *Motion control*, *Arm control* and *Motor control*, each defined for a certain type of control engineering. Some further arguments for this design now follow.

Considering a specific implementation, the motion control for all joints of a robot can of course be viewed as a single multi-variable controller. The standpoint taken here is that this is not a suitable formulation of the problem, at least not for industrial development. We claim that the choice of control structure and engineering tools is not entirely free due to demands on modularity. The proposed software layer is an attempt to impose such a modular structure. The reasons are the following:

Control complexity: A single non-linear multi-variable controller gets unsuitably complex when dealing with all control constraints and

physical effects. Examples:

- There are constraints on speed, torque, and torque derivative to protect the mechanical system.
- Run-time supervision and start-up diagnostics (to detect run-away motion, malfunctioning sensors, etc.) are easier to get reliable when implemented together with a control structure that in a simple way reflects the hardware and the states of the physical system.
- Many control difficulties due to non-linear and/or unmodeled dynamics are because of the physical properties of individual joints. Typical examples are torque ripple, motor friction and gear-box play. Dealing with locally appearing phenomena by local (inner) control loops is a fruitful approach within control design.

One could claim that this is just a matter of judicious choice of control engineering tools. But according to industrial experience of control development, demands on reliable system operation/supervision indicate the need for architectural support.

Hardware structure: The architecture should support a control structure that promotes efficient hardware solutions. This motivates introduction of the Motor control layer.

System development: The structure of the control system should support modular development in projects focusing on well defined problems like multivariable arm control. In other words, user views (as defined in Chapter 4) are appropriate also for structuring the (low-level) motion control system. The scenario expressed in Figure 4.3 includes:

- Control of individual motors, perhaps using less expensive hardware.
- Control of the manipulator/robot arm, perhaps to achieve improved decoupling of joint motions.
- Incorporation of external motions, i.e., control of other work-cell motions like servo-controlled welding fixtures.

Note that even if external motions typically were application specific, indicating that such control should be included in the application control layer, support for integrated external motions affects the design of the embedded system to a larger extent. Therefore, this is part of the motion control layer, whereas configuration and control parameters are selected from the application layer. Control of integrated external joints is today dealt with by the robot manufacturer.

The object oriented paradigm [96, 126, 44] appears to be well suited for design and implementation of the control systems. Classes will then en-

capsulate 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.

6.2 Desired properties of the software layers

As an outline of a functional specification, a coarse description of the involved software layers and interfaces are described in this section.

The motor control layer

The design of the multi-variable 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, for instance, computed torque is not used in commercial robots today. The design should instead be divided into joint-wise controllers (implemented in the motor control layer) and a multi-variable part (implemented in the arm control layer). Therefore, the purpose of the *motor control* layer is to support controller implementation in two ways:

- A complex multi-variable arm controller can be better modularized if axis-specific features are put in separate modules, as mentioned in the previous section.
- The motor 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 motor control will now be described. The control blocks as such are well known [29]. Interface aspects due to hardware distribution or multiprocessing are ignored for simplicity.

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 motor 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, possibly with decreased performance.

A general control structure of an axis controller in the motor control layer is shown in Figure 6.1. Figure 6.2 exemplifies the structure

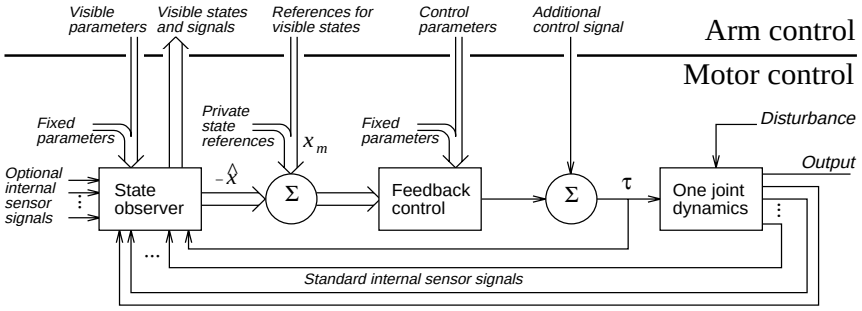

Figure 6.1 General structure of one motor controller.

with a simple controller of the type commonly used today. Torque limitation is not shown 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.

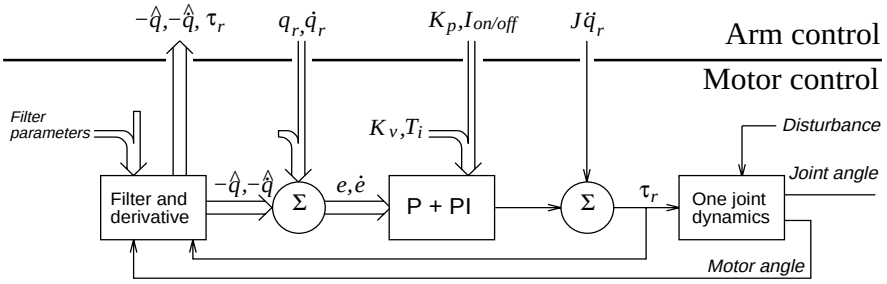

Figure 6.2 Motor control of a type commonly used today with a P-type position control and a PI-type velocity control. The notation of signals are standard for robot control, or please refer to [59].

Both standard sensor signals and optional ones are connected to the state observer, which also includes filters etc. An optional sensor for motor control may be an acceleration [73] or torque sensor [214] used for high performance joint control. 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 importance 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 motor 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 motor control layer.

The optional internal sensors, see Figure 6.1, 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.

The application layer – motion control layer interface

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 acquisition of motion properties.

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

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

The acquisition 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, functionality, and flexibility of the robot and its programming environment. The need for such knowledge has also been noticed both from the field of off-line programming, and from development of high performance motions and application packages.

In the off-line programming case, the reason is that robot manufacturers do not want to reveal their control solutions and software models for the robot due to proprietary reasons and software maintenance reasons (improving the control system should not impose updating of software

tools distributed all over the world). The situation will be the same with future open robot control systems admitting external (to the robot manufacturer) advanced users to implement control strategies. A key idea for the implementation of these services is therefore that the same executable code should be used as for the proprietary motion control of the robot. This is preferably handled in the following two different ways, depending on if it is for simulation purposes or not:

1. 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. In C++ notation, 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.
2. Acquisition of dynamic and kinematic properties, utilizing models internal to the motion controller, for motion planning and optimization purposes is preferably supported via special functions. In this way, the same interface for both simulated and real robot motions is achieved.

The idea to let the robot manufacturer supply compiled code for their (secret) dynamic models and controllers is quite natural. Exactly for this purpose, the Realistic Robot Simulation (RRS) specification [169] was developed as a joint effort between designers of robot control systems and of off-line programming systems. (The RRS interface can be considered as an interface to the "Simulation engine" in Figure 3.12, p. 42.) The claim here is that the RRS approach can be questioned along the following lines:

- The support for application specific motion primitives and so called sensor-based motions is too limited. It is possible to subscribe to events that are generated when the value of time or position passes a certain threshold. This should be extended according to the ideas on open control as described in Section 6.6.
- A black-box model of the robot arm and its control is difficult to combine with simulation of equipment mounted on the robot arm. Instead, an object oriented modeling and simulation technique should be used [21, 158] for the mechanical part of the system.
- Properties of control utilizing dedicated hardware, like finite word-length computations coded in assembly language, get hard or inefficient to resemble in a general purpose software package.

Realistic simulation should therefore instead be achieved by having a simulation version of the robot controller connected to an object oriented simulator of the mechanical robot. Alternatively, an available robot controller could be used. The interface is anyway the same as for the motion

control as mentioned above.

Encapsulated arm control

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. Having a specific software layer for this type of control will hopefully ease the implementation of such algorithms.

The purpose of the application layer is to keep the basic motion control clean from application specific 'fixes'. However, for major applications it may still be a good idea for the robot manufacturer to develop special built-in control solutions. The reason may be that access restrictions and inappropriate data flows prevent an efficient implementation in the application layer. The only example found, so far, is short arm motions for spot-welding robots. The characteristics of this example are:

- The robot dynamics can be assumed to be constant during the motion. The load is also constant and well known.
- Performance of the motion is limited by torque, torque derivative, and resonance excitation. Speed is not a limiting factor.
- The performance of the short moves, which typically take less than $0.5s$, are crucial for productivity. Shorter sampling periods may be needed to increase the time resolution of trajectory generation and for position control.

Note that less CPU time is needed due to the first two points, whereas the last point implies a higher CPU load. We can therefore consider the need for special arm control as a way of improving performance by better hardware utilization. Let us name the special short-move function `ShortMove`. That function will be referred to later in Section 8.2 treating spot welding.

The motion control – arm control interface

The following features are useful for the built-in control of the robot arm, but also as services provided for the motion control layer:

- 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 [61] should be provided. 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 parameterized 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.

This list should probably be extended if, for instance, integrated force control of multiple arms should be supported. The arm control software layer can for instance be encapsulated in a class.

6.3 Remaining problems

Each specified feature so far is based on known techniques. In other words, the individual features as such are known; it is the organization of the control software that is new. It is then natural to question whether the imposed structure hinders employment of control algorithms.

There are by definition no problems concerning the motor control layer. Only control that can be performed joint-wise is included in the motor control. It is the encapsulation of the arm control that we should be concerned with.

Both the desire to have the arm control encapsulated and to have added external joint control integrated with the arm control may be a problem. The algorithm should to be formulated in a way that makes it modular with respect to the individual external joints. It is not clear if this is generally possible. To prepare for a treatment of this problem in Section 6.4, an introduction to path velocity control is included finally in this section.

Another observed problem is about real-time performance of motion planning. Lower layers of the system must be designed for efficient real-time execution in micro processors and signal processors. Robot motion algorithms have therefore traditionally been split into two types. One type includes algorithms that can be used at run-time and thereby also for motions which depend on sensor inputs, whereas the second type is algorithms that require substantial CPU-time (before motion can start). Such algorithms have only been possible to use in an off-line manner for motions that are known in advance. It would of course be desirable to admit such algorithms also for sensor-based motions that are only slightly adjusted at run-time. This means that the motion control layer should include management of precomputed motions. A special issue is how sensor objects should behave since the actual sensor signals are available at run-time but not at plan-time. The problem of finding a software design that combines simulated motions, precomputed motions, sensor-based motions, and incorporation of sensors is the subject in Section 6.5.

Introduction to on-line trajectory time scaling

The aim of the on-line (i.e., at run time) trajectory scaling [62, 61] 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 dynamic effects like friction (in the robot or to the work-piece) varies. Some spot welding robots, for instance, must be exercised several minutes after maintenance 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 6.3, in such a way that the required torque matches the torque limits. If no joint exceeds its torque

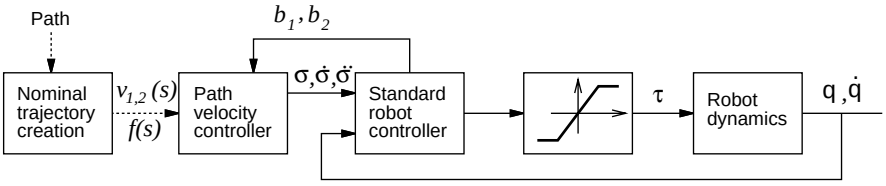

Figure 6.3 Path following according to [61].

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 desired path is followed. The basic idea behind the algorithm is to parameterize the standard robot controller in a new path coordinate s. It is assumed that the path is specified as a vector-valued function $f(s)$, the nominal trajectories being 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 [61, 62]. Figure 6.3 is adapted from [61], but the data flow for the path specification is explicitly shown.

6.4 Arm control – external control interplay

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 [61], 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 over-speed 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

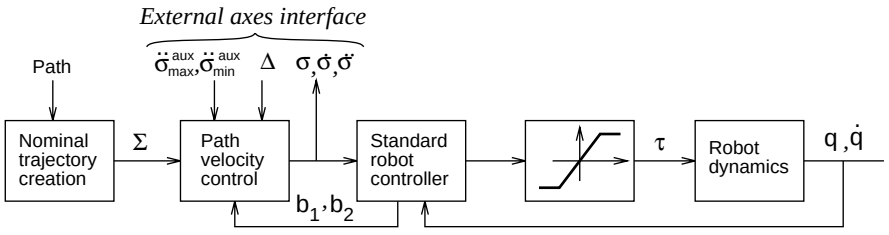

Figure 6.4 Path following as in Figure 6.3, but with proper signals (i.e., depending on time) and an interface for external axes.

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 6.3 was that the motion specification was considered to be preplanned nonlinear functions [61]. The signal Σ on the other hand is a proper signal as shown in Figure 6.4. 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 permitting 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 [61], but in stages as shown in Figure 6.5. The time lag Δ is also computed as the maximum of the input Δ and the time lag for the axis itself.

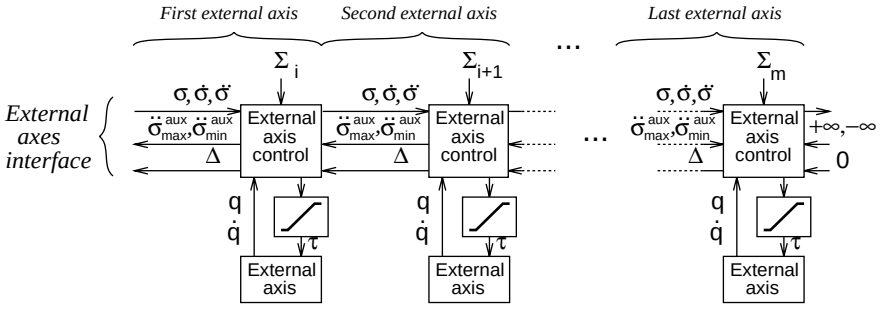

Figure 6.5 Proposed "bit-slice" style for incorporation of external axes.

These modifications illustrate how one algorithm can be modularized to fit into a structure suitable for industrial use with external servos connected to the robot controller. It should even be 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.

6.5 Pipelining and caching sensor-based motions

As mentioned above, advanced algorithms for optimization of motions sometimes take so much computing time that it may delay the motion more than improved performance can speed it up. However, much of the computations can often be performed in advance, that is, 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 implication is that run-time support for precomputations is necessary for optimum time algorithms and the like.

Temporal properties

A trivial approach to pre-plan motions would be to let the motion control interface contain explicit functions to obtain computed motion data without performing the motion (as mentioned in the description of the interface between the arm control and motion control layers). Such a feature may be useful for simulation and high level planning, but for ordinary usage we want something simpler.

Another too limited approach would then be to request “suspended motion”. That is, motion data should be computed but transfer to the motion control should be suspended until it is released from the caller. This approach would, however, not permit a specific motion to be reused later (next cycle of the task), and computing in advance would make the robot program unnecessarily complex.

Instead, the idea here is to use software solutions resembling principles used for similar problems in modern computer hardware (like pipelining, paging, and cache memories). This means that we want the system (the motion layer in this case) to internally manage the performance improving features. Analogous to the computer hardware case [75], the system should be able to

- fetch motion instructions in advance, start processing them, and put results in a buffer (from which set-points to the feedback control easily can be determined),
- save key data that have required substantial computing time in temporary storage,
- reuse computed data if available in the temporary storage.

The first of these items, which we may call a *motion pipeline*, is being done in systems today. The other two items, which we may call a *motion cache*, has not been seen elsewhere.

Also in analogy with the hardware case, we need means to specify how a specific instruction should be handled. That is when we want to influence the caching based on knowledge about the application. Therefore, in addition to internal functionality, the motion commands/functions available to the application layer should, among other arguments, take an argument that specifies the desired temporal property of the motion. Considering time, there are in principle only three cases; past, present, and future. So, to each move-procedure there should be a predefined parameter with an enumeration type consisting of the following three values:

`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 the motion cache.

`done`: The motion can start immediately using data that have already been stored in the motion cache.

Computed motions stored in the cache can be referred to by an identification number which we call *move descriptor* (compare with file descriptors in UNIX). Use of the wrong move descriptor for a motion, or if the descriptor points to data that have 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. A verification of this principle was obtained by guiding and supporting an implementation [191] using Matlab on the host computer as an RPC server for computing optimized trajectories.

Specification of sensor dependency

Motions, as specified in the call of the `Move` function, to some extent depend on sensor inputs. A motion that does not depend on any external sensors is known in advance, and the precomputation of motions described above can be directly applied. If, on the other hand, the desired motion is completely determined by sensor signals, its trajectories cannot be computed in advance. Possible performance improvements depend on the degree of sensor dependency in some (complicated) way, but for the management of sensor readings and motion pipelining we only need a logical specification

of how the system should handle the motion request. Each case needs to be explicitly considered internally in the motion control layer. It is therefore desirable to have few alternatives to simplify the implementation. On the other hand, having only one or two cases (depending or not depending on external sensors) as in available systems is not enough. In the ORC architecture, the application layer program passes the information about the degree of sensor dependency by supplying an additional enumeration parameter (in a way similar to the temporal specification above). This enumeration type is proposed to consist of the following four elements:

`completely`: A completely known motion does not depend on sensors at all. All services available from the motion control (like optimal trajectory planning in advance) can therefore be used.

`almost`: An almost known motion may be subject to modifications based on sensor information, but only to such a limited extent that motion planning is still appropriate. One example may be gluing along almost known paths that are followed using a path tracking sensor. To leave control authority for the sensor-based adjustments, the nominal motion may be planned with some margin according to supplied parameters.

`partly`: A partly known motion basically depends on sensor information, but some nominal path exists. It is however, not useful to plan or optimize this type of motion. For instance, 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 when the path is lost.
- 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 for supervision purposes.

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

`not`: A not known motion is entirely specified by sensor signals. One such case is manual move of the robot arm using 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 6.6.

Solutions to the run-time trajectory planning and generation problem [116] are unusual in the literature. (Do not confuse this with, for example, the solutions in [59] where the trajectory generation is done at run-time,

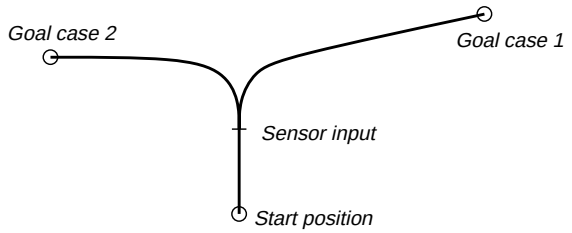

Figure 6.6 Path and trajectories not known in advance. The goal position depends on the input from a sensor.

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.

Deduced properties of sensor objects

We have so far considered the fact that motions may depend on external sensor signals, 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, using an object-oriented language, is a good way to provide language support for management of sensors.
- Application-specific or task-specific control implying that the filtering of sensor signals should be possible to change from higher levels of the system.
- External sensors should be possible to combine with the proposed motion pipelining.
- The simulation support should apply also for sensor-based motions.

It is quite natural to model sensors with a general base class, and then create derived subclasses for each special type of sensor. The idea here is that the base class should specify the following three behaviors to support motion pipelining and simulation:

- *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 (computation of minimum time trajectories for instance) of almost known motions can be carried out exactly the same way as for completely known motions. The dummy sampling typically returns a worst case value (possibly depending on time, position 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. Using the software techniques that will be described in Chapter 7, the behavior of each sensor (given by the application or task) can be defined, compiled, and passed to the sensor object in the motion control layer.

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 required functionality can be conveniently expressed in a base class. Further implementation details were presented in [144], but this presentation concentrates on the principles.

6.6 Open motion control

In order to make a layered system flexible, the interfaces between the layers must be open in a suitable way. Considering the interface between the motion control system and higher levels, we need some means to send down motion specifications that are general enough for advanced robot applications, and some means to obtain motion data for use on higher levels of the system. The interface specifications outlined in Section 6.2 are believed to be state-of-the-art, but not enough. Considering the real-time properties of the system, this section first describes how more general motion specifications can be handled without incorporating the world management into the motion control system. Secondly, benefits of application specific access to internal control signals will be described.

Motions determined by world model relations

The motion specification issue is by itself a research topic, particularly if compliance and force control should be supported [177]. Even if we limit the discussion to ordinary position specifications, the situation may be quite complex. Consider for example object-level programming (described in Chapter 3) for objects that are varying with time or moving around in a way not known in advance. External sensors must of course be used to obtain information about the environment at run-time. To handle cases like this, the position equations in RCCL [81] may contain frames that depend on sensor signals. The frames are also part of the world model. The development towards more flexible/intelligent robots has therefore implied a tighter integration between the world modeling and the motion control.

Opposed to such an integrated approach, the proposed layered ORC architecture specifies that the feedback control part of the system should be separated from the world modeling and the high-level programming system. That is to achieve the proposed user views. Another reason is to make it possible to have a simpler embedded system for applications not requiring the advanced features. A more attractive approach is therefore to have a fixed hierarchy with a few predefined frames that can be efficiently handled in the embedded system as in the new system from ABB [2], and then (as a proposed extension) allow these frames to be evaluated according to functions supplied from the application layer. Those functions can then, for instance, make accesses via an additional CPU board to obtain world model information in the most advanced cases. This was presented as motions in function space in [144] where also some more details were given, but a further analysis and a full implementation remain to be done.

Software sensors

Industrial robots are mechanically rather precise machines driven by servo controlled motors. They are not as precise and rigid as NC machines, so desired motions are not quite precisely achieved. The same applies when compliant motions are used on not quite known work-pieces (like the casting in Figure 2.5, p. 12). In many cases, the actual motion or production result therefore needs to be measured in order to compensate for deviations and errors.

The problem with adding external sensors is that the hardware gets more expensive, and it decreases the MTBF (Mean Time Between Failure). However, to meet the performance requirements on the motion control, the dynamics of the robot is reflected in the control system (by state

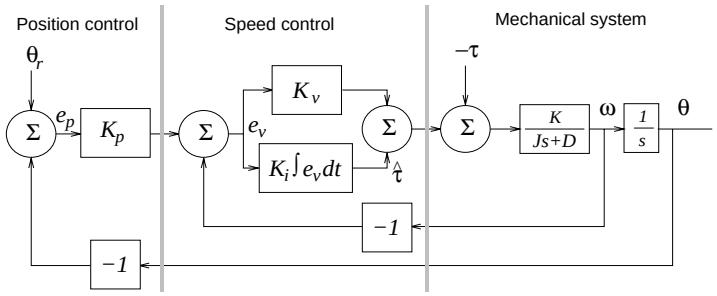

Figure 6.7 Simplified servo control.

observers, linearizing feedback, etc.). Therefore, states of importance for the robot task exist within the control system. This is known in principle by control engineers, but using such (internal) information to solve robot application problems has not, as far as known, been suggested until now. Furthermore, it is usually not clear to application engineers that such useful information exists. The following simple example further illustrates this idea of utilizing internal control information instead of external sensors.

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 6.7. It is basically this type of control that is used for the joint-wise control of most commercial 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.

The following examples illustrate how the control signals can be useful. 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.
- 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 extract it. 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.

From the user programming point of view it makes no difference, if a procedure called to get sensor data samples the signal from an external device, or if data acquisition is performed internally in the system. Therefore, the treatment of sensors in the previous section also applies to software sensors. Implementation techniques will be developed in Chapter 7.

6.7 Implementation

From the user programming level, the motion control system can be seen as a driver for the manipulator device. Such a driver can then be implemented in an ordinary operating system manner [81], it can simply be viewed as an abstract data type [215], or it can be encapsulated by a class in an object oriented framework[133]. However, the needs for openness in advanced manufacturing applications indicates that a more general type of interface is appropriate. The operating system driver model can of course be kept in principle, but the generality of the services (the io-controls in a UNIX framework) would not be well expressed to the programmer. Instead, an object oriented framework extended with the principles of open embedded systems presented in Chapter 7 is suitable for the motion control interface.

Considering robot programming within the layers for built-in motion control, this is the control engineering topic. In this field, good tools for identification, modeling, synthesis, and code generation have emerged in recent years. The aim in this work has been to connect such (host computer based) tools to the embedded robot control system.

Motor control implementation

The motor control was mapped to the hardware in such a way that the DSPs were utilized for the high-frequency properties of the control, and the master M68040 CPU was used for the low frequency part. The reason is that the DSPs execute pure filter algorithms very efficiently, but integrating action and the necessary wind-up protection contains logic that breaks the floating point pipeline in a very undesirable way. High sampling frequencies are also possible because the control is interrupt driven without the overhead of a full real-time kernel. The M68040 CPU on the other hand is better suited for general control tasks [53]. Currently, using our department's real-time kernel [18], the sampling frequency cannot exceed 1 kHz, which is slow compared to the 24 kHz currently used for the DSPs.

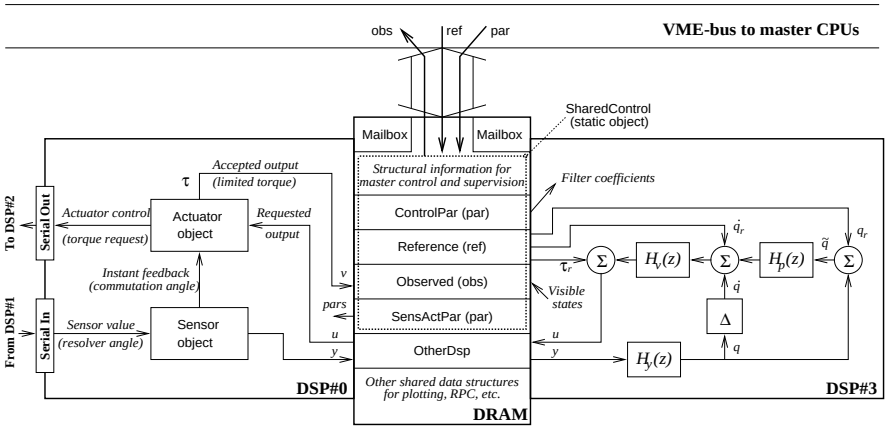

Figure 6.8 Motor control implementation on the DSP hardware level. Only DSP#0 and DSP#3 are shown. Connections to the other DSPs and to the controlled robot are shown in Figure 5.7 (p. 79) and Figure 5.2 (p. 71)

Figure 6.8 shows the data flow for the DSP part of the control. The DRAM memory (see also Figure 5.7, p. 79) is central for the multiprocessor implementation. Control parameters and references are written by the master processor to the DRAM, the two DSPs shown in the figure exchange control data via the DRAM, observed states are updated by the DSPs and it is then used both by the master part of the control and by the supervision part (Figure 5.6, p. 77). The programs were written in C++ for the DSPs as explained in Section 5.3. The filters $H_y(z)$, $H_v(z)$, and $H_p(z)$, however, were available as highly optimized assembly code from the application library for the DSPs.

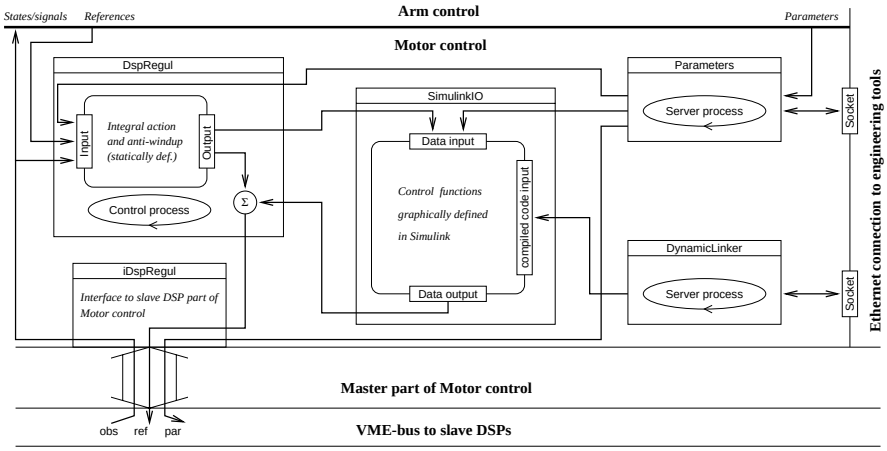

Figure 6.9 Master part of the motor control implementation.

The master control part was mainly written in Modula-2, but since the Modula-2 programs were translated to C before cross compilation, data structures in common only needed to be declared once. Special care was taken to handle the differences in address spaces, real-time primitives, floating point representation, and byte numbering as described in Section 5.3. Data logging services are not shown for clarity.

The master control part running on the M68040 board is shown in Figure 6.9. It contains a fixed implementation of an I regulator which can integrate speed or position error according to parameters set at run time. The I-part takes care of the low frequency behavior of the control, whereas the HF-control takes care of the high-frequency part. Advanced motor control like active damping typically takes place in a mid-frequency range [83]. The maximum sampling frequency of 1 kHz for the master control is enough for this, given the HF-control part. To rapidly prototype new control principles, code generation from block diagram descriptions on the host computer form a valuable tool. That brings us into the next subject.

Engineering tools

Embedded (robot control) systems should be as small and efficient as possible. For engineering workstations, it is more important to have good user interfaces and plenty of computing power. That is to save costs for production equipment and engineering time. The approach in this work has been to have the real-time control part well separated from the engineering tools running on the host computer. A network connection between

the two is of course slower than using bus-to-bus adaptors, but it corresponds well to the situation for manufacturing systems. A factory network connects embedded control systems to the central manufacturing control and to the engineering workstations, possibly located on a remote site. Without going into details, the following software tools were developed:

- Using available low-level networking primitives and UNIX sockets, processes in the embedded system were connected to Matlab running on the host computer. Opposed to ordinary pipes and streams which work on a byte level, the developed connection is typed in the sense that it works with matrices which fit well with Matlab.
- Data-logging modules were developed for the embedded system (both for the M68k and for the DSPs), and a Matlab-based front end was written.
- Development of services in the embedded system for excitation signals was initiated and guided. That facilitates the experimental part of system identification.
- An embedded system server for control parameters was specified, and the implementation was initiated and guided. A powerful user interface was implemented by the author using the Matlab graphics. The parameter interface is layered in such a way that parameters can be accessed on five levels; the embedded level, the network level, the Matlab script level, the Matlab graphical level, and the Matlab script/graphical level. The script/graphical level means that the graphical user interface is run and updated from a script.
- An operator interface for manual control and robot program execution was implemented using Matlab graphical objects. The reason to use Matlab is to make it simple to include the 'manual' operation in scripts expressing control experiments.
- Using the embedded dynamic linkage according to Chapter 7, it was made possible to change at run-time the cross-compiled programs from Simulink/Real-Time-Workshop [125]. See Figure 6.9.

Thus, a tool like Matlab with the possibility to link in external (networking) functions is very useful. It is of course also important to make use of other available tools whenever appropriate. Evaluation of one such tool [172] is therefore currently going on. Another system developed within our department [66] provides high level description of control blocks, code generation, and reconfiguration of control blocks during run-time. These tools are well suited for development of the arm control and the motion control, whereas the more limited Matlab-based tools are suitable for the single-joint control.

The reason for focusing on the engineering tools, instead of on new algorithms, is that improved control performance is more a matter of employing known control principles in an efficient/convenient way than finding some new theory. On the other hand, improved algorithms for control synthesis is needed to reduce the engineering effort, for instance to adopt to changes in the mechanics. The developed system will hopefully be used for such research. Experience from our own tools so far is that interactive operation of the controlled process connected to a control design environment greatly facilitates the control engineering work.

Experiments and discussion

The presented implementation was developed in connection with student and master thesis projects [129, 84], in which successful experiments also were carried out. Experiments on individual joints have been made with sampling frequencies ranging up to 24 kHz for position and velocity control, and up to 48 kHz for torque control. The present standard sampling frequency when all six joints are used is 4 kHz. The hardware interfaces are built for 8 kHz (i.e., 48 kHz for all joints together), but we preferred to use a well structured and flexible C++ implementation using only half the rate. Controlling all joints which the rate of 4 kHz implies that the order of the position/velocity controller should be less than ten. The Simulink generated controllers, running on the master M68040 board, can usually not be run in more than 500 Hz.

As an example of a measurement on joint one of the Irb-2000, Figure 6.10 shows the position output when the input torque was a logarithmically swept sine-wave with constant amplitude. Estimation of transfer

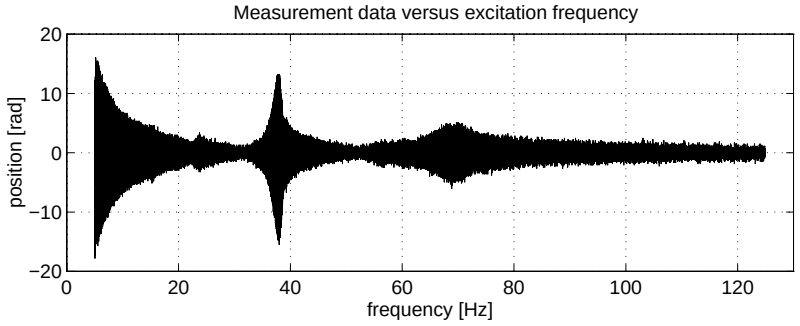

Figure 6.10 Measurement performed on the fist joint [129] using the developed experimental environment. To both excite resonances and capture a wide frequency range in a single experiment, 35000 samples were recorded over a time period of 70 seconds (500 Hz sampling frequency in this case).

functions from this type of data is straight forward [99]. Doing the experiments interactively, or via Matlab scripts that also serves as documentation, has proven to be very convenient. The main 380V drive power can also be turned on and off from the host workstation. Note that we do not use any intermediate storage in files; measurement data is directly sent from the embedded controller via the network to the Matlab workspace.

Experiences so far indicate that the layered architecture is quite convenient to work with. We have, on the other hand, not used the layers rigidly because the interface design has been part of the work. It would of course be desirable to have detailed specifications of the interfaces. That would permit independent development of complete software layers or modules. Such a standardization should, however, wait until more experience of open, layered, **and** efficient implementations is available. For instance, available implementations [133] are appropriate for the situations they are designed for, but they are not directly applicable in an industrial manufacturing context. The requirements on multi-layered programming and run-time efficiency are hardly fulfilled, and defined interfaces [133] are not general enough. As pointed out on Page 15, standards most often come with maturity.

The question about interaction between the arm control and the control of external motions, as stated earlier in this chapter, is partly left open. The study of the path velocity control did not reveal any major restrictions due to the imposed control layers. Further evaluation of state-of-the-art robot control algorithms, both with respect to the software structure and the control properties, is in progress.

6.8 Summary

The design of the standard, i.e., application independent, part of the motion control system has been developed in this chapter. The aim was to support control engineering in a framework considering industrial demands, and to make it possible to add high-level control on top of the built-in control. To achieve this, the following new principles were proposed:

- The motion control software should be divided into motor control, arm control, and overall motion control. That is to manage control complexity, hardware structure, and system development.
- It was described what services the motion control system needs to provide for higher levels of control. The client/server design combines encapsulation of control solutions with support for advanced high-level control.

- The need to structure the control for incorporation of external joints was pointed out, and an example how that can be done was given.
- A concept to allow use of more CPU-time consuming algorithms for embedded control was proposed. This includes a straight forward way to save and refer to computed motions. It also contains an important classification of sensor dependency.
- Sensors (software) objects of course include information about physical input channel, sampling, filtering, etc. Additionally, to facilitate the two previous items, sensor objects should have methods for real, dummy, and simulated sampling. This is a key idea for advanced use of sensor based motions.
- The possibility to use control signals as sensors was pointed out. This is typically not realized among application engineers.
- A structure for motor control was described, and software tools for motion control engineering were developed.

The contribution of this chapter is a motion control structure with unique support for: Encapsulation of control properties, appropriate programming views and tools for control engineers, flexible tailoring to application specific demands, and management of (sensor-based) motions to achieve efficiency without complicated programming. Major parts of the structure have been implemented for control engineering purposes.

7

Open Embedded Control

Embedded control devices today usually allow parameter changes, and possibly activation of different pre-implemented algorithms. Full reprogramming using the complete source code is not allowed for safety, efficiency, and proprietary reasons. For these reasons, embedded regulators are quite rigid and closed concerning the control structure.

In several applications, like industrial robots, there is a need to tailor the low level control to meet specific application demands. In order to meet the efficiency and safety demands, a way of building more generic and open regulators has been developed. The key idea is to use pieces of compiled executable code as functional operators which in the simplest case may appear as ordinary control parameters. In an object oriented framework, this means that new methods can be added to controller objects after implementation of the basic control, and even while the controller is running.

The implementation was carried out in industrially well accepted languages such as C and C++. The dynamic binding at run-time differs from ordinary dynamic linking in that only a subset of the symbols can be used. This subset is defined by the fixed part of the system. The safety demands can therefore still be fulfilled. Encouraged by results from some fully implemented test cases, we believe that extensive use of this concept will admit more open, still efficient, embedded systems to be developed.[4]

7.1 Introduction

Making machines programmable has added flexibility to manufacturing equipments. Typical examples are industrial robots and NC-machines.

4 A slightly different version of this chapter has been submitted for publication[149]

They can be reprogrammed for applications that were not defined at the time the equipment was developed. Improved functionality/performance and cost efficiency are other major driving forces for use of computer control, for instance in vehicles. Re-programmability is very desirable also in this case, which means that the lifetime of the product can be prolonged without having to exchange the components when new (e.g., environmental) demands are imposed.

In mechatronics and process control, demands on safety, efficiency and simplicity are often contradictory to flexibility. Products providing features for end-user programming, such as industrial robots, are therefore preferably layered in such a way that the ordinary user can do the application programming in a simple and safe way, and the expert programmer can tailor the system to fulfill unforeseen requirements from new applications. The same also applies to many mechatronic products that do not provide end-user programming. For example, the multivariable engine control in a car should be possible to update by the car manufacturer. Furthermore, within mechatronics it is in general desirable to use components that can satisfy the needs of many applications and products. That will contribute to cost reduction. This means that we have the same situation as for the robot control; *open components* are needed.

A system that allows such low-level reconfiguration to meet requirements that it was not specifically designed for, is called an *open system*. Most machine controllers used in industry today are *closed*. The reason is that they have been specifically designed to provide an easy to use, cost efficient, and safely performing control for today's standard applications. In present and future more advanced applications, however, customization of closed systems becomes time consuming and thereby costly. Development towards more open control systems are therefore going on, see [144, 151, 70] and references therein. Still, there are proprietary, efficiency, and safety reasons for not having a completely open system. The key issue is to find the right tradeoffs between an open and a closed design of a system. The best possible solution will, of course, depend on the available software mechanisms. We include both (partly) open systems and components in the term *open embedded control*. The term control is used since the purpose of behaviors subject to change are typically to control the system, or parts of it.

The trend towards more open systems has been going on for some time within operating systems, computer networks, interactive software, and graphical user interfaces. This development has resulted in software techniques that have inspired our approach to open embedded systems. We do, however, not want to impose the use of standard (heavy) operating systems and process communication principles on an otherwise lean

design of a cost-sensitive mechatronic product.

The desire to create software mechanisms that support simple implementation of open embedded control can now be formulated as: Given appropriate hardware, how should the software be organized to allow efficient extension (in terms of improved functionality, performance, etc.) by changing open parts of the software only, while some parts of the system may remain closed. As the examples in the next section will show, only allowing parameter changes and use of pre-implemented algorithms does not provide enough flexibility.

One approach, used in PLC systems, is to down-load new code for interpretation to each hardware unit. In many applications, full utilization of the computer hardware is required, and executable code (cross-compiled to the specific hardware) is therefore preferred, and also used in a modern process control system like SattLine [97]. Use of compiled libraries admits the implementation of certain components to be hidden, but often we need closed subsystems with open 'slots'. Recall, however, that we want to keep part of the software closed. The question is then how to obtain a suitable border between the open and closed parts. This chapter presents a solution to this problem, without requiring any special language or run-time model. We therefore believe that the proposed way of building open real-time control systems is an innovative simplification compared to more general concepts (like [87, 154, 132, 23, 201, 178]) for open systems.

7.2 Applications

Some typical situations when an embedded system needs to be open are presented in this section. These examples are all in the area of manufacturing systems, but they also capture typical properties of many other types of embedded systems.

Feedback control

The most common type of feedback controller in industry is the PID-regulator [28]. In most process control applications, PID-controllers give sufficient performance and there are simple rules and schemes for how to do the tuning without requiring extensive knowledge about the process dynamics. Although the basic PID algorithm is very simple, industrial application demands have resulted in enhancements like gain-scheduling, auto-tuning, and adaptivity. Such features have been built into more advanced PID regulators.

Within mechatronics and machine control, ability to compensate for known nonlinearities is often more important for performance than the

ability to handle unknown changes in process dynamics (called adaptivity above). Compensation of nonlinearities in the process typically results in nonlinear control functions. Such functions are normally specific for the application. Nevertheless, like in the process control case, it is desirable to use standard control blocks in terms of hardware and software. That is usually not possible within mechatronics today, but such flexibility concerning the embedded control functions could reduce the cost of the controller in many cases. Furthermore, the sampling frequencies required in mechatronics are typically much higher than those used in process control. Special care has to be taken in order to achieve the flexibility in an efficient way.

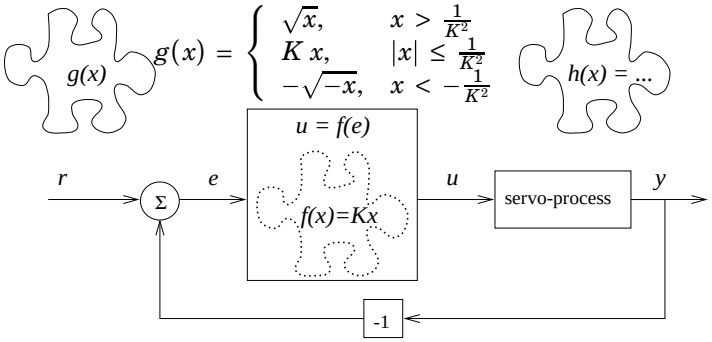

Figure 7.1 Block diagram of a P-controller allowing the gain to be replaced (at run-time) by a nonlinear function. The puzzle pieces illustrate type-checking during dynamic binding.

As a simple example of industrial motion control, assume we want to control the motor angle in a mechanical servo system. Further assume that we do not accept any overshoot when the desired angle r is approached, and that an inner analog velocity controller is used together with drive electronics that controls the motor torque within a certain range. This means that we have a servo process (see Figure 7.1) with an inner nonlinearity due to the limited torque available. Only the P-part of the PID controller is used in this case.

With some basic knowledge in physics and control, one can easily see that the nonlinearity can be almost fully compensated for using the function $g(x)$ shown in Figure 7.1. The trick is to use the square-root function to compensate that the required braking energy is proportional to the square of the speed. For stability reasons, the function g contains a linear part for small arguments. This type of nonlinearity is used in some industrial drive systems, but implemented as a built-in special purpose feature.

If we instead make an embedded PID-controller in which the proportional gain constant can be exchanged by an external function, much greater flexibility is achieved. These functions need not be known when the controller is shipped. They can be written and inserted when the controller is to be used. In the next section, we will return to how this can be achieved, but it means that we can use the same embedded controller as a building block for controlling very different processes. Type checking ensures that only a conformant function can be inserted. The type of the functions is illustrated in Figure 7.1 by the shape of the pieces.

It is common within mechatronic applications to make use of a dynamic model of the controlled system to achieve the best possible performance. Control techniques like state feedback, state observers, optimal control, linearizing control, etc., are used for this purpose. Control of manufacturing equipment and many military control problems are typical applications for such control strategies. In such systems, like industrial robots, it is even more valuable to have software slots admitting the behavior of the machine to be tailored to different needs.

As an example, assume we have a well functioning embedded regulator comprising state feedback control. For brevity, only two states x_1 and x_2 are used in the sequel. State feedback means that we compute the (in this example scalar) control signal u as the negative scalar product of a gain vector L and the state vector $x = [x_1 \quad x_2]^T$, i.e., the control signal u is defined as $u(t) = -L \cdot x(t)$. The state feedback can in a simple application be used to control a servo, as in the simple control case above. The following example of exact linearization is from [105], which also contains more advanced applications where this type of control is useful.

Consider the pendulum equation

$$\ddot{\theta} = -a \sin\theta - b\dot{\theta} + c\tau$$

where $a = g/l$, $b = d/m$, $c = 1/ml^2$, θ is the angle subtended by the rod and the vertical axis, and τ is the torque applied to the pendulum. Here l denotes the length of the pendulum, d the viscous friction, m the mass, and g is the gravity constant. View the torque as the control input and suppose we want to stabilize the pendulum at an angle $\theta = \delta$. This can be achieved by the control law

$$\tau = \frac{a}{c}[\sin(x_1 + \delta) - \sin\delta] + k_1 x_1 + k_2 x_2$$

where the first term takes care of the nonlinearity, and the rest is a linear state feedback that is used to place the closed loop poles using the gains k_1 and k_2 from the measured states position and velocity respectively.

It would be desirable to allow an efficient implementation of this control using a standard embedded state feedback regulator. This means that the regulator by default should provide linear feedback, but the gains should be possible to replace by nonlinear functions. There should be no significant overhead due to such a feature. Furthermore, in more complex applications, it should be allowed to have selected parts of an algorithm hidden. The hidden (pre-implemented) part could then conceptually belong to the actual mechatronic device or component.

Supervisory control

A distributed control system is composed of separate local controllers and at least one supervisory controller. In such a system it is preferable to have a vendor independent interface to the local units. It is also important to minimize data flow and reduce the computational load on the supervisory controller. However, standard controllers for local use have different signals available. At construction time of a controller it is impossible to know which signals might be desirable to use, since this depends on the specific application. In other words, distributed control systems today are sometimes burdened with more communication and computations than necessary.

An attractive solution would be to have a slot in the local controllers for additional computation. This could be in the form of algorithms on a smartcard supplied by the vendor or executable code loaded over a local network. With this approach standard controllers could be customized to fit different environments, and the computational load on the supervisory controller could be reduced.

This type of functionality partially exists in todays process control systems, but requires that the local controller is integrated with the supervisory control which either sends down new code for interpretation, or restarts the local controller with new compiled code. Dynamic binding of functions would be a more attractive solution. These functions should not need to be defined in a special process control language defined by the supervisory system. Using a standard computer programming language instead means that the local embedded controller more easily can be integrated into quite different control systems.

In terms of PLC programming [89], we want to use a vendor independent method to change function blocks, or parts thereof, at run-time. Such a principle seems to be compatible with the evolving standard [89] for PLC and control system programming. (The standard does allow use of externally defined functions and we have not found any requirement on static binding of such functions.)

Intelligent low-level sensing

Intelligent sensors are equipped with electronics and data processing capabilities so that data can be processed locally, thus requiring less data transfer and computations for the main control. Performance can also be improved by using intelligent sensors. Filtering algorithms in the sensor signal processing unit may run at a higher sampling rate, and operate on data that has not been delayed by data transfers.

High volume production of sensors like accelerometers and laser sensors have resulted in very low cost for the sensor. Field busses and interface electronics for connection of sensors are also becoming affordable. Such sensors today may provide a protocol for changing filter parameters and threshold levels. For a sensor to be considered intelligent, we also require that more general nonlinear and application specific algorithms can be performed in the sensor. There are, however, also parts of the filtering that are specific for the sensor technology, and should not be exposed to the user. This means that we want to have a hidden part of the filtering hardware/software, and an open part where application specific software should be possible to plug in. Such algorithms sometimes need to change over time, depending on the type of information needed by the high level control (like task planning etc.).

As a simple test case, consider an accelerometer mounted on a robot arm. Normal robot motions result in smooth changes in acceleration. More rapid changes result if the robot hits an obstacle. We then want this event to be detected by the sensor and reported to the main control computer. If more information about the collision is requested, the sensor should also be able to reply with a recording of the signal during the collision. Such information can be provided by a special purpose intelligent sensor, or by an off-the-shelf intelligent **open** sensor.

7.3 Embedded dynamic binding

We will now investigate how to bind functions at run-time to software already running in the embedded system, considering demands on efficiency and predictability for typical mechatronic applications.

Alternatives

The problem we are facing can be described in terms of a client and a server. The server is the embedded software already implemented, and possibly running. The basic service is first of all to perform the control, but also to service requests to change the control behavior. It is the

latter aspect that is treated here. The client acts on a higher hierarchical level, which means that it contains more application specific knowledge. Depending on what is appropriate to solve the particular application, the client requests the lower level control (i.e., the server) to use different new control functions. We then need some means of expressing these functions and to dynamically let the server evaluate them. As throughout this work, we try to apply well proven software techniques, but in the context of mechatronics and embedded control. Some of the alternatives we have considered for the implementation and dynamic binding are:

Remote Procedure Calls (RPC) are widely used and understood, but since the input/output arguments of the procedure have to be transferred to/from the computer that executes the procedure, the resulting data flows and communication delays can be substantial. Unpredictable timing would also be a problem, for instance when Ethernet connections are used.

Interpreted languages make it straightforward to pass functions to some target software containing an interpreter. Some languages to consider are:

Lisp is a flexible and powerful alternative, but tends to be one or two orders of magnitude slower than compiled languages unless the Lisp code is compiled. Its dependency on garbage collection also makes it hard to use for real-time tasks.

Forth is powerful and relatively efficient, but hard to debug and maintain due to its lack of structure. A solution useful for our needs could probably use Forth as an intermediate language, but with no apparent advantages.

Erlang [25] is an interpreted parallel language developed for telecommunication systems. It compiles to reasonably fast intermediate code that can be passes between processes running in different address spaces or on different CPUs, but as Lisp it relies on garbage collection, and we therefore decided against using it in our prototype implementation.

Java is a relatively new language with support for object-orientation, threads, and their relationship [141]. It is compiled into efficient bytecode, but the drawbacks of Erlang remains.

Compiled functions written in a widely used language like C would be an attractive solution. The problem of binding the used symbols to the external environment then has to be solved. If this can be done, we will hopefully achieve almost the same flexibility as the other alternatives, and almost the same performance as for builtin functions.

Another drawback with the interpreted languages is that the system must always contain the (compiled) implementation of the interpreter, even when no additional software are used. The **Compiled functions** alternative, on the other hand, gives a wide range of language choices since almost any cross compiler can be used. In the sequel, the term *action* will be used for such a compiled function that is to be dynamically bound in the embedded system. This is the solution we have chosen for our project, and in the following sections we will investigate two different implementations of **actions**.

Function-based actions

Function-based actions are a chunk of code that has been compiled as position independent code (PIC). All interaction with the control system is done via an environment pointer that is passed as a parameter to the down-loaded function. The following implementation of a small example based on the application shown in Figure 7.1 further illustrates this approach. The header file `environment.h` contains the following C declaration of the environment:

```
typedef struct {
  float r, y, gain;      /* Names according to figure. */
  float (*sqrt)(float);
  /* more math functions here.. */
} Environment;
```

The following lines are then part of the standard implementation of the regulator:

```
#include <environment.h>

float defaultK(Environment *env) { return env->gain; }

float (*K)(Environment *) = defaultK; /* <- Action type. */

void Pcontrol() {
  Environment env;
  MakeDownloadable(K, "K");           /* <- Action slot. */
  env.sqrt = sqrt;
  for (;;) {
    env.r = GetRef();
    env.y = Sample();
    P = K(&env) * (env.r - env.y);    /* <- Control law. */
    SetOutput( P );
  }
```

```
}
```

where, as C-programmers know, the action declaration specifies that `K` is a pointer to a function that as its argument takes a pointer to the `Environment` specified above, and returns a float.

Since automatic code generation of control algorithms nowadays is an important control engineering tool, it is of course important not to impose special requirements on the way control laws are expressed. The fact that we use the structure `env` makes the control law (only the one line above in this simple example) different from the text-book version `P=K*(r-y)` in two minor ways: First, accesses of variables that are part of the environment are preceded by '`env.`'. That can for example be handled by preprocessor directives in C and C++, or by use of `WITH` in Modula-2. Secondly, the number `K` is represented by the function taking the address of the environment as an argument. (The "`(&env)`" can be avoided by using C++ operator overloading, but here we will be explicit and not confuse the reader by any syntactic sugar.)

Now the target system is prepared for a new function `K` to be down-loaded. Let us replace the `defaultK` with a separately compiled nonlinear gain function. Since the control law is the nonlinear gain times the control error, we have to divide the function g in Figure 7.1 by the control error. This means that the implementation of the action will be:

```
#include <environment.h>

float sqrtK(Environment *env) {
  float e = env->r - env->y;
  float a = 1 / (env->gain * env->gain);
  if      (e >  a) { return  1.0 / env->sqrt( e); }
  else if (e < -a) { return -1.0 / env->sqrt(-e); }
  else             { return env->gain; }
}
```

Further improvements concerning the control are possible, but outside the scope of this paper. Considering the software aspects the solution seems to be everything we need, but there are a few catches:

- The compiler must be capable to generate PIC code. On some platforms this can be prohibitively expensive in terms of execution speed or code size. It may also be the case that a certain compiler cannot generate PIC code.
- It is hard to make changes to the environment structure since all elements has to be in the same place, and no checks are done when the code gets down-loaded. Environment mismatches will result in runtime errors.

- Some language constructs need access to global data or global functions (e.g., operators **new** and **delete** in C++), which is not easily implemented in this scheme. This problem can be solved by a smarter compiler, but such a modification is probably not worthwhile.

In practice, this means that actions implemented this way are restricted to be single functions. This type of actions are therefore called function-based actions. Despite the problems, successful experiments have been carried out, as will be described in Section 7.4.

Dynamically linked actions

We will now introduce a linker to overcome the limitations with the function-based actions. Conceptually this seems more complicated, but in practice one can often use an existing linker and with minor effort generate a small file that contains the symbols that are needed to link with the control system. In our prototype system the linking is done with the GNU linker [55] and a small Perl [216] script (300 lines) on the host system, and a few small procedures in the control system (400 lines + 1 line per symbol). The action client performs the following algorithm (by running the Perl script on the host computer):

1. The script requests all available interface symbols of a specific action slot from the control system.
2. A preliminary linking with these symbols is done to check that all references can be resolved. If any undefined symbols remain, linking is aborted.
3. Using the resulting code-size from step 2, the control system is instructed to allocate a chunk of memory with the appropriate size, and its start address is sent back to the client.
4. The final linking is done and the address of the entry point of the action is retrieved.
5. The finished code is sent to the control system along with the address of the entry point, see the function `install` below.

The implementation is straightforward.

To make it possible to have multiple actions in the same code segment we will also change the way actions are installed in the example application. The following is the same example as above but somewhat modified, starting with the file `environment.h`:

```
typedef struct {
  float r, y, gain;
} Environment;
```

```
extern float sqrt(float);
/* more math functions here.. */

extern float (*K)(Environment *);
```

The following lines are part of the embedded control software:

```
#include <environment.h>

float defaultK(Environment *env) { return env->gain; }

float (*K)(Environment *) = defaultK;

void SendSymbols() {
  SendSymbol("sqrt", &sqrt);
  SendSymbol("K", &K);
}

void Pcontrol() {
  Environment env;
  MakeLoader(SendSymbols);
  for (;;) {
    env.r = GetRef();
    env.y = Sample();
    P = K(&env) * (env.r - env.y);  /* <- Control law. */
    SetOutput( P );
   }
}
```

As we can see, the things that have changed are:

- The **sqrt** function is no longer part of the environment structure, it is instead declared as an external function. By doing it this way, it is easy to add new functions to the interface, and the removal of old functions that are used by old actions will give diagnostic messages when down-loading is attempted. In the C++ case we can also detect signature changes since the parameter types are encoded in the function names (name mangling). When Modula-2 is used, module time-stamp mismatches are detected in the same way.
- The action pointer is made visible to the down-loaded action. The reason is that the down-loaded action is now responsible for the installation, a fact that makes it possible to replace multiple actions simultaneously. This also makes it straightforward to add static C++ objects to a code segment.

Note that even if we could have placed all variables of the `Environment` outside the structure (as done with the math functions), the controller specific variables are kept in the `Environment` to allow several instances of a regulator to use the same action.

The clients implementation of the down-loadable action is slightly changed, and an install function is added.

```
#include <environment.h>

float sqrtK(Environment *env) {
  float e = env->r - env->y;
  float a = 1 / (env->gain * env->gain);
  if      (e >  a) { return 1.0 / sqrt(e); }
  else if (e < -a) { return -1.0 / sqrt(-e); }
  else             { return env->gain; }
}

void install() { K = sqrtK; }
```

In this case `install()` only connects the loaded action by assigning the address of the action to the action pointer. In more complex applications, `install` would probably take a number of arguments to be used for initialization of the loaded action segment. Such initialization may include calls of C++ constructors.

Experience

For each one of the two ways of implementing the dynamic binding, there are some advantages and drawbacks. First, the following benefits and problems with the **function-based** approach have been experienced:

+ Easy to understand and implement
+ Code gets reentrant automatically, since all persistent data have to be kept in a pointer block. This means that the same code can be used at many places without any additional work.
+ Code blocks can be moved in memory after down-loading (compare Macintosh code resources [22]). This is a favorable property if memory compaction is needed.
– Global data can not be used.
– There are certain demands on the compiler; with GNU cc we had problems with non-optimized code and floating-point constants.
– On some architectures PIC-code is more expensive in terms of space and execution time.

– All calls to interface procedures have to be done by constant offsets into a jump-table; this makes it virtually impossible to remove anything from the interface (because old code would down-load correctly but use the old offsets which are incorrect for procedures located after the removed one).
– C++ devotees: Dynamic objects are hard to use as operator `new` and `delete` have to be global functions.

There are also some benefits and problems with the **dynamic linking** approach:

+ Any compiler can be used since all special work is done during linking.
+ Errors can be better detected when the interface is changed. In the C++ case signature changes are detected, and in the Modula-2 case time-stamp changes are detected.
+ The same programming model as in ordinary programs can be used. Global data and procedures are available, and even static C++ objects can be used.
– Special care has to be taken to write reentrant code. This is the same problem as in ordinary programs.
– The implementation in the target system is somewhat more complex.
– Code cannot be moved in memory since it is linked by the client to a specific address. This may lead to fragmentation problems.

We will now discuss the use of these techniques on the applications that were presented in Section 2.

7.4 Experiments

The two principles presented in Section 7.3 have been fully implemented. Full implementations of the application examples have also been done. Some features of these implementations will be presented in this section. The presentation is hardware oriented, using the experimental environment presented in Chapter 5.

Distributed systems

The use of actions to solve the supervisory control application exemplifies the benefits of actions in systems with distributed hardware connected via a computer network. An embedded control signal logging tool was developed in C++, and connected to the well known program Matlab running on the host workstation. A Matlab script was written and the

graphics handling was tweaked for real-time performance (Figure 7.3 on Page 130 was created using this tool). The implementation showed that the use of actions is a good solution, but also the drawbacks mentioned in Section 7.3 were experienced. To avoid unnecessary details, we will now describe how the action concept can be conveniently handled in an object-oriented framework.

The client side communicates with the server that receives and executes the down-loaded actions. Under such circumstances, the supervisory control system running on a host workstation sends down actions to the server that executes the actions, thus computing the variables that are to be supervised. The software can be divided into the following five parts.

1. The client's compilation and linking strategy according to Section 7.3.
2. The transfer of the action from the client to the server. This part is standard computer communication and is therefore not further described.
3. The embedded software implementing allocation, initialization, storage, call, and deallocation of actions. This part corresponds to a generic holder of actions.
4. An interface part defining the environment (data and operations) that is available for actions. This defines the actual openness of the control object.
5. The specific application or control code executing the actions. Except for the initialization part, the control source code does not need to reflect that actions are allowed.

This means that we want to do an implementation for the embedded system of items 3, 4, and 5. An object oriented design and implementation using the C++ language was found to be well suited for this.

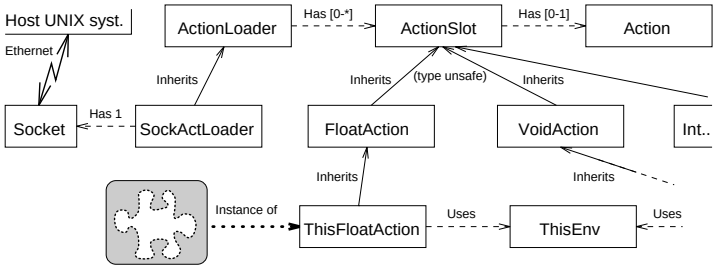

Figure 7.2 Classes for simple use of actions. The (type unsafe) inheritance from `ActionSlot` is part of the system, whereas the resulting action slot (`ThisFloatAction` in this case) is strongly typed.

Figure 7.2 shows our object design. The class ActionLoader is independent of the way actions are transferred to the computer. This is instead encapsulated in the SockActLoader which uses some communication class, denoted Socket in the figure. The ActionSlot is a holder of a generic action. A number of type-safe classes derived from ActionSlot were written, one for each built-in type. These derived classes provide inline type-conversion operators for convenient use of actions in algorithms. For instance, the FloatAction contains an operator `float`.

Shared memory systems

Motion control systems often use multiple processors on a common bus. In less demanding applications when only one CPU working in one address space is used, a fixed linking and use of function pointers can provide the required flexibility. When using multiple CPUs that share memory via a bus, like the VME bus in Figure 5.2, we have to cope with different address spaces. Assume this is the case in our simple control example, and we want to use dynamically linked actions.

Using dynamically linked actions in a stand-alone system, as motion control systems often are, may seem to require the linker to be ported to the target architecture. Note, however, that the required functionality is very limited. It does not need to depend on file systems and operating systems, command line decoding can be omitted, only a few options are needed, etc. Hence, such a linker is easy to write.

Source code for the simple control application has been presented in Section 7.3. The `&env` in the control law `P=K(&env)*(r-y);` is possible to omit by using overloaded operators in C++. In our case it was done by declaring `K` as a FloatAction (see Figure 7.2), which in C++ contains `public: inline operator float();`. This means that the action can be used wherever a float can be used, and the control law can be written as `P=K*(r-y);`, exactly as in C without actions. Standard tools for generation of control code can therefore still be used.

More advanced control problems can be solved in a similar manner. We can then keep the control law without any changes due to the use of actions. A perhaps better solution is to change our view of the elements of the feedback vector from being constants that are multiplied by the corresponding state, to being functions that take a state variable as an argument and by default returns the product of the state and the gain factor. The nonlinear pendulum control can then be achieved by using the following feedback gain functions:

$$\begin{aligned} l_1(x) &= a\left[\sin(x_1+\delta)-\sin\delta\right]/c + k_1 x_1 \\ l_2(x) &= k_2 x_2 \end{aligned}$$

where k_1 and k_2 could get their values from a standard parameter interface, or whatever was used for the constant L vector values. An even more flexible solution would be to let a feedback vector L be a function of the state vector and the time, i.e. $L(x,t) = ...$ That would for instance allow a non-stationary LQ-controller [29] to be used.

Dedicated hardware systems

The following implementation of the low-level sensing illustrates the use of actions for special purpose hardware. As shown in Figure 5.2, an accelerometer connected via an AD-converter to a DSP is available. For brevity, we will only use a simple action detecting an acceleration level. The following design tries to show how actions can be supported with a minimum of support from the embedded sensor software.

To reflect typical properties of intelligent sensors, our hardware was used as follows: 1) There is a memory area (called `action_storage`) reserved for loaded actions. 2) The `action_storage` is accessed by using the built-in DMA capabilities of DSP32C. This means that no support from the DSP program is needed; access of `action_storage` from outside only imposes some cycle stealing controlled by the hardware. 3) The address and the length of `action_storage` must be known by the action client. In our setup, we simply get this information from the symbol table of the linked sensor code. 4) The loaded action must obey any restrictions on the use of resources like CPU-time and memory. We used the DSP simulator (on the host computer) to check the execution time and use of memory.

We will now use function-based actions to catch collisions and plot the recorded signal in a host computer tool. Since they do not require any addresses of internal symbols (recall that the pointer to the environment is passed as a function argument at run-time), and the management of the `action_storage` is done by the master, the following C++ code is sufficient. First the header file included by both the action client (the master) and the action server (the sensor unit):

```
struct Environment {
  struct Signals {
    float time, acc;
  } *this_sample;
  enum status {pending, recording, transferring, waiting};
  status trig_state;
};

typedef int (*PlotAction)(Environment*);
```

As C-programmers know, the final `typedef`-line specifies that `PlotAction`

is a pointer to a function that as its argument takes a pointer to the `Environment` specified above, and returns an integer. The basic sensor software then contains

```
int DefaultAction( register Environment *env ) {
  // Default behavior implemented here.
}

long action_storage[0x400];  // Size hard-coded for brevity.
PlotAction plot_action = DefaultAction;
```

The return value of the action tells whether sensor data output should be supplied or not. Our built-in data-supply object of type `AccPlot` contains the following member function:

```
int AccPlot::PutSample( float signals[] ) {
  // <Checking conditions that has to be fulfilled..>
  // <..16 lines omitted here..>
      // Space available, check trig condition.
      environment.this_sample = (Environment::Signals*)signals;
      if ((*plot_action)(&environment)) { // <- CALL OF ACTION.
        // Put data and let HW convert to IEEE output format.
        // <Some work done here.>
        return 1;
      } else {  // <Seven final lines omitted ..>
```

where the line containing `(*plot_action)(&environment)` shows how the action is called. As the reader with experience from the C language knows, this statement means that the function pointed to by `plot_action` is called, passing a pointer to the environment data structure as an argument. The following function was written and compiled, and then position independence, execution time, and memory usage was checked. This was done on the host workstation.

```
int AccPlotAction( register Environment *env ) {
  int ilevel = 25; float level;
  switch (env->trig_state) {
    case Environment::pending:
      level = float(ilevel);
      if (env->this_sample->acc > level) {
        env->trig_state = Environment::recording;
      } else { return 0; };
    case Environment::recording:  return 1;
    default:                      return 0;
  };
};
```

The threshold value, which in this example is fixed to $25 m/s^2$, should in a real case get its value from some kind of parameter interface. The simple > test on the level can of course be replaced by more complex conditions, using for example filtering functions (provided via the `Environment`) from the application library of the DSP. The size of the compiled code for the `AccPlotAction` function is 212 bytes, and worst case execution time is $3.4 \mu s$. It was loaded into the DSP memory using DMA, and it was activated by assigning the address of the loaded action to the `plot_action` variable (done as one atomic operation using the DMA facility). With a sampling rate of 8 kHz, a recording of the transient when a slow downward robot motion touches a cover plate made of metal was requested. For proper assembly, there should be a gasket applied on the plate. The intelligent sensor catches the contact transient and sends the captured data to the host computer. The rigidity/compliance of the surface affects the response quite significantly as shown in Figure 7.3. It is therefore a trivial task for the high level control to decide if the gasket is missing or not.

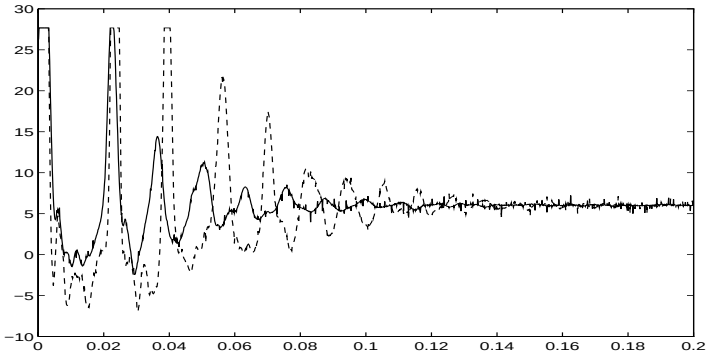

Figure 7.3 Acceleration $[m/s^2]$ versus time $[s]$ after contact event. The solid line shows the desired response with the gasket applied, and the dashed line shows the metal contact when the gasket is missing.

There are, however, also problems with the above solution. As the observant reader has noticed, the threshold level in the `AccPlotAction` function is first given an integer value 25 which is then converted to a float. It would of course be better to declare floating point values directly, but the standard DSP32C C-compiler does then not produce position independent code. This is typical for RISC architectures, which uses a fixed instruction length. In this case, the instruction length is 32 bits. That is also the length of a `float` (and also of a `long`), whereas an integer occupies 24 bits. The C-compiler therefore includes the constant

integer values in the instructions in the code-section. A `float`, on the other hand, will be put in the data section, thus requiring linkage to the executable code. The same problem occurs, as mentioned in Section 7.3, when auxiliary functions are included in the action.

Even if the problem is due to the compiler, it is still relevant since compilers generally work in this way, and actions are preferably written in a high-level language. Special languages and/or compilers are sometimes used for safety critical systems to get deterministic execution properties. Such a special compiler could provide position independence also for the above function. It is also easy to achieve that using assembly language. Despite the problems, we therefore think that function-based actions can be a useful alternative in some applications. But for portability and execution efficiency reasons, the dynamic linking version is probably the best alternative in most cases.

7.5 Safety and predictable real-time performance

The proposed mechanism is a powerful way to add flexibility to embedded systems, but how are safety and predictability affected? The following investigation is made with some typical properties of hard real-time systems in mind. The reader may think of the Spring Kernel [194] and the Synchronous approach [187] as relevant examples.

Since we have the same requirements on safe programming of actions as for implementation of the standard part of the system, some way to ensure that safety has to be devised. If a language as C is used to implement the actions, the system has to protect itself from invalid memory accesses that otherwise could compromise the entire system. Proper programming of the MMU can prevent illegal memory accesses but that is an expensive solution, especially concerning worst case execution time for dynamically linked actions. A more convenient (for the programmer) and efficient (for the computer) approach is to use a language (like Java [141]) where memory accesses can be checked a priori.

To ensure that deadlines in the system are met, the timing of individual actions is of paramount interest. If the system is statically scheduled, the schedule has to take into account the extra time actions are allowed to take. The timing constraints is then a property of the action slot, and the constraints have to checked when actions are installed. In dynamically scheduled systems, rescheduling has to be done when actions are loaded. In both cases, the execution time of the action has to be determined.

In some cases, as in our DSP system, execution times can be determined by the client, and the action loader can check that timing con-

straints are fulfilled. Thus, no additional functionality/software for timing analysis is needed on the target side. In other more general cases, for instance when the action may be down-loaded to differently configured hardware (but with compatible instruction sets), it is more suitable to perform timing evaluation of the action in the target system. To determine the actual timing of the action, it either has to be statically analyzable like in Real-Time Java [141], or the timing has to be checked at runtime by measuring the actual time taken for some appropriately chosen test runs of the action. Assume that each action contains a function that calls the loaded function with special parameters to get the maximum execution time. It is not appropriate to just let the `install` function (see the code example on Page 124) call the `MaxTime` function and then compare with the bounds by calling some evaluation function defined in the action's environment. This is because timing analysis may also require the hardware (interrupts, caches, etc.) to be setup in the same way as when the action has been installed. Therefore, we propose that action slots in statically scheduled systems require actions to have a `MaxTime` entry (and `MinTime`, etc.), and that the server calls it before the `install` function. This is a minor and straightforward extension of our present implementation. Of course, the program must have a properly scheduled time slot for the timing evaluation, but that slot can serve all actions managed by that CPU.

When new code has been properly installed, it can be useful to monitor signals emanating from the the action to make sure that it fulfills predefined control constraints. That was proposed in the Simplex Architecture [179], which has been developed to support online upgrade of hardware and software components. The action mechanism facilitates also such a feature; actions can of course be removed (fully implemented but description omitted) when conditions of activation are not fulfilled.

In conclusion, we claim that the proposed mechanism is very useful both for improving flexibility to existing hard real-time systems, and for development of new open embedded systems.

7.6 Conclusions

A development towards more open computer systems is motivated by increased demands on flexibility, i.e., that the system should be possible to tailor to meet new customer demands. The need for open systems also within mechatronics and embedded control was illustrated by some application examples. To permit feedback control based on information from external sensors, and for the design of reactive systems in general,

open control systems are needed. Special demands on efficiency, safety, etc. motivated use of dedicated software techniques for this type of systems.

The most basic way to allow changes is by providing parameters that the user can tune for his/her purposes. Such parameters have been restricted to data, but the demands on flexibility often require the program to be changed. Our idea is to use pieces of compiled executable code as functional operators. Such operators may in the simplest case be a linear function, which combined with operators provided by the programming language may appear as an ordinary numerical control parameter. In more complex situations, even if encountered after implementation of the basic system, we may introduce a function with additional arguments (states, time, etc.), or even a 'function' with internal states. A new parameterization of the control is then obtained. (Our implementation even supports parameterization of the parameterization, that is, action parameters supplied at plug-in-time defined the actual parameterization used during run-time.) In an object-oriented framework, this means that new methods can be added to controller objects after implementation of the basic control, and even while the controller is running. Two ways to do the required dynamic binding of functions and objects were introduced and discussed.

A major effort was put into full implementation of the test cases. The results were encouraging; it was possible to achieve flexibility and efficiency, and to combine open and closed system designs. In other words, an embedded may indeed be open, thus justifying the title of this chapter. The application examples, the proposed principles for open embedded control, and the implementations verifying the performance are the main contributions.

In conclusion, we propose a way to build **open and efficient** embedded systems using industrially accepted software techniques. It was also described how other research results, concerning for example predictability and static scheduling [194, 195], can benefit from the proposed mechanism. We think that a major advantage of the concept is its simplicity (concerning programming and operating system requirements) and the possibility to combine it with other solutions within mechatronics and real-time systems. Since flexibility is likely to be even more important in the future, for example in manufacturing systems, we hope that the proposed software technique will contribute to that.

8

Applications

We are now ready to further investigate special applications that are non-trivial to handle with current systems. A short introduction to such applications was given in Section 2.5 (p. 10), where one industrial and one benchmark problem were presented. This chapter contains a further treatment of those examples, and more industrial examples are also presented. These application examples are today, in many 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.

The approach taken here is as explained previously; the robot control systems should be open for the experienced user on a fairly low level tightly connected to the motion control system. Actions, as defined in Chapter 7, will be used to accomplish such a tight connection. The term *robot skill* will be used as a short term denoting actions loaded into the application layer for the purpose of extending the capabilities of the robot. The experimental platform presented in Chapter 5 is instrumental to obtain the results reported.

The first application example, which is deburring of castings, illustrates the aspects of open robot control very well. The second example treats high performance motion control for spot-welding. There is one example about object identification for materials handling. Two examples in the section about assembly, describe how creative use of the application layer can improve the performance of assembly operations without additional sensors or hardware. The final industrial example is about arc welding, focusing on support for special control of the welding process, and on the use of such control from a task-level programming system. The ongoing implementation of the pendulum benchmark problem is then presented. Finally, we will look at some related industrial development in recent years and put the development of application specific control into an industrial context.

8.1 Deburring of castings

As already pointed out on Page 10, we want deburring to 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. By appropriate tuning of speed etc., the tool will make the surface smooth. In the ideal situation when the variation in burr size is small, and when the grinding path is well known, successful grinding has been achieved [205, 85, 103, 160, 137]. Even in the case of more unknown burr sizes, successful experiments have been made using special tooling and external control [159]. The special tooling includes passive compliance in well defined directions relative to the grinding forces, and the tools is equipped with sensors for accurate position and force measurements. Such very special end-effectors [161] are needed because the robot controller cannot be tailored to accomplish the desired grinding control.

The situation is even more difficult in the general case when the location of the work-piece is not accurately known, and when remaining bulges remain due to exceptional burr sizes. This is the case that has to be solved before robotized grinding gets industrially useful. Major robot vendors are aware that current systems cannot handle this type of application, and many potential customers have therefore not been offered any technical solution [33]. Thus, robots capable of handle realistic deburring applications are industrially very tractable.

Even if the long-term goal of this research is to develop a system for real industrial use, such a complete implementation is beyond the scope of this thesis. Instead, let us look at unsolved specific problems. As mentioned above, deburring along well known paths with small variations in burr size has been accomplished in other laboratories by well designed end-effectors and well tuned feedback control. The major unsolved difficulty is to make the robot recognize when and where additional grinding is required and, if so, to determine additional grinding motions, like the ones a human worker would have performed.

Desired behavior

Considering our example from Chapter 2 (Figure 2.5, p. 2.5), we have the profiles according to Figure 8.1. Note that the following solution will not require any additional sensors to measure the deviation from the nominal path. Instead, that can be computed from the position errors during deburring. Such (internal control) data are normally not possible to use in robot programs. Here, it is possible due to the open control architecture as described in earlier chapters.

A simple grinding strategy to handle exceptional burr sizes would be to apply a grinding force that is proportional to the deviation from the nominal (i.e., programmed) path, or to regulate the grinding speed along the path. However, to avoid work-piece-burns [137] such control can only be used to a limited extent. A more general solution is to:

1. Perform some initial grinding to estimate the actual location of the work-piece [68] and adjust the nominal path to get a new desired path. That compensates for misalignments of the work-piece. Thereby we permit larger tolerances for work-pieces and fixtures without requiring special fixtures (which easily cost around 20000USD [51]). The adjusted path is the Desired profile in Figure 8.1.
2. The desired profile will define the set-points to the motion control. During the first grinding the Cartesian position error in the force-controlled direction is to be computed based on the joint position errors. That gives the difference between the After first grinding and the Desired Profile in Figure 8.1. When this difference exceeds a certain threshold, a remaining bulge has been detected.

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 8.1. The detection and recording of the bulge is to be implemented in an action (as defined in Chapter 7) by the deburring expert programmer, possibly after measurements from human demonstration of the task [182]. 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. Note that such

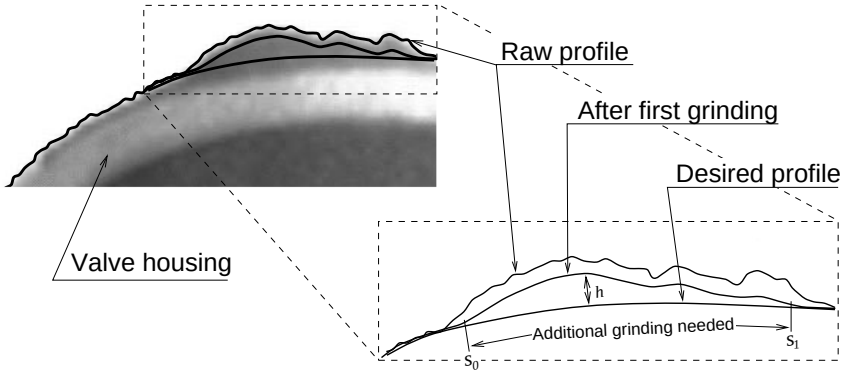

Figure 8.1 Part of work-piece and contours for the deburring example.

computation in the application layer 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 by moving the tool back and forth over the bulge (like a human worker would have done). Grinding can then be resumed at s_2.

Interruption of the normal grinding to perform the additionally required grinding means that a minimum of time is wasted on moving the grinder (or the work-piece in the case that is held by the robot) back to remaining bulges. Path planning problems (obstacle avoidance, avoidance of singularities, and proper selection of arm configurations) are thereby avoided. Another alternative that also avoids the path planning problem is to first complete the first grinding, and then (using measurements from the first run) compute an additional grinding program with iterated (and possibly increased) grinding forces at the remains. Successive runs can then be made until the desired profile is achieved. This alternative solution, using a standard robot systems and an advanced grinding tool which also performs the sensing, has recently been implemented by Persoons and Van Brussel in Leuven, Belgium [160]. The solution suggested here[5], utilizing the adequate support offered by our proposed software organization, should be more productive because less external hardware is needed and the required feedback is accomplished more locally in the system.

Status of implementation

Our treatment of the deburring application has so far resulted in a description of the problem, an analysis of possible approaches, review of available research results, and a description of a desired behavior that we want to implement. Research efforts towards a full implementation are suggested to be done in the following stages:

1. Attainment of knowledge about the deburring process, and implementation of preliminary grinding. This includes design of the deburring control loop.
2. Integration of the grinding control and the robot motion control is desirable to obtain (cost and performance) efficiency. It must be investigated if application specific requirements concerning data flows and programmability are fulfilled.

---

5 The proposed solution was presented internationally in 1992 [150]. That paper (and [144]) also includes some source code and further implementation details that have been omitted here for brevity.

3. Incorporation of the grinding control (according to item 1) in the robot control system (according to item 2). Improvements of the control and end-effector design may be necessary to obtain full experimental grinding.

Item 1 has, as described above, been solved in other laboratories, giving valuable know-how about the deburring application. Here, with the limitations of current industrial systems in mind, we want to investigate if our software architecture supports implementation of the deburring specific control. This is Item 2 which will be treated below.

Item 3 requires grinders and fixtures that are currently not available within our laboratory. An important subproblem that, on the other hand, is being pursued is integrated position/force control. In particular, the transient behavior when the grinder gets in contact with the work-piece is important. The experimental platform therefore includes a force sensor for an ongoing implementation based on the impedance approach [100]. The developed interface for experimental control of ABB robots is also being used for hybrid position/force control at the University of Coimbra, Portugal [162]. To examine the applicability of the ORC architecture, without having the grinding tool and force control available, we found the following contour tracking experiment to be relevant.

The gripper mounted on the Irb-6 in our laboratory is equipped with a laser distance sensor [157]. The sensor emits a laser beam and senses the reflection. The distance is obtained (internally in the sensor) by triangularization. The accuracy is better than 0.1 mm. The sensor output is connected to the control computer via an analog interface (marked Laser IO in Figure 5.1, p. 70). Instead of defining a `GRINDMOVE` primitive, we will implement a `TRACKMOVE` primitive. This will of course be much simpler concerning the force control and its interaction with the grinding process, but the characteristics of the compensation of possible misalignments and the handling of exceptional bulges can be tested. Therefore, the test is relevant for evaluation of the proposed ORC architecture.

Configuration of software layers The ORC programming layers were configured/utilized as follows:

- The top layers for task-level, off-line, and on-line programming were omitted for simplicity since they are not subject to evaluation here (that was done in Chapter 3). This means that the robot was programmed very much like a computer.
- The executive layer was configured for robot programming in Modula-2. This was straight forward using the action concept proposed in Chapter 7.

- The application layer was extended with a module/package for the contour following. This was also implemented in Modula-2 using actions. It was done in such a way that the trackmove primitive could be installed and removed while the robot was performing (but of course not when the primitive was in use). For safety reasons (the laser beam is harmful for the eyes), the orientation of the sensor was fixed and only vertical (Z) on-line adjustments were allowed.
- The motion control was done joint-wise, and no external joints were supported. The motion control layer was therefore not used, and the arm control layer only included an arm-specific trajectory generator. Whereas the basic motion control and the application specific control was done in the same (M68040) CPU and in the same address space, callback procedures were used for simplicity instead of full actions to pass the contour following algorithm to the arm control.

Programming To admit both robot programs (i.e., the task description) and skills (i.e., the tracking behavior) to be changed at run-time, skills loaded into the application layer are registered in a table (internally in the application layer). Each slot in the table includes a pointer to the "`Symbols` function" (corresponding to the `SendSymbols` function in the example on Page 123) of the skill. That function returns the symbols (functions and data) that can be accessed from the executive layer. This is used when the robot controller is tailored to this specific application, which was done in the following steps [16]:

1. The robot skill was written and compiled referring to static information (modules or header-files describing types and functions) on the host computer. Type checking and generation of time stamps for the compiled code is done at this stage.
2. The contour tracking skill was installed as an action in the application layer. This includes checking that only allowed symbols are referred to by the compiled code, and the time stamps are used to check that all interfaces (used by the action) within the embedded system were defined before the action was compiled. We made the socket interface to the network available to robot skills, admitting contact with engineering tools on the host computer to be established at run-time.

   The installed skill (dynamically) exports the symbols that should be possible to access from the user-program. This is done by calling a function in the built-in part of the application layer. Thereby, the `Symbols` function of the loaded skill is registered in the table mentioned above. The call performing this registration is done from

the dynamically loaded code but in the context of the DynamicLinker server process (visualized in Figure 6.9 where actions defined from SimuLink were loaded) not to disturb the execution of ongoing control.

3. The robot program, including the task description (path coordinated etc.), was compiled in the same way as in item 1. The interface to the skill is statically available, and assumed to be available also in the embedded system.
4. The robot program was loaded as an action into the robot controller, similar to item 2. The only special extension is references to symbols that are defined by dynamically loaded skills. Those symbols are retrieved by the executive layer (from the application layer) by calling a generic function that calls every `Symbols`-function that have been registered in the table mentioned above. The assumption from the previous item can therefore be checked. If not fulfilled, the loading of the robot program is aborted. If the robot program is accepted, the execution of it started as a new software process. An execution control interface with a Matlab-based graphical front-end was also implemented [16], but for brevity that is not described here.

Thus, both type checking and time-stamp checking (detecting references to old interfaces) of dynamically loaded robot skills were done. Furthermore, well designed software modules and 'makefiles' take care of the internal complexity; the user only needs to know a few simple commands.

Experiences and ongoing work Several practical difficulties were encountered during the implementation and test. This had to do with an unfortunate combination of sensor and work-piece. (The problems were **not** due to the hardware or software organization.)

One difficulty encountered was to program the circular motion so that the laser always senses the top of the 1-2 mm wide burr. The laser beam is between 0.45 mm and 0.9 mm wide depending on distance and orientation, and it is not visible which makes the programming hard. This difficulty would not be present if a grinder were used. This problem was solved by writing a program that made the robot move twice (in different direction) over the casting with the sensor activated. The sensor data and end-effector coordinates were captured via the software installed in the application layer, and the data were sent to Matlab. Then we wrote a Matlab script which read the data from the robot and computed the location of the work-piece. That is, the desired edge was assumed to be circular and the burrs were assumed to be perpendicular to the desired edge, and the coordinates for the circle were computed. Those coordinates

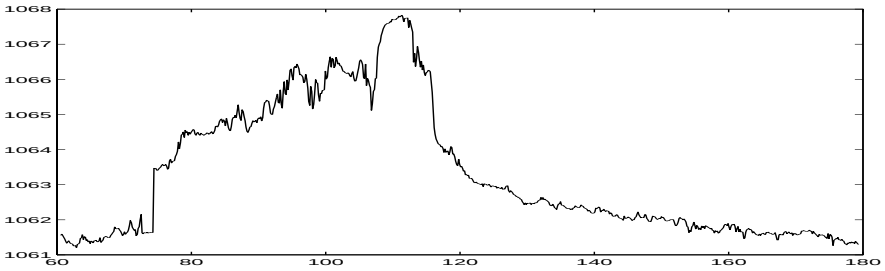

Figure 8.2 Part of casting edge as detected by the robot using a laser sensor. The plotted height (in mm above the ground) versus path coordinate (in mm along the path from start) catches the main part of the bulge. The deficient quality is due to irregular reflection of the laser beam on ragged edge.

were then transferred to the robot controller and used for the tracking motion.

Data were logged also during the final run around the edge of the casting. The contour data used to detect the exceptional bulge was sensed and sent to Matlab for documentation purposes, see Figure 8.2. Two additional problems were then observed. First, the ragged edge of burr did not reflect the laser beam as required. Secondly, the top of the largest burr was leaning outside the assumed circle. The latter of these problems could of course easily be fixed, but being disappointed concerning the laser sensor, we did not think it was worthwhile. Instead, we decided to try compliant control and let the robot be in contact with the work-piece. At the moment of writing, the practical arrangements are going on. The contour detection and computation will be the same as in the contour following (and in the deburring) case. This key feature was accomplished in the following way.

The action (the callback routine in this case) passed from the loaded skill to the arm control layer was allowed to modify the derivative of the path coordinate (always computed by our trajectory generator) during the motion. Changing the sign of this derivative therefore results in a motion backwards along the path. The desired behavior for additional grinding motions can thereby simply be computed in terms of a trajectories for additional motions as functions of the grinding path derivative. In conclusion, programming at different levels could be done conveniently in the proposed architecture, but practical issues remain to be solved.

8.2 Spot welding

Spot welding is a task well suited for industrial robots. As the application has become mature, the performance demands have become clearer. Competition and benchmark tests (made by major car manufacturer) are further reasons. Customers make performance evaluations because 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.

The need to tightly connect the motion control system with the timing of the control of the welding equipment, as described below, became clear to the author already around 1984 at ABB Robotics. An important car manufacturer (SAAB) required that motions between weld-spots may not take more than 0.4 seconds, including the time needed to control the weld-gun and the time to start and stop the welding. At that time we had to extend the built-in motion control system with special features.

Such an experience naturally inspires a more structured approach, as was proposed by the author in 1992 [144, 150]. The SpotWare package [7] developed since then within ABB Robotics very well agrees with the proposed principles. Technically, the new S4 system fulfills the requirements for this application; the added application-specific control can subscribe on time events from the built-in trajectory generator (as specified in the RRS interface [169]).

The problem

Consider welding of some part of a car. The spots to weld are typically equally spaced and placed in a row along the edges of the sheet-metal parts that are to be joined. A typical distance between the weld-spots is around 50 mm, and a typical time for the motion is (today) around 0.3 seconds, which still is significant compared to the welding time.

Assume that welding of one spot is almost completed. Upon completion, the robot should, as quickly as possible, open the weld-gun, move to the next spot, and start welding. Let us call this new location `spotx`. Hence, the statement `WELDMOVE WeldGun TO spotx` is to be executed. The `WELDMOVE` statement on the user level will be interpreted and a compiled function will be called. What are the requirements on this function?

We may call the function `WeldMove`. It has to be built on existing motion primitives. It needs to use information from the basic servo loops, to handle signals from the tool, and to include other types of 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 solution

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. Motion control optimized for this type of short motions may further improve performance. Assume that there is a `ShortMove` function available in the arm control layer as described on Page 93. We will then make use of this function whenever the motion to a new welding position is "short" (in joint space).

Figure 8.3 Timing for stop of motion and start of welding.

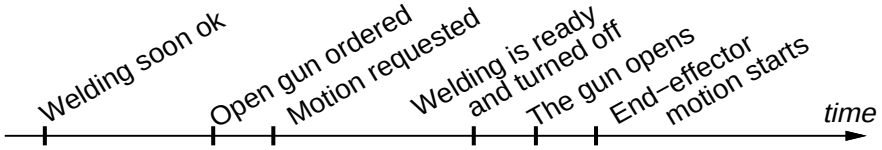

Figure 8.4 Timing for stop of welding and start of motion.

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 [167]. Since some time passes, after the signal has been set, until the weld-gun is almost closed (that is, until the motion has to be completed), it is

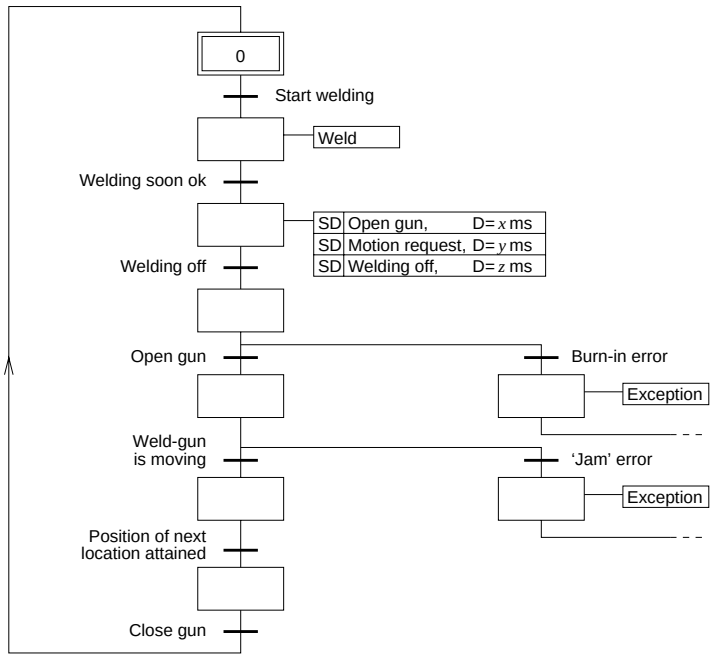

Figure 8.5 Grafcet [63] description of timing for high performance spot-welding.

desirable to signal "close-and-weld" some specified time before the motion is completed, as shown in Figure 8.3.

Similar methods can also be applied at the start of the motion, where it is required that the welding equipment generates a "welding OK" signal a certain specified time before the welding is estimated to be complete [167]. (Welding is event driven and depends on the welding current and other signals monitored by the welding equipment.) In Figure 8.4 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. Omitting the timing in Figure 8.3 for brevity, Figure 8.5 further explains the sequencing and timing that must be handled by the control software [86].

8.3 Materials handling

Robots used in so called pick-and-place applications often need to identify the presence of objects or what kind of object that is present. Also other applications may include sorting of pieces, thus also requiring this functionality. In some cases beam sensors can be used [52]. A more general but costly solution is to use vision systems. Another promising alternative is ultrasonic sensors [189]. In some cases, however, there are reasons to believe that external sensors can be avoided by utilizing internal control information.

Desired behavior

Estimation of dynamic properties using measured data is a common and important technique within automatic control. Several algorithms and methods have been developed for this purpose [99]. Such system identification techniques have also been used to estimate robot arm inertias for control design purposes. It is then natural to ask: Why not use system identification also as a way to avoid external sensors. This, of course, applies to the case when the motions to grasp the objects do not depend on the type of object. That may be the case when the objects are of similar shape, when they only differ in weight due to different materials, or when we want to check if an object of fixed type has been grasped or not. The problem today is that signals required for system/object identification, are not possible to access from any user level of the system.

Thus, the desired behavior is that the robot after grasping an object should be able to identify the object, if any, only using internal (built-in sensors). This feature should of course be well encapsulated and easy to use for the ordinary robot programmer, whereas the algorithms should be implemented by the control engineer in the application layer. The end-user program may contain the following lines:

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

where `IDENTMOVE` and `IDENT` are application specific features of the system. The parameters `pars` for the system identification algorithms include data like length and magnitude of the excitation signal. 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[]`.

Prototype implementations

The proposed principle was first presented in [144]. A full implementation, proving the applicability of the concept, has more recently been carried out by others [138]. In our laboratory, system identification experiments for control purposes have been carried out [129] using the developed experimental platform. An excitation and data logging tool (developed within our laboratory), with a Matlab-based graphical front-end, was used for these experiments. The tool was, however, connected directly between the motion control and the engineering workstation, and not to the user programming layers. On the other hand, successful uses of actions and experience from the system identification experiments show that excitation and data logging can be conveniently controlled from outside the motion control system.

Object identification controlled from the user level has been implemented also in our laboratory. This was done as a two weeks project by four students (with guidance from the author) within a course in adaptive control [218]. The Irb-6 robot was used, including its original analog speed control (PI-mode selected, see Figure 5.1 on p. 70). The accuracy of the inertia estimation was not very good due to unmodeled dynamics (within the harmonic-drive gear boxes) and due to interference with the analog speed control, but the robot was able to sort pieces depending on material (steel or aluminum). The added instructions were made compatible with the old ABB programming language ARLA [4]. The syntax and semantics of the instructions exposed to the end-user were therefore much simpler than proposed above.

Another prototype experiment, illustrating that intelligent external sensors can be used for 'quick' object identification, was described on Page 130. With experiences from these three experiments, it is very likely that the proposed architecture facilitates the employment of system identification as a means to avoid (unnecessary) sensor hardware for object identification, and also that necessary sensors can be efficiently used.

8.4 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

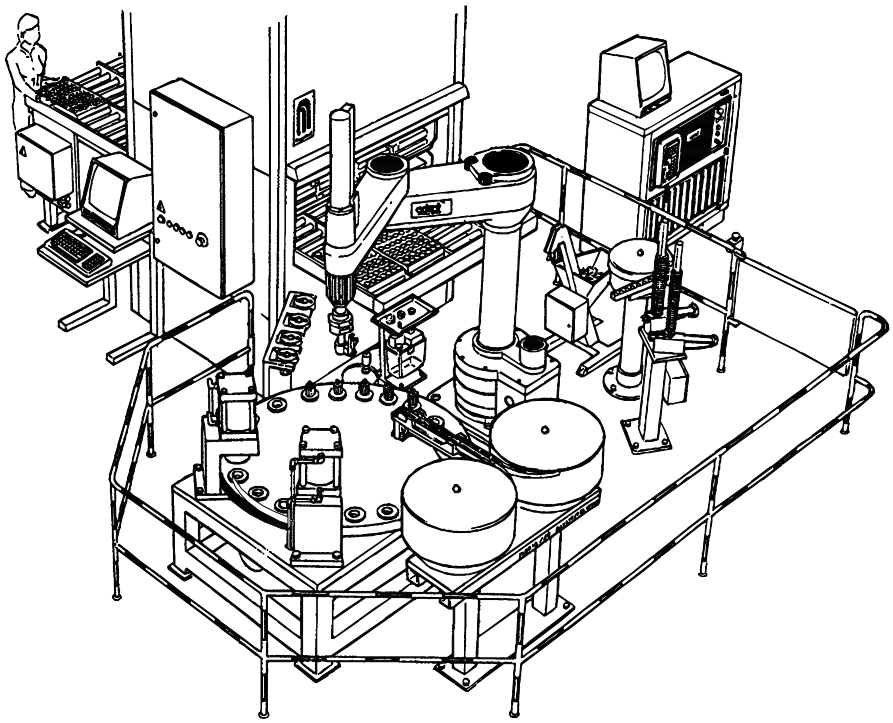

Figure 8.6 Example of an assembly work cell, from [74] with permission from IVF.

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), whereas the next one 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 fixed servo tuning). 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 8.6. Motions among stations in the work cell are performed at maximum speed. Before a position can be reached

accurately, the robot either has to make a smooth deceleration, or make a fast deceleration and, then, a slow approaching motion. Dynamic effects from the high speed motion will otherwise make the accuracy deficient. Today, it is not practically possible to have the servo so accurately tuned that a fast motion can end directly at the desired final point. Such a tuning would be dependent on pay-load, actuator and gear temperature (which in turn depend on the task, time, ambient temp.), etc.

Control problems with time-varying unknown parameters are sometimes approached with adaptive control [102]. This means, however, that the servo parameters will vary during motion, and depend on recent motions. Hence, adaptive control for industrial robots requires special care, as repetitive accuracy is very crucial. Another approach is to apply auto tuning [102] 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 is idle, waiting for new work pieces. Request for such tuning in the end-user program may look like the following few lines:

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

where the `AUTOTUNE` primitive should be implemented in the application layer (by the control engineer). The computed control parameters `BoltTuning` may 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 control 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 “time optimal”, i.e., torque demanding, motions sometimes need energy saving modifications. The nominal torque demanding motion can be developed with formal methods [42], 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 typically of on-off type. In our experimental Irb-2000 system we can access the output of these sensors, but that is normally not possible. Furthermore, for safety reasons, the robot is emergency stopped when overheat is detected, so the outputs from the built-in sensors only supplies information when it is too late.

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. Experience from the ABB S2 system, which uses analog simulation built into the drives to simulate the motor temperature, shows that this can be quite accurate if the ambient temperature is known. The dynamic model 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. 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. If the built-in motion control does not contain motor temperature estimation, it should be possible to add from the application layer of the system.

Suggested approach

Our desire to apply explicitly requested auto-tuning to improve peak performance is a straight forward extension of the principles used in the materials handling application. Specifically:

- The same control signals are needed. Access to those signals was shown, in previous application, not to be any problem.
- Computation of control parameters from measured data is a pure algorithmic problem. Deficient availability of CPU resources and servo control parameters may impose some restrictions, but implementation only has to deal with algorithms within the application layer. Thus, there are no restrictions imposed by the proposed architecture.
- Features admitting the end-user to obtain and use several sets of tuned parameters can be implemented in several ways. One simple alternative has been successfully implemented [218].
- The built-in motion control must support on-line change of control parameters as specified on Page 93. This is standard, using our multi-layered parameter interface mentioned on Page 108. Such parameter changes are even possible in the Adept system [9].

The system requirements for the overall performance problem is a special case of the peak performance problem; only the torque signals are needed, estimation is a simple function, and the obtained information only needs to be propagated to higher levels. Therefore, the proposed layered architecture appears to be well suited for special application support as desired.

8.5 Arc welding

There are several needs for intermediate level programming within arc-welding applications. The following situations have been studied:

Basic functionality: A basic application package for arc welding includes control and programming features for waving motions and control of the welding equipment and interfaces to common types of welding equipments. The ArcWare package [5] developed within ABB Robotics is an example of well designed software for this purpose.

Path tracking: Available industrial systems most often also support weld-seam-tracking using a laser-scanner sensor. The sensor is then integrated with the motion control system, but this can so far only be done by the robot manufacturer (because available systems do not provide an open application layer). The path tracking control problem includes estimation of the weld seam based on (often very noisy) data from the laser scanner, and control of robot motion in such a way that the seam is tracked. The stochastic nature of the problem, and the desire to perform high-speed welding of thin-sheet metal without loosing track of the seam, motivates a minimum risk approach [78]. The path-tracking problem was also described by the author in [144].

Welding specific functionality: Arc-welding is a quite complex electrical, mechanical, and chemical process which can be influenced by voltages, currents, and weld-tool motions. For special materials, or for welding with special demands on qualities or productivity, special welding principles have been developed [11]. Sensing and control (of voltages, currents, weld-joint geometry, etc.), and a close interaction with the robot motion control are of key importance.

Task level programming: Even if on-line programming in this thesis is claimed to be superior concerning operator adjustments (Chapter 3), it is also realized that the initial programming of, for instance, advanced arc-welding programs are often better done off-line. That is because CAD data for the work-pieces can be used for the definition

of welding paths and end-effector orientations. The off-line programming system also provides a platform for task-level programming using knowledge-based techniques. So far this does not imply any need for intermediate level control, but our desire to use "hand-crafted agents" (as described in Chapter 4) does. Special welding techniques according to the previous item should be possible to load and refer to from the task-level system.

There are no known indications that the proposed control system design should not suit the needed basic functionality and path tracking, but no implementation has been done and these aspects are therefore omitted here. Furthermore, industrially available systems can handle these items. Instead, let us focus on the desired "Welding specific functionality" and on the "Task-level programming".

Thus, we want a system admitting incorporation of welding control at an intermediate level (here, in the application layer), and an on-line connection to a task-level programming system. The welding control should be possible to define and install from the task-level system, and it should be possible for motion commands from the task-level system to utilize the loaded welding skill.

Implementation status

Efforts to support advanced arc-welding was inspired by the welding experts within the next-door laboratory at the Department of Production Engineering. Implementations were done in close collaboration. This activity is ongoing. At the moment of writing, the status is the following.

Techniques for control of special welding has been developed and experimentally verified within the Department of Production engineering [11]. Because available systems are not open enough, such research adds

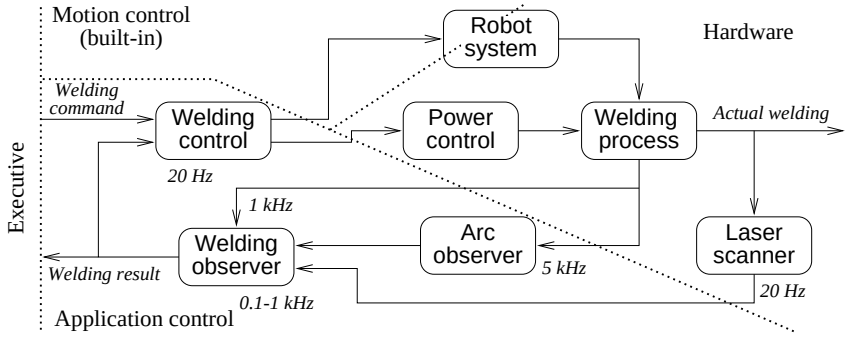

Figure 8.7 Block scheme interpretation of welding control according to [11].

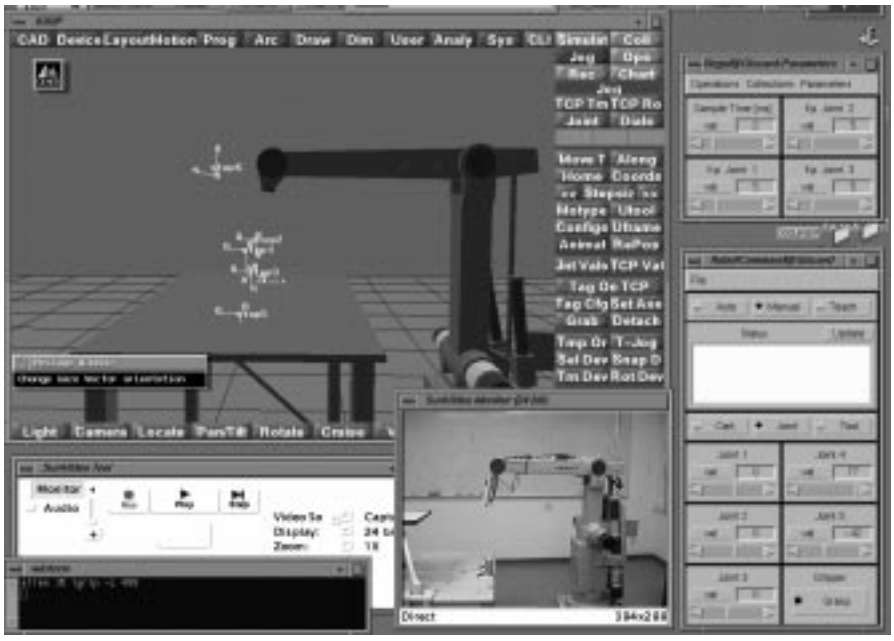

Figure 8.8 User interface to off-line robot simulation (at the Department of Production Engineering) with on-line connection to the physical robot (at the Department of Automatic Control). The robot (located 100m away) is observed via a video interface (available on a Sun close to the robot and connected back to the X-server on the SGI running IGRIP). The robot can also be manually controlled and tuned using our Matlab-based interface shown to the right (and mentioned on Page 108).

controllers external to the robot controller. The solutions have been studied to obtain requirements on open robot control systems. The feedback control nature of the welding control is visualized in Figure 8.7. Clearly, implementation of the welding control at the end-user level of the system would be very inefficient. (Many additional function calls and data transfers for each sample.) Instead, we want to implement this within the application layer. The observers may require additional hardware to be plugged in (on the VME-bus in our case), which is a standard thing to do, but principles for the application layer programming would be similar to what has been described above for the deburring control. Installation of a special welding controller will return an identification number to the host (actually the index in the skill table as described in the deburring application).

Implementation of task-level programming for arc-welding is being

done based on the IGRIP [65] off-line programming system. Results so far are presented in [48]. An on-line connection between the task-level system and the robot was needed. Using the features of the ORC architecture and our experimental robot control system, this was accomplished as follows:

- The SHLIB in IGRIP was extended with functions that during robot simulation transmitted the robot joint trajectories to the Hardware interface, see Figure 3.12 (p. 42). The actual hardware interface was located 100 meters away in our laboratory. The IGRIP part of the interface is based on UNIX sockets connected to Internet.
- A server running on a workstation in our laboratory was developed. This server connects the off-line system to the embedded control system whenever these clients are available and responding properly.
- All control engineering tools and operator interfaces running on our Sun workstations could be used directly also on the SGI workstation, as shown to the right in Figure 8.8. This is thanks to UNIX and the X-windows.
- An IGRIP server for the embedded controller was developed. This server is installed at run-time into the executive layer. The server accepts trajectories from IGRIP as robot commands. Call of embedded system primitives from the IGRIP system is accomplished by escape codes (negative numbers) in the time column of a supplied trajectory. One such code is used to refer to robot skills loaded into the application layer. (The identity of the skill is supplied as the second number of the trajectory sample.)
- Video cameras, a frame grabber in the Sun workstation, and the SunVideo software were used to remotely observe the robot motions.

Though Internet-based teleoperation of the robot is possible, it is hard to rely on dynamic feedback information transmitted through the network due to the often quite limited data rate (bandwidth). Actually, to speed up the video interface, a dedicated 10 Mbit/s connection was purchased. The simpler Irb-6 robot has been used for the initial tests, see Figure 8.8, but the Irb-2000 is to be used in the future. Experience from initial tests indicate that the ORC architecture and the experimental system form a very good platform for task-level programming.

8.6 The inverted pendulum benchmark problem

Motivated by our study of principles, robot skill has been used as a qualitative term in this chapter. It is of course desirable to also put

some kind of measure on the skill. Such a measure will, however, very much depend on the specific application. Furthermore, we primarily want to measure how well a robot control system supports incorporation of application specific control. For this purpose it is interesting to consider a well defined control problem capturing important properties of industrial applications. For this purpose, the inverted pendulum benchmark problem was suggested on Page 13.

For some defined magnitude of disturbances, and with the requirement that the pendulum control should be possible to add without changing the built-in motion control, the minimum length of a balanced pendulum will be a measure of the performance that application engineers can achieve. A system not permitting external loops to be closed tightly together with the built-in motion control will be less useful; sampling rates, response times, and high data flows will limit the performance (if the hardware has not been oversized).

Pendulum control has been carried out within our department [26]. Implementation of the control using the experimental Irb-2000 system is currently going on. We decided to build a pendulum that can be grasped by the robot using an ordinary gripper. At the moment of writing, manufacturing of both the pendulum and the gripper are about to be completed.

8.7 Industrial development

Most robot applications today are programmed on top of a fixed and closed motion control system, possibly using sensors to determine what motion to do (as described in Section 2.4). In recent years, however, there has been some industrial development towards open systems. This development will now be shortly described, and then we will put the development of application specific motion control into an industrial context.

Systems

Our approach to let the advanced user introduce application specific feedback control may, of course, be achieved by using a completely open system. But, for reasons explained in Chapter 4, let us use an industrially compatible approach here. Ideally, an industrially available system would admit us to introduce the required control in an efficient way. Two of the most promising systems used in industry today are available from ABB Robotics and from Adept Inc. These systems are programmed in RAPID [6] and in AIM/V+ [181, 10, 9] respectively.

The Adept system is flexible in several ways. The control system can be reconfigured for different mechanical manipulators, and parameters

for the built-in motion control can be changed. Using the built-in motion control, new motion primitives can be composed by programming in the V+ language, and new features can be given a nice graphical presentation to the end-user. The control structure is, however, fixed and control signals (like joint torque) are (as far as known from the manuals) not possible to access. Thus, the system is open but mainly on an executive level.

The ABB S4 system [2] is another so called open system. The software design of the system is object oriented, and the implementation is object based (that is, inheritance and polymorphism are not used). The basic design of that system was made independently of, and in parallel with, the development of the fundamental ideas in this work [144]. After 1992, there has been some exchange of information. It turned out that many basic ideas are in common, which makes the system an excellent platform for further extensions in the direction proposed here.

In conclusion, future support for advanced applications using industrially available systems appears to be feasible. Actually, implementation and packetizing of the spot-welding features (Section 8.2) has been done recently at ABB through in-house development. Still, the application problems presented in this chapter, and the approaches proposed to solve these problems, are contributions that deserve more attention within robotics research.

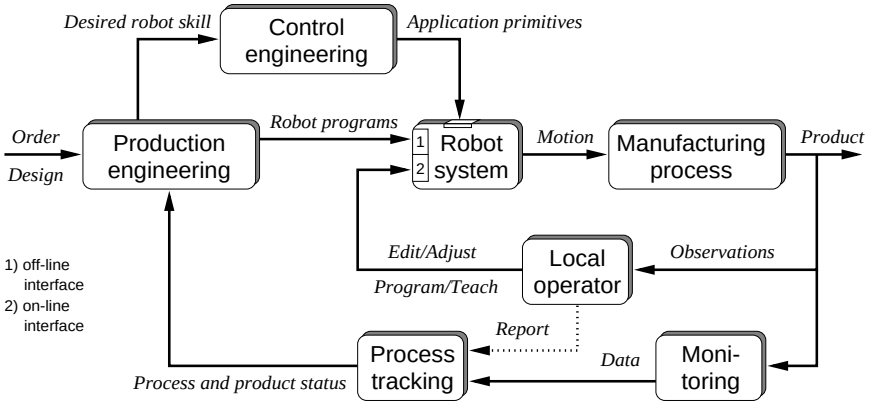

Figure 8.9 Flow of information/programs/operations in manufacturing control and robot programming. The development of application primitives is done to meet new control requirements on the robot system.

Industrial development of application primitives

Development and modification of robot programs were given a control interpretation is Section 3.4, where robot programs only utilized standard motion primitives in the control system. The application examples in this chapter, on the other hand, are all preferably solved by introducing control loops (or processing of control signals) in tight connection with the built-in motion control. How does this relate to the situation shown in Figure 3.6 (p. 30)?

Needs for new application specific control result from production engineering. Alternative solutions may be do develop special machines, or special end-effectors for standard robots. Development of special motion control of course requires a control engineering activity. Here, that results in control solutions that make the robot more skilled for the particular application or task. The approach has been to let the advanced user accomplish this by developing new *application primitives*. Figure 8.9 illustrates the development process, which now also includes control engineering. In the ORC architecture, it is the application layer (Figure 4.4, p. 57) that provides the programming interface for such application specific control.

8.8 Summary

Some case-studies of typical industrial applications have been carried out in this chapter. This was done with the focus on possible benefits of a close interaction between robot programming and robot motion control. From the results obtained, we can with some confidence conclude the following:

- Limitations in industrially available robot control systems today restrict the applicability of industrial robots.
- A control system may technically provide the features required to solve an application problem, like spot welding, but lack of well defined programming interfaces (user views) and supporting tools restrict development to take place in-house (that is, still done by the robot manufacturer). By contrast, the proposed ORC architecture seems to provide appropriate features and programming views. Of course, it is far from being a product.
- Further development, in directions proposed here, of the new S4 system from ABB appears to be feasible.
- An open and powerful experimental system, like the one presented in Chapter 5, is of key importance for prototyping of application specific control.

Several of the application demands presented are to some extent known by engineers in industry. Nevertheless, these aspects have most often been overlooked within robot programming research. We can therefore summarize the contributions of this chapter as:

- The problem formulations concerning application specific motion control are contributions in themselves.
- It was shown how the ORC architecture supports implementation of such control.
- It was described how use of control signals in the built-in motion control can be utilized to avoid external equipment and sensors, and thereby also achieving tighter external control loops.
- It was described how system identification, auto-tuning, and adaptive control should be handled to obtain performance, functionality, and programmability of the robot.
- A connection of a task-level/off-line programming system to a robot system was done. The unique property is that new application primitives can be installed from the engineering workstation (via a separate network connection) into the embedded robot controller, thus extending the capabilities of the task-level system.
- The development of application specific motion control was put into an industrial perspective.

Thus, open multi-layered robot control systems with well defined programming interfaces are of key importance for the applicability of industrial robots.

9

Conclusions

We can now sum up the contributions of this thesis, and propose directions for further research. But first, the background, motivation, and approach of the work is recalled for the reader using this chapter as a summary of the thesis.

Summary of topics and industrial relevance

Industrial robots play a key role in flexible manufacturing systems. The software in a robot control system contains a software level for simple end-user (re)programming, and a software level for efficient computation of the low-level control. It turns out that application programming of many applications implies the need for an intermediate software level linking these two levels. Such an intermediate level must handle aspects such as dynamic effects, sensor integration, timing between the robot, its equipment, and external equipment, and so on. These robot functions are favorably hand crafted due to effects like dynamics, part tolerance, physical characteristics such as friction or stiction, and the like.

Also on the end-user programming level, there are problems that hinder improvements of manufacturing practices. Such a problem is deficient integration of engineering style (off-line) programming and production style (on-line) robot programming. Another problem is lack of programming language support for special applications.

High performance low-level servo control is important for productivity and high utilization of the equipment. This has inspired extensive research within robot motion control. Even if more remains to be done, many research results have been presented. Industrial usage has, however, been limited. First, it is hard for the robot manufacturer to introduce new concepts in a controller that is highly optimized for current main applications. Secondly, such motion controllers form closed systems, i.e., new control solutions based on feedback from external sensors (or

internal states) cannot be externally defined. Third, suitable experimental systems based on modern and commonly used industrial manipulators have not been available to the academic research, and industrial demands on programmability, efficiency and a partly closed structure have not been observed until now.

With a background from industrial robot development, the purpose of this thesis project was to tackle the above problems concerning the control principles, the structure of the system, and the programming methods. That would hopefully lead to more applicable robots and less repetitive and monotonous manual work. Note, however, that the purpose is not to replace all manual task by fully automated manufacturing plants. Instead, the desire is to have machines that allow human friendly industrial work.

Contributions

In order as presented, the contributions of the thesis are:

- The analysis and the control interpretation of end-user robot programming. Integrated on-line and off-line programming allows an efficient cascade control type of production control.
- The proposed technique for transformation of robot programs, using incremental tools from computer science, is novel in this context.
- The problem of physical context sensitivity and the need for support from the on-line programming language were identified. An approach and a system for full implementation were developed.
- Existing control system architectures were investigated, and the new Open Robot Control (ORC) architecture was proposed. ORC defines user/programmer views of the system.
- A unique experimental platform based on widely used industrial robots has been developed. This makes it easy to perform control experiments using a modern and industrially relevant robot dynamics. It has also been connected to a test-level programming system for arc welding, thus making that research possible.
- Improvements and extensive verification of the used public domain real-time platform developed within the department, makes the experimental environment even more attractive for research and teaching. New tools for parameter handling and control signal logging were also developed.
- The employment of high-level principles for special purpose hardware, and for its interaction with other types of CPUs, was tested. A

special object design and compiling strategy were fully implemented and successfully used.

- The performed multi-processor implementation of robot motion control illustrates how heterogeneous real-time software and hardware can be combined to efficiently execute different parts of the control algorithms.
- To accomplish an appropriate blend between efficiency and flexibility in embedded control systems, a concept called actions was introduced. The associated operator here means a piece of compiled code that is sent as an executable control function between different parts of the controller, typically using different address spaces and CPU types.
- Based on the use of actions and an appropriate software design, an open motion controller was proposed. Built-in control algorithms can still be mainly closed. It is, however, open in the sense that the low-level servo control can be tailored to special application demands, and external control loops can be tightly connected to it.
- Some industrial applications that can benefit from application specific motion control were presented. The way such applications can be solved using the proposed principles is a major contribution.

Apart from being directly applicable to robot control systems, several of these contributions are also useful for other type of control or embedded systems.

Reflections

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, despite certain drawbacks that were 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 for 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.

Future work

Further research and development should preferably start by making a more product-like experimental system. A second generation circuit boards including some improvements would promote general usage. A software module containing the latest generation software from ABB Robotics, extended by some RPC servers to make it more open, would be valuable. Such a system would allow full industrial evaluation of the proposed control system and programming principles.

Furthermore, formal abstraction and specification methods of the structuring principles encountered may prove fruitful both for mathematical capacity analysis (in the section of discrete-event systems and feedback control systems) and for standardization purposes.

Further development of control algorithms and control engineering tools is, of course, very important. First, the robot and its external axes need improved performance and simplified tuning. Secondly, combined position/force control, including programming and tuning, needs further enhancements to be generally applicable. These problems are the subject to ongoing control research, but its incorporation into an industrially useful system should perhaps be more emphasised. Such control strategies would ease full implementation of new strategies for deburring, welding, etc.

Thus, the contributions and some further development in this direction will hopefully lead to more flexible and efficient industrial robots, making human-friendly industrial work possible as desired.

References

[1] ABB CORPORATE RESEARCH, DEPT. OF AUTOMATIC CONTROL, LUND, SWEDEN. *Knowledge-Based Real-Time Control Systems. IT4 Project: Phase II*, 1991.

[2] ABB FLEXIBLE AUTOMATION, S-721 68 Västerås, Sweden. *Product Manual Robot Control System S4*.

[3] ABB FLEXIBLE AUTOMATION, S-721 68 Västerås, Sweden. *Service manual Irb-2000/3*.

[4] ABB FLEXIBLE AUTOMATION, S-721 68 Västerås, Sweden. *Programming Manual Robot Control System S3*, 1991. Art. No. 6397 013-129.

[5] ABB FLEXIBLE AUTOMATION, S-721 68 Västerås, Sweden. *ArcWare User's Guide*, 1995. Art. No. 3HAB 0011-13.

[6] ABB FLEXIBLE AUTOMATION, S-721 68 Västerås, Sweden. *RAPID User's Guide*, 1995. Art. No. 3HAB 0002-24.

[7] ABB FLEXIBLE AUTOMATION, S-721 68 Västerås, Sweden. *SpotWare User's Guide*, 1995. Art. No. 3HAB 0002-24.

[8] H. ABELSON, G. H. SUSSMAN, and J. SUSSMAN. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1985.

[9] ADEPT TECHNOLOGY INC., San Jose, CA, USA. *AdeptMotion Servo*.

[10] ADEPT TECHNOLOGY INC., San Jose, CA, USA. *AIM Users Guide*, 1991.

[11] B. ÅGREN. *Sensor Integration for Robotic Arc Welding*. PhD thesis TMMV-1023, Lund Inst. of Technology, Lund, Sweden, Department of Production and Materials Engineering, 1995.

[12] A. V. AHO, R. SETHI, and J. D. ULLMAN. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1985.

[13] 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.

[14] ANALOG DEVICES. *Variable Resolution, Monolithic Resolver-to-Digital Converter (AD2S80)*, 1992.

[15] R. J. ANDERSON. "Smart: A modular architecture for robotics and teleoperation." In *IEEE Conference on Robotics and Automation, Atlanta*, pp. 416–421, May 1993.

[16] J. ANDERSSON. "Ett öppet system för programmering och styrning av robotar," (An open system for robot programming and control). Master thesis ISRN LUTFD2/TFRT--5557--SE, April 1996.

[17] L. ANDERSSON and A. BLOMDELL. "A real-time programming environment and a real-time kernel." In ASPLUND, Ed., *National Swedish Symposium on Real-Time Systems*, Technical Report No 30 1991-06-21. Dept. of Computer Systems, Uppsala University, Uppsala, Sweden, 1991.

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

[19] R. L. ANDERSSON. "Computer architectures for robot control: A comparison and a new processor delivering 20 real Mflops." In *IEEE International Conference on Robotics and Automation*, 1989.

[20] R. L. ANDERSSON. "System design for robot control with a scalar supercomputer." In *IEEE International Conference on Robotics and Automation*, pp. 1210–1215, Cincinnati, OH, May 1990.

[21] P. ANELL. "Modeling of multibody systems in Omola." Master thesis ISRN LUTFD2/TFRT--5516--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, September 1994.

[22] APPLE COMPUTER INC. *Inside Macintosh*, volume I. Addison-Wesley, 1985.

[23] APPLE COMPUTER INC. *OpenDoc for Macintosh, An Overview for developers*, 1994.

[24] ARCHITECTURE TECHNOLOGY CORPORATION, P.O.Box 24344 Minneapolis, Minnesota 55424, USA. *Flexible Manufacturing Systems*, 4 edition, September 1991.

[25] J. ARMSTRONG, R. VIRDING, and M. WILLIAMS. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[26] K. J. ÅSTRÖM and J. EKER. "A nonlinear observer for the inverted pendelum." In *Submitted to the IEEE Conference on Control Applications*, 1996.

[27] K. J. ÅSTRÖM and K. FURUTA. "Control principles with applications to pendelums.". Under preparation., 1995.

[28] K. J. ÅSTRÖM and T. HÄGGLUND. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, Research Triangle Park, NC, second edition, 1995.

[29] K. J. ÅSTRÖM and B. WITTENMARK. *Computer Controlled Systems—Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1990.

[30] AT&T. *Cfront – a C++ Translator*, 1990.

[31] AT&T. *DSP32C Digital Signal Processor - Information manual*, January 1990.

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

[33] ABB FLEXIBLE AUTOMATION Discussion with Åke Madesäter (responsible for Customer Focus), September 1995.

[34] A. J. BARBERA. *An architecture for a robot hierachical control system*. U.S. Department of Commerce, National Bureau of Standards, Special publication 500-23, Washington, D.C. 20234, USA.

[35] A. E. BEJCZY. *Parallel Computation Systems for Robotics Algortihms and Architecture*. New York: World Scientific, 1992.

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

[37] D. G. BIHN and T. C. S. HSIA. "Universal six-joint robot controller." *Control Systems Magazine*, February, February 1988.

[38] R. BIRD and P. WADLER. *Introduction to Functional Programming*. Prentice Hall Int (UK), 1988.

[39] E. BJARNASON and G. HEDIN. "A grammar-interpreting parser in a language design laboratory." Technical Report, Department of Computer Science, Lund University, Sweden, September 1995. Draft submitted to CC'96.

[40] H. W. BLACK. "Stabilized feedback amplifiers." *Bell Syst. Tech. J. (and US patent 2102671, 1927)*, **13**, pp. 1–18, 1934.

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

[42] 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.

[43] G. S. BOLMSJÖ. *Industriell Robotteknik*. Studentlitteratur, Lund, Sweden, 1989. In Swedish.

[44] G. BOOCH. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991.

[45] M. BRADY, J. M. HOLLERBACH, T. L. JOHNSON, T. LOZANO-PÉREZ, and M. T. M. (EDS.). *Robot Motion*. MIT Press, Cambridge, Massachusetts, 1982.

[46] H. BRANTMARK and I. REYIER. "Implementation of application oriented function packages in a robot control system." In *International Symposium on Industrial Robots (ISIR)*, 1995.

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

[48] K. BRINK. *Event-based Control of Industrial Robot Systems*. Lic Tech thesis, Lund Inst. of Technology, Lund, Sweden, Department of Production and Materials Engineering, 1996.

[49] K. BRINK, M. OLSSON, and G. BOLMSJÖ. "Event based robot control." In *The first ECPD International conference on Advanced Robotics and Intelligent Automation*, September 1995.

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

[51] H. V. BRUSSEL and W. PERSOONS. "Robotic deburring of small series of castings." In *Annals of the CIRP 45/1*.

[52] J. F. CANNY and K. Y. GOLDBERG. "A RISC approach to sensing and manipulation." In *Manuscript submitted for publication*, Dept. of Computer Science, UC Berkeley, USA, 1995.

[53] S. CASELLI, E. FALDELLA, and F. ZANICHELLI. "Performance evaluation of processor architectures for robotics." In *Compeuro*, pp. 667–671. IEEE, 1991.

[54] S. CAVALIERI, A. D. STEFANO, and O. MARIBELLA. "Pre-run-time scheduling to reduce schedule length in the fieldbus environment." *IEEE Transactions on Software Engineering*, **No 11**, November 1995.

[55] S. CHAMBERLAIN and R. PESCH. *Using LD, the GNU linker*. Free Software Foundation, Inc. and Cygnus Support, URL: http://www.cygnus.com/library/ld/ld_toc.html, March 1993.

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

[57] W. F. CLOCKSIN and C. S. MELLISH. *Programming in Prolog*. Springer-Verlag, 1987.

[58] "ESPRIT 6805 COMPLAN public domain report.". URL reference: http://www.mech.kuleuven.ac.be/pma/project/complan, July 1995.

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

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

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

[62] O. DAHL and L. NIELSEN. "Stability analysis of an on-line algorithm for torque limited path following." In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pp. 1216–1222, Cincinnati, Ohio, May 13–18 1990.

[63] R. DAVID and H. ALLA. *Petri Nets and Grafcet: Tools for modelling discrete events systems*. Prentice-Hall, 1992.

[64] J. DENAVIT and R. S. HARTENBERG. "A kinematic notation for lower-pair mechanisms based on matrices." *Journal of Applied Mechanics*, pp. 215–221, 1955.

[65] DENEB ROBOTICS, Detroit, USA. *IGRIP Users Manual*, 1995. IGRIP is a registred trademark of Deneb Robotics Inc.

[66] J. EKER and A. BLOMDELL. "A structured interactive approach to embedded control." In *Submitted to the 4th International Symposium on Intelligent Robotic Systems, Lisbon, Portugal*, Dept. of Automatic Control, Lund, Sweden, 1996.

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

[68] A. FEDELE, A. FIORETTI, C. MANES, and G. ULIVI. "On-line processing of position and force measures for contour identification and robot control." In *IEEE International Conference on Robotics and Automation*, pp. 369–374, 1993.

[69] 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.

[70] W. E. FORD. "What is an open architecture controller." In *9th International Symposium on Intelligent Control*, 1994.

[71] C. F. L. GENE H. GOLUB. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 1989.

[72] GMF INC., Troy, Mich., USA. *KAREL Language Reference*.

[73] I. GODLER, A. AKAHANE, K. OHNISHI, and T. YAMASHITA. "A novel rotary acceleration sensor." *IEEE Control Systems*, February, February 1995.

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

[75] B. GUDMUNDSSON. *Virtuellt Minne*. Studentlitteratur, Lund, Sweden, 1977.

[76] N. HALBWACHS. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

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

[78] A. HANSSON and L. NIELSEN. "Control and supervision in sensor-based robotics." In *Proceedings Robotikdagar*, Linköping, Sweden, 1991.

[79] S. Y. HARMON. "Architectures: Designers vs. implementors." In *Workshop on architectures for intelligent control systems*. IEEE International Conference on Robotics and Automation, 1992.

[80] B. HAYES-ROTH, K. PFLEGER, P. LALANDA, P. MORIGNOT, and M. BALABANOVIC. "A domain-specific software architecture for adaptive intelligent systems." *IEEE Transactions on Software Engineering*, March, pp. 288–301, March 1995.

[81] 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.

[82] G. HEDIN. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Science, Lund University, Sweden, 1992.

[83] M. HEMMINGSSON. "Digital reglering av ett resonant servo med friktion," (Digital control of a resonant servo with friction). Master thesis ISRN LUTFD2/TFRT--5515--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, September 1994.

[84] B. H. J. HENDRIKS. "Implementation of industrial robot control." Master thesis ISRN LUTFD2/TFRT--5555--SE, March 1996.

[85] H. HIRABAYASHI, S. OHWADA, I. YOSHIDA, and M. MIKI. "Force-controlled deburring robots." pp. 19–1 – 19–11, Production Engineering Research Laboratory, Hitachi Ltd., Yokohama, Japan. unknown conference.

[86] C. A. R. HOARE. *Communicating Sequential Processes*. Prentice Hall Int., London, UK, 1985.

[87] IBM CORP. *SOMobjects Developer Toolkit, Users Guide*, 2.0 edition, June 1993. An introductory guide to the System Object Model and its accompanying frameworks.

[88] INTEGRATED MOTIONS, INC., 758 Gilman St., Berkeley, CA 94710, USA. *Zebra-ZERO Force Control Robot*, 1992.

[89] INTERNATIONAL ELECTROTECHNICAL COMMISSION. *IEC 1131-3, Programmable controllers, Part 3: Programming languages*, 1992.

[90] THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO), http://www.iso.ch/cate/35100.html. *Open systems interconnection (OSI)*, 1995.

[91] THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO), TECHNICAL COMMITTEE TC184/SC2, http://www.iso.ch/. *Manipulating industrial robots – Intermediate Code for Robots (ICR)*, ISO/TR 10562:1995.

[92] J. ISH-SHALOM. "Task level specification of a robot control." In *IEEE International Symposium on Intelligent Control, Philadelphia*, 1987.

[93] Y. ISHIKAWA, H. TOKUDA, and C. W. MERCER. "An object-oriented real-time programming language." *IEEE Computer*, October, pp. 66–73, October 1992.

[94] ISRA SYSTEMTECHNIK GMBH, MorneWegstraße 45A, D-6100 Darmstadt, Germany. *Robot Command & Teach Ball*, 1992.

[95] IVF-KTH. *Montering, idag och bortom år 2000 (Assembly, today and beyond year 2000)*, Monteringsenheten, 100 44 Stockholm, Sweden, September 1995. In Swedish.

[96] I. JACOBSON. *Object-Oriented Software Engineering, A Use Case Driven Approach*. ACM Press, Addison-Wesley, 1992. ISBN 0-201-54435-0.

[97] G. JOHANNESSON. *Object-Oriented Process Automation with SattLine*. Chartwell Bratt Ltd, 1994. ISBN 0-86238-259-5.

[98] R. JOHANSSON. "Lund research programme in autonomous robotics." NUTEK project 9309758, Swedish National Board for Industrial and Technical Development, 117 86 Stockholm, Sweden, 1993.

[99] R. JOHANSSON. *System Modeling and Identification*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[100] R. JOHANSSON and M. W. SPONG. "Quadratic optimization of impedance control." In *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 616–621, San Diego, California, May 1994.

[101] JR3 INC., 22 Harter Avenue, Woodland, CA 95695, USA. *JR3 Multi-Axis Force-Torque Sensors*.

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

[103] H. KAZEROONI. "Automated robotic deburring using impedance control." *IEEE Control Systems Magazine*, February, February 1988.

[104] B. W. KERNIGHAN and D. M. RITCHIE. *The C programming language*. Prentice-Hall, 1978.

[105] H. K. KHALIL. *Nonlinear Systems*. Macmillan, 1992.

[106] R. KLEPKO and A. MALOWANY. "A rule-based hierachical robot control system." In *Proceeding of the 1987 Summer Computer Simulation Conference, San Diego, USA*, pp. 646–652, July 1987.

[107] J. L. KNUDSEN and ET.AL. *Object oriented environments: the Mjolner approach*. Prentice Hall, 1993.

[108] 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.

[109] A. J. KOIVO. *Fundamentals for Control of Robotic Manipulators*. John Wiley & Sons, 1989.

[110] K. KOSUGE, K. FURUTA, and T. YOKOYAMA. "A control architecture for intelligent mechanical system: Virtual internal model following control." In *IEEE International Symposium on Intelligent Control, Philadelphia*, 1987.

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

[112] C. S. G. LEE, R. C. GONZALEZ, and K. S. FU (ED.). *Tutorial on Robotics*. Number ISBN 0-8186-0658-4. IEEE Computer Society Press, 1986.

[113] C. S. G. LEE, T. N. MUDGE, and J. L. TURNEY. *Tutorial on Robotics*, chapter Hierachical control structure using special purpose proces-

sors for the control of robot arms. Number ISBN 0-8186-0658-4. IEEE Computer Society Press, 1986.

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

[115] L. I. LIEBERMAN and M. A. WESLEY. *Tutorial on Robotics*, chapter AUTOPASS: An automatic Programming System for Computer Controlled Mechanical Assembly. Number ISBN 0-8186-0658-4. IEEE Computer Society Press, 1986.

[116] J. LLOYD and V. HAYWARD. "Real-time trajectory generation in multi-RCCL." Technical Report, McGill Research Centre for Intelligent Machines, McGill University, Montreal, Canada, April 1992.

[117] P. LOBERG and A. TÖRNE. "A layered architecture for real-time applications." In *The 7:th Euromicro Conference on Real-Time Systems, Odense, Denmark*. IEEE, June 1995.

[118] T. LOZANO-PÉREZ. "Robot programming." *Proceedings of the IEEE*, **71:7**, pp. 821–841, July 1983.

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

[120] LuTH M4ME-81/82. "Styrsystem för industrirobot ASEA Irb-6." Technical report 1982:31, Luleå Technical University, Luleå, Sweden, 1982. In Swedish.

[121] B. MAGNUSSON. "Application specific real-time systems – programming environments." NUTEK project 9403737, Swedish National Board for Industrial and Technical Development, 117 86 Stockholm, Sweden, 1993.

[122] B. MAGNUSSON and R. HENRIKSSON. "Garbage collection for control systems." In *Proceedings of IWMM'95 (International Workshop on Memory Management)*, Kinross, Scotland, September 1995.

[123] 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.

[124] THE MATHWORKS, INC., Natick, MA. *Matlab Reference Guide*, 1992.

[125] THE MATHWORKS, INC., Natick, MA. *Real-time workshop Users guide*, 1994.

[126] B. MAYER. *Object-oriented Software Construction*. Prentice Hall, 1988.

[127] A. MAYSTEL. "Architectures for intelligent control systems.". Discussion List: aics-l@ubvm.cc.buffalo.edu, 1992-1996.

[128] S. M. MEERKOV. "MANUFACTURING SYSTEMS: Too many problems..., too few solutions...". Plenary seminar, American Control Conference, Baltimore, July 1994.

[129] J. P. MEEUWSE. "Algorithms and tools for control of flexible servo systems." Master thesis ISRN LUTFD2/TFRT--5531--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, July 1995.

[130] G. MESSINA and G. TRICOMI. "Standard programming tools for robots." In *Proceeding of the Third international Symposium on Measurement and Control in Robotics*, Torino, Italy, September 1993. IMECO Technical Committee on Robotics (TC 17).

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

[132] MICROSOFT CORPORATION. *Object Linking Embedding*, 1993. Version 2.0.

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

[134] M. S. MUJTABA and W. D. FISHER. "HAL, a work station based intelligent robot control environment." *Robots 11/ISIR 17 Conference Proceedings*, 1987.

[135] S. MUJTABA and R. GOLDMAN. "AL Users' Manual." Technical Report STAN-CS-81-889, Stanford, Dept. of Computer Science, December 1981.

[136] F. N-NAGY and A. SIEGLER. *Engineering Fundations of Robotics*. Prentice Hall, 1987.

[137] H. NASRI. *A Process Model for Robotic Grinding*. Lic Tech thesis TMMV-1014, Lund Inst. of Technology, Lund, Sweden, Department of Production and Materials Engineering, October 1992.

[138] H. NASRI and G. BOLMSJ . "Using robot dynamics for part weight measuring and developing search function." *Submitted to Int. J. of Intelligent Mechatronics*, 1995.

[139] E. J. NICOLSON. "Standardizing i/o for mechatronic systems (sioms) using real time unix device drivers." In *Proc. 1994 IEEE Conf. Robotics and Automation*, pp. 3489–3494, 1994.

[140] L. NIELSEN. "Computer implementation of control systems." Report TFRT-7476, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 1991.

[141] K. NILSEN. "Real-Time Java." Technical Report, Iowa State University, Ames, Iowa, USA, February 1996.

[142] B. NILSSON. *Object-Oriented Modeling of Chemical Processes*. PhD thesis ISRN LUTFD2/TFRT--1041--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, August 1993.

[143] K. NILSSON. "Analys och syntes av en industrirobots dynamik.". Master's thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1982. In Swedish.

[144] K. NILSSON. *Application Oriented Programming and Control of Industrial Robots*. Lic Tech thesis ISRN LUTFD2/TFRT--3212--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, June 1992.

[145] K. NILSSON. "Application specific real-time systems – programming of control systsems." NUTEK project 9302726, Swedish National Board for Industrial and Technical Development, 117 86 Stockholm, Sweden, 1993.

[146] K. NILSSON. "DSPs moving up to object-oriented programs." *Electronic Engineering Times*, **No 765**, September 1993.

[147] K. NILSSON. "Object oriented DSP programming." In *Proceedings of The Fourth International Conference on Signal Processing Applications & Technology*, volume 1, pp. 561–570. DSP Associates, Santa Clara, California, Sept/Oct 1993.

[148] K. NILSSON. "Software for embedded DSPs." In *Proceedings from The American Control Conference*, 1994. Invited Paper.

[149] K. NILSSON, A. BLOMDELL, and O. LAURIN. "Open embedded control." *Submitted to: Real-Time Systems – The international journal of time critical computing*, 1996.

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

[151] K. NILSSON and L. NIELSEN. "On the programming and control of industrial robots." In *International Workshop on Mechatronical Computer Systems for Perception and Action*, pp. 347–357, Halmstad, Sweden, 1993.

[152] M. NYBERG. "Integration of robot programming systems." Master thesis, April 1996.

[153] S. OBARA, T. OHMAE, M. WATANABE, and K. KUBO. "Servo system using serial transmission in control loop for industrial robots." *IEEE Transactions on Industrial Electronics*, August, 1987.

[154] OBJECT MANAGEMENT GROUP (DIGITAL, HP, HYPERDESK, NCR, OBJECT DESIGN, SUNSOFT). *The Common Object Request Broker: Architecture and Specification*, document number 91.12.1 edition, December 1991.

[155] D. E. OKHOTSIMSKY and V. A. KARTASHEV. "Development of a robotic assembly complex." In *IEEE International Conference on Robotics and Automation*, Nice, France, 1992.

[156] G. OLSSON and G. PIANI. *Computer Systems for Automation and Control*. Prentice Hall, 1992.

[157] OMRON TATEISI ELECTRONICS CO., Control components H.Q., Osaka 541, Japan. *Laser Displacement Sensor (Z4W-A29)*.

[158] M. OTTER, H. ELMQVIST, and F. E. CELLIER. "Modelling of multibody systems with the object-oriented modeling language dymola." In *Computer Aided Analysis of Rigid and Flexible Mechanical Systems*, Troia, Portugal, 1993. NATO Advanced Study Institute.

[159] W. PERSOONS. "Design, manufacturing and calibration of a hydraulic tool for the robotic fettling of castings." Technical Report 94R40, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Celestijnenlaan 300 B, B-3001 Leuven, Belgium, September 1994.

[160] W. PERSOONS and H. V. BRUSSEL. "Off-line programming of a force controlled robot for the deburring of castings." In *13th Symposium on Engineering Applications of Mechanical Manufacturing Science and Technology*, McMaster, Canada, May 1996.

[161] W. PERSOONS and P. VANHERCK. "A process model for robotic cup grinding." In *Annals of the CIRP 45/1*, 1996.

[162] J. N. PIRES and J. S. DA COSTA. "A real-time system for position/force control of mechanical manipulators." In *To appear in: Procedings of the 7th International Machine Design Conference*, Ankara, Turkey, September 1996.

[163] W. H. PRESS, B. FLANNERY, S. A. TEUKOLSKY, and W. T. VETTERLING. *Numerical recepies*. Cambridge University Press, 1986.

[164] M. RAHIMI and W. KARWOWSKI. *Human-Robot Interaction*. Number ISBN 0-85066-809-3. Taylor&Francis, 1992.

[165] U. RASPER and R. ANGERBAUER. "Heterogeneous distributed real-time achitecture for numerical and robot control." In *Proceeding of*

the Third international Symposium on Measurement and Control in Robotics, pp. 493–498, 1995.

[166] E. RICH. *Artificial Intelligence*. McGRAW-HILL, 1983.

[167] ROBERT BOSCH GMBH, INDUSTRIAL EQUIPMENT DIVISION, ELECTRONIC CONTROLS, P.O.Box 1162, D-6120 Erbach, Germany. *Bosch welding control PS-2000 – The new modular range*, 1987.

[168] L. ROSSOL. "Nomad open architecture motion control software." In *Proceedings of the International Robotics and Vision Automation Conference*, Detroit, MI, April 1993.

[169] *Realistic Robot Simulation Interface Specification*. IPK Berlin, Dr. R. Bernhardt, Pascalstr. 8-9, 10 587 Berlin, Germany, 1994.

[170] E. SANDEWALL, J. HULTMAN, and E. TENGVALD. "Software architecture and programming paradigms for robotic applications." Technical Report LiTH-IDA-R-88-12, Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1988.

[171] G. N. SARIDIS. "Intelligent robotic control." *IEEE Trans. Automatic Control*, **No AC-28**, pp. 547–557.

[172] S. A. SCHNEIDER, V. W. CHEN, and G. PARDO-CASTELLOTE. "The ControlShell component-based real-time programming system." In *IEEE Conference on Robotics and Automation*, 1995.

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

[174] M. SCHOPPERS. "Robotic whole-body dexterity and a software architecture for robotic task performance in uncontrolled environments." Technical Report, Robotics Research Harvesting, 1993. Phase 1 SBIR Final Report under NASA contract 9-18861.

[175] M. SCHOPPERS. "The use of dynamics in an intelligent controller for a space faring rescue robot." *Artificial Intelligence*, **No 73**, pp. 175–230, 1995.

[176] J. D. SCHUTTER and H. V. BRUSSEL. "A method for specifying and controlling complient robot motion." In *Proceeding of 25th Conference on Decision and Control*, pp. 1871–1876, Athens, Greece, 1986.

[177] J. D. SCHUTTER and H. BRUYNINCKX. "Model-based specification and execution of complient motion." *IEEE Conf. on Robotics and Automation*, May 1992.

[178] J. E. S. SEAN M. DORWARD, RAVI SETHI. "Adding new code to a running C++ program." In *USENIX C++ conference proceedings*, pp. 279–292, April 1990.

[179] L. SHA, R. RAJKUMAR, and M. GAGLIARDI. "A software architecture for dependable and evolvable industrial computing systems." Technical Report CMU/SEI-95-TR-005, Software Engineering Institute, CMU, Pittsburg, USA, July 1995.

[180] M. SHAW, R. DELINE, D. V. KLEIN, T. L. ROSS, D. M. YOUNG, and G. ZELESNIK. "Abstractions for software architecture and tools to support them." *IEEE Transactions on Software Engineering*, March, pp. 314–335, March 1995.

[181] 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.

[182] K.-I. SHIMOKURA and S. LIU. "Programming deburring robots based on human demonstration with direct burr size measurement." In *IEEE International Conference on Robotics and Automation*, pp. 572–577, 1994.

[183] SILMA INC., 1601 Saratoga-Sunnyvale Rd., Cupertino, CA 95014, USA. *The CimStation User's manual*, 1989.

[184] SILMA INC., 1601 Saratoga-Sunnyvale Rd., Cupertino, Calif. 95014. *SIL Language Manual*, 1991.

[185] September 1993. Information from John J. Craig during meeting at SILMA Inc.

[186] D. SIMON, B. ESPIAU, E. CASTILLO, and K. KAPELLOS. "Computer-aided design of a generic robot controller handling reactivity and real-time control issues." Technical Report 1801, INRIA, 2004 route des Lucioles, B.P. 93, 06902 Sophia-Antipolis, France, November 1992.

[187] D. SIMON, B. ESPIAU, E. CASTILLO, and K. KAPELLOS. "Computer-aided design of a generic robot controller handling reactivity and real-time control issues." *IEEE Transactions on Control Systems Technology*, December, pp. 213–229, December 1993.

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

[189] L. SOBRAL, R. JOHANSSON, G. LINDSTEDT, and G. OLSSON. "Ultrasonic detection in robotic environments." In *Proc. 1996 IEEE Conf. Robotics and Automation*, Minneapolis, MN, 1996.

[190] THE SOFTWARE DEVELOPMENT ENVIRONMENT GROUP, DEPARTMENT OF COMPUTER SCIENCE, LUND UNIVERSITY, SWEDEN, orm@dna.lth.se. *Applab User's Guide*, 1995.

[191] J. SONNERFELDT. "Matlab som 'compute server' för inbyggda system – beräkning av robotrörelser," (Matlab as a compute server for embedded systems—trajectory computation). Master thesis ISRN LUTFD2/TFRT--5527--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, February 1995.

[192] M. W. SPONG. "Swing up control of the acrobot." In *IEEE International Conference on Robotics and Automation*, pp. 2356–2361, 1994.

[193] M. W. SPONG and M. VIDYASAGAR. *Robot Dynamics and Control*. John Wiley & Sons, 1989.

[194] J. A. STANKOVIC and K. RAMAMRITHAM. *Hard Real-Time Systems*. IEEE Computer Society Press, 1988.

[195] J. A. STANKOVIC and K. RAMAMRITHAM. *Advances in Real-Time Systems*. IEEE Computer Society Press, 1993.

[196] D. B. STEWART, D. E. SCHMITZ, and P. K. KOSLA. "CHIMERA II: A real-time multiprocessing environment for sensor-based robot control." In *IEEE International Conference on Robotics and Automation*, 1989.

[197] D. B. STEWART, D. E. SCHMITZ, and P. K. KOSLA. "Implementing real-time robotic systems using CHIMERA II." In *IEEE International Conference on Robotics and Automation*, 1990.

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

[199] P. STRUSS. *Expert Systems in Computer-Aided Design*, chapter Multiple Representation of Structure and Function, pp. 57–84. Elsevier Science Publishers B.V., SIEMENS Corp., Otto-Hahn-Ring 6, D-8000 Munich 83, Germany, 1987.

[200] J. W. SULLIVAN and S. W. T. (EDS.). *Intelligent User Interfaces*. Addison Wesley, 1991.

[201] SUN MICROSYSTEMS. *SunOS Linker and Libraries Manual*, November 1993.

[202] 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.

[203] R. TAYLOR. "A synthesis of manipulator control – programs from task-level specifications." Technical Report AI Memo 282, Stanford University, July 1976.

[204] R. H. TAYLOR, P. D. SUMMERS, and J. M. MAYER. *Tutorial on Robotics*, chapter AML: A manufacturing language. Number ISBN 0-8186-0658-4. IEEE Computer Society Press, 1986.

[205] T. THOMESSON and O. EGELAND. "Integrated force-control system for grinding and deburring applications.". Center for Robotic Research, The Norwegian Institute of Technology, N-7034 Trondheim, Norway.

[206] A. TOPPER and B. ENG. "A computing architecture for a multiple robot controller." Technical Report TR-CIM-91-5, McGill Research Centre for Intelligent Machines, McGill University, Montreal, Canada, June 1991.

[207] M. TÖRNGREN. *Modelling and Design of Distributed Real-Time Control Applications*. PhD thesis ISRN LUTFD2/TFRT--KTH/MMK- -95/7- -SE--SE, Royal Institute of Technology, S-100 44 Stockhom, Sweden, 1995.

[208] S. TRUVÉ. "A new H for real-time programming.". Carlstedt Research & Technology AB, Stora Badhusgatan 18-20, 411 21 Göteborg, Sweden, 1995.

[209] T. L. TURNER, J. J. CRAIG, and W. A. GRUVER. "A microprocessor architecture for advanced robot control." pp. 407–416, Unidentified conference around 1984.

[210] S. G. TZAFESTAS. *Intelligent Robotic Systems*. Number ISBN 0-8247-8135-X. Marcel Dekker Inc., 1991.

[211] D. UNGAR, R. SMITH, C. CHAMBERS, and U. HÖLZLE. "Object, message, and performance: How they coexist in Self." *IEEE Computer*, October, pp. 53–64, October 1992.

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

[213] E. VERHULST. "Virtuoso: Providing sub-microsecond context switching on DSPs with a dedicated nanokernel." In *Proceedings from The International Conference on Signal Processing Applications and Technology*, Santa Clara, CA, USA, September 1993.

[214] D. VISCHER and O. KHATIB. "Design and development of high-performance torque-controlled joints." *IEEE Transactions on Robotics and Automation*, **No 4**, August 1995.

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

[216] L. WALL and R. L. SCHWARTZ. *Programming Perl*. UNIX Programming. O'Reilly & Associates, nuts@ora.com, August 1991.

[217] P. WEGNER. "Dimensions of object-oriented modeling." *IEEE Computer*, October, pp. 66–73, October 1992.

[218] B. WITTENMARK. "Projekt i Adaptiv reglering VT 93," (Projects in Adaptive Control). Project number 13: "Adaptiv robotstyrning" ISRN LUTFD2/TFRT--7508--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, June 1993.

[219] B. WITTENMARK, J. NILSSON, and M. TÖRNGREN. "Timing problems in real-time control systems." In *Proceedings of the 1995 American Control Conference*, Seattle, Washington, 1995.

[220] B. WITTENMARK and M. TÖRNGREN. "Timing problems in real-time control systems: Problem formulation." Report ISRN LUTFD2/TFRT--7518--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 1994.

[221] W. M. WONHAM. *Linear Multivariable Control – A Geometric Approach*. Springer-Verlag, New York, 3rd edition, 1985.

[222] S. H. ZWEBEN, S. H. EDWARDS, B. W. WEIDE, and J. E. HOLLINGSWORTH. "The effects of layering and encapsulation on software development cost and quality." *IEEE Transactions on Software Engineering*, March, pp. 200–208, March 1995.