Realization of Expert System Based Feedback Control

Årzén, Karl-Erik

1987

Realization of Expert System Based Feedback Control

Karl-Erik Årzén

Lund 1986

To Gunilla and Pernilla

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

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* Doctoral Dissertation
	Date of issue November 1987
	Document Number CODEN: LUTFD2/(TFRT-1029)/1-199/(1987)
Author(s) Karl-Erik Årzén	*Supervisor* K.J. Åström
	Sponsoring organisation The National Swedish Board of Technical Development (STU)

Title and subtitle

Realization of Expert System Based Feedback Control

Abstract

There is currently a significant interest in expert system techniques in the process control community. A common application is to use the expert system as an operator aid for process monitoring. The topic of this thesis is to explore the use of expert system techniques in feedback control systems. Even simple regulators need a substantial amount of heuristic logic. The development of this safety-jacket is time-consuming. In the thesis, the heuristics is separated from the algorithms and implemented using expert system techniques. This gives a better logical structure and it leads to control systems with new and interesting properties. The concept is referred to as knowledge-based control. The initial tuning of a regulator is a good example of a problem where heuristics are mixed with deep knowledge and numerical algorithms. New theoretical result on relay tuning are given. They are used as the basis for heuristic rules for classifying processes according to their dynamics. An architecture is presented where two concurrent processes are used to implement the heuristics and the control algorithms. A prototype system is described where the standard expert system framework OPS4 is used on a VAX 11/780. Experiences from experiments with the prototype led to the design of a real-time expert system framework better suited for the problem. A modular, blackboard-based approach is taken. This allows the decomposition of the problem into subtasks which are implemented as separate knowledge sources that can be rule-based with different inference strategies or procedural. The framework can be compared with a real-time operating system and has similar real-time primitives. An example is shown were the framework is used to implement an elaborated version of relay auto-tuning. Knowledge is extracted about three points on the open-loop Nyquist curve. The tuning procedure chooses process model and controller depending on the extracted knowledge.

Key words

Real-time expert systems, Feedback control, Auto-tuning, Intelligent control, Relay feedback,

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title		*ISBN*
Language English	*Number of pages* 199	*Recipient's notes*
Security classification		

Contents

Preface

The following thesis belongs somewhere in the borderland of Computer Science, particularly Artificial Intelligence, and Automatic Control. Writing an interdisciplinary thesis has many potential pitfalls. The scope necessarily becomes broad and the audience, with experience from both fields, is small. This thesis is primarily written for a reader with knowledge of automatic control at the graduate level. To provide the necessary background and terminology, the thesis contains a survey on expert systems. The thesis is broad in the sense that it touches upon several different areas that each, in its own right, could be the topic of further research. The intention behind the work has been to tackle the entire problem in all its width rather than to dwell into the details of all the subproblems. Another pitfall of interdisciplinary theses is that they run the risk of being considered inappropriate for both fields. Hopefully, this is not the case here. Instead, my hope is that the work may generate new ideas and concepts.

Acknowledgements

This work has been carried out at the Department of Automatic Control, Lund Institute of Technology, Sweden. It has been a pleasure to work in an inspiring and dynamic atmosphere where projects that reach outside the borders of traditional control theory are encouraged. I would like to thank all my colleagues at the Department, who have made this thesis possible. The ideas behind this work originate from my supervisor Professor Karl Johan Åström. His support and confidence has been invaluable. I am indebted to Sven Erik Mattsson, Lars Nielsen, Per Hagander, Jan Eric Larsson, and Per Persson who all have given valuable criticism on the manuscript. A special thanks goes to Sven Erik Mattsson and Per Hagander for their proof-reading on Chapter 3. One of the seeds of this work is the relay auto-tuner. I would like to thank Tore Hägglund and Lars Bååth who have commercialized it and shared their experiences with me.

The thesis is type-set with the type-setting program TEX (Knuth, 1984). Leif Andersson has written the macros that were used in the thesis and contributed much to the typography of the thesis. He has also assisted me in mastering the EUNICE system. The building blocks in the expert system framework developed has been Flavors and YAPS. They have been provided by the Department of Computer Science, Univ. of Maryland. I would like to thank Liz Allen for this well-written software. The financial support has been provided by the National Swedish Board of Technical Development (STU). The work has been carried out in the Computer-Aided Control Engineering Project under contract 85-3084.

1

Introduction

Process control systems have developed significantly during the last decades. The control equipment has changed from analog controllers and relay networks during the late 1950s to distributed computer-based control systems. Computer-based systems increase the functionality and allow larger and more complex plants to be automated.

The computer revolution has affected process control at all levels. Microprocessors are used for implementation of local controllers. Programmable logic controllers (PLC) have replaced the relay networks. On the plant-wide level, computers are used for supervision and set-point control as well as for the actual control of the plant in the form of direct digital control (DDC) systems. Powerful tools for information presentation are an important part of the user interfaces.

In spite of this development the basic controllers are still the same, although implemented with new techniques. The control systems do not in general give any support in choosing controller structure or controller parameters. The majority of the control loops in process industry are still PID loops. A few adaptive and auto-tuning controllers have entered the market but these systems are still rather rigid.

Artificial intelligence (AI) is an area that long has attracted interest. Since the late 1950s the area has evolved into a separate, now and then, heavily discussed and questioned, research discipline. Expert systems or knowledge-based systems are a branch of AI that has gained increasing interest during the last decade. An expert system can loosely be described as a computer program that tries to emulate the problem solving behavior of a human expert in some limited domain. The reason for the increased interest in expert systems is partly that the technique has matured and partly that new and more suitable software and hardware have emerged. Expert system development tools, so called expert system shells or frameworks, are now available on conventional computers. Powerful workstations with dedicated hardware for AI languages such as, e.g., Lisp are in the market and the prices are dropping.

The topic of this thesis is to investigate the use of expert system techniques at the local control level. The aim is to use expert system techniques for the implementation of the heuristic logic that is an important part of all controllers. The separation between numerical algorithms and heuristic logic hopefully gives an environment that simplifies the development of complex controllers and where heuristics can be exploited to a higher degree. The concept will be referred to as *knowledge-based control* and the resulting controller will be named a *knowledge-based controller.* The reason for this terminology is the use of knowledge-based system techniques. It does not imply that it is the author's opinion that traditional controllers are based on ignorance. The ideas behind knowledge-based control were first presented in Åström and Anton (1984) and Åström *et al* (1986).

Functionally, the knowledge-based controller can be viewed as an extension of auto-tuners and adaptive regulators. The controller should on-line build up enough process knowledge to choose and tune an appropriate controller. The knowledge is mainly built up through different dynamic tuning experiments. The controller should have the capacity to select among several different control design principles. The knowledge-based system is used on a supervisory level on top of a set of numerical control, identification, and monitoring algorithms. The identification algorithms may consist of both simple algorithms for estimation of, e.g, ultimate gain and frequency, and ordinary on-line identification algorithms such as, e.g., a recursive least-squares algorithm. The main task of the knowledge-based system is to orchestrate different numerical algorithms on-line. This involves starting and stopping algorithms, calculating algorithm parameters, analyzing results from identification algorithms, react correctly on alarms from monitoring algorithms, etc.

Control of a process basically involves two types of knowledge: control knowledge and process knowledge. Control knowledge contains the textbook knowledge about automatic control, e.g., knowledge about different control strategies. Process knowledge contains knowledge of the actual process that should be controlled. This knowledge can, e.g, be acquired through identification experiments or from experienced process operators.

The knowledge-based controllers can be described as an attempt to supply a controller with enough control knowledge to be able to extract the process knowledge needed automatically. Ideally, the knowledge-based controller should be able to act and reason in the same way as an experienced control engineer does when he designs a controller and evaluates its performance. This involves several different types of knowledge. Some examples are: knowledge about tuning procedure procedures that can be used to extract process information, knowledge about dynamic models, knowledge about different control design techniques, knowledge about the numerical algorithms involved, e.g., what their parameters mean and how their results should be interpreted, etc. The structure is shown in Figure 1.1.

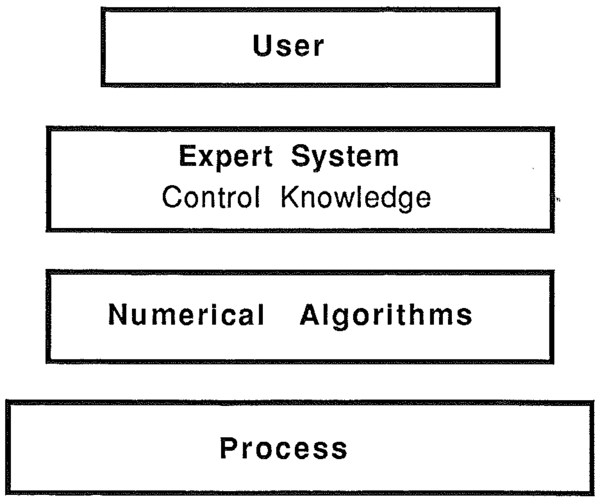

Figure 1.1 Knowledge-based controller

An entirely different approach, though quite similar with respect to implementation, is taken in the fuzzy control area. Here, the control algorithm is replaced by a set of rules for how the control output should be modified in different situations. The rules contains quantified, linguistic values of the measured signals. The intended applications for fuzzy control are control of complex processes for which either appropriate models do not exist or are inadequate. Work in this area also goes under the names linguistic control and rule-based control. The fuzzy controller may be viewed as an attempt to model the behavior of the operator who is manually controlling the process. Due to this, the resulting controller will be based mainly on process knowledge. It also implies that each controller is designed especially for a certain plant. The situation is shown in Figure 1.2.

The approach taken in this thesis is more general in that the controller immediately can be used on many different processes. However, it has also limitations. The assumption that enough process knowledge to control a process can be built-up automatically is probably only true for small control problems. This thesis concerns only the single-input, single-output case. It might be possible to extend the idea to systems with a few inputs and outputs. If, however, a process with multiple interconnected control loops is considered, the correct way is definitely to build-in as much a-priori process knowledge as possible into the controller.

A more well-known expert system application in process control is to use an expert system as a complement to a conventional control system for process monitoring and alarm analysis. The aim of such systems is to package the knowledge of experienced process operators and provide it 24-hours a day. Expert systems of this type are usually only used as advice giving systems. The architecture of

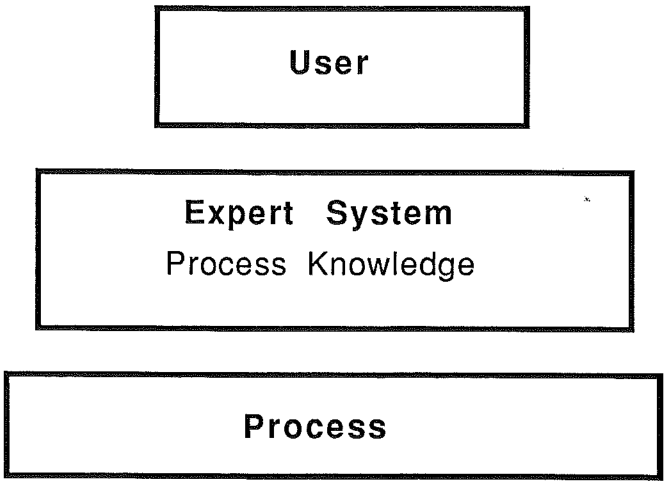

Figure 1.2 Fuzzy controller

this type of system is shown in Figure 1.3.

The knowledge-based controller can be viewed as a learning system. Learning has long been a major area of AI, (Michalski *et al*, 1983). In automatic control, learning ideas have been used before. Adaptive controllers are sometimes labeled as learning systems. Rule-based controllers sometimes have the possibility to modify the control actions of the rules based on accumulated experience, (Michie and Chambers, 1968). The learning in both these contexts consists of adjusting numerical parameters in expressions of a fixed form in order to obtain desired performance. The learning in the knowledge-based controller is instead focussed on acquiring higher level process knowledge, e.g., process models, by performing experiments with the process and analyzing the results.

Contributions

The main contribution of this thesis concerns the architecture and design of a knowledge-based controller. An architecture is proposed where the numerical algorithms parts and the knowledge-based system are separated into different concurrent processes. Most existing expert system frameworks are not suited for real-time operation. In this thesis, an expert system framework is described that is modelled after a conventional real-time operating system with similar real-time primitives. It allows rule-based and procedural knowledge representation. The framework supports the separation of a problem into a number of subtasks which may be implemented as separate modules.

The process knowledge needed for the control design is extracted through on-line tuning experiments. Relay feedback is one tuning method that is used in existing auto-tuners, (Åström and Hägglund, 1984b). The thesis also contains

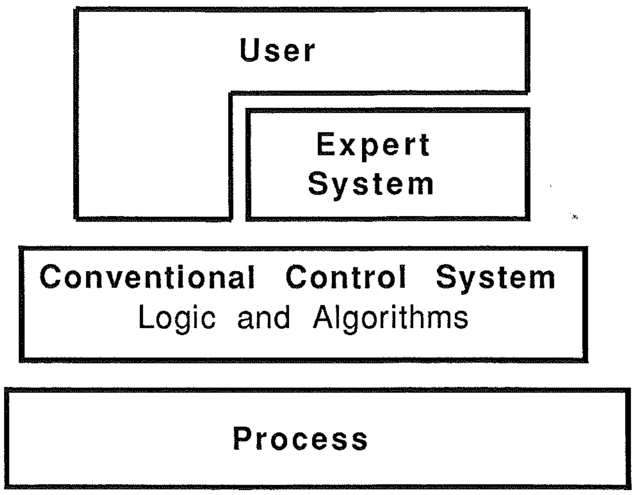

Figure 1.3 Expert system for process monitoring

new theoretical results regarding systems under relay feedback. Theorems are given which partially can be used to classify systems according to how they should be controlled.

The goal in the long-run for the work in knowledge-based control is to build an "intelligent" controller. This thesis has not reached that far. Instead, it contains some examples of how the real-time framework that has been developed may be used in the initial tuning of the controller. In general, knowledge-based control consists of more than just tuning. On-line monitoring is an equally important area.

Outline of the thesis

The thesis has the following organization. Chapter 2 gives further background and motivation for the knowledge-based controller. The chapter also contains an overview of related expert system applications in process control.

Tuning is a typical example of a problem that consists of both deep, theoretical knowledge, heuristic considerations and numerical algorithms. Chapter 3 contains new results on relay feedback. The results can be used to detect processes which may be approximated by first order systems.

Chapter 4 discusses the overall architecture of the knowledge-based controller. The subprocess decomposition and the interprocess communication are described in detail for a VAX 11/780 implementation. The implementation consists of three separate concurrent processes according to Figure 1.4. The chapter also describes the internal structure of the numerical algorithm subprocess.

What distinguishes the knowledge-based controller is the use of expert system techniques for implementing the heuristic logic. A brief overview of expert

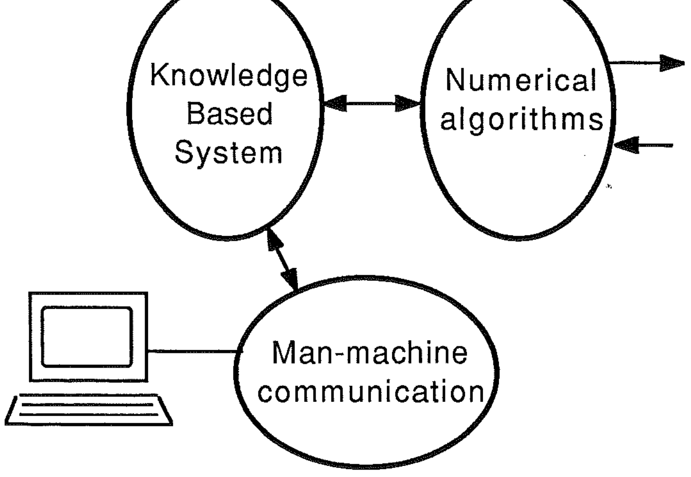

Figure 1.4 Implementation structure of a knowledge-based controller.

system techniques is given in Chapter 5.

A first prototype where the knowledge-based part of the system was implemented with an off-the-shelf expert system framework is described in Chapter 6. The prototype is evaluated and the lessons learned are discussed. In Chapter 7, an expert system framework suited for real-time operation is described. Chapter 8 describes some knowledge-based controller examples where this framework is used. A relay based tuning procedure is presented which is based on knowledge of three points on the Nyquist curve of the open-loop process. Finally, conclusions and directions for further work are given in Chapter 9.

2

Background and Motivation

When this thesis project started, it was called Expert Control. This name may, however, cause confusion. Many other expert system applications in automatic control go under the same name. Section 2.1 is an attempt to give a background to this thesis and to classify different expert system applications in automatic control.

The knowledge-based control approach can be motivated from two different viewpoints. The first relates to the intended functionality of the controller, i.e., to successively build up process knowledge. In Section 2.2, the knowledge-based controller is described in relation to conventional adaptive controllers and auto-tuners. The second viewpoint concerns the implementation technique. Section 2.3 contains motivation for the use of expert system techniques and a discussion of alternatives. The section emphasizes the heuristic elements of process control. Finally, Section 2.4 discusses possible implications that the expert system approach may have for the user interface.

2.1 Related applications

Following the expert system boom during the last years, several expert system applications in control, and specially process control, have been proposed. The idea of trying to combine expert system ideas with controllers is, however, not new. Already in 1962, a Heuristic Decision program was discussed (Crossman and Cooke, 1962). It was mainly intended for manual control problems.

During the beginning of the 1960s, the interest and optimism were high in artificial intelligence and cybernetics. Much research concerned systems modelled after the neuron-based structure of the human brain. These ideas were also applied to control problems. Systems such as the Perceptron (Rosenblatt, 1961) and ADALINE (Widrow, 1962) belong here. These ideas are also the origin of what is called learning control. Other names for this are intelligent control and self-organizing control.

In learning control systems, the controller should be able to estimate unknown information during its operation and determine an optimal control action from the estimated information. Different learning schemes have been proposed. Pattern classification techniques, sometimes with adaptable decision thresholds, are one example. Other examples are bayesian estimation and stochastic approximation. Much of the work in learning control systems could just as well fall into the adaptive control area. Examples of work in the area are Fu (1970, 1971) and Saridis (1977, 1983).

The area of fuzzy control has its origin in the work of Zadeh (1965) on multiple-valued logic in the form of fuzzy sets. The fuzzy control or rule-based control approach has, e.g., been applied to steam engines (Mamdani and Assilian, 1975) and cement kilns (Holmblad and Ostergaard, 1981). In Francis and Leitch (1985), a PROLOG based fuzzy control shell is described. A survey of the area is found in Tong (1977, 1984).

Expert system applications in process control

Expert system applications in process control can be classified according to three different criteria:

Plant-wide — single loop

Off-line operation — on-line operation

Operator assistance — closed loop operation

The cases in each pair above should be regarded as extreme cases and an actual application usually falls somewhere in between. The first criterion states if the application is intended for the global, plant-wide level with several control loops or if it is used on the single loop level. The second criterion states if the application is used in off-line mode, e.g., in an analysis or design phase, or if it is used for on-line operation. The last criterion tells whether the expert system is used mainly for information and advice presentation to the operator or if itself actually carries out the actions. This criterion is mainly applicable for the on-line case.

The following classifies different types of expert system applications in process control with respect to the above criteria.

Process Design: An expert system could be used as an off-line tool during process design. The aim is to capture knowledge about how the process should be designed in order to fulfill various objectives, e.g., good control prospects, serviceability, economy etc. Work along these lines in the case of factory design is described in Fisher, (1986).

Control design: Another typical off-line application. At the plant-wide level expert systems have been proposed for selecting appropriate input-output pairings given process configuration and control specifications (Niida and Umeda, 1986). In Shinskey (1986), an expert system is used for design of distillation controls based on relative gain techniques.

At the single loop level, expert systems are often combined with computer aided control design packages. The aim is to capture knowledge about different control design techniques such as SISO lead/lag compensation (James *et al*, 1985), multivariable design using linear quadratic theory (Birdwell *et al*, 1985), multivariable frequency domain design (Pang and McFarlane, 1987), state feedback and estimation (Trankle *et al*, 1986). In Larsson and Persson (1987), the expert system plays the role of an intelligent help system for system identification.

Process monitoring: Process monitoring is perhaps the most well-known expert system application in process control. The expert system is usually used on top of a conventional control system. The applications are typically plant-wide and the expert system is used more or less on-line. The expert system is mainly used as an operator assistant. Special cases of monitoring are fault diagnosis and alarm analysis. Fault diagnosis systems use plant-specific knowledge such as, e.g., cause-effect relations and trouble-tracing diagrams to locate the fault that caused the error symptoms and to give advice on how to correct it. In complex monitoring systems, an initial alarm often results in a lot of secondary alarms that drown the operators in information and hide the original alarm. An alarm analysis system with knowledge of, e.g., alarm patterns, dynamics of fault propagation and process configuration could assist the operators in resolving these matters. This work is motivated by accidents such as the Three Mile Island nuclear reactor accident. Examples of different monitoring applications are numerous. A few examples are Nelson (1982) on nuclear power plants, Sakaguchi and Matsumoto (1983) on electrical power systems and Palowitch and Kramer (1985) on chemical plants.

Process scheduling and optimization: Control optimization and scheduling are areas which have been proposed as potential expert system candidates. On a global level the expert system could, e.g., be used to give advice on set-points in order to optimize plant operation. Apart from using the expert system as an operator assistant, it has been suggested that the loop should be closed and that the expert system should directly interact with the controlled plant. The number of actual expert system projects that have been carried out along these lines is, however, small.

Expert system scheduling has been suggested for batch processes. It is also widely discussed in the discrete manufacturing industry for flexible manufacturing systems (FMS), (Smith *et al*, 1986). Possible applications in continuous process industries are process start-ups and shut-downs or major production changes. Actual implementations are also here rare.

Control: Here, the expert system typically is used on a small part of the plant, on-line, and in closed loop. This category contains the work in this thesis. Fuzzy controllers also belong here.

A few expert system projects have a closer relationship to knowledge-based control than others. These will be briefly described in the following.

In Trankle and Markosian (1985), Expert System Adaptive Control (ESAC) is described. This is in spirit similar to knowledge-based control. The intended application was a Reconfigurable Intelligent Flight Control System (RIFCS). The goal of the system was to enable adaptation to catastrophic plant changes that require changes in the structure of the control law. The system was modeled after an ordinary self-tuning regulator with three different expert system modules: the system identifier, the control system designer and the control implementation supervisor. No real-time version of the system has been implemented.

In Sanoff and Wellstead (1985), the combination of a self-tuning regulator and an expert system was proposed. The expert system was suggested to consist of two parts. A configuration expert guided the operator in the parameter setting of the controller and a run-time expert monitored the control.

The PICON system (Moore *et al*, 1984a, 1985) and the G2 system (Gensym, 1987) are two related expert system frameworks, specialized on process monitoring, which could be compared to the expert system framework described in this thesis. An expert system based on PICON basically consists of a set of rules where each rule is tested periodically. The condition parts of the rules typically specifies logical conditions of sensor values which must be fulfilled in order for the rule to be applicable. An example of rules from a catalytic cracker process (PICON, 1985) could look as follows.

```
let condition reactor-temperature-high =
    reactor-temperature > 990 deg-f

if reactor-temperature-high then
    send "The reactor temperature is high;
          gasoline production is below optimum."
          to console
  and
    conclude possible-increase-in-carbon
```

The above is a typical example of a rule based on shallow knowledge. Deep knowledge, e.g., predictions based on mathematical models, or validation of balance equations, is more difficult to represent.

2.2 Functionality of the knowledge-based controller

From a functional point of view the knowledge-based controller belongs to the class of auto-tuning and adaptive controllers. Through different tuning experiment, the controller should on-line build up and refine process knowledge. This knowledge is explicitly represented in the controller. The controller should also, to some extent, have the opportunity to exploit existing a priori information. This means that the operator should have some possibility to enter knowledge he has about the process into the controller.

In many aspects, this resembles what existing auto-tuners and adaptive controllers already are doing. A short overview will be given on current industrial control practice to point out the differences and to motivate the knowledge-based approach.

PID control

The choice of controller structure and the setting of the controller parameters always require some process knowledge. Modelling based on physical laws and off-line process identification (Eykhoff, 1974; Åström and Eykhoff, 1970; Ljung, 1987) can be used to acquire the knowledge needed. These methods are, however, time and resource extensive and are mainly justified for processes with high potential economic benefits or where the control specifications require accurate process models. This is usually not the situation in process industry. An ordinary industrial process contains a large number, perhaps hundreds, of control loops. Most control loops belong to a standard set of control problems, e.g., flow control, level control, etc. The majority of the loops are controlled by PID controllers. The controller structure is chosen based on experience from similar loops. For example, in pressure and level control, proportional action is usually sufficient, whereas in temperature control both proportional, integral and derivate action is needed. The controller parameters are manually tuned by rule-of-thumb methods, e.g., introduce integral action in order to eliminate the static error. In spite of this, or perhaps due to this, it is a well-known fact that many control loops in process industry are badly tuned or run in manual mode.

Auto-tuners

Auto-tuning controllers (Åström and Hägglund, 1984a, 1984b; Hoopes *et al*, 1983; Yarber, 1984) have an initial tuning phase which is used to determine the parameters of the controller. An alternative approach is to let the auto-tuner monitor the process behavior with respect to disturbances and based on that adjust the controller parameters (Bristol, 1977; Kraus and Myron, 1984).

Although existing auto-tuners to some extent extracts and builds up process knowledge, they have many limitations. The auto-tuners usually have a fixed controller structure, typically only PID, and use only one design method to calculate the PID parameters. This implicitly limits the class of processes which

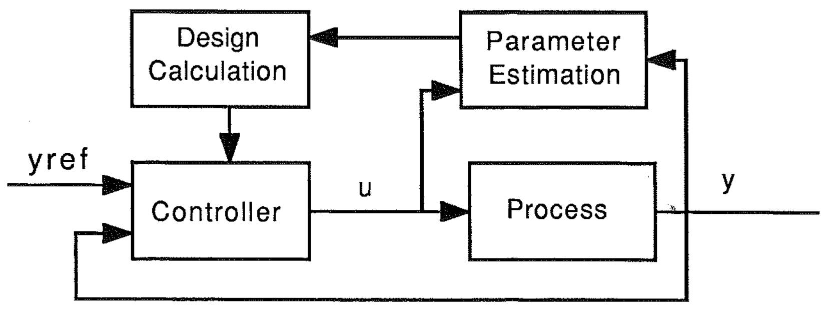

Figure 2.1 An explicit self-tuning regulator.

they can handle. They usually extract only the specific process knowledge needed for the control design used. An overview of auto-tuning controllers can be found in Åström and Hägglund (1988).

Adaptive controllers

A problem with many control loops is that they are badly tuned due to changes that have occurred since the tuning. A solution to this is the adaptive controller (Åström, 1987a; Åström and Wittenmark, 1988). An explicit self-tuning regulator contains a recursive identification algorithm that periodically updates an internal process model and adjusts the controller coefficients according to the model. This is shown in Figure 2.1.

The adaptive controllers are quite general. The process knowledge they extract is the parameters in a predetermined process model or, in some cases, the parameters in a predetermined control law. They need much a priori information about, e.g., model orders and details in the control and estimation methods used. Such information can be difficult to provide and process operators typically lack the intuitive understanding that they have with conventional PID controllers. The choice of these parameters is, however, crucial. Adaptive controllers can be viewed as gradient methods that perform well locally. If they, however, are started outside the local neighbourhood, the system may become unstable. The trend among commercial suppliers is to predetermine as many parameters as possible. The drawback of this is that the generality is lost and the class of processes which can be controlled is diminished.

Operator knowledge

Experienced process operators and process engineers are an important source of process knowledge. From experience of the process, the operator usually has some mental model of how the process behaves and responds to changes. He knows the type of control problem, i.e., whether it is a temperature loop or flow loop,

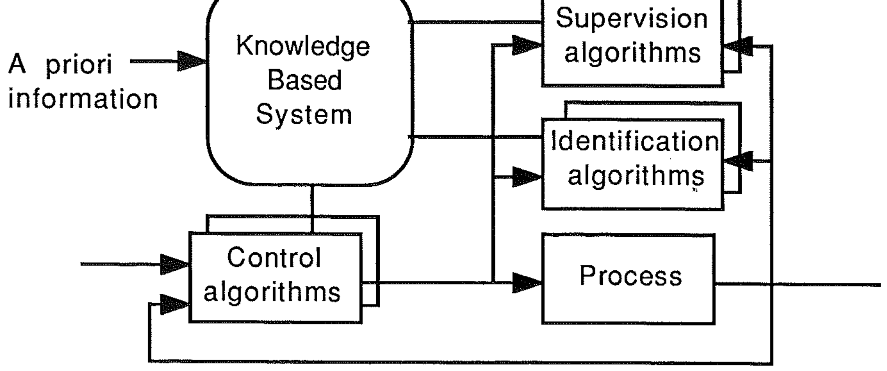

Figure 2.2 A knowledge-based controller

etc. He has qualitative information of the process, e.g., if the open-loop system is stable or highly oscillatory. He usually also has some feeling for the gross nature of the dynamics, e.g., the time scale of the process and the magnitude of the static gain. Information of this kind cannot be trusted too heavily since it is based on subjective judgements. Used with care it can, however, provide useful indications. Information of this kind is usually difficult to directly exploit in conventional controllers.

The knowledge-based approach

This project is based on the hypothesis that a universal adaptive control algorithm without requirements on prior information does not exist or can at least not be implemented in practice. Fundamental theoretical work concerning the amount of process knowledge that is needed to stabilize a process has been done by, e.g., Byrnes (1985) and Mårtensson (1986). Unfortunately, the resulting control algorithms have not proved practically usable. The approach in this project is to use different control algorithms for different purposes and to use different identification and monitoring algorithms, each specialized on one aspect of the process behavior. The overall controller consists of an "intelligent" combination of these algorithms, as shown in Figure 2.2. Such an approach is found in many engineering disciplines. One example is image analysis. When a digital image is analyzed, the standard way is to use different algorithms each one specialized on one aspect of the analysis. Examples are feature extracting algorithms of different types or algorithms for different parts of the image understanding problem.

Functionally, the knowledge-based controller very much resembles a conventional auto-tuner. The difference is that is should be more flexible. The controller should be able to approximate the process dynamics with different model structures, e.g., by either a first-order system, a first-order system with a time delay or a second-order system. Depending on the model structure, the model parameters and the specifications, the controller designs different control laws. This involves both different methods for determining the parameters in a given control algorithm, e.g., a PID controller, and different ways of choosing control structure, e.g., a PID controller or a pole-placement controller.

Another difference is the monitoring aspects. The knowledge-based controller should have the possibility to monitor the steady-state control performance on-line and if necessary adjust parameters, switch between different controllers, or initiate re-tunings.

A possible application for the knowledge-based approach is startup and monitoring of adaptive control laws. Different tuning experiments could be performed to built up enough knowledge to safely start a self-tuning regulator.

2.3 Implementation technique

Heuristics play an important role in many levels of process control. The use of a knowledge-based system as a part of the controller is an attempt to manage the heuristic elements of the controller in a structured way.

The word heuristic has the following interpretation due basically to Brownston *et al* (1985).

Heuristic: *A principle (sometimes called a rule-of-thumb) that embodies some problem-solving knowledge and has some likelihood of successing more rapidly than a theoretically based algorithm for solving the problem, but that is not guaranteed to work in all situations.*

Heuristic elements in process control

Consider, to begin with, a conventional PID controller. It can be described by the model:

$$u(t) = k\Big[e(t) + \frac{1}{T_i}\int^t e(s)ds + T_d\frac{de(t)}{dt}\Big],$$

where $u(t)$ is the controller output and $e(t)$ is the error signal. From this equation, a basic understanding of the algorithm could be gained. To actually implement a PID controller requires much more. First, a realization must be chosen. Commercially available PID controllers may differ quite substantially here. The derivate term may act only on the measured variable, the derivate and the proportional term may act only on the measured variable, the controller could be combined with a lead network, and so on.

The second point to consider is operational issues. It should be possible to switch safely between manual and automatic mode, controller parameter changes must not cause process upsets, precautions against integral wind-up must be taken, and so on. An industrial controller typically consists of an implementation of the basic control algorithm plus heuristic logic that take care of these issues. The logic shows up as selectors or branching statements e.g. *if – then – else* statements and *case* statements. Although these issues are important for good controller performance they are with a few exceptions, e.g., Glattfelder and Schauffelberger (1983), seldom tackled theoretically. Instead they are designed mainly from intuition, experience, and simulation.

In more complex controllers such as, e.g., adaptive controllers, the amount of heuristic logic is increased. Another name for it is safety nets or safety jackets (Clark, 1981; Isermann, 1982; Wittenmark and Åström, 1984). The safety net part of the controller code may be much larger than the actual algorithm. Experience has shown that the safety nets tend to become complex even for rather small problems. A consequence of this is that design and testing is quite time consuming.

As mentioned previously, the choice of controller structure is often based on experience from control of similar loops, i.e., based on heuristics. The controller parameters are adjusted according to heuristic rules-of-thumb. Heuristics plays a role even if the controller parameters are calculated based on some design method. Many possible design methods exist for a given process model and controller structure. The choice of method is not obvious. Most the design methods also contains parameters that must be chosen. Examples of this are given in Chapter 3.

Heuristic considerations play an even more important role on higher levels of the plant. For example, how to pair control signals and measured variables in order to fulfill some global control objective is to a large extent solved by methods based on experience. Multivariable control loops follow the same pattern. How to avoid undesired effects due to couplings between different controllers is one example, (Shinskey, 1986).

The knowledge-based approach

It is the author's experience from implementations of relay based auto-tuners, that the combination of numerics and heuristics in such problems tends to result in code where the numerical parts and the logic are mixed up, when implemented with conventional languages such as, e.g., Pascal or Modula-2. The mixture quickly causes the programs to become difficult to understand and a modular development is difficult to maintain.

The approach taken in the knowledge based controller is to separate the heuristics from the numerics to as a large degree as possible, and to implement the heuristics with expert system techniques. This will give a more structured implementation. The approach taken in the knowledge-based controller, where the overall performance depends on the combination of several algorithms both in sequence and in parallel, increases the logical complexity of the problem and rise even higher demands for a structured implementation.

Expert systems are based on knowledge that consists both of facts and of heuristics. The knowledge is explicitly implemented in a knowledge database that easily can be expanded. An inference engine which draws conclusions based on the available knowledge is implemented separately from the knowledge base. A very common knowledge representation is to use *if* – *then* – rules. In rule-based expert systems, knowledge is represented as collections of rules. This gives both a declarative programming style and a modular system where new rules can be added relatively independently. A description of different expert system techniques is given in Chapter 5. Furthermore, expert systems have the reputation of being suited for solving complex problems dominated by heuristics and where conventional procedural programming techniques have not proved successful. Based on this, expert system techniques seemed to have the qualifications that were needed for the project.

Modes of application

The knowledge-based controller has two potential application modes. It can be used as an actual controller or as a testbench for experiments with new controller structures. The current status of expert system hardware does not allow an expert system based controller to be used on a wider level. This is, however, changing rapidly as computing power increases and expert system tools become available also on small standard processors.

To use the system as an laboratory testbench has many interesting possibilities. The rapid prototyping possibilities of expert systems makes it a suitable environment for experiments with new controller structures. This would especially involve the development and testing of the logic safety nets. When a relatively stable system has been developed and the need to further extend the system is small, parts of it or the whole system may be implemented with conventional programming techniques. The knowledge-based implementation could here provide good guidance on how to choose abstract data types or objects.

Alternative implementation methods

One implementation alternative is to use an available distributed control system. Distributed control systems often consist of a combination of a function block language and a programmable logic controller (PLC). The PLC part typically implements the sequence control problems with logic functions, timers and switches. The function block language implements the continuous control problems with standard blocks for control functions frequently used. These systems are, however, mainly intended for large control problems where the individual controllers are implemented in one or perhaps a few blocks and the PLC system is used to implement the logical connections among the different control loops. The knowledge-based controller instead is a very complex single controller. If this was to be implemented with a commercial control system, there are two possible solutions which reflect the way function block languages can be classified.

The first group of function block languages are the one where the function blocks are small. Each function block performs only a low-level operation. Single blocks are combined to form a functional module. The knowledge-based controller would in such a system consist of a very large number of combined function blocks with the branching logic implemented in the PLC system. The logic safety net part of the controller typically requires a richer expressibility than is provided by a standard PLC system. This and the mere number of function blocks needed, make the solution intractable.

The second group of function block languages contains large, specialized function blocks. In some of these languages, it is possible for the user to program the function blocks. The languages provided for this are usually either languages common in process industry already, e.g., FORTRAN or interpreted languages such as BASIC or FORTH. To implement a knowledge-based controller in this way would, however, probably create the same or even worse problems as if a high-level, real-time language was used from the start.

2.4 User interface aspects

The knowledge-based approach has interesting potentials for the user interface. Ideally, a knowledge-based controller would start by giving the human operator the possibility to enter his process knowledge. This can be done in different ways. The traditional expert system approach is to ask questions to the user. A less rigid form of interaction would be to allow the operator to enter the information on his own initiative. If questions are used, they must be related to the language and the mental models the operator uses for the process. Explanations must be provided for each question. The explanations could also contain guidelines for how the required information could be gathered or calculated if it is not known by the operator. Explanation facilities would give a tutoring facility that perhaps could be used for educational purposes. The operator must always have the possibility to refrain from answering the questions. The decision of what information that is needed and what information the knowledge-based system could take advantage of is not evident. It is also difficult to compose relevant questions that extracts this information from the operator.

A second important source of information is the control specifications. The knobs on a conventional controller are usually controller specific parameters, e.g., regulator gain, integration and derivative times in a PID controller. On a few controllers it is possible to use the knobs to specify the closed loop performance, e.g., bandwidth or overshoot (Åström, 1979). A desired feature of a knowledge-based controller would be the possibility to enter qualitative measures of the closed loop performance, e.g., as high bandwidth as possible. It is then the task of the controller to calculate the control parameters to fulfil this, with the constraints given by, e.g., controller saturations, acceptable overshoot, etc.

The knowledge based controller must also be able to explain its operation to the user, i.e., be able to give answers on various questions. The explicit process knowledge representation simplifies the implementation of extensive query facilities. Possible question types that might be supported are:

What process model have you come up with?

Which control design method was used and why?

Is the disturbance situation normal right now?

What is the variance of the control error?

These examples are mainly intended for the process engineer. The knowledge-based approach also has other implications for the interface to the process and control engineers. The knowledge-based approach allows direct interaction with the knowledge-based system. It may be possible to inspect and manipulate the contents of the knowledge-base during operation. This could, e.g., involve the addition of new rules to the system.

3

Tuning Methods

The automatic acquisition of process knowledge through different tuning experiments is an essential part of the knowledge-based controller. Tuning is also a good example of a problem where numerical algorithms are combined with heuristic logic. Theoretical results exist, but are not complete. Heuristic rules based on experience and sound engineering are used to cover the theoretical gaps. This chapter concentrates on relay based tuning. Theoretical results are presented which can be used as a partial basis for the classification of processes according to their dynamics.

The process knowledge needed, in order to design a controller for a given process, basically consists of knowledge about the process dynamics and about the disturbances that affects the process. The process dynamics may include linear as well as non-linear dynamics. This chapter mainly considers automatic tuning methods for acquiring knowledge about the linear process dynamics. Non-linear dynamics is approached via linearization around different operating points.

Disturbances are basically of two different kinds: low-frequency load disturbances and high-frequency measurement noise. A more exact knowledge about the disturbances may be used to improve the control. Internal disturbance models may be used by the controller for disturbance compensation. For example, a sinusoidal signal of known frequency that disturbs the system can easily be compensated for. Typically, disturbance compensation of this kind requires accurate process models.

Systems can be classified according to their dynamics. Strictly positive real systems are one class. Such systems are easy to control since they are stable under proportional feedback with arbitrarily high gain. Systems with a positive impulse response, or equivalently, a monotonously increasing step response, are easy to deal with if they don't have too large time-delays. Such systems appear naturally when describing, e.g., mixing phenomena. Simple and safe design methods exist

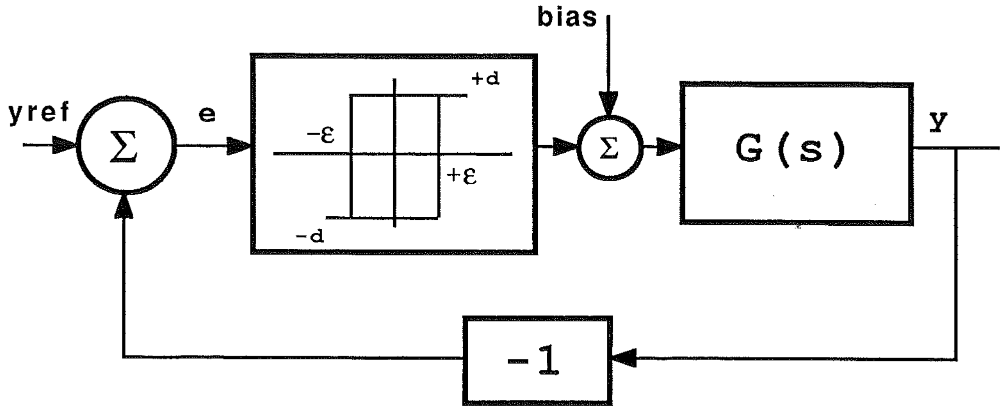

Figure 3.1 Relay feedback

for this class. Systems with time-delays or with zeros in the right half plane impose constraints on the achievable control performance.

The majority of the control loops in industry contain PID controllers. Proportional control is basically sufficient only for systems that can be well approximated by a first order system. Derivative action is typically needed when a second order process model must be used. For systems with a dominating time delay, derivative action does not help much. Instead, a PI controller with small gain in order to maintain stability, and a large integral part must be used. For these systems much can be gained by using a controller with dead-time compensation, e.g., a discrete time pole-placement controller. Systems with poorly damped oscillatory modes are another class of systems that are difficult to handle with PID controllers. Notch filters are typically used to reduce the signal transmission at the resonance frequency.

3.1 The relay tuning method

The idea behind the relay tuning method is to introduce a relay in the feedback loop. This will in most cases cause the system to oscillate. Measurements of the oscillations give information about the process dynamics that can be used to compute the appropriate controller parameters.

The idea of using a relay for tuning purposes was proposed by Åström (1982). Åström and Hägglund have further developed the ideas in Hägglund (1981), Åström and Hägglund (1984a), (1984b), and Hägglund and Åström (1985). The work on relay auto-tuning was actually the seed of the work on knowledge-based control and this thesis. Early work along these lines is described in Åström (1983).

A process under relay feedback is shown in Figure 3.1 where d is the relay amplitude and ε is the relay hysteresis. The process information that can be acquired from a relay experiment is basically knowledge of one point on the open-loop Nyquist curve of the process. In many cases, this point is the intersection of the Nyquist curve with the negative real axis. This point is usually described by the ultimate gain k_c and the ultimate period t_c. Knowledge of this point is important since it is the basis for the Ziegler-Nichols scheme, (Ziegler and Nichols, 1943), for determining PID parameters.

Describing function methods

An approximative analysis of a system under relay feedback can be done using describing function techniques, e.g., Atherton (1975). In the describing function method, the non-linearity is replaced by its describing function. The describing function is defined as the fundamental component of the Fourier series expansion of the nonlinearity output divided by the sinusoidal input. The describing function becomes a function of the amplitude of the sinusoidal input and is denoted $N(a)$ where a is the amplitude of the input. The existence of stable limit cycles can be predicted using the Nyquist stability criterion. It is done by plotting the values of $-1/N(a)$ in the s-plane and examining the intersections with the process Nyquist curve.

Consider the ideal relay, i.e., $\varepsilon = 0$. The describing function is

$$N(a) = \frac{4d}{\pi a}$$

The value of $-1/N(a)$ is thus the negative real axis. Information about other points on the Nyquist curve can be obtained by connecting a system with known dynamics between the relay and the process. For example, an integrator gives information about the point where the Nyquist curve intersects the negative imaginary axis.

A relay with hysteresis has the describing function

$$N(a) = \frac{4d}{\pi a^2}(\sqrt{a^2 - \varepsilon^2} - i\varepsilon) \qquad ; \qquad a \geq \varepsilon$$

The value of $-1/N(a)$ is in this case a straight line parallel to the negative real axis and with the imaginary value $-\pi\varepsilon/4d$. Using a relay with positive hysteresis, it is possible to extract knowledge about points on the Nyquist curve of the open-loop process that lie in the third quadrant. The choice of d and ε is constrained by the dynamics of the process. For example, for a process with monotone step response the following condition must hold for the relay to start oscillate, (Åström and Hägglund, 1984b).

$$\frac{\varepsilon}{d} \leq G(0) \qquad G(0) > 0$$

Describing function methods are approximative and in particular require that the process has low-pass filter characteristics in order to attenuate all harmonics but the fundamental. This is not always the case. Consider for example a first order system, $G(s) = k/(s+a)$. The Nyquist curve for this system lies entirely in the fourth quadrant. The describing function method, thus, does not predict any oscillations. In spite of this, first order systems do oscillate under relay feedback with positive hysteresis.

Exact solutions

Methods for determining the exact oscillation period exist. Tsypkin, (1958, 1984) has given conditions in the frequency domain. Equivalent time domain conditions have been given by Hamel (1949).

Using notation from Atherton (1975), Tsypkin's conditions for symmetric relay oscillations with oscillation period ω are

$$\frac{4d\omega}{\pi}\sum_{n=0}^{\infty}\mathrm{Re}\{G_1(i(2n+1)\omega)\} - d\lim_{s\to\infty} sG_1(s) < 0 \tag{3.1}$$

and

$$\frac{4d}{\pi}\sum_{n=0}^{\infty}\frac{\mathrm{Im}\{G_1(i(2n+1)\omega)\}}{(2n+1)} - dG(\infty) = -\varepsilon \tag{3.2}$$

where $G_1(s)$ is the strictly proper part of $G(s)$, i.e.,

$$G(s) = G_1(s) + G(\infty)$$

The first condition says that the derivative of the output should be negative immediately before the relay switches from negative to positive. The second condition says that the output should have the value $-\varepsilon$ when the relay switches from negative to positive. Tsypkin has defined a locus, $\Lambda(\omega)$, which is an analog of the Nyquist locus, as follows:

$$\mathrm{Re}\{\Lambda(\omega)\} = \sum_{n=0}^{\infty}\mathrm{Re}\{G_1(i(2n+1)\omega)\}$$
$$\mathrm{Im}\{\Lambda(\omega)\} = \sum_{n=0}^{\infty}\frac{\mathrm{Im}\{G_1(i(2n+1)\omega)\}}{(2n+1)}$$

Using $\Lambda(\omega)$, the conditions for limit cycle oscillations can be reformulated as

$$\mathrm{Re}\{\Lambda(\omega)\} < \frac{\pi}{4\omega}\lim_{s\to\infty} sG_1(s)$$
$$\mathrm{Im}\{\Lambda(\omega)\} = \frac{\pi}{4}\{G(\infty) - \frac{\varepsilon}{d}\}$$

Åström and Hägglund (1984a) have given a condition that is equal to condition (3.2) for systems without direct term, i.e., $G(\infty) = 0$. The condition is derived from the observation that the oscillating system can be described as a discrete time system that is sampled at the relay switching times. The condition for symmetric oscillations is

$$H(T/2, -1) = -\frac{\varepsilon}{d}$$

where $H(h, z)$ is the pulse transfer function for zero-order-hold sampling of $G(s)$ with sampling period h, and where T is the oscillation period.

The complete sampled system conditions equivalent to (3.1) and (3.2) can be formulated as follows. Let the system $G(s) = G_1(s) + G(\infty)$ be controlled by a relay with hysteresis, i.e.

$$u(t) = \begin{cases} d & \text{if } e > \ \ \varepsilon \text{ or } (e > -\varepsilon \text{ and } u(t-) = \ \ d) \\ -d & \text{if } e < -\varepsilon \text{ or } (e < \ \ \varepsilon \text{ and } u(t-) = -d) \end{cases} \tag{3.3}$$

where $e = -y$.

THEOREM 3.1 Consider the system $G(s) = G_1(s) + G(\infty)$ with the feedback law (3.3). Assume that the closed loop system oscillates with a stable limit cycle with period T. Assume further that the oscillation is symmetric, i.e. $u(t+T/2) = -u(t)$ and $y(t+T/2) = -y(t)$. It then follows that

$$\widehat{H}_1(T/2, -1) - 2 \lim_{s \to \infty} sG_1(s) < 0 \tag{3.4}$$

and

$$H_1(T/2, -1) - G(\infty) = -\frac{\varepsilon}{d} \tag{3.5}$$

where $H_1(h, z)$ is the pulse transfer function for zero-order-hold sampling of the system $G_1(s)$ with sampling interval h and $\widehat{H}_1(h, z)$ is the pulse transfer function for zero-order-hold sampling of the system $sG_1(s)$.

Proof: The proof is based on the relation between the z-transform of a continuous function and its Laplace transform, (Jury, 1964). When the system contains a direct term this relation is

$$H(h, e^{sh}) = \frac{1}{h} \sum_{-\infty}^{\infty} F(s + in\omega_s) \ + \frac{1}{2} f(0+)$$

where

$$F(s) = \frac{1 - e^{-sh}}{s} G(s)$$

and $f(t)$ is the corresponding time function. The term $\frac{1}{2} f(0+)$ is omitted if the system has no direct term.

In condition (3.4), it follows that

$$\widehat{H}_1(h,e^{sh}) = \frac{1}{h}\sum_{-\infty}^{\infty}(1-e^{-h(s+in\omega_s)})G_1(s+in\omega_s) + \frac{1}{2}\lim_{s\to\infty} s(1-e^{-sh})G_1(s)$$

Using the fact that

$$\omega_s = \frac{2\pi}{h}$$
$$e^{sh} = z$$

and by putting $z=-1$, i.e., $s=i\pi/h$, this can be written as

$$\begin{aligned}\widehat{H}_1(h,-1) &= \frac{1}{h}\sum_{-\infty}^{\infty} 2\mathrm{Re}\{G_1((2n+1)\frac{\pi}{h}i)\} + \frac{1}{2}\lim_{s\to\infty} s(1-(-1))G_1(s)\\ &= \frac{4\omega}{\pi}\sum_{0}^{\infty}\mathrm{Re}\{G_1((2n+1)\omega i)\} + \lim_{s\to\infty} sG_1(s)\end{aligned}$$

where

$$\omega = \frac{\pi}{h}$$

From this and since $d > 0$, it follows that condition (3.4) is equivalent to condition (3.1).

In condition (3.5), it follows that,

$$H_1(h,e^{sh}) = \frac{1}{h}\sum_{-\infty}^{\infty}\frac{(1-e^{-h(s+in\omega_s)})}{(s+in\omega_s)}G_1(s+in\omega_s)$$

Using the fact that

$$\omega_s = \frac{2\pi}{h}$$
$$e^{sh} = z$$

and by putting $z=-1$, i.e., $s=i\pi/h$, this can be written as

$$\frac{1}{h}\sum_{-\infty}^{\infty}\frac{2}{\frac{\pi}{h}(2n+1)}\mathrm{Im}\{G_1((2n+1)\frac{\pi}{h}i)\} = \frac{4}{\pi}\sum_{0}^{\infty}\frac{\mathrm{Im}\{G_1((2n+1)\omega i)\}}{(2n+1)}$$

where

$$\omega = \frac{\pi}{h}$$

From this it follows that condition (3.5) is equivalent to condition (3.2). □

Remark 1. Using relay feedback for systems with direct term is rather unrealistic. An algebraic loop occurs if $G(\infty)d > \varepsilon$.

Conditions for local stability of the oscillations are also available, (Tsypkin 1984). A frequency domain criteria is

$$\frac{d\mathrm{Im}\{\Lambda(\omega)\}}{d\omega} > 0$$

where ω is the oscillation frequency. A corresponding time domain version also exists (Åström and Hägglund, 1984a). The proofs of these conditions are based on local analysis.

Common for all the exact methods is that they only give necessary conditions. Sufficient conditions do not seem to exist yet.

Theorems for the oscillation curve form

A collection of additional results for symmetric relay oscillations will be presented. The previous results only concerned the oscillation period T. The new results also describe the curve form of the limit cycles. The proofs of the two theorems are based on state-space formalism. Let the system $G(s)$ be described by the minimal realization

$$\mathcal{S}(A, B, C, D) \tag{3.6}$$

THEOREM 3.2 Consider the system (3.6) with the feedback law (3.3). Assume that the closed loop system oscillates with a stable limit cycle with period T. Assume further that the oscillation is symmetric, i.e. $u(t + T/2) = -u(t)$ and $y(t + T/2) = -y(t)$. It then follows for any τ, $0 < \tau < T/2$, that

$$H_\tau(T/2, -1) = \frac{y_\tau}{d} \tag{3.7}$$

where y_τ is the y value the time τ after the relay has switched from plus to minus and $H_\tau(h, z)$ is the pulse transfer function for zero-order-hold sampling of the system $\mathcal{S}(A, B, C, D)$ with the input delayed $T/2 - \tau$.

Proof: The general form of the signals u and y are shown in Figure 3.2. From the symmetry and the minimal realization of (3.6), it follows that $x(t) = -x(t-T/2)$. Assume that $x(t_{2n}) = b$. It then follows that $x(t_{2n+1}) = -b$ and $x(t_{2n+2}) = b$.

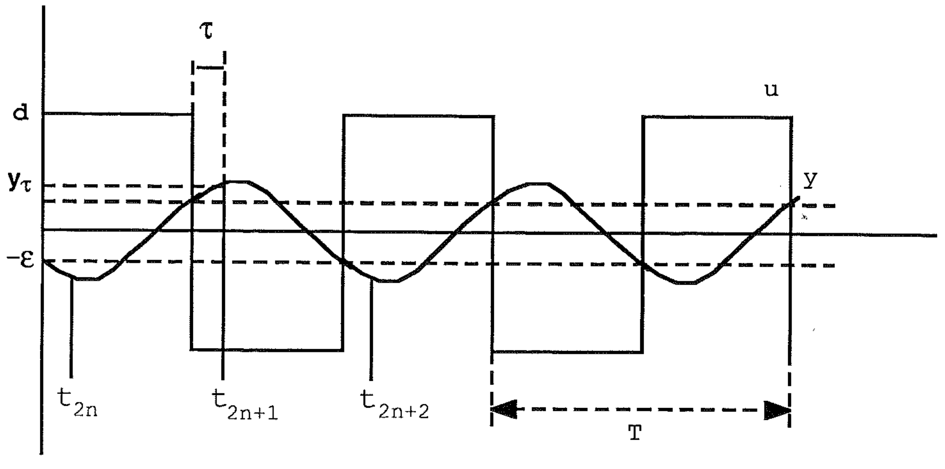

Figure 3.2 Signals under limit cycle conditions

Integration of the state equations over the half period $[t_{2n}, t_{2n+1}]$ gives

$$-b = \Phi^{1/2} b + \Gamma_1 d - \Gamma_0 d$$

where

$$\begin{aligned}\Phi^{1/2} &= e^{AT/2}\\ \Gamma_1 &= \int_\tau^{T/2} e^{As} ds\, B = e^{A\tau} \int_0^{T/2-\tau} e^{As} ds\, B\\ \Gamma_0 &= \int_0^\tau e^{As} ds\, B\end{aligned}$$

Thus

$$-b = -[I + \Phi^{1/2}]^{-1}[\Gamma_0 - \Gamma_1]d$$

Using the assumption that the output is equal to y_τ at t_{2n+1} it follows that

$$y_\tau = -C[I + \Phi^{1/2}]^{-1}[\Gamma_0 - \Gamma_1]d - Dd$$

The pulse transfer function for system (3.6), with the input delayed $T/2 - \tau$, and sampled with period $T/2$ looks as

$$H_\tau(T/2, z) = C[zI - e^{AT/2}]^{-1}[\Gamma_0 + \Gamma_1 z^{-1}] + Dz^{-1}$$

From this the theorem easily follows. □

THEOREM 3.3 Consider the system (3.6) under the constraint that $D = 0$, together with the feedback law (3.3). Assume that the closed loop system oscillates with a stable, symmetric limit cycle with period T. Assume further that the derivative of the output, $\dot{y}$, has the value $\dot{y}_\tau$, τ units after a relay switch from plus to minus where $0 < \tau < T/2$. It then follows that

$$\widehat{H}_\tau(T/2, -1) = \frac{\dot{y}_\tau}{d} \qquad (3.8),$$

where $\widehat{H}_\tau(h, z)$ is the pulse transfer function for zero-order-hold sampling of the system $\mathcal{S}(A, B, CA, CB)$ with the input delayed $T/2 - \tau$.

Proof: The preceding theorem can be used where

$$\dot{y}(t) = C\dot{x}(t) = CAx + CBu$$

is realised as

$$\mathcal{S}(A, B, CA, CB)$$

$\square$

The proof of these theorems are based on state-space formalism and are thus not valid for systems that contain a time delay. The results can, however, be extended to the general case.

Information content

The relay method is primarily used to obtain information about one point on the open-loop Nyquist curve. The oscillation curve form, however, gives additional information about the open-loop dynamics. Figure 3.3 shows the relay output u and the output y from the controlled process for the following standard cases: a first order system, a second order system, a first-order system with time delay and a high frequency roll-off system. The resulting curve forms vary quite significantly. The describing function assumptions are best fulfilled by the high frequency roll-off system. The phase shift is here close to $-180°$ as expected. The first order system has a phase shift of approximately $-90°$ which is natural. An interesting feature of the first order system is that the derivative of the output changes sign at the relay switching time. This feature can be used to detect first order systems or systems that can be successfully approximated by a first order system.

A complete characterization of the class of systems that give a unique stable limit cycle is not available. In Åström and Hägglund (1984b), it is shown that the equation

$$H(T/2, -1) = -\frac{\varepsilon}{d}$$

always has at least one solution for stable systems where $G(s) \to 0$ as $s \to \infty$.

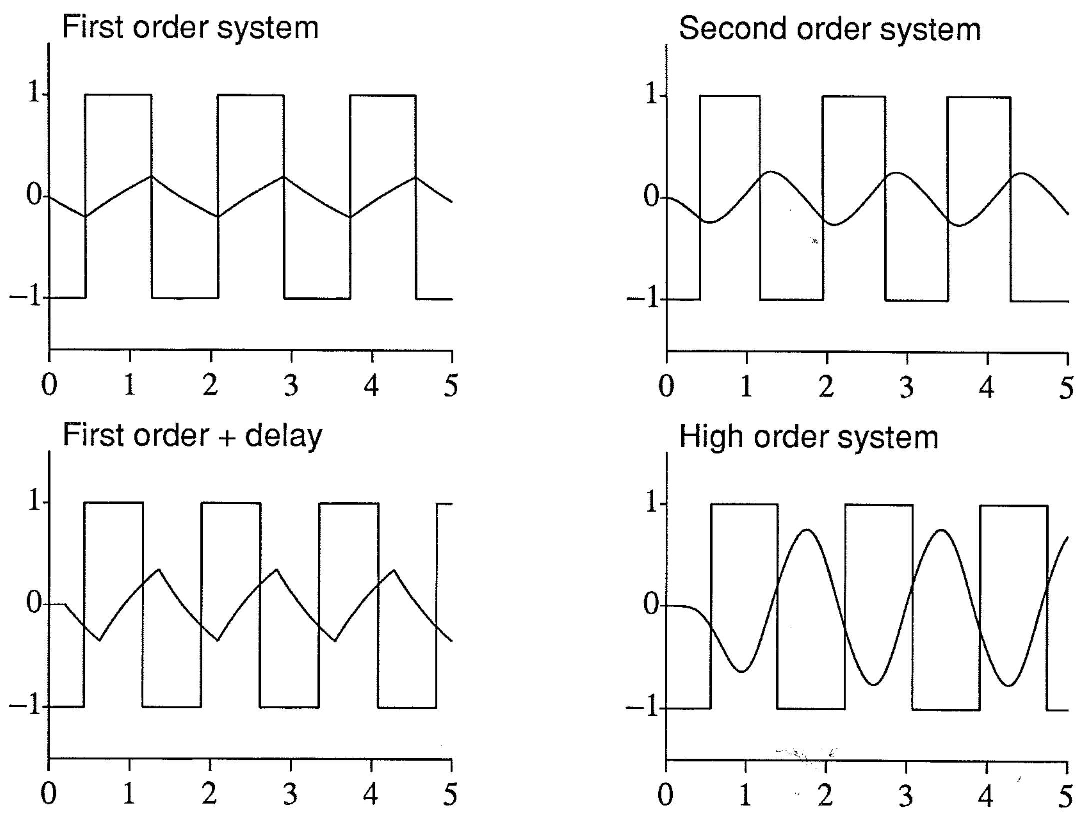

Figure 3.3 Oscillation curve forms. $\varepsilon = 0.2$ and $d = 1$.

In the same paper it is also conjectured that there exists a unique, stable limit cycle for stable systems. This seems, however, not to be true. The simulation curves in Figure 3.4 show a stable system with poorly damped oscillatory modes and a large time delay that oscillates with different periods for different initial values. A theoretical analysis of the case gets very complicated. It may be argued that the simulations may not have reached stationarity. Extended simulations, however, show the same limit cycles. With regard to the relay tuning experiment this does, however, make no difference. Processes exist for which the relay autotuning will have problems.

Stable systems with a monotone step response have a more pleasant behavior. In Åström and Hägglund (1984b), it is shown that they will have a stable limit cycle provided ε is large enough. It is also shown that the oscillation period T must be smaller than the time $2t_0$ where t_0 is given by

$$S(t_0) = \frac{1}{2}(S(\infty) + \frac{\varepsilon}{d})$$

and $S(t)$ denotes the step response of the system.

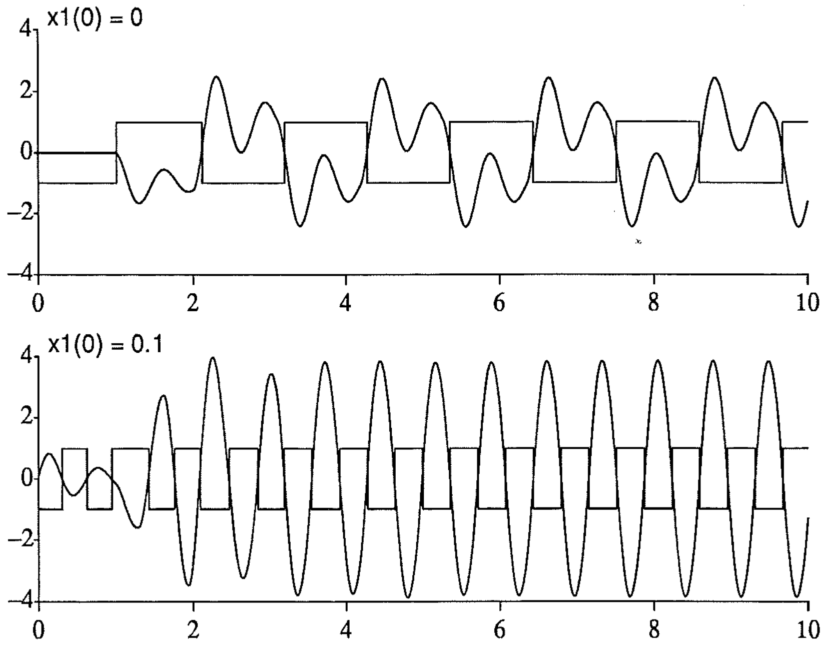

Figure 3.4 Relay control of the system $(s+\omega^2)e^{-sL}/(s^2+s\zeta\omega s+\omega^2)$ with $\omega = 10$, $\zeta = 0.13$ and $L = 1$ for different initial values.

Conjecture 1: If a stable system with monotone step response contains a pure time delay e^{-sL}, it is conjectured that the oscillation period must be larger than $2L$.

The conjecture is motivated as follows. Assume first that the system consists only of a time delay. The oscillation period will then be $2L$. If dynamics with a monotone step response is added to the system, the resulting oscillation period will increase.

The conjecture does not necessarily hold for systems with non-monotone step responses. An example of this is the second curve in Figure 3.4. Systems with poorly damped oscillatory modes are in general difficult to analyze and they may give strange oscillation curve forms. An example of this is shown in Figure 3.5.

The following conjecture can be stated about the approximative phase-shift for a system under relay control.

Conjecture 2: For a minimum-phase system, the derivative of the output will change sign at a time τ after the relay switching time with $0 \leq \tau < T/4$.

Consider first an open-loop system where all but the first harmonics are attenuated, i.e., the output y is a perfect sine. The phase-shift will then approach $-180°$, i.e. $\tau \to T/4$. Higher order harmonics that are not attenuated will cause the oscillation curve form to more and more resemble a square-wave and thus decrease τ. The reverse of the condition is not always true. Systems with small

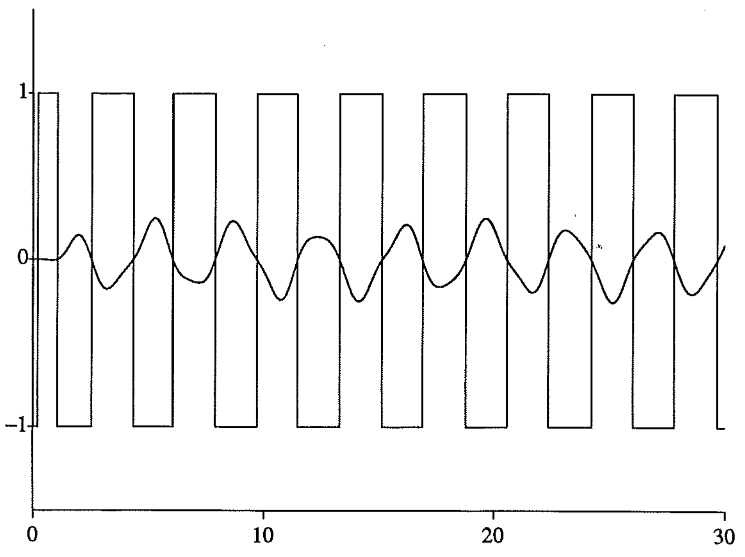

Figure 3.5 Oscillation curve form for the system $G(s) = \omega^2/(s+1)^3(s^2+2\zeta\omega s+\omega^2)$ with $\omega = 4$ and $\zeta = 0.005$.

time delays will, e.g., behave almost as minimum-phase systems. The conjecture will be demonstrated for a very special system.

EXAMPLE 3.1
Consider the following system under relay control.

$$G(s) = \frac{a}{s(s+a)} \qquad a > 0$$

Theorem 3.3 gives an expression for when the output derivative $\dot{y}$ changes sign. The system that should be investigated is

$$sG(s)e^{-s(T/2-\tau)} \qquad 0 < \tau < T/2.$$

The pulse transfer function for this system sampled with $h = T/2$ and evaluated in -1 is

$$H_\tau(T/2,-1) = \frac{e^{-aT/2}(2e^{a(T/2-\tau)}-1)-1}{1+e^{-aT/2}} \equiv f(\tau).$$

This expression can be shown to be a monotonously decreasing function of τ with the following values.

$$\lim_{\tau\to 0} f(\tau) = \frac{1-e^{-aT/2}}{1+e^{-aT/2}} > 0$$
$$f(T/4) = -\frac{(1-e^{aT/4})^2}{1+e^{-aT/2}} < 0$$
$$\lim_{\tau\to T/2} f(\tau) = -\frac{1-e^{-aT/2}}{1+e^{-aT/2}} < 0$$

This shows that the derivative of output signal always changes sign for $\tau \in (0, T/4)$.

Detection of PI-controllable systems

Minimum phase systems with relative degree one form an interesting class. These systems can theoretically be controlled arbitrarily well by constant high gain proportional feedback. In practice, control signal saturation and measurement noise limits the maximum allowed feedback gain and thus integral action might be needed.

PI controllers are also used for control of processes with significant time delays. These systems will not be detected with the method that will be presented. On the other hand dead-time compensation is a better solution for these systems.

Systems having relative degree one can in theory be detected by examining the value of the derivative of the process output at the relay switching times. If the system has relative degree one, the derivative will have a discontinuity. Discontinuities are however very difficult to measure, in particular when they are small.

A characteristic which is easier to measure is whether the derivative of the output changes sign at the relay switching time or not. This determined by the following theorem.

THEOREM 3.4 Consider the system $G(s) = G_1(s) + G(\infty)$ where $G_1(s)$ is strictly proper, under the feedback law (3.3). Assume that the closed loop system oscillates with a stable limit cycle with period T. Assume further that the limit cycle is symmetric, i.e., $u(t + T/2) = -u(t)$ and $y(t + T/2) = -y(t)$. The derivative of the output will change sign at the relay switching time if and only if

$$\widehat{H}(T/2, -1) > 0 \tag{3.9}$$

where $\widehat{H}(h, z)$ is the pulse transfer function for zero-order-hold sampling of the system $sG_1(s)$.

Proof: Let t_{2n} be the time when the relay switches from negative to positive. The value of the derivative at $\dot{y}(t_{2n}-)$ is, according to Tsypkin,

$$\dot{y}(t_{2n}-) = \frac{4d\omega}{\pi} \sum_{n=0}^{\infty} \mathrm{Re}\{G_1(i(2n+1)\omega)\} - d \lim_{s\to\infty} sG_1(s)$$

From condition (3.1), it follows that this value is negative. The derivative will change at the switching time if

$$\lim_{s\to\infty} sG_1(s) \neq 0$$

and the magnitude of the discontinuity will be

$$2d \lim_{s\to\infty} sG_1(s)$$

The value of the derivative immediately after the switch is

$$\begin{aligned}\dot{y}(t_{2n}+) &= \frac{4d\omega}{\pi}\sum_{n=0}^{\infty}\operatorname{Re}\{G_1(i(2n+1)\omega)\} + d\lim_{s\to\infty} sG_1(s)\\ &= \widehat{H}(T/2,-1)d \end{aligned} \tag{3.10}$$

The derivative changes sign if (3.10) is positive. □

Remark. The proof also shows that under symmetric limit cycle conditions

$$\widehat{H}(T/2,-1) > 0 \Rightarrow \lim sG_1(s) > 0$$

The converse is not true. This corresponds to the case when the discontinuity is too small to change the sign of the derivative.

The given result contains the oscillation period as a parameter and thus implicitly depends on the relay characteristics, i.e., the relay amplitude and hysteresis. For certain systems it is, however, possible to show that $\dot{y}$ will change sign at the relay switching time independently of the oscillation period and thus of the relay characteristics.

EXAMPLE 3.2 — SPR systems

Consider a system that is strictly positive real (SPR), (van Valkenburg, 1960 Chapters 3 and 4), and has no direct term. It then follows from Equation (3.10) in the proof of Theorem 3.4 that $\dot{y}$ will change sign at the relay switching times independently of the relay characteristics.

The fact that SPR systems belongs to the class that we can detect is satisfactory. These systems will be stable under proportional feedback for all values of the gain.

EXAMPLE 3.3

Consider the system

$$G(s) = \frac{s+a}{s(s+b)} \qquad a, b > 0.$$

If $a < b$, the system is SPR and $\dot{y}$ will change sign independently of the relay characteristics. When $a = b$, the system is a pure integrator for which $\dot{y}$ always will change sign at the switching time.

If $a > b$, the system is not SPR. The Nyquist curve of the system lies entirely in the third quadrant. Using Theorem 3.4 gives

$$\widehat{H}(h,z) = 1 + \frac{(a-b)(1-e^{-bh})}{b(z-e^{-bh})}$$

and

$$\begin{aligned}\widehat{H}(T/2,-1) &= 1 - \frac{(a-b)(1-e^{-bT/2})}{b(1+e^{-bT/2})}\\ &= 1 - \frac{a-b}{b}\tanh\frac{bT}{4} \equiv f(T)\end{aligned}$$

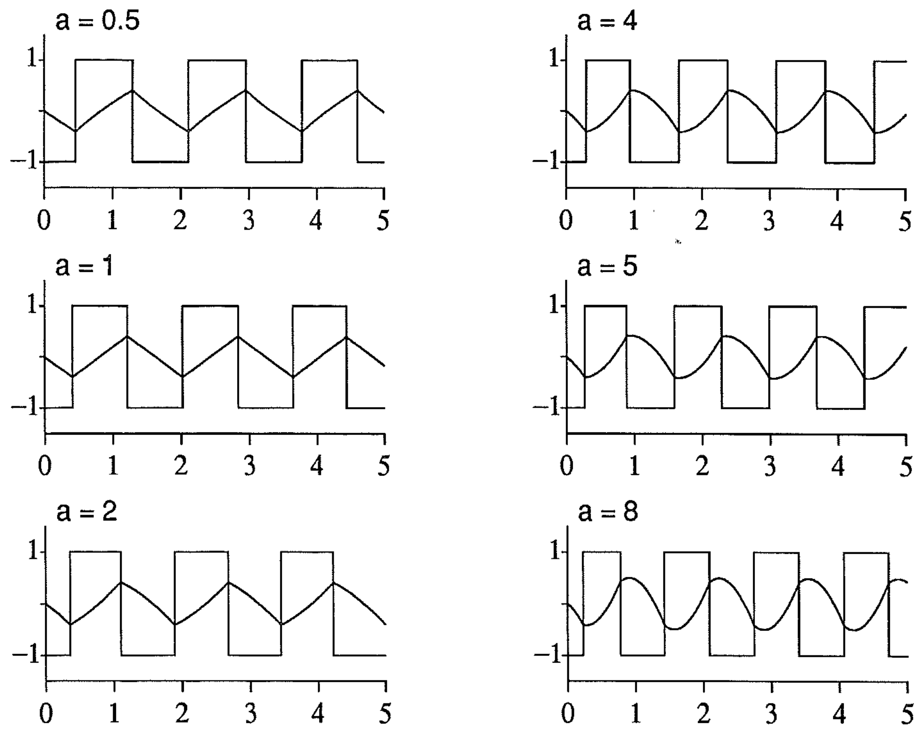

Figure 3.6 Oscillation curves. $b = 1$, $\varepsilon = 0.4$ and $d = 1$.

$f(T)$ is a monotonously decreasing function of T with the limits

$$\begin{aligned} f(0) &= 1 \\ f(\infty) &= 1 - \frac{a-b}{b} = 2 - \frac{a}{b} \end{aligned}$$

It thus follows that if $a/b < 2$ then $\dot{y}$ will change sign at the relay switching times independently of the relay characteristics. If $a/b > 2$, the relay characteristic determines whether $\dot{y}$ changes sign or not. Figure 3.6 shows the oscillation curve forms for different ratios of a/b. The discontinuity of $\dot{y}$ at the switching instants is clearly seen. In this example, the derivate changes sign also for higher values of a/b than 2. The limit value for a/b can be calculated by first computing the limit cycle period as a function of a for the particular values of b, d and ε used in Figure 3.6. This can, e.g., be done with the condition (3.5). In this case, Equation (3.5) has no closed solution. Figure 3.7 shows a plot of the oscillation period T as a function of a. Figure 3.8 shows $f(T(a))$ which has been computed using the plot in Figure 3.7. From this it is seen that $f(T(a))$ is positive for approximately $a < 3.9$.

These examples and the condition (3.10) indicate that the changing of sign at the relay switching times depends on how far into the left half plane the Nyquist

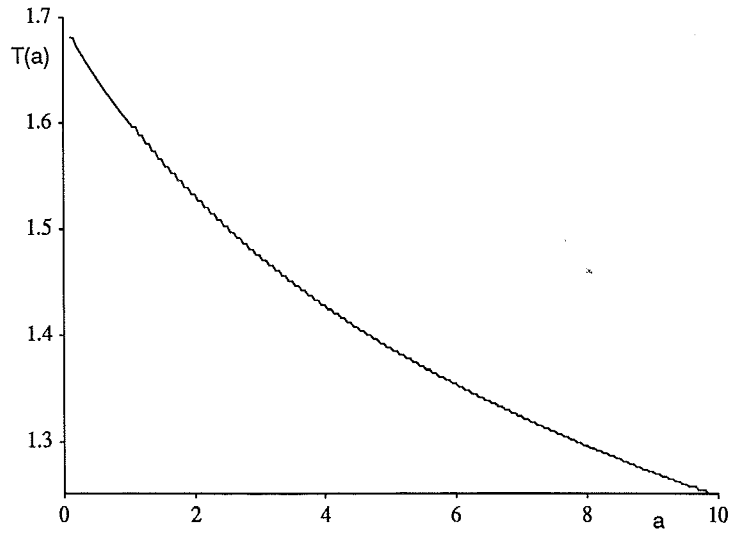

Figure 3.7 $T(a)$

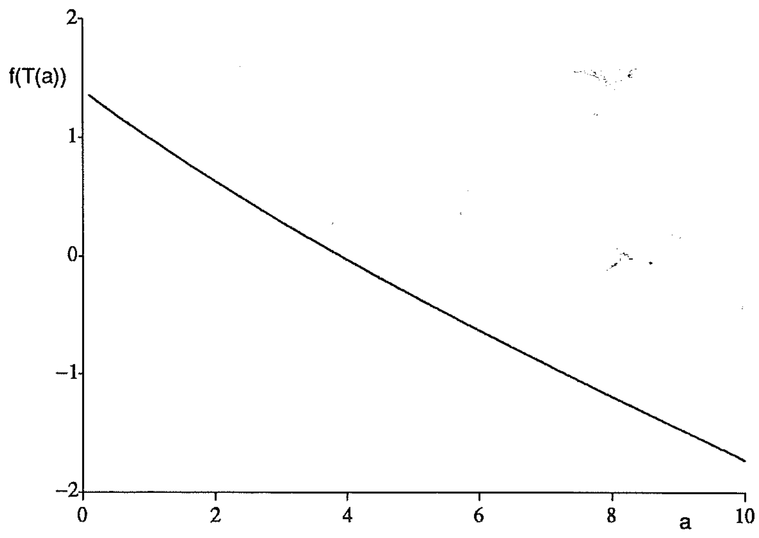

Figure 3.8 $f(T(a))$

curve, and thus Tsypkin's Λ-locus, lies for the frequencies above the oscillation frequency. The Nyquist curve and Λ-locus for the system in Example 3.3 are shown in Figure 3.9. The diagrams are plotted for the values $a = 0.5, 1.5, 2, 4, 6, 8$. In each diagram, the leftmost curve has the value $a = 0.5$.

Theorem 3.4 is applies also to systems that cannot be controlled with a PI controller. For example, non-minimum phase systems with zeros in the right half plane may satisfy the theorem if $\lim_{s\to\infty} sG_1(s) > 0$. Such systems will

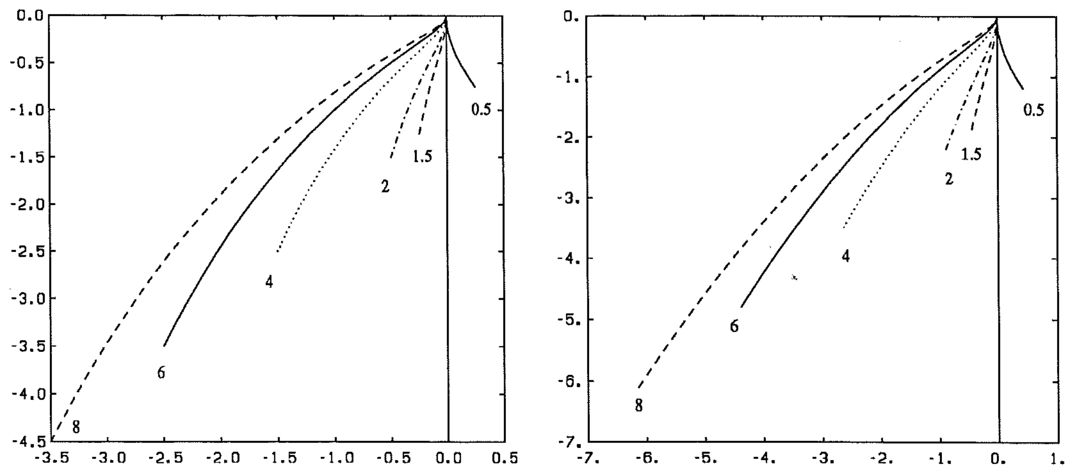

Figure 3.9 Nyquist curve and Λ-locus for different values of a.

become unstable with high gain proportional feedback. It is also possible to find minimum-phase systems with relative degree one for which P or PI control is inadequate.

Based on the insight derived from the theoretical analysis the follwing empirical rule is used to classify a system as PI-controllable. If the time it takes for the signal y to reach its peak value after a relay switch is close to zero, the process is considered as PI-controllable. This rule reduces the risk of erronously detecting systems with non-minimum phase zeros. The oscillation curve for a non-minimum phase system where the derivative changes sign at the relay switching time is shown in Figure 3.10. This system will not be classified as PI-controllable.

Control design methods

Several control design methods that are based on relay oscillations exist. Several of them will be used in the experiments in Chapter 8. A survey of the different methods is given in Appendix A.

Practical considerations

The relay experiment requires a relatively small amount of prior information. The input parameters that must be given is the initial value of the relay hysteresis, the maximum allowed relay amplitude gain, the bias level and the desired oscillation amplitude. The latter is needed in order to ensure that the oscillation exceeds the measurement noise level. After a few half-periods the oscillation amplitude is compared to the desired amplitude and, if needed, the relay amplitude and hysteresis are adjusted. A correct bias value is needed in order to get symmetric oscillations. Experience has, however, shown that this is not crucial. In most cases the method works well even if the oscillations are slightly unsymmetric.

The relay method is, as most estimation techniques, sensitive to load disturbances that occur during the experiment. The occurrence of a load disturbance

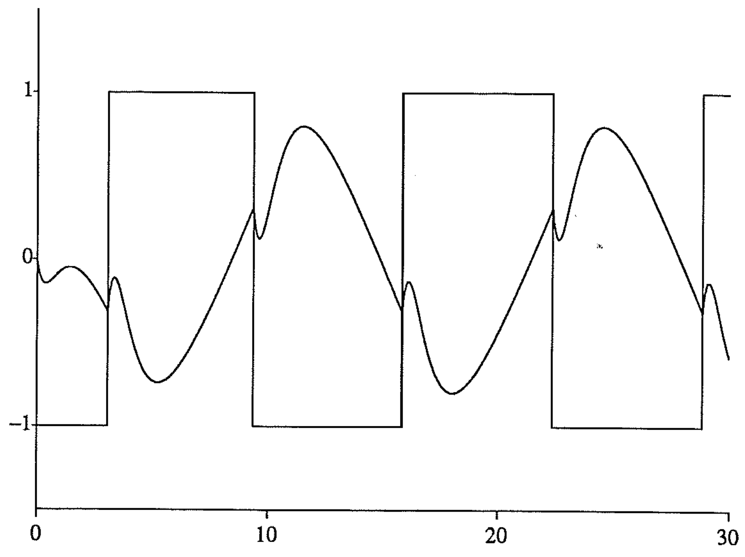

Figure 3.10 Relay oscillation curve form for a non-minimum phase system with $\lim_{s \to \infty} sG_1(s) > 0$.

can, however, be detected. When a load disturbance occurs, the relay switching period is disturbed, i.e., the relay does not change sign when it is expected to. This could then be reported to the supervisory logic. The presence of sinusoidal load disturbances will still give rise to difficulties.

3.2 Step and pulse response methods

An alternative method to determine the dynamics of a process is to examine its response to a deterministic signal such as a step or a pulse. A prerequisite for these methods is that the process is in stationarity and that no drastic disturbances influence the process. Step and pulse experiments are usually performed on the open-loop system. The deterministic signal is entered in the process input and different features of the response are measured. The measurements are either used to fit a process model or for direct calculation of the controller parameters in, e.g., a PID controller.

The step and pulse response methods are, at least in principal, easy to carry out. Many characteristics of the process dynamics can be read out directly from the response, e.g., the static gain and whether the process has oscillatory modes or not. A drawback with the step and pulse response methods is that the time scale of the process must be known in advance. This is necessary in order to determine when the transient response has died out and, thus, when the measurements can be finished.

The step and pulse response methods can also be performed on the closed-loop system. In that case, the deterministic signal is entered as a change in reference value. The complexity of the computations is, however, increased since the controller dynamics must be taken into account.

Several different versions of step and pulse response methods have been proposed in the literature. An overview of the different approaches will be given in Appendix B.

3.3 Summary

This chapter together with Appendices A and B describe alternative tuning methods. A method which not has been described is to use full-fledged parameter estimation with proper selection of input signals and validations. This is, e.g., used in Turnbull Control Systems' TCS 6355 auto-tuner and Leeds & Northrup's Electromax V. The method gives good results. It needs, however, a significant amount of prior information. The commercial systems have therefore added a "pre-tune" mode to acquire the prior information.

The described methods are mainly useful for the initial tuning phase where enough process knowledge is gathered in order to choose a controller structure and compute the controller parameters. In order to handle varying process dynamics the controller must be combined with one or several monitoring algorithms that may initiate a re-tuning if the control performance deteriorates.

A common characteristic for the different tuning methods is that they do not give a single answer on how a process model should be chosen and how the controller should be designed. Instead, some methods perform better for some systems and other methods perform better for other systems. Disturbances may affect the tunings and different methods may give contradictory results. All in all, the tuning phase contains a large amount of heuristics. In Chapter 8, experimental results will be shown where several of the described methods are used.

4

Implementation Structure

The knowledge-based controller is an example of how to separate the logic safety net from the pure numerical algorithms and to implement it with expert system techniques. This chapter is devoted to the question of how this separation should be done.

A number of prototype systems have been developed in the project. The basic architecture has remained the same. What has changed is the internal structure of the knowledge-based system. Chapter 6 describes a prototype system where an off-the-shelf expert system framework was used to implement the knowledge-based system and Chapter 7 describes a real-time expert system framework dedicated for knowledge-based control.

The prototypes have all been implemented on a VAX 11/780 running under VMS. The system is divided into a number of subprocesses running concurrently. The subprocesses communicate by sending messages through mailboxes. The main subprocesses are the numerical algorithms, the knowledge-based system and the man-machine communication. This chapter gives motivation for the chosen division and discusses its implementation. Section 4.1 discusses the separation of the problem into different processes. The division between numerical algorithms and symbolical computations is given special treatment. In Section 4.2, the communication between the processes is described. Section 4.3 describes the internal structure of the algorithm process.

4.1 Subprocess decomposition

Applications where numerical and symbolical computations are combined have attracted increased attention in the expert system community lately. Mimicing the expertize of e.g. an experienced engineer almost inevitably requires that the expert system is combined with programs for numerical calculations. In spite of this it is not clear how symbolical and numerical computations should be combined. Most expert system applications are purely symbolical stand-alone systems that interact only with the human user.

Several different approaches for the combination exist. It is often possible for expert systems to call routines written in conventional languages. In this way, numerical, calculation extensive tasks could be implemented in conventional procedural languages. The opposite solution is to embed an expert system in a conventional language such as Pascal or C.

Another way of combination is to use an expert system as an intelligent front-end (Carroll and McKendree, 1987; Larsson and Persson, 1987; Gale and Pregibon, 1982) to existing numerical programs. The expert system can act as an goal-directed help system that gives practical advice on both how to use the actual program and how to correctly apply the program and interpret its results.

The described approaches are, however not suitable for this project. In knowledge-based control, the combination of the expert system and the numerical algorithms must meet certain real-time demands. The execution of rules in an expert system is basically a large search problem and as such typically slow compared to the time scales determined by the controlled process. It is also less deterministic with respect to timing. For example, the control algorithm must be able to compute a control output at a fixed sampling interval and cannot be halted by the expert system searching among different rules. The solution is therefore to implement the separate parts as communicating concurrent processes with different priorities. The numerical algorithms have the highest priority in order to ensure that the controller is not delayed. The implementation structure is described in Figure 4.1. The man-machine communication is implemented as a separate subprocess. This process provides the user interface. Currently, the interface is purely alpha-numerical using a VT-100 terminal. From the interface the user can interact directly with the knowledge-based system and indirectly with the different algorithms, e.g., manually change controller parameters or switch between different modes, This is described in more detail in Section 8.4.

Implementation languages

The knowledge-based system is implemented in Lisp and the numerical algorithms are implemented in Pascal. The reasons for using different languages are numerous. The majority of existing expert systems and programming tools for expert systems are written in Lisp. Using Lisp for the knowledge-based process in this project makes it possible to use existing software products or parts of such in the system.

There is no impediment in principle to writing an expert system in a conventional language. Two approaches can be used. The first is to hard-code the rules of the expert system as program statements in the language. By doing this, however, much is lost. The modular, incremental nature of expert systems

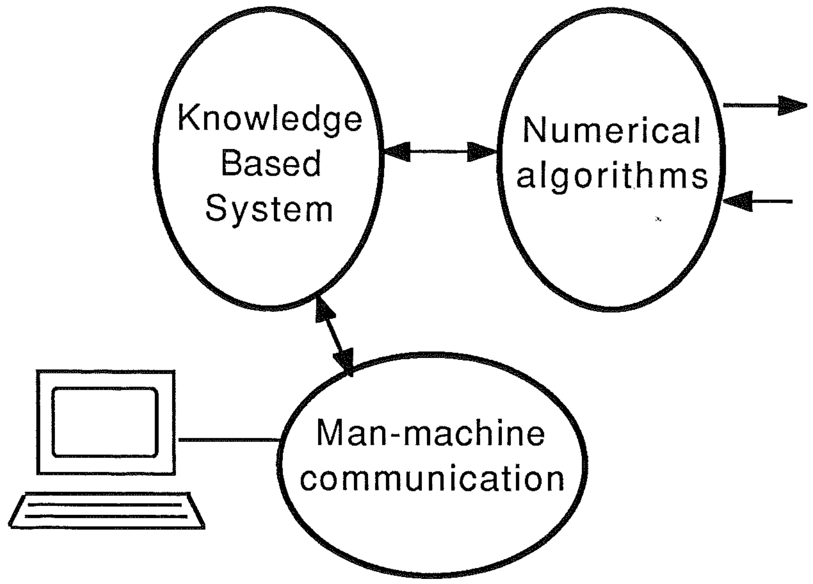

Figure 4.1 Implementation structure.

disappears. The control structure of the implementation also becomes different. The data-driven feature of forward chaining systems is, e.g., difficult to emulate.

The second approach is to implement a rule interpreter in the conventional language. This, however, tends to force the implementation of several features, important for symbolic processing, already available in Lisp, e.g., dynamic memory allocation combined with garbage collection. The advantages of conventional languages are then to a large extent lost. The programming environments available for conventional languages are also in general much inferior to what is available in Lisp systems.

An objection against Lisp is that it is slow and difficult to integrate with existing software. The first objection is principally true. Programming languages with run-time type checking are slower than languages with compile-time type checking at least when executed on standard processors. The development of dedicated Lisp processors has, however, resulted in powerful Lisp workstations that can execute Lisp at high speed. Good Lisp implementations are also evolving for small computers like, e.g., the IBM PC. The trend among expert system framework developers is to supply delivery versions of their frameworks written in conventional languages for standard computers. The development of the expert system is meant to take place on Lisp machines with powerful programming environments. When a system is ready, systems based on conventional languages can serve as delivery vehicles with smaller demands on, e.g., modifiability.

The Lisp used in the project is the Unix dialect Franz Lisp, (Foderaro *et al*, 1983). The software package EUNICE, (Kashtan, 1982), is used to create a

Unix environment under VMS. EUNICE functions as an interface that performs the necessary translations between applications written for Unix and the VMS operating system. It is, e.g., possible both to use VMS system services and Unix system services in an EUNICE application program. This combination of different operating system has been somewhat of a nuisance during the project. The reason is the EUNICE system which has not been fully reliable. The combination was, however, necessary since no good Lisp implementation for VMS was available to us when the project started.

The reason for the use of a conventional language in the algorithm part is mainly that most numerical algorithms are coded in conventional languages. By using Pascal, existing algorithms can easily be included in the system. The use of a compiled language gives a less interactive environment than what had been the case if, e.g., Lisp had been used. It is, however, favorable with respect to execution speed. One of the assumptions of the project has been that it is the heuristic logic that is the most difficult part to develop and where the need for an interactive environment is most evident. In a different situation that, e.g., was focused more on algorithm development, the considerations would be different.

4.2 Inter-process communication

The knowledge-based control task contains different types of information of both symbolical and numerical nature. Symbolic information could, e.g., be process models structures, information about the outcome and parameter settings of different numerical algorithms, etc. Measured signal values and computed control signals are examples of numerical information.

The approach behind the system is to store the symbolic information in the knowledge-based system and the numerical information in the numerical algorithm process. In some cases, it would be useful to be able to store information in shared memory. An example of such information is stored values of old measurements and control signals. In the VAX implementation, however, this is difficult. Instead the information is either stored in both places or exchanged when necessary.

Communication between the different processes occurs for different reasons. The knowledge based system must be able to notify the algorithms that an algorithm should be started or stopped. It is also necessary for it to be able to give initial values for the different internal parameters in the algorithms as well as having the possibility to change the parameters while the algorithms are running. The knowledge-based system must also have the possibility to request the algorithms for information, i.e., ask questions to them and await their answers.

The numerical algorithms on the other hand, must consequently have the possibility to deliver answers to the knowledge-based systems. They must also be able to report information on their own initiative, e.g., the results of some algorithm.

The man-machine communication must be able to transfer commands to the knowledge-based systems and to the algorithms. These subprocesses must, accordingly, have the possibility to return results to the human user.

VMS communication facilities

The operating system VMS has three different methods for interprocess communication: event flags, global sections, and mailboxes.

Event flags provide a means for event synchronization between processes. They can, however, not be used for exchanging data and are thus not suitable for this application.

A global section is an area of memory containing data or code that is shared between different processes. This is done by mapping virtual memory areas from the different processes onto the same physical memory area. Global sections are the most general communication on VAX/VMS. Using global sections with assembler or conventional high-level languages works well since in these languages there is a direct correspondence between the variables in the program and the memory cells where they are stored. A Lisp symbol, that corresponds to a variable in a conventional language, does not, however, directly correspond to a memory cell. Lisp implementations use internal data structures to represent Lisp symbols. The garbage collection in Lisp also causes Lisp symbols to be moved around in memory. The effect is that it is difficult to associate memory locations with Lisp symbols and thus to perform the memory mapping.

The communication method used in this project is the mailbox mechanism. The communication structure is shown in Figure 4.2, where the rectangles represent mailboxes.

The mailbox **Inbox** is used for messages to the knowledge-based system from both the numerical algorithms and the man-machine interface. The knowledge-based system handles all user commands also those that regard the numerical algorithms. The mailbox **Outbox** is used for messages to the numerical algorithms. A separate mailbox, **Answerbox** is needed for sending answers to information requests from the knowledge-based system. **Resultbox** is used by the knowledge-based system to return results to the man-machine interface.

A mailbox is an I/O device that can be used by cooperating processes for synchronous or asynchronous message passing. There are primitives for creating, writing to and reading from a mailbox. Mailboxes can hold multiple messages which are stored in a first-in, first-out queue. It is also possible to check the number of messages in a mailbox. Mailboxes can be accessed at several different levels. In the implementation of the knowledge-based controller, the highest level is used. Here, mailboxes can be viewed as ordinary files with the same accessing methods. By treating the mailbox as a text file, a message is regarded as a line of text. Appendix C contains the details of how the mailboxes are created and how they are interfaced to the Lisp and Pascal processes.

A demand on the system is the possibility to insert messages into mailboxes

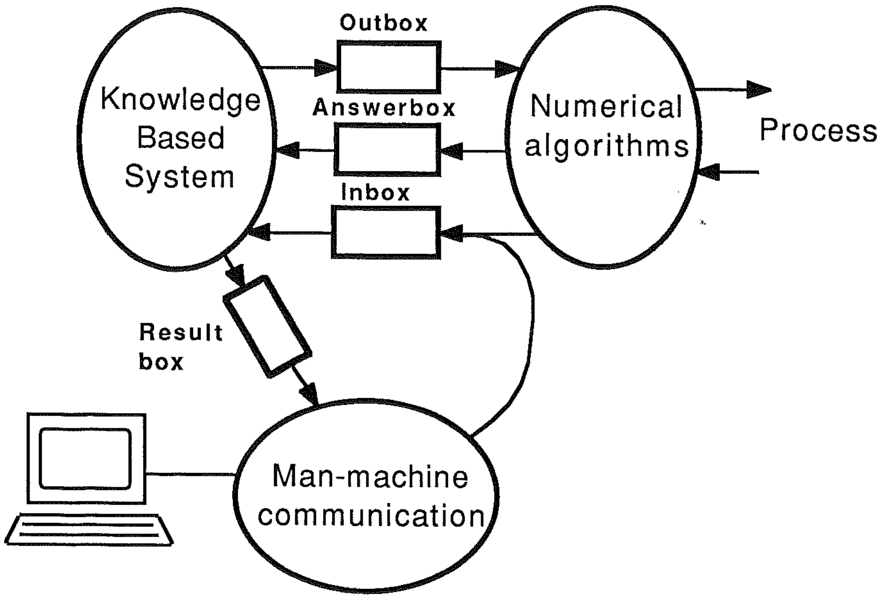

Figure 4.2 Communication structure. The rectangles represent mailboxes.

according to an associated priority instead of by FIFO scheduling. This is needed to assure that for example alarm messages from monitoring algorithms are taken care of by the knowledge-based system before less important messages. This is not allowed with VMS mailboxes. Instead, it is solved by an internal mailbox structure in the knowledge-based system process. Instead of only reading a message from **Inbox**, the process starts by emptying **Inbox** and inserting all messages into the internal mailbox according to their priority before returning the first message in the internal mailbox.

Associating a mailbox with a text file has a powerful impact on the communication between two Lisp processes. Arbitrary Lisp expressions, e.g., lists can be exchanged between the processes. Moreover, in Lisp there is no syntactical difference between data objects and program code, i.e., Lisp functions. Lists are used to represent both. This makes it easy to implement remote evaluation of Lisp functions. An arbitrary Lisp expression can be sent to another process, evaluated there, and the result of the evaluation sent back. The Lisp code for this is very simple.

```
    Process 1                    Process 2

(print message
       mailbox-1)
```

```
                                  (print (eval (read mailbox-1))
                                         mailbox-2)

    (read mailbox-2)
```

The value of the Lisp symbol `message` could be an arbitrary Lisp expression. The symbols `mailbox-1` and `mailbox-2` are bound to the actual mailboxes. The example will not work if an error occurs during the evaluation of the message. Such errors can, however, be caught by the Franz Lisp function `(errset expression)`. This function evaluates its argument; if no error occurs during the evaluation, the resulting value is returned in a list. If an error occurs, the value `nil` is returned. `Process 2` now instead looks like

```
       Process 2

    (let
      ((result (errset (read mailbox-1))))
       (print
         (cond (result (car result))
               (t 'ERROR))
        mailbox-2))
```

The unrestricted message format makes it possible to evaluate Lisp functions in other processes almost as if they were run interactively. It is used for the communication between the man-machine interface and the knowledge-based system.

The remote evaluation facility has fascinating possibilities. In particular, it is very easy to change code in an executing process while it is running. This gives, theoretically, the possibility to on-line change arbitrary parts of an executing process. In practice, however, it is only minor modifications to the code that are made due to the risk for run-time errors caused by changes not thoroughly thought-out. When the knowledge-based controller is used as an experimental development environment, however, the feature is important.

Message protocol

Messages to the mailbox **Inbox** have the following syntax.

(*priority* (*message-tag body*))

The integer-valued priority tells how important the message is. The message-tag can take either the value `regul` meaning that the message comes from the numerical algorithms, or the value `eval` which implicitly tells that the message comes from the man-machine interface. In the latter case the body, a Lisp expression, is evaluated and the result is returned in **Returnbox**. Messages that come from the numerical algorithms have different bodies depending on which algorithm they origin from. These messages are taken care of by the knowledge-based system.

The messages from the knowledge-based system to the numerical algorithms are of four different types.

> `Start` ⟨algorithm⟩ ⟨parameter⟩ ⟨real-value⟩ ...
> The periodic execution of an algorithm is started. Legal parameter values are real numbers. Default values are given to parameters that not are explicitly initialized.
>
> `Par` ⟨algorithm⟩ ⟨parameter⟩ ⟨real-value⟩ ...
> The parameters of the algorithms are changed. This message is also used for sending questions.
>
> `Stop` ⟨algorithm⟩
> An algorithm is stopped.
>
> `Execute` ⟨algorithm⟩
> A single execution of an algorithm is performed. This is used for algorithms that are not executed periodically.

The knowledge-based system can also change some of the global variables in the numerical algorithms process. Typical examples are sampling period, set point, or the control variable when the controller is in manual mode. This is done with the message

> `Global` ⟨parameter⟩ ⟨real-value⟩ ...

4.3 The numerical algorithms subprocess

This section describes the internal structure of the numerical algorithms subprocess. The actual algorithms are described in Section 8.2.

The aim of the process is to provide a flexible library of numerical algorithms. An algorithm performs some limited calculations based on numerical signals. It is usually executed periodically.

The algorithms can principally be divided into three groups: control algorithms, identification algorithms and monitoring algorithms. The control algorithms all compute a control signal. Only one control algorithm can be running at a time. The identification and monitoring algorithms all in some sense extract information from the numerical signal flow. This information is sent to the knowledge-based system The algorithms in these two groups can be viewed as filters that send information to the knowledge-based system only when something significant has happened. During steady-state operation, the knowledge-based system is not involved and the system resembles a conventional controller. The separation between the numerical algorithms and the knowledge-based system is favorable from the point of information flow. If a knowledge-based system was interfaced directly to a physical process or to an existing control system, numerical information would have to be sent forth and back again at a high rate. The knowledge-based system also had to itself extract all useful symbolic information from the signals. This is a task that often is expressed in the form of numerical algorithms. Using expert system techniques for such tasks is often inefficient.

The intention has been to code the algorithms as cleanly and as closely to the textbook as possible. The heuristic safety-jacket parts of the code should be left to the knowledge-based system. This separation cannot be carried out completely. Some logic conditions may require instant actions when they have been detected. In such cases, the delay caused by the knowledge-based system could be unacceptable. The algorithms may also contain heuristic parts that need to be tested every sampling instant. These parts should be included in the algorithms. An example of this is the anti reset-windup code of a PID controller.

Another intention has been to implement the different algorithms as independently as possible. The reason for this is that it should be easy to switch different algorithms in and out. This is solved by having as few common variables as possible shared by the different algorithms. This has, e.g., the effect that the same information sometimes is stored in more than one algorithm and thus increases the storage required. Total independence among the algorithms is undesired. For instance, all algorithms share the values of the measured control variable and error signal. Some algorithms must share additional variables. An example of this is the situation where an explicit self-tuning regulator is formed by combining one pole placement control algorithm, one least-square identification algorithm and one algorithm that computes new controller parameters. Other examples are algorithms that monitor the execution of other algorithms.

The most flexible implementation of a library of this type would probably be in the form of interconnected function blocks. Each function block type would then represent an algorithm. Multiple instances with different parameters and input signals could then be created. This would, e.g., facilitate the implementation of a multi-loop system. The computation order for the individual blocks in a conventional function block language is based on causal dependencies and is often calculated when the function blocks are configured. Function blocks that generate signals must be evaluated before function blocks which use these signals, etc. The causal dependencies are explicitly expressed in the blocks. The input signal to a block is declared as the output signal from another block, etc. This obstructs the desire for independence among the algorithms. Consider the following simple example. A PID controller block needs the value of the controlled variable y. The controlled variable is filtered through a digital low-pass filter before it is used. Figure 4.3 shows a standard function block solution for this. This solution is not sufficient since in a knowledge-based controller it should be possible to activate and deactivate the low-pass filter without affecting the PID controller. To achieve this, the measured control variable could be stored in a register according to Figure 4.4.

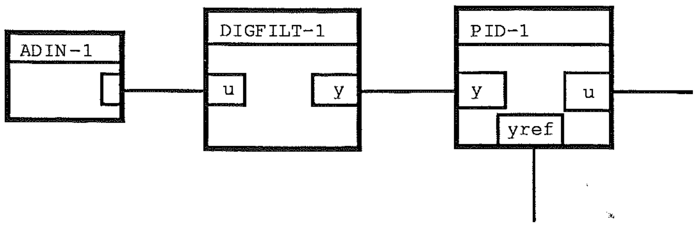

Figure 4.3 Standard function block solution.

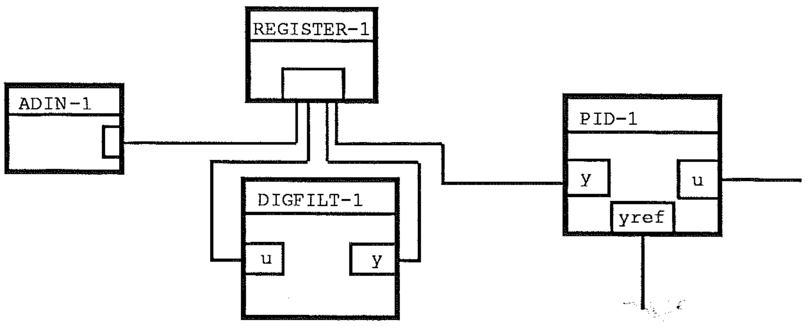

Figure 4.4 Modified function block solution.

In this configuration the causal ordering is ambiguous. One way to resolve it is to have prespecified partial orderings among different blocks. This is what is used in the system. Algorithms are viewed as filters that may transform the incoming signals as shown in Figure 4.5. This is similar to the pipe facility of Unix (Ritchie and Thompson, 1978).

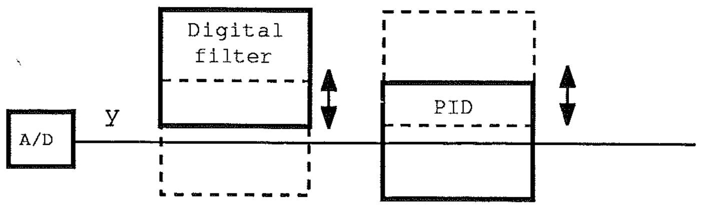

Figure 4.5 Pipe oriented solution

Another demand on the function block language is that it must be possible for the knowledge-based system to perform reconfiguration on-line. This is usually not possible in commercial control systems. The technique for implementing function block languages is well known and commercial control systems often use function blocks (Lukas, 1986). The work involved in developing a function block language is, however, not neglectable. Therefore, since this part of the project is not central, a less ambitious approach has been taken.

The solution uses a fixed number of numerical algorithms with a prespecified computation order. The main loop of the subprocess has the following structure.

```
begin
  initiate_mailbox_communication;
  while true do
  begin
    set_sampling_eventflag(main_sampling_time);
    y := analog_in(inchannel);
    for all algorithms do
      if active[algorithm]  then algorithm(execute);
    if message_in_outbox then read_message;
    wait_for_sampling_eventflag;
  end;
end.
```

The order of the algorithms is determined by their function. For example, a digital filter on the measured signal is checked before the control algorithm. Identification algorithms are in general processed after the control algorithms, etc.

Each algorithm is coded according to the same pattern. The pattern looks as follows.

```
procedure algorithm_1(operation : operationtype);

  procedure algorithm_1_execute;

  {* Declarations *}

  begin
    with algorithm_1_variables do
    begin
      if counter > 0 then counter := counter - 1
      else
      begin
        counter := sampling_time;

        {* Execution body *}

      end;
    end;
```

```
end;

procedure algorithm_1_parameter_change;

{* Declarations *}

begin
  with algorithm_1_variables do
  begin
    while more_new_parameter_values do
    begin
      read parameter_identifier;
      read parameter_value;
      case parameter_identifier of
        parameter-1 :  .....
                 .
                 .
      end;
    end;
  end;
end;

procedure algorithm_1_start;

{* Declarations *}

begin
  with algorithm_1_variables do
  begin
    active[algorithm_1] := true;

    {* Initializations *}

    algorithm_1(parameters);

    {* Decode initial parameter settings *}

  end;
end;

procedure algorithm_1_stop;

{* Declarations *}

begin
  with algorithm_1_variables do
  begin
```

```
    active[algorithm_1] := false;

    {* Terminations *}

  end;
end;


begin
  case operation of
    start      : algorithm_1_start;
    stop       : algorithm_1_stop;
    execute    : algorithm_1_execute;
    parameters : algorithm_1_parameter_change;
  end;
end;
```

Each algorithm is separated into four parts. These parts are implemented as separate procedures. The start procedure is executed when an algorithm is started. This also involves a call to the parameter procedure to take care of initial parameter settings. The parameter procedure is also executed when a parameter message for this algorithm has been received. The execute procedure contains the actual body of the algorithm. It is executed by the main program every sampling loop as long as the algorithm is active. It can also be executed on demand from the knowledge-based system by the means of an execute message. The stop procedure is executed when the algorithm is stopped.

Apart from the above body, an algorithm also consists of its own constant declarations, type declarations and variable declarations. These four parts are stored in separate files which are inserted in the main program. This structure simplifies the addition of new algorithms to the program.

Associated with each algorithm is a sampling interval. This is implemented by the `counter` variable in each algorithm. The fastest allowed sampling interval is the main sampling interval which determines the execution speed of the main loop. The synchronization between different algorithms is solved automatically by the program. Algorithms that have functional connections are executed at the same sampling instant.

5

Expert System Techniques

The objective of this chapter is to explain and discuss some of the characteristics and programming techniques that distinguish expert systems from conventional programs. Section 5.1 discusses the common characteristics of expert systems. Section 5.2 focuses on object-oriented representation and Section 5.3 on rule based programming. The ideas behind blackboard systems are explained in Section 5.4. Expert systems for real-time operation are discussed in Section 5.5.

Most of the material in this chapter should be well-known for a reader with some experience of AI. The motivation for the chapter is to provide the necessary background for the discussions in Chapter 7. This has influenced the presentation of the subject. Hence, this presentation does not pretend to give a complete treatment of the expert system area. This can be found in Waterman (1986), Hayes-Roth *et al* (1983) or Harmon and King (1985).

5.1 Expert system characteristics

Expert systems or knowledge-based systems is an area of Artificial Intelligence that has grown rapidly during the last few years. The basic definition of the term *expert system* is a program that solves problems within a specific, limited domain that normally would require human expertise. This is a wide and vague definition that also covers many traditional computer programs. As an example, both compiler-compilers and programs for Fast Fourier Transform calculations without doubt require expertise and thus fulfill the definition. Still these examples are normally not considered to be expert systems. A clear definition of an expert system is difficult to state. The situation instead is that an expert system more or less fulfills a number of different characteristics.

The most significant characteristic is the expert problem solving capacity within limited application domains and for problems where conventional programming techniques have not been successful. The reason why conventional techniques do not work is mainly that the problem lacks a clear analytical and/or algorithmical solution or that the existing algorithm is computationally intractable. The expert system instead tries to emulate the problem solving behavior of the human expert. This means that the system tries to represent and execute the expert's knowledge and reasoning strategy.

Another very common characteristic is that the domain knowledge is represented explicitly in an identifiable, separate part of the program. This so called knowledge base is separated from the *inference engine* that actually runs the program by operating on the knowledge. A system of this type is referred to as a *knowledge-based system.* The explicit knowledge representation gives an expert system a declarative nature in contrast to traditional procedural programming languages such as Pascal or Fortran where domain knowledge is expressed in the form of program statements. Knowledge representation is a key issue in expert systems. Common representation forms are rules and objects which are discussed in the following sections.

Well-developed explanation facilities are another expert system characteristic. It is necessary that an expert system can explain its reasoning in order to be accepted by the user. It is usually possible to get an explanation for why the system asks a certain question to the user and how a certain conclusion has been reached.

The possibility to reason with uncertainty is another feature that might be present. Knowledge might be uncertain in certain applications and expert systems can then support the representation of, and reasoning with, this uncertainty. This is often implemented in the form of probability measures that reflect the reliability of the knowledge and which are propagated through the reasoning.

Another expert system aspect is the modularity that is provided through the explicit knowledge representation. The knowledge base is built incrementally and can relatively easily be expanded with, e.g., new rules. This makes exploratory programming possible, where prototypes rapidly can be developed and later be used as a part of the final implementation. This is perhaps the main reason why expert systems have the reputation of allowing implementation of very complex systems.

Implementation languages

Most existing expert systems are implemented in a symbolical language such as Lisp (Winston and Horn 1984; Steele 1984) or Prolog (Clocksin and Mellish 1984) or in some language implemented on top of those. In the sequel the reader is assumed to have some knowledge about Lisp. The separation between the knowledge base and the inference engine has led to the development of so called *expert system shells* or *frameworks.* A framework is an empty expert system without any domain knowledge. It provides an inference engine and a knowledge repre-

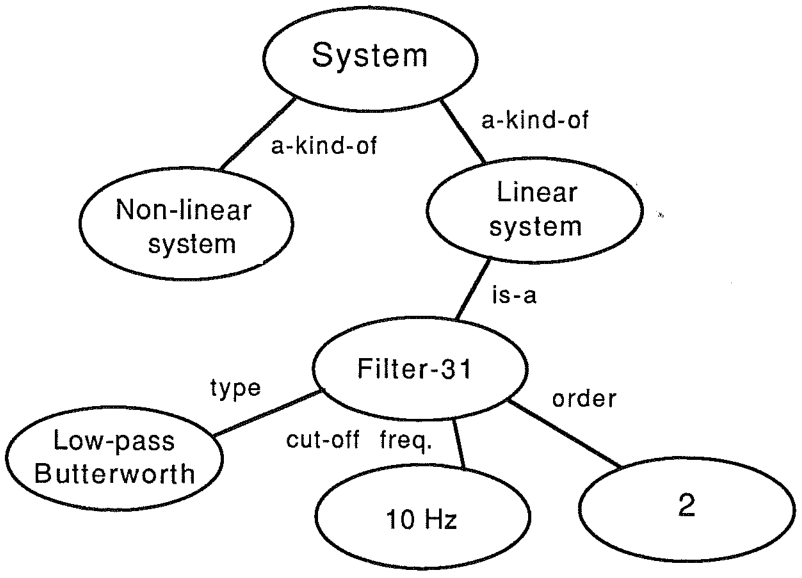

Figure 5.1 Semantic network example

the sometimes very high expectations on expert systems from people not well-informed on the subject. It should always be kept in mind that expert system frameworks are nothing more than high level programming languages, run on conventional von-Neumann architectures and suitable for certain applications. Today's expert systems cannot exhibit true intelligence in the meaning the term has to most of us, e.g., to be able to come up with solutions in unforeseen situations or to be able to perform common-sense reasoning. Systems with these capabilities are way into the future.

5.2 Object-oriented representations

To represent knowledge as objects with associated attributes is common in existing expert systems. The basis is the *semantic network*, (Quillan, 1966), which represents knowledge as a network of nodes. A node could represent the concept of objects, events, ideas, etc. Associative links represent the relations among the nodes. An example of a semantic network is shown in Figure 5.1. A survey is given in Brachman (1979). Semantic networks represent the combination of a superclass-subclass hierarchy and the description of properties (attribute - value pairs). Superclass-subclass or generalization versus specialization is often represented with the *a-kind-of* link. Another well-known aspect of the network formalism is the instance relation that associates a particular individual with a class of which it is a member. This is often represented with the *is-a* link.

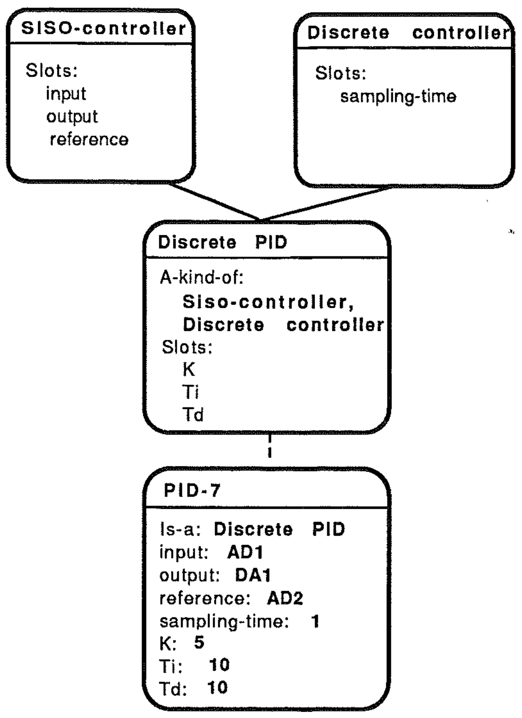

Figure 5.2 Frame system example

Sometimes no distinction is made between *a-kind-of* links and *is-a* links.

Frame systems

The idea of frame systems, a variation of the semantic networks, was introduced by Minsky (1975). A frame system consists of three different building blocks: frames (sometimes called units), slots and facets. A frame is the equivalent to a node in a semantic network, i.e., it represents concepts of objects, events, ideas, etc. With some abuse of language, frames are often referred to as objects. The slots describe the properties or attributes of a certain frame. In the same way, facets describe the different slots. One of the facets is the actual value of the slot. Others facets could be used to specify which type the slot value may take, default value for the slot or to give additional description of the slot.

Frames are often divided into two types: those which describe classes and those which describe individual instances. An important concept of semantic networks and frame systems is the inheritance of properties. Inheritance allows class frames which can pass their slots along to subclass frames and to instance frames. Multiple inheritance, i.e., that a frame is a subclass of more than one superclass is common. A simple frame system example is shown in Figure 5.2.

Procedures can be attached to frames by associating the procedures with slots. These procedures are called *demons* and they provide for so called *access-*

oriented programming. Demons are for example used to compute the value of a slot when a reference is made to it and no previous value exists. Another possibility is to have demons that are executed each time a slot is given a new value or each time a frame is created or destroyed.

Several frame based knowledge representation languages exist. Some examples are KRL (Bobrow and Winograd, 1977), Units (Nilsson, 1982), PAUL (Hein, 1983) and KEE (Intellicorp, 1984).

Object-oriented programming

At the same time as the frame based knowledge representation languages were developed, a very similar development took place in the area of object-oriented programming. This area has its historical background in the work on SIMULA (Dahl and Nygaard, 1966) and has its most extreme representative in the programming language SMALLTALK-80 (Goldberg and Robson, 1983). A good overview of object-oriented programming is given in (Stefik and Bobrow, 1986). The basic entity of object-oriented programming is the object which has a local state and a behavior. Objects are asked to perform operations by sending appropriate messages to them. Objects have associated procedures called *methods* that respond to the messages. Message passing supports data abstraction and generic algorithms. A protocol, i.e., a set of messages is defined, which specifies the external behavior of the object. The internal implementation of the object can thus easily be changed without affecting the calling programs.

In the same way as in frame systems, objects are divided into classes and instances of classes. The classes builds up a superclass - subclass hierarchy with inheritance. The inheritance is more focused on inheritance of behavior, i.e., of methods than on inheritance of properties as is the case in the frame systems. The analogue of the slots in frame systems are the variables. Variables are often divided in two types: instance variables that are inherited by the instances and class variables that are attached to a specific class and common to all the instances of this class.

Several different object-oriented add-ons to Lisp exist. Some examples are Flavors (Cannon, 1982), Common-Loops (Bobrow *et al*, 1986) and Object Lisp (Drescher, 1985). The programming language C is undergoing a similar development with the add-ons Objective-C (Cox, 1986) and C++ (Stroustrup, 1985). Although very similar in spirit to frame systems the main purpose of the object-oriented systems is to use the objects for data abstraction in computer programming. The frame systems are instead focussed on using similar facilities for knowledge representation.

5.3 Rule-based representations

Rules are the main knowledge representation method used in existing expert systems. This is reflected by the use of the name rule-based system synonymously to expert system. Another name that is used is *production systems* where a production is the equivalent of a rule. The primary reference for production systems is Newell and Simon (1972). Rules often look like

```
if <antecedents>
then
   <consequents>
```

or like

```
if <conditions>
then
    <actions>
```

The antecedent or condition part of a rule is also called the left hand side (LHS) of the rule. In the same way the consequent or action part is called the right hand side (RHS).

The three main parts of a rule-based expert systems are

1. Database or working memory
2. Rulebase
3. Inference engine

The database is used to represent facts about the application domain. The data structures in the database vary between different systems. The simplest form is a collection of variables that can take different values. Another quite common data structure is the list. Lists are used in OPS4 (Forgy, 1979) and YAPS (Allen, 1983). Object-oriented data structures are very common. The EMYCIN (van Melle, 1981) class of systems usually use object-attribute-value triplets. Basically the same is used in the OPS5 (Forgy, 1981) system. Other systems use more elaborate frame based systems with inheritance and procedural attachment. Examples of those are KEE (Intellicorp, 1984) and ART (Inference Corp., 1984).

The rulebase contains the rules of the system. It is sometimes partitioned into different rule-groups according to different contexts. The left hand side of the rules typically consists either of patterns that should match the contents of the database or of predicates on the database that should be fulfilled. Most systems allow rules to use pattern matching variables. This makes it possible to write generic rules that can be used by the system for many different facts. The right hand side of the rule either modifies the database in some way or performs some external input or output.

The inference engine applies the rules to the database according to some strategy. The dominating strategies are forward chaining and backward chaining. In some systems these are combined. Since the inference strategy is the real core of the system, these two strategies will be explained in detail later.

The rule-based programming style is especially well suited for certain problems. Suitable problems as well as advantages and disadvantages of rule-based systems are described in, e.g., Brownston *et al* (1985). The power of the rule-based systems is most evident for complex, ill-structured applications that lack efficient algorithmical solutions. The decomposition of the system into a number of relatively loosely coupled rules makes it suitable for problems that are decomposable into subproblems which have no fixed or apparent order. The rule-based approach supports parallel lines of reasoning as opposed to the primarily sequential execution of conventional languages.

Forward chaining

In a forward chaining system, the left hand sides of the rules are examined to see whether or not they are fulfilled. If so, the modifications to the database in the right hand side of the rules are executed and then the system examines the rules again. Forward chaining systems are sometimes called *data-driven systems.* A survey of forward chaining expert systems is Brownston *et al* (1985). Most forward chaining systems use pattern matching to express when rules are applicable. The LHS of the rules contain patterns that must match the contents of the database for the rule to be fulfilled. An example could look like

```
if (father -x -y)
   (father -y -z)
then
   (add (grand-father -x -z)).
```

In this notation, `-x,-y` and `-z` are matching variables that can match arbitrary symbols. When the same matching variable occurs at more than one place it must match against the same symbol at all occurrences.

The reasoning is performed in what is called a *recognize-act cycle* that has three states. During the match state all rules that are fulfilled are collected into the *conflict set* together with the corresponding database elements. If rules with pattern matching variables are allowed, the same rule can appear in the conflict set several times with different matching database elements. During the select phase, one rule is chosen for execution. If the conflict set contains more than one element, the conflict is resolved according to some *conflict resolution strategy.* During the act state the right hand side of the selected rule is executed.

The conflict resolution strategy is crucial for how the rule execution proceeds. Conflict resolution strategies can be divided into two groups. The first group consists of strategies that order the rules in a predetermined way. Examples of this are strategies that select rules according to the order in which the rules were created or according to a priority associated with each rule. Another example is strategies that favor more complex rules, e.g., rules with many condition elements, before simpler rules. The other group contains strategies where the choice of a particular rule depends on the state of database. An example of this are strategies that select rules on the basis of the recency of the matched database elements. Rules matched by more recent added information are usually favored. It is only systems with conflict resolution strategies of this type that really deserve the name data driven systems. Systems that use predetermined ordering could actually be implemented in conventional programming languages as a sequence of if-then statements with each if-then statement representing one rule.

The conflict resolution also determines the search strategy that the inference engine implements. A recency based resolution implements depth-first search of a knowledge base and a resolution based on predetermined order implements a breadth-first search.

Matching each rule against the contents of the database every recognize-act cycle is time consuming. It can be avoided by saving matching information between the cycles and by matching only the database elements that are changed at each cycle. This is efficient because typically the database changes very little between the cycles. The most well-known matching algorithm of this type is the RETE algorithm (Forgy, 1982) used in the OPS family. The rules are basically compiled into a network where each node represent a test of one left hand side condition. Rules that contains more than one left hand side condition give rise to nodes that join the nodes for each single condition element. When database elements are added or removed, they are propagated through the network and result in the addition or removal of partial matches and conflict set elements. Matching algorithms of this kind usually also allows negated left hand side conditions, i.e., patterns that must not match against the contents of the database.

An effect of the network based rule interpreters is that the recognize-act cycle now has a different ordering. The cycle starts with the select state where a fulfilled rule is selected. During the act state the right hand side of the selected rule is executed. This causes database elements to be added or removed and it is during these database alterations that the actual matching occurs. The network approach to pattern matching affects the addition of new rules to the system in a serious way. Since the rules build up the network used in the matching, this network must exist before database elements are added or removed. This means that a new rule which is added during rule execution will only recognize database elements that has been altered after the rule was entered. In order for the new rule to operate on the total database, the database elements already added to the database have to be refreshed, i.e., removed and added anew.

Another implication of network based systems is the effect they have on the use of predicates in the left hand sides of the rules. The intended use of LHS predicates is to further specialize the rules beyond what is possible through pure pattern matching. An example of this is the trivial rule

```
if (number -n)
 predicate
   (>= -n 1000)
then
   (add (large-number -n))
```

that simply marks a number as being large if it exceeds 1000. Predicates are tested as soon as enough partial matching information is available for the predicate arguments to have values. The reason for this is the wish to prune the network, and thus the search space, as early as possible. This works well as long as the LHS predicates act only on the database contents. It is, however, sometimes desirable to have predicates that act upon information outside the database, e.g., predicates that test if some measured signal exceeds a certain threshold or if a certain global variable takes a given value. An example is the following rule.

```
if (check-signal -signal)
   (channel-number -signal -channel)
   (signal-limit -signal -limit)
 predicate
   (>= (analog-in -channel) -limit)
then
   (remove 1)
   (message "The signal " -signal
            " exceeds the value " -limit)
```

The intended meaning of this rule is to check if the value of a signal exceeds a certain limit whenever the database element **check-signal** is added. Due to the network based system this will not work as intended. The greater-or-equal predicate will be evaluated as soon as enough partial information is available for its arguments to have values, i.e., as soon as the database elements **channel-number** and **signal-limit** are available, and not when **check-signal** is added. The delay of the predicate testing until the complete LHS is matched would solve this specific example but gives no general solution. The main problem is that the network, which in a way contains the state of system, can only be changed by database alterations and not by external events. With a small trick, the above rule would behave as intended. The trick is to replace the above predicate with

```
(and -signal
     (>= (analog-in -channel) -limit))
```

which has the same effect but cannot be tested until a value for **-signal** is available.

Backward chaining

Backward chaining systems are sometimes called goal-directed systems. A backward chainer tries to achieve a goal, or alternatively stated, to verify a hypothesis by trying to prove rules that confirm this hypothesis. A goal could, e.g., be expressed as the need to compute the value of a certain object attribute in the database, and a hypothesis could be expressed, e.g., as a certain value for an object attribute that needs to be verified. If the goal is not immediately available in the database, the backward chainer tries to find the rules with consequents that deduce the goal. A rule is selected and the antecedents of this rule become new goals that must be fulfilled. This causes new rules with these goals in their consequents to be selected and so on. The user is usually asked when a goal cannot be directly proven and no rules are found with the goal in their consequents. If a goal cannot be fulfilled the system backtracks and chooses another rule. The way in which the rules are selected and in which order the subgoals are analyzed determines the search strategy. During depth-first search the first applicable rule is chosen and its first antecedent immediately becomes the new subgoal. In a breath-first scheme all the antecedents of the chosen rule are checked before eventually a subgoal is selected. In a best-first scheme the rule most likely to succeed, e.g., with fewest antecedents, is selected first.

Backward chaining systems are often used for classification problems, see Shortliffe (1976) or Weiss and Kulikowski (1981). In these applications, the number of possible outcomes, i.e., the values of the goal attribute, is typically small. A feature of many backward chaining systems is the possibility to reason with uncertainty. Database elements have associated numerical certainty factors that reflect the amount of belief or disbelief in a certain fact. Rules also have certainty factors denoting to which degree a certain inference can be trusted. The inference engine propagates the certainty factors during the rule execution. The uncertainty feature is often combined with the possibility to let object attributes simultaneously take different values with different amount of certainty.

Backward chaining systems have traditionally well-developed explanation facilities. The explanation facilities are built into the inference engine and the explanations are generated automatically. The two standard types of explanation facilities are the "How?" and the "Why?" questions. When the system has drawn a conclusion the user may ask how the conclusion was reached. This typically results in a trace of the rules that were used in the reasoning. If the system asks the user for additional information, the "Why?" question explains why this information is needed. The user has sometimes the possibility to investigate the outcome of different answers with a "Whatif?" question.

5.4 Blackboard systems

Forward chaining and backward chaining are both quite rigid reasoning strategies that fit well only for rather stereotype applications. In so called *oppurtunistic reasoning systems*, the most appropriate reasoning strategy, e.g., forward chaining or backward chaining, is chosen at every time. The blackboard based problem solving paradigm is a special case of such a system.

A blackboard system consists of a global database, the blackboard, and a set of logically independent knowledge sources. The knowledge sources operate on and respond to changes on the blackboard. The knowledge sources contain the domain knowledge for a certain part in the problem solving. Knowledge could be expressed either as rules together with an appropriate inference strategy or as ordinary procedures. The choice of which knowledge source that should be executed is determined dynamically depending of the contents of the blackboard.

The blackboard is usually hierarchically organized into objects. The relations between the objects are expressed through named links. This is somewhat similar to the frame-attribute-value structure in frame systems. A control module monitors the changes on the blackboard and decides what action to take, i.e., which knowledge source to activate. Knowledge sources that may be activated are contained in an *agenda.*

Blackboard based systems are difficult to classify any further. The blackboard based reasoning model can rather be viewed as a guideline for problem solving structuring. A survey can be found in Nii (1986a; 1986b). In the HASP/SIAP project, (Nii *et al*, 1982), a blackboard system was used for interpreting sonar signals collected by hydrophone arrays in some area of the ocean. This project is interesting since the system was used autonomously, in real time. The system was however only passively recording incoming information and thus had no feedback element.

5.5 Real time expert systems

The over-whelming majority of existing expert systems are off-line applications. The human user provides the expert system with some amount of initial information which possibly is completed later on during the consultation. In autonomous systems, the interaction with the human is minimized but still the expert systems are mainly off-line systems. In certain fault-diagnosis applications for, e.g., nuclear power plants or computer systems, the expert system apparently monitors the plant operation continuously. If, however, these systems are scrutinized, they are often basically off-line systems. When a fault has occurred and the expert system has been activated, the system only looks at the history leading to the current fault situation and this is similar to how an off-line system works from some initial information. A few true on-line systems have been implemented. These are mainly in the area of signal analysis. The systems are typically used for interpretation and analysis of measured signals. A few examples are process monitoring (Moore, 1985), the previously mentioned HASP/SIAP project and the VM (Ventilator Manager) intensive care monitoring system (Fagan, 1978). These applications rarely contain any feedback action, i.e., can rarely themselves affect the incoming information flow.

The logical background for most existing system are a standard logic, i.e., propositional logic or first order predicate calculus, together with the *modus ponens* rule. This simply says that whenever a fact `A` is known to be true and there is a rule `If A then B`, it is permitted to conclude that `B` is true. Another characteristic of expert systems is the monotonicity property. The beliefs of the expert system, i.e., the contents of the database, are considered to be true and these truths are not allowed to change. The task of the reasoning system is to monotonically draw new conclusions from the existing ones. The monotonicity shows up for example in classification systems where the user rarely has any possibilities to later change some of the information he has provided. A true on-line expert system must provide some way of *non-monotonic reasoning*. In real life we are often faced with the need to draw conclusions based on incomplete or uncertain information. Later, as new information comes in, the basis for the drawn conclusions may turn out to be wrong. The system then must be able to retract these conclusions.

Standard logic systems are all monotonic. If the logical statement `A` can be proved from a set of initial axioms, additional axioms or information must not cause the negation of `A` to be provable. If this was the case, the logical system would be inconsistent. The approaches to non-monotonic reasoning can be divided in two groups. The first group contains solutions where the logic is extended in several ways. The second approach is to include the logical system in a meta-system that handles the non-monotonic issues. Examples of extensions to the logic system are, e.g., the work on circumscription, (McCarthy, 1980), default reasoning (Reiter, 1980), the UNLESS operator (Sandewall, 1972) and the non-monotonic modal logic of McDermott and Doyle (1980). A compilation of non-standard logics is found in Turner (1984). Truth maintenance systems (TMS) (e.g., Doyle (1979); de Kleer (1986)), are examples of the meta-system approach. The overall system consists of an ordinary inference system and a TMS that serves as a sort of intelligent database. The task of the TMS system is to determine which data that are to be believed when a new inference has been made and to ensure that the database is consistent. These theoretical approaches to non-monotonic reasoning have not yet matured. Many problems are unsolved and they have so far only been used in micro-world examples.

The expert system frameworks PICON (Moore *et al*, 1984a, 1985) and G2 (Gensym, 1987) take another approach. Associated with each measured process variable is a duration time that reflects how long every measurement can be assumed valid. The duration times are propagated to all conclusions drawn based on these measurements. If no further information arrive all conclusions will sooner or later become invalid. This feature is combined with the possibility to check rules, and thus update information, with a fixed time period.

The usual approach taken in applications is to use various *ad hoc* methods to circumvent the problems. One method that can be used is to explicitly, for each case, ensure that the database is consistent; this is basically what is done in conventional real-time computer programs. This is also the approach taken in this thesis. TMS techniques have not been tried due to several reasons. The main task for a TMS is to automatically maintain the consistency of database when information changes. This is feasible for systems which only passively record incoming information. In our application, the knowledge-based system takes different actions, e.g., starts a certain tuning experiment, based on the information it has at a certain time. If this information later changes, it is not a trivial task to automatically maintain the consistency of the database. That would in this example involve the automatic interruption of the ongoing tuning experiment as well as some method to undo the effects it has had on the controlled process.

6

An Off-the-shelf Framework

A standard off-the-shelf expert system framework, OPS4, was used to implement the knowledge-based part of the controller in the first prototype (Årzén, 1986a; 1986b). This chapter describes that prototype and the experiences gained.

The OPS4 framework is briefly described and motivation for the choice is given. An experiment with the resulting controller is described. During the experiment, several lessons were learned, both positive and negative. The most important lesson was that conventional expert system frameworks are not suited for real-time operation. This was, however, more or less anticipated.

6.1 Motivation

The OPS4 framework, (Forgy, 1979), was used for two reasons. OPS4 is a pure rule-based, forward-chaining expert system framework. It has a network based pattern matching algorithm and is, thus, relatively fast compared to other expert system frameworks. The second reason was simply that it was available to us in source-code and that we wanted to test the basic ideas underlying knowledge-based control rapidly.

The rule-based representation was motivated by its declarative nature and by the modularity given by the decomposition of the program into a set of rules. Apart from that, the fact that the majority of existing expert systems are rule-based speaks for this choice.

The forward-chaining inference strategy of OPS4 also seemed well suited. The knowledge-based control application is basically data-driven. Data in the form of significant events detected by the algorithms are sent to the knowledge-based system which should react and generate some response. Forward chaining systems have a parallel behavior. All rules have equal status and the focus of attention shifts among them depending on the incoming data. This gives a execution strategy where parallel activities can be kept active. The recency based conflict resolution strategy allows interrupts among the different activities when new data arrive.

6.2 OPS4

OPS4 is a pattern-matching forward-chaining system written in Lisp that uses recency-based conflict resolution. The database consists of arbitrarily nested list expressions. OPS4 uses incremental pattern-matching implemented by the RETE algorithm.

Negated patterns are allowed in the LHS of the rules. Predicates can be used in the LHS but this requires that the Lisp functions used are explicitly declared as predicates. OPS4 also poses restrictions on how the predicates may be used.

The RHS of the rules must be composed out of a set of predefined functions or actions. The most important actions are <ADD> which adds a new database element and <DELETE> which deletes a database element. The action <BUILD> creates a new rule and the action <EXCISE> removes a rule. It is possible to have user-written actions. This requires that the Lisp functions are written in a special way and explicitly declared as RHS actions. An example of the rule syntax of OPS4 looks as follows.

```
(SYSTEM

   RULE12 ((State is =X) & =Y
           (Alarm has occurred) & =Z
      -->
           (<DELETE> =Y =Z)
           (<ADD> (State is alarm)
                  (Oldstate is =X)))
```

This rule simply switches to alarm state when an alarm notification has arrived. Pattern matching variables are prefixed with an equal sign. The variables =Y and =Z are bound to the entire expression preceding them by the ampersand.

OPS4 has no means for partitioning neither the rulebase nor the database.

6.3 A Ziegler-Nichols auto-tuner

The experiments performed with this system consisted of the implementation of a relay auto-tuner as described in Chapter 3. The control design was based on modified Zieger-Nichols rules. Gain-scheduling based on set point was later added.

The numerical algorithms used for this were a PID algorithm, a relay algorithm, an oscillation analyzer and a noise estimator. The oscillation analyzer measured the oscillation period and amplitude. The noise estimator measured the noise level. The controller had three different modes: manual mode, tuning mode, and PID mode. The operator had commands for switching between the different modes. The experiment was started in manual mode. The operator controlled the process manually until it was in steady state at the desired set point. The mode was then changed to tuning. When the tuning was finished, the operator had to confirm the computed PID parameters. If they were accepted, the system changed to PID mode. From PID mode the operator could change to manual or tuning mode. When the set-point was changed, new PID parameters were switched in from the gain-schedule table if available, i.e., if a tuning experiment had been performed in the neighbourhood of the new set-point.

The operator also had commands for changing parameters, e.g., PID parameters and relay parameters such as step size and hysteresis. It was also possible for the operator to change the contents of the database and to add new rules on-line.

A total number of about 70 rules were used for this experiment. The rules can be grouped according to their context as follows.

Noise estimation rules: Rules for start and supervision of the noise-estimation.

Relay rules: Rules related to the relay experiment for such things as determining the relay parameters, adjusting the relay parameters, judging when the oscillation is steady, and computing the ultimate gain and frequency.

PID design rules: Rules for computing the PID parameters.

PID monitoring rules: Rules for handling the gain-schedule table.

Command decoding rules: Rules for handling operator commands.

The OPS4 system was integrated with the real-time environment described in Chapter 4 using a built-in restart feature. When no matching rules were found, the element `(RESTART)` was added to the database. A rule with `(RESTART)` as its condition element then became satisfied. This rule read an incoming message which was either inserted into the database or treated as a Lisp expression and evaluated. New elements in the database caused rules to be matched and the recognize-act cycle to continue.

A laboratory tank system was used as the controlled process during the experiments and the objective was level control. As an alternative to control a physical process, the simulation program SIMNON (Elmqvist, 1975) was interfaced to the system. Simnon is a program for simulation of non-linear differential or difference equations. The program has been modified to allow real-time simulation. The simulation facility gave a convenient way to experiment with a variety of different processes.

6.4 Experiences and conclusions

The experiment with this first prototype gave many results concerning both the feasibility of the knowledge-based approach and the demands on a expert system framework for knowledge-based control.

The most important conclusion was perhaps that we were reassured in that the approach is feasible. The response times for the knowledge-based system were acceptable. The sampling rate for the numerical algorithm process was 1 second. It took approximately 2-3 sampling periods from that a message was sent to the knowledge-based system until a responding message was returned on a moderately loaded VAX 11/780. This figure of course depends on the number and complexity of the rules.

The second positive result of the experiment was a clean implementation that clearly benefited from the separation of logic and algorithms. The system was flexible and easy to expand. An example of this is the addition of the gain-scheduling facility. This was done after the system had been developed and it required only the addition of about 10 new rules. The time and effort to make these extensions were significantly smaller than for comparable implementations in Pascal and PL/M.

The negative results all regarded the expert system framework. The experiences can be divided into general experiences of OPS4 and application dependent experiences.

A general experience of OPS4 is that it is more a programming language than an expert system framework. The rule syntax can sometimes be very cryptic which diminishes the declarative nature of the system and makes the rules difficult to understand.

Another drawback with OPS4 is its lack of structuring facilities. It is not possible to structure neither the database nor the rulebase. The knowledge-based control task can be divided into relatively loosely coupled subtasks. There is however no possibility to arrange the rules into according groups. This leads to the need for a special state element in the database which determine the actual state of the system. Rules for the actual state all have the common database element, `(State is "actual state")`.

The need to structure the database is also apparent. One desired possibility would be to have the database partitioned into one long-term memory and one or several scratch-pad memories for volatile information.

The single database elements also lack structuring facilities. Arbitrary list expressions are very general. They do, however, not support the representation of information which naturally is grouped together. This is better expressed by, e.g., frame structures. This also facilitates automatic correctness checking of the condition elements in the rules. This not at all supported in OPS4 and leads to frequent errors due to, e.g., misspellings or sheer oblivion.

The most important drawback with OPS4, which it with a few exceptions shares with other expert system frameworks, is that it not designed for real time operation. It has, e.g., no possibility to have time-outs associated with database elements, no means for halting the rule execution for a certain time, and no possibilities to check rules at a given time interval.

The main experience of the tuning application is its large sequential nature. First a noise estimator is used to gather noise information. This information is then used to determine the relay parameters. When a steady state oscillation has been obtained, the PID parameters are determined from the oscillation data. Production systems are in general weak at sequencing problems. The way to achieve sequencing is to explicitly code it into the rules, i.e., to implement a kind of finite state machine with states and state transformations. This has the effect that the actual domain knowledge is obscured by the control knowledge, i.e., knowledge about when to apply a certain rule. It also increases the number of rules. The pure sequential part of the problem is probably better implemented by a procedural representation of some kind. It is however important that the advantages with rule-based representation not are lost.

The sequential nature of the problem partially explains the large number of rules needed for the relatively simple experiment performed. The extensive use of rules, even for tasks where it was not needed such as command decoding, also added up the number of rules.

Another insight that the experiment gave was that only forward chaining is not enough. Even though the problem basically is data-driven, several subtasks can be identified where an goal-driven approach may fit better. The monitoring phase can be stated as a diagnosis problem where backward chaining is more appropriate. The same is true for the phase where the system tries to extract process knowledge from the operator.

7

A Real-time Expert System Framework

One of the experiences from the OPS4-based prototype system was that conventional expert system frameworks lack many of the features desirable in a knowledge-based controller. For that reason, a real-time expert system framework has been developed which will be described in this chapter.

The framework is based on the blackboard principle with a set of cooperating knowledge sources working against a common global database. Different knowledge representations techniques may be used in the knowledge sources. The real-time features of the system have partly been inspired by conventional real-time operating systems.

The framework is described conceptually in Section 7.1. The operating system analogy is also described there. The implementation is based on YAPS (Yet Another Production System) (Allen, 1983) and the object-oriented framework Flavors (Cannon, 1982). YAPS is described in Section 7.2. Several extensions to YAPS have been made. These are described in Section 7.3. Section 7.4 describes the overall implementation of the system.

The structure of the implementation supports the use of different knowledge representation techniques. Forward chaining based knowledge sources are described in Section 7.5. This section also describes the real-time primitives which have been added to the system and their implementation. Section 7.6 describes backward chaining knowledge-sources and Section 7.7 discusses procedural knowledge sources. The combination of knowledge sources into sequences or plans is described in Section 7.8.

7.1 Motivation and functional description

The experiences from the OPS4 prototype indicate that a suitable expert system framework for knowledge-based control should allow different knowledge representation techniques, support a modular representation of distinct subtasks, have structuring possibilities for the database, and allow for real-time constructs.

Different knowledge representation techniques are needed since the system has to perform tasks of totally different nature. These subtasks are often relatively independent. Sometimes they are also separated in time.

The knowledge extraction phase where the operator is questioned for a priori process knowledge can be viewed as a classification problem where the expert system tries to classify the process. When a controller has been started, the system should monitor its performance. Finding the cause of bad performance can be stated as a diagnosis problem where different hypothesis are tried. The need for goal-driven reasoning is therefore evident.

In many situations, the system should be able to react on and reason from incoming data. This implies data-driven reasoning in the form of a forward chaining system. The design phase were the system should make a controller design can be seen as a configuration problem which is often solved with forward chaining.

Other subtasks are dominated by a sequential element. The conventional procedural representation is therefore needed for some tasks. The selection and aggregation of different subtasks can be described as a planning problem.

The reasoning model chosen as the basis for the knowledge-based control framework is the blackboard model. A global database, the blackboard, is available to different, cooperating knowledge sources. The database allows for frame-attribute-value structures for storing associated information. The knowledge sources can be thought of as different actors, each of which solves some subtask of the problem. The knowledge sources also have their own local databases. Different knowledge representation strategies may be used in the knowledge sources. The structure of the framework is shown in Figure 7.1. A knowledge source implements the domain knowledge for a certain task. It is often associated with one or more numerical algorithms. It could for example contain the heuristic logic surrounding an algorithm. Another examples of subtasks suited for implementation in a knowledge source are the choice of controller structure, calculation of controller parameters, the operator interrogation, etc. Knowledge sources can either immediately perform their task, e.g., compute a result of some kind and finish, or they need to wait for incoming information either from the algorithms or from the operator.

The operation of the knowledge-based controller involves the activation of different knowledge sources both in sequence and in parallel. A typical case when knowledge sources are active in parallel is during the steady state control of the process. One knowledge source takes care of the actual control algorithm while other knowledge sources implement different monitoring aspects.

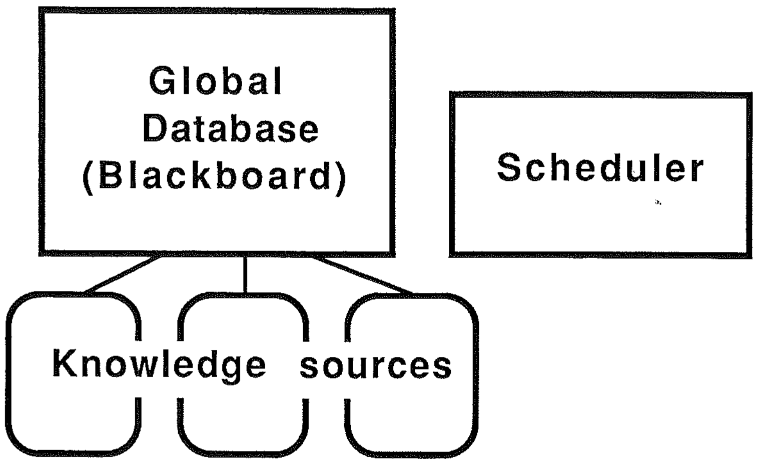

Figure 7.1 Knowledge based control structure

A separate module schedules the selection of knowledge sources at two different levels. The first level involves the sequential activation of different knowledge sources. This is treated in more detail in Section 7.8. The second level involves the scheduling between different knowledge sources that are active simultaneously. This resembles the scheduling in an ordinary real-time, multi-tasking operating system.

The operating system analogy

The separation of the knowledge base into several knowledge sources has two purposes. The first is an attempt to structure the implementation into parts that have natural connections to the subtasks of the problem and to use the most appropriate reasoning strategy for each subtask.

The second purpose is to use the knowledge sources to model the different parallel activities going on in the knowledge-based controller. This can be compared to an ordinary real-time operating system. The knowledge sources are the equivalents of concurrent processes and the scheduler resembles the scheduler of an operating system. Concurrent processes can call real-time primitives to wait a certain time or for a certain event. The equivalents of these primitives have been implemented in the knowledge-based control framework and will be further described in Section 7.5.

The implementation of the knowledge sources can be done with different degrees of parallelism. The most extreme way is to implement the global database in shared memory and distribute the knowledge sources on separate processors. Another possibility is to implement the knowledge-sources as concurrent processes. This requires a programming environment that allows concurrent Lisp

Figure 7.1 Knowledge based control structure

A separate module schedules the selection of knowledge sources at two different levels. The first level involves the sequential activation of different knowledge sources. This is treated in more detail in Section 7.8. The second level involves the scheduling between different knowledge sources that are active simultaneously. This resembles the scheduling in an ordinary real-time, multi-tasking operating system.

The operating system analogy

The separation of the knowledge base into several knowledge sources has two purposes. The first is an attempt to structure the implementation into parts that have natural connections to the subtasks of the problem and to use the most appropriate reasoning strategy for each subtask.

The second purpose is to use the knowledge sources to model the different parallel activities going on in the knowledge-based controller. This can be compared to an ordinary real-time operating system. The knowledge sources are the equivalents of concurrent processes and the scheduler resembles the scheduler of an operating system. Concurrent processes can call real-time primitives to wait a certain time or for a certain event. The equivalents of these primitives have been implemented in the knowledge-based control framework and will be further described in Section 7.5.

The implementation of the knowledge sources can be done with different degrees of parallelism. The most extreme way is to implement the global database in shared memory and distribute the knowledge sources on separate processors. Another possibility is to implement the knowledge-sources as concurrent processes. This requires a programming environment that allows concurrent Lisp processes. This is, e.g., found on Lisp machines. The hardware available for this project is a VAX 11/780. Running parallel Lisp processes on a VAX is very resource consuming. The solution chosen in this project is to simulate the concurrent processes in a single process. In the current implementation, the effect of this is that the knowledge sources may not interrupt each other. A knowledge source runs until it explicitly returns control to the scheduler, e.g., if it has to wait for some information or if it is finished. An extension which allows interrupts among the knowledge sources is outlined in Section 7.9. Using a single process to implement the entire knowledge based part has advantages with respect to portability.

A concurrent implementation of different reasoning modules have been proposed before. In Ensor and Gabbe (1985), a blackboard system is described where the knowledge sources are distributed on separate processors. The same is discussed in Stenerson (1986). The control structure of the KRL system (Bobrow and Winograd, 1977) is based on "the belief that the next generation of intelligent programs will integrate data-directed and goal-directed processing by using multiprocessing". The KRL system, intended for use in language understanding systems, provides for a multiprocess agenda with user-provided scheduling strategies. The Loops system manual (Bobrow and Stefik, 1983) also describes the use of concurrent tasks to represent the invocation of different rule sets.

7.2 YAPS

The main building block in the implementation of the knowledge-based control framework is the forward-chaining expert system framework YAPS. This section will describe YAPS rather extensively. Most of the material is collected from Allen (1983).

Overview

YAPS is a pattern-matching forward-chaining system in the same spirit as the OPS family. There are, however, very important differences.

The database in YAPS may contain arbitrarily nested list expressions. The lists may contain atoms and numbers. They may also contain Flavor instances. This is described more in Section 7.4. Associated with each database element is a unique cycle number which acts as a time tag. YAPS allows multiple coexisting databases.

A set of rules is associated with each database. The rules are stored in a single discrimination network common to all YAPS databases. Rules used in multiple databases are only stored once. The discrimination network principally follows the description in Section 5.3.

YAPS patterns are arbitrarily nested list expressions with variables. Variables begin with the hyphen character '-'. A single hyphen may match anything.

A list of values can be bound to a single matching variable using the dot '.' notation. For example, in the pattern

```
(-car . -cdr)
```

the variable **-cdr** is bound to the list of the but-first arguments similar to the Lisp case.

Arbitrary Lisp functions can be used as predicates in the LHS of the rules. The rule for checking if a signal value exceeds a certain limit will with some small extensions have the following look in YAPS.

```
(p check-signal-value

  (check-signal -signal)
  (channel-number -signal -channel)
  (signal-limit -signal -limit)
test
  (and -signal
      (>= (analog-in -channel) -limit))
-->
  (remove 1)
  (fact exceeded-value -signal -limit)
  (message
     " The signal " -signal " exceeds the value " -limit))
```

The keyword **test** separates patterns from predicates. The RHS of the rule begins after the keyword **-->**. The RHS may contain ordinary Lisp expressions. The pattern matching variables are bound during the evaluation of the RHS. The YAPS function **remove** removes the database elements matching the enumerated LHS patterns. The **fact** function adds a new database element. The **fact** function only evaluates its arguments if they are pattern matching variables. The character ^, however, forces the evaluation of the succeeding argument.

Negated conditions are allowed and can be combined with predicates as in the following example taken from Allen (1983).

```
(p find-largest
    (print-largest)
    (data -x)
 (~ (data -y) with (> -y -x))
-->
    (remove 1)
    (message "The largest data element is " -x))
```

This rule finds the largest data element and prints it. Negated patterns are preceded with the ~ character. A negated condition may contain arbitrarily many patterns. The keyword `with` begins the predicates of the negated condition. If all negated patterns are matched and all their tests return non-nil values then the rule is prevented from execution. Predicates are tested as early as possible in order to prune false partial matches early.

YAPS has two different conflict resolution strategies: goal-directed and age-only. Both are based on recency of database elements and complexity of rules. The algorithms are as follows.

1. If a rule already has fired with a given set of facts, then it is prevented from firing again with the same facts.

2. The goal-directed strategy sorts, for each binding, i.e., a rule together with the facts that it matches, the ages of the facts in two lists. The first list contains the ages of all facts which begin with the keyword **`goal`**. The second list contains the ages of all the rest of the facts. The age-only strategy sorts for each binding all the facts in the first list.

3. The bindings are first compared by looking at the first lists. The most recent ages are compared and the binding with the most recent fact is chosen. If the conflict still is unresolved, successive ages are compared. If this does not help, take the binding with the longest list, i.e., the binding whose rule contains the largest number of patterns. If the lists have the same length, go to the next step.

4. The goal-directed strategy compares the second list in the same way as the first. If there is still a tie, go to the next step.

5. A binding is chosen at random.

The conflict resolution strategy also influences the way YAPS reacts when facts are added to a database which is not currently executing. By default, the rule execution starts as soon as a new fact has been added which generates a binding in the conflict set, i.e., an implicit `(run)` is performed. If, however, the goal-directed strategy is used without any goal database elements, the rule execution does not start until an explicit `(run)` has been performed. This gives a strategy which from a conflict resolution aspect is equivalent to the age-only strategy but which react differently when new facts are added. This is favorable in some cases which will be explained in Section 7.3.

YAPS is implemented in Flavors (Cannon, 1982). A YAPS database is created by instantiating the **`yaps-database`** flavor. This flavor defines methods for adding rules, adding facts, etc. There is always one database which is active. Initially this is a default database. The variable **`*yaps-db*`** always points to the active database. Databases can be manipulated either by sending messages to a specified database or by calling functions which by default operate on the currently active database. Message-passing in the Flavors system used has the syntax

```
(<- <object> <message> <argument> ...).
```

An overview of the most important messages and functions is given in Appendix D.

OPS comparison

YAPS is with respect to functionality very similar to the OPS family of expert system frameworks. It has, however, some important advantages which makes it useful as a building block for knowledge-based control.

LHS predicates: Arbitrary Lisp functions can be used as predicates. No declarations are needed and no restrictions are imposed.

RHS: The right hand side actions can be general Lisp expressions. No declarations are needed. The right hand sides are internally represented as Lisp functions and can be compiled in order to speed up execution.

Syntax: The syntax is less awkward than in OPS. The difference is not, however, of very significant importance.

Integration: The object-oriented implementation simplifies the integration of YAPS with other Lisp tools. For example, arbitrary user-written flavors may inherit the YAPS database flavor and thus combine object-oriented programming with rule-based programming.

Implementation: The object-oriented implementation with clean message interfaces between the different internal objects simplifies extensions and modifications of the plain YAPS system. The modifications that have been performed during the project are described in the next section.

7.3 YAPS extensions

The YAPS system has been extended in different ways during the project. Most of the extensions have been aimed at increasing the expression power of the rules. The extensions have in some cases had a negative effect on execution efficiency. This has, however, been considered acceptable since one of the aims for the development of the knowledge-based control framework is to explore which rule constructs that are needed.

Structured database elements

As in OPS4, the list is the only allowed database element. Closely related information is however better represented with structured data types. Consider the following example. Assume that the YAPS system reasons about PID controllers. Assume for simplicity that a PID controller only is represented by its name and by the three parameters: K, T_i, and T_d. This information could basically be represented in two different ways using lists. The first alternative uses one list as follows.

```
    (<name> PID-controller K <value> Ti <value> Td <value>)
```

The second alternative use multiple list in an object-attribute-value style as follows.

```
    (<name> PID-controller)
    (<name> K <value>)
    (<name> Ti <value>)
    (<name> Td <value>)
```

Assume also that during the control design, a rule is used which always sets the integration time to $T_i = 4 * T_d$ if its previous value is **off**. In the two different representations the rule would look like either like

```
   (p compute-Ti-1
      (-name Pid-controller K -k Ti off Td -Td)
   -->
      (remove 1)
      (fact -name
             PID-controller
             K -k
             Ti ^(* 4 -Td)
             Td -Td))
```

or like

```
   (p compute-Ti-2
      (-name PID-controller)
      (-name Ti off)
      (-name Td -Td)
   -->
      (remove 2)
      (fact -name Ti ^(* 4 -Td)))
```

The first alternative results in rules that have to refer to irrelevant information, e.g., the value of `K`, in order to update the database. The second alternative causes closely connected information to be spread out in the database. This increases the number of condition elements needed in the rules.

The extension made to YAPS allows frame structures in the database. This is somewhat similar to what is available in OPS5. Unlike OPS5, database elements are not restricted to frames. The standard list format is allowed as well. A drawback with the OPS5 approach is that it forces the definition of a lot of unnatural frames with perhaps only a single attribute.

Frames must be explicitly declared before they are referred to. This is done according to the following syntax.

```
<frame-definition> :== (defframe <type> <description>
                         (<attribute-form> ...)
                         ([<super-frame>] ...))

<type>             :== <symbol>
<description>      :== <string>
<attribute-form>   :== <attribute>|
                       (<attribute> [<description>]
                            [<keyword> [<value>]] ...)
<attribute>        :== <symbol>
<keyword>          :== :default|:askable|:duration
<value>            :== <lisp value>
<super-frame>      :== <symbol>
```

The definition of the PID controller would look like

```
(defframe PID-controller "PID controller frame"
  ((K "Proportional gain" :default 1)
   (Ti "Integration time" :default off)
   (Td "Derivation time"  :default 0))
  ()).
```

The `:default` keyword specifies the initial value of an attribute. If this option is not used, the initial value is `nil`. Attribute inheritance can be specified as a list of super frames. Attributes are collected on a left to right, depth-first scheme with duplicate removal. This is the same inheritance scheme as in, e.g., Flavors.

Several functions for frame manipulation have been added. The most important are

```
(make [<name>] <type> <attr1> <val1> <attr2> <val2> ...)
(<- '<database> 'make '<body>)
```

Creates a frame in a database. An optional identifier, `<name>`, can be associated with each frame. If omitted, a unique identifier is automatically generated. Attribute values given override the default values. The `<body>` in the message-passing form of make is a list of the arguments in the functional form of `make`. The functions return the frame identifier.

`(modify <number> <attr1> <val1> ...)`
Modifies a frame in the current database. The number correspond to the patterns in the LHS of the rule. May only be used in the RHS.

`(modi <name> <attr1> <val1> ...)`
`(<- '<database> 'modi '<body>)`
Modifies a frame in a database. The frame is identified through its unique identifier. The `<body>` in the message passing form is a list of the arguments of the functional definition of `modi`.

Frames are removed with the standard functions **`remove`** and **`rm`**.

Frames can be referred to in the rules by the following patterns.

```
(frame <type>
   [<attribute> <value>] ...)
```

The `<attribute>` must be a symbol which belongs to the attributes of the frame `<type>`. The `<value>` may be an arbitrary YAPS pattern. Unique frames can be referred to by substituting the unique frame identifier for the word **`frame`**. The previous `compute-Ti` rule would now look as follows.

```
(make PID-controller K 5 Td 6)

(p compute-Ti-3
  (frame PID-controller
     Ti off
     Td -Td)
-->
  (modify 1 Ti ^(* 4 -Td)))
```

Frames can also be used as attribute values. In the following small example, a `PID-controller` frame is used as the value of the `controller` attribute of the frame `control-loop`.

```
(defframe control-loop ""
  (controller inchannel outchannel)
  ())
```

```
(make control-loop
  controller ^(make PID-controller)
  inchannel 0
  outchannel 1)
```

The patterns that match the above construct would look like.

```
(p frame-as-attribute-example
  (frame control-loop
    controller -frame)
  (-frame PID-controller
    K -k)
-->
    ....)
```

The reason why this works is that the **make** function returns the unique frame identifier and this becomes the value of the **controller** attribute.

Frames are internally represented as single lists. The PID-controller created previously is internally represented as

```
(PID-controller-frame-0 PID-controller K 5 Ti off Td 6).
```

The control-loop frame created above is represented as two lists

```
(PID-controller-frame-1 PID-controller K 1 Ti off Td 0)

(control-loop-frame-0 control-loop
  controller PID-controller-frame-1
  inchannel 0
  outchannel 1)
```

The rules which refer to frames are preprocessed and the frame patterns are modified. The word **frame** is changed to a hyphen and the pattern is expanded to include all attributes. The preprocessed **compute-Ti-3** rule looks like

```
(p compute-Ti-3
  (- PID-controller
    k -
    Ti off
    Td -Td)
-->
```

```
    (modify 1 Ti ^(* 4 -Td))).
```

The `modify` functions removes the corresponding database element and adds it in the modified form. The described way to represent frames correspond to the first list alternative presented earlier. A drawback is that, since during modification, the entire database element is added again, rules which were not directly affected by the modification may be instantiated again with "old" attribute values. This drawback is, however, shared with OPS5.

An alternative representation that circumvents the problem would be to use multiple object-attribute-value lists to represent a frame. This would however complicate the preprocessing. Each frame pattern would be transformed into a number of list patterns and the enumeration of the patterns would change. The removal of an entire frame would also be more difficult. The chosen approach is also favorable with respect to rule execution efficiency. According to Brownston *et al* (1985), rules with a few large conditions execute faster than rules with many simple conditions when RETE-type algorithms are used.

A tempting approach for frame representation would be to use Flavors. Since YAPS allows Flavor instances in the database this is principally possible. Consider the following example where a rule sets **Ti to off** when its value is below zero. Here, the PID-controller is stored as an instance of the PID-controller flavor in the list

```
  (<pid-instance> PID-controller).


  (defflavor PID-controller
     (k Ti Td)
     ())

  (p turn-Ti-off
    (-PID-instance PID-controller)
  test
    (<= (<- -PID-instance 'Ti) 0)
  -->
    (<- -PID-instance 'set-Ti 'off))
```

This solution is dangerous. The changes which are made to the PID instance by message passing will not be notified by the pattern matching network. The network is only affected by explicit database additions and removals. The only way Flavor instances can be referred to is as atomic entities.

How? explanations

Extensive explanation facilities are normally associated with backward chaining frameworks. In these systems the explanation facilities are built-in and generates explanations automatically. The user has the possibility to ask how a certain conclusion was derived or why the system needs certain information from him.

Asking questions to the user is not an inherent feature of forward chaining systems and therefore, 'why' questions are not appropriate. 'How' explanations can, however, be implemented. This has been done in the following sense. Each fact in the database has been associated with information about which rule that added that fact, which database elements that caused this rule to fire, and in which YAPS database the rule was fired. Each frame element has information regarding both the frame as a whole and each individual attribute. The explanation information is associated with the fact when the fact is added to the database. Frame modifications require special attention. The explanation information from the old frame must be transfered to the modified frame.

The following commands have been added for explanation requests.

`(explain '<number> '<number> ...)`
: Generate explanations for the database elements that match the numbered pattern elements. May only be used in the RHS.

`(expl '<cycle> '<cycle> ...)`
`(<- '<database> 'expl '<list of cycle-numbers>)`
: Generate explanations for the facts with the given cycle numbers in the database.

`(explain-attribute '<number> '<attribute>)`
: Generates an explanation for an individual attribute of the frame which matches the numbered pattern element. May only be used in the RHS.

`(expl-attribute '<cycle> <attribute>)`
`(<- '<database> 'expl-attribute '<cycle> '<attribute>)`
: Generate an explanation for an individual attribute of the frame with the given cycle number.

The explanation format is shown in the following example.

```
(p compute-Ti-5
  (New controller created)
  (frame PID-controller
     Ti off
     Td -Td)
test
  (> -Td 0)
-->
```

```
  (remove 1)
  (modify 2 Ti ^(* 4 Td))
  (fact Ti changed))

->(make PID-controller K 7 Td 2)

->(fact New controller created)

->(db)
Facts in db <database f0>
Cycle  Fact

1.     (PID-controller-frame-0 PID-controller K 7 Ti off Td 2)
2.     (New controller created)

->(run)

->(db)
Facts in db <database f0>
Cycle  Fact

3.     (PID-controller-frame-0 PID-controller K 7 Ti 8 Td 2)
4.     (Ti changed)

->(expl 4)

(Ti changed) was concluded by <database f0>
using rule:

(p compute-Ti-5
     (New controller created)
     (frame PID-controller  Ti off Td -Td)
test (> -Td 0)
-->  (remove 1)
     (modify 2 Ti ^(* 4 Td))
     (fact Ti changed))

together with the facts

1. (PID-controller-frame-0 PID-controller K 7 Ti off Td 2)
2. (New controller created)

->(expl 3)

(PID-controller-frame-0 PID-controller K 7 Ti 8 Td 2) was
entered from top-level.
```

```
->(expl-attribute 3 k)

The attribute
k
of the fact
(PID-controller-frame-0 PID-controller K 7 Ti 8 Td 2) was
entered from top-level.

->(expl-attribute 3 Ti)

The attribute
Ti
of the fact
(PID-controller-frame-0 PID-controller K 7 Ti 8 Td 2) was
concluded by <database f0>
using rule:

(p compute-Ti-5
     (New controller created)
     (frame PID-controller  Ti off Td -Td)
test (> -Td 0)
-->  (remove 1)
     (modify 2 Ti ^(* 4 Td))
     (fact Ti changed))

together with the facts

1. (PID-controller-frame-0 PID-controller K 7 Ti off Td 2)
2. (New controller created)
```

The value of automatically generated explanations are debated (Swartout, 1983). A disclaimer often heard is that the explanation given is too dependent on the actual representation of the domain knowledge than on the domain knowledge itself. This results in explanations which are unnatural to the user and thus difficult to understand. For debugging purposes during development, however, the feature is valuable. Another fact that points in direction of mainly using the explanation feature during development is that it is time and space extensive. Due to this, the generation of explanation information can be turned on and off with a switch.

Rules as methods

Another slight modification that has been done allows the association of methods to frame structures. The methods are implemented as rules.

Normally, the **(run)** command only returns **nil**. It has been modified to return the value returned by the RHS of the last rule executed. The value returned by a RHS is the value returned by the last Lisp expression in the RHS. This modification in combination with the possibility to start the rule execution automatically when a new fact is added enables a programming style which resembles object-oriented programming with message passing. Consider the following example.

```
(defframe ship ""
  (x-velocity y-velocity)
  ())

(p speed-method-of-ship
   (send -ship :speed)
   (-ship ship
     x-velocity -x
     y-velocity -y)
-->
   (remove 1)
   (sqrt (+ (* -x -x)
            (* -y -y))))


->(make titanic ship x-velocity 5 y-velocity 5)

->(fact send titanic :speed)
7.071
```

This only works if the messages are added from outside the database. Facts added from rules within the database will not cause **(run)** to be called since the database already is running.

Other modifications

The rule syntax of YAPS can sometimes be difficult to understand. To somewhat remedy this, an optional description string can be included in the rule definition. This looks as follows.

```
(p rule-name
   ["Textual description"]
  <rule-body>)
```

Apart from the major modifications described, several new functions have been added to the system. The most important of these are as follows

(<- '<database> 'facts)
    Returns a list of all the facts in the database.

`(remove-fact '<fact>)`
`(<- '<database> 'remove-fact '<fact>)`
Removes a fact equal to `<fact>` from a database.

`(fact-nr '<number>)`
Returns the fact matched by pattern element `<number>`. RHS function.

`(describep '<rule-name> ...)`
Prints the textual description of the rules.

The possibility to easily modify the YAPS system is mainly credited to the clean Flavor implementation. It is the author's conviction that access to good quality source code is necessary if a conventional expert system framework is going to be used for non-standard applications.

7.4 The scheduler

An implementation of the knowledge-based control framework described in Section 7.1 has been performed on a VAX 11/780. The implementation is based on the YAPS system and the key feature exploited is the ability to use Flavor instances as database elements.

The scheduler is implemented as a flavor which inherits a YAPS database. The scheduling strategy is represented with rules. Each knowledge source is represented as a knowledge source frame in the scheduler database. The definition of the knowledge source frame looks as follows in a slightly simplified form.

```
(defframe knowledge-source

  "Contains domain knowledge for a certain task"

  (name
   (type "The type of the knowledge source e.g.
          forward, backward ..")
   (status "External state of the knowledge source,
            active or inactive" :default inactive)
   (state "Internal state of the knowledge source,
           inactive, waiting, ready or running"
           :default inactive)
   (description "Textual description of the knowledge
           source")
   (instance "Flavor instance that actually implements
              the knowledge source"))
  ())
```

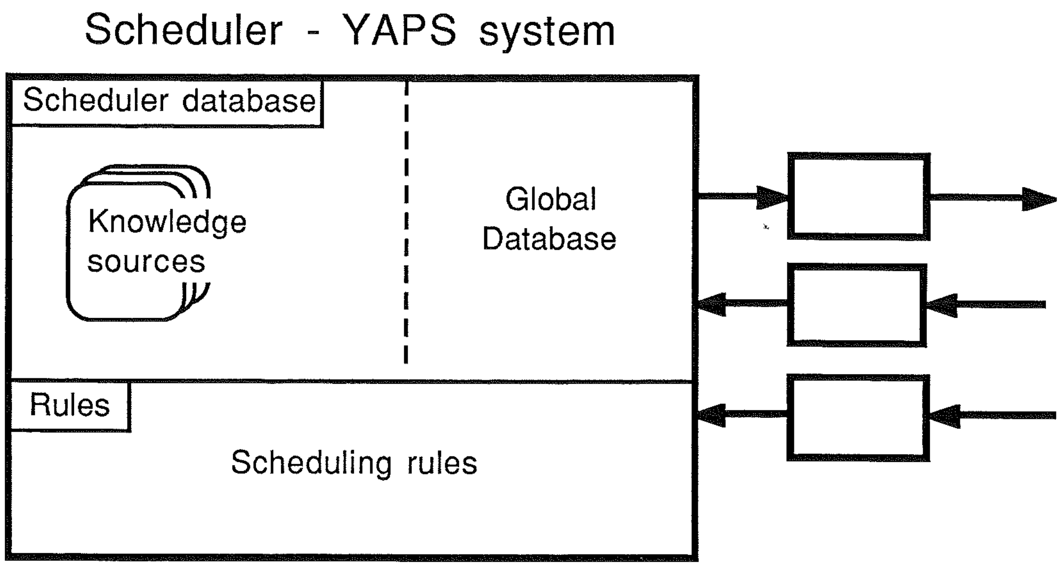

Figure 7.2 Implementation structure

The different types of knowledge sources are represented as different flavors. Each individual knowledge source is implemented as an instance of the corresponding flavor. The implementation structure is shown in Figure 7.2. The actual interface between the knowledge sources and the scheduler consists of a relatively small set of messages for which the knowledge source flavors should supply methods. This makes it easy to add new types of knowledge sources to the system. The message set contains, e.g., the messages `run`, `fact`, `make`, `modify`, `rm-fact`, `db`, etc.

The attribute `status` has the value `active` when the knowledge source has been started and not yet is finished. The attribute `state` indicates the internal state of the knowledge source when it is active. Legal values are `running`, `ready`, and `waiting`. The meaning of these values are the same as in a conventional operating system.

A slightly simplified example of a rule in the scheduler is given below.

```
(p schedule1
  "If a knowledge source is ready and no other operator is
   running then run this operator"
   (frame knowledge-source
      status active
      state ready
      instance -x)
(~ (frame knowledge-source
        state running))
  -->
     (modify 1 state running)
     (<- -x 'run))
```

The recency-based conflict resolution strategy of YAPS gives rise to a last-in, first-out (LIFO) queue of ready knowledge sources in the scheduler. Other strategies such as first-in-first-out (FIFO) queues or priority-based queues are, however, easily implemented with only a few rules. This is one example were the flexibility of rule-based programming shows up. Concurrent process scheduling follows a well-defined algorithm and could thus just as well be implemented with conventional programming techniques. During the development phase of the knowledge-based control framework, when frequent changes were made to the scheduler, the rule-based approach, however, showed very suitable.

Database elements belonging to the global database are represented both in the scheduler and in every single knowledge source. Database elements local to a knowledge source are only represented there. The scheduler also contains database elements that are specific to the scheduling activity, e.g., the knowledge source frames. Other such elements are described in the next section.

Adding a fact to the global database or modifying a frame in the global database has the effect that the modifications are performed in every single knowledge source and in the scheduler. This is ineffective but has the advantage of a clean implementation. It also has the advantage that local and global database elements are treated equally within each knowledge source. The effectivity issues are discussed more in the next section.

The scheduler flavor implements methods for a number of messages. Some of them are `global-fact`, `global-remove`, `global-make`, `global-modify` and `global-db`. These messages correspond to the standard YAPS messages described in Sections 7.2 and 7.3 and operate on the global database. The `global-db` message prints out the database elements which are members of all the knowledge sources and the scheduler. Apart from the `db` method, every knowledge source must also implement the `local-db` method which prints the database elements of a knowledge source that not belongs to the global database.

The scheduler is connected to the other processes by a rule that waits for incoming messages. This rule is executed whenever no other rules are matched. A local mailbox with priorities is used according to the description in Section 4.2. Messages from the man-machine communication are evaluated as Lisp expressions and their results are returned. All other messages are added as database elements locally in the scheduler. The rule looks as follows.

```
  (p restart-rule
    (restart)
  -->
```

```
    (refresh 1)
    (getmessage))
```

The getmessage function performs the actual reading from the internal mailbox. The refreshing of the restart element before the call to getmessage will give the restart element a smaller cycle number than elements which are added due to the incoming message. The recency based conflict resolution strategy causes this rule to be run again when no other rules are matched. If the order of the refreshing and the call to getmessage were interchanged, an infinite loop would be created where the restart-rule was executed all the time.

7.5 Forward chaining knowledge sources

The forward chaining knowledge sources are implemented as instances of a flavor that inherits a YAPS database. This gives a structure where several YAPS systems reside as database elements inside the scheduler YAPS system. The goal-directed conflict resolution strategy is used without any goal elements, i.e., the rule execution must be started by an explicit `(<- <knowledge source> 'run)` command. The reason for this is that the scheduler should have control over which knowledge source that is currently running.

Several new functions are provided for database manipulations apart from the ones already available through YAPS. The most important of these are the following.

`(global-fact  <expression> ..)`
: Encapsulated the expressions in a list and adds them to the global database. Calls the global-fact method of the scheduler.

`(fact-in <knowledge source>  <expression> ..)`
: Encapsulates the expressions in a list and adds it to the specified knowledge source.

`(scheduler-fact  <expression> ...)`
: Encapsulates the arguments in a list and adds it to the scheduler. Equivalent to
`(fact-in scheduler  <expression> ..)`.

`(global-make <make-body>)`
: Creates a frame instance in the global database. Calls the global-make method of the scheduler.

`(make-in <knowledge source> <make-body>)`
: Creates a frame instance in the given knowledge source.

`(scheduler-make <make-body>)`
: Creates a frame instance in the scheduler. Equivalent to
`(make-in scheduler <make-body>)`.

`(global-remove '<number> ...)`
: Removes the database elements corresponding to the enumerated patterns globally. Calls the scheduler's **`global-remove`** method.

`(remove-in <knowledge source> '<number> ...)`
: Removes the database elements, corresponding to the enumerated patterns, in the given knowledge source.

`(global-remove-fact <fact>)`
: Removes a fact equal to `<fact>` globally. Calls the scheduler's **`global-remove`** method.

`(remove-fact-in <knowledge source> <fact>)`
: Removes a fact equal to `<fact>` in the given knowledge source.

`(global-modify '<number> <attr1> <val1> ...)`
: Modifies a frame globally. The number corresponds to the patterns in the LHS of the rule.

`(modify-in <knowledge source> '<number> <attr1> <val1> ...)`
: Modifies a frame in the given knowledge source. The number corresponds to the patterns in the LHS of the rule.

`(global-modi <name> <attr1> <val1> ...)`
: Modifies a frame globally. The frame is specified through its unique identifier.

`(modi-in <knowledge source> <name> <attr1> <val1> ...)`
: Modifies a frame in the given knowledge source. The frame is specified through its identifier.

Efficiency

Global database alteration is potentially ineffective since it requires that the alterations are performed in every single knowledge source. Knowledge sources implemented in YAPS improves the situation. YAPS stores rules and partial pattern matchings in a discrimination network. All YAPS systems share the same discrimination network. This is favorable with respect to space requirements. Different knowledge sources which contain the same rules or rules with similar condition elements will share the corresponding parts of the network. It is, however, still inefficient with respect to speed. Consider the following example. Suppose that the system contains `n` YAPS knowledge sources. A frame modification would then require `2*(n + 1)` network traversations. The extra one comes from the scheduler.

The situation can be improved with a slight modification. What is needed is network traversing functions for database addition and removal that perform the alterations in all YAPS databases. This would decrease the number of traversions to 2. The implementation of these functions is not a major effort.

The described solution would only decrease the time taken into account for the network traversation. The time taken into account for the actual pattern matching which is performed during the traversation would still be the same. In the current system, facts are added and removed to a knowledge source independently if it is active or inactive. This results in a lot of, sometimes unnecessary, matching activity in inactive knowledge sources. Consider, e.g., the situation where global database modifications cause rules to be matched and later unmatched in an inactive knowledge source. One solution to this would be to associate a small database buffer to each YAPS knowledge source. For inactive knowledge sources, database alterations are stored in this buffer in their order of appearance. When the knowledge source eventually gets activated, the database alterations are performed. The important thing is that not all database alterations need to be performed. Examples of this are facts which are added and later removed or several modifications of the same frame which can be merged to a single modification. The drawback of the solution is that the activation of a knowledge source will take longer time.

Real-time primitives

Several real-time primitives have been added to the forward chaining knowledge sources. The primitives basically halt the execution of the knowledge source for a certain time or until a certain database element is added. They are the forward chaining equivalents of the waittime and waitevent primitives of an ordinary operating system.

`(waittime '<time> '<tag>)`

The knowledge source suspends its operation after the current RHS has been executed. After the given time, the fact `(wakeup <tag>)` is added to the knowledge source and it is resumed. The format of `<time>` is a string, `"hh:mm:ss"`. The `<tag>` could be either a string or a symbol. Since the RHS of a rule is considered atomic, several waittime requests can be pending simultaneously if they are issued in the same rule.

`(timer-request '<time> '<tag>)`

A `waittime` function where the knowledge source is not suspended until all executable rules have been fired. The knowledge source is resumed and the fact `(wakeup <tag>)` is added after the specified time.

`(waitentry '<symbol1> ... [:timeout '<time> '<tag>])`

The knowledge source is suspended after the current RHS has been executed. The knowledge source is resumed when a fact that matches `(<symbol1> <symbol2> <symbol3> ...)` has been added to the scheduler, i.e., locally in the scheduler or in the global database. The matching fact is added to the knowledge source if it is not already there. A maximum number of three significant symbols in the beginning of a fact are allowed. General patterns would be desired instead of the sequence of constant symbols above.

This would, however, require that facts with variables were allowed in the database and thus a unification type of pattern matching. If the optional timeout arguments are given, the fact **(timeout <tag>)** is added to the knowledge source if the wait condition is not fulfilled during **<time>**.

(waitmessage '<symbol1> ... [:timeout '<time> '<tag>])
The same as **waitentry** with the matching fact restricted to an incoming message.

(entry-request '<symbol1> ...[:timeout '<time> '<tag>])
Is to **waitentry** as **timer-request** is to **waittime**.

(message-request '<symbol1> ... [:timeout '<time> '<tag>])
Is to **waitmessage** as **timer-request** is to **waittime**.

(waitattribute '<name> '<attribute> '<value> [:timeout ...])
The knowledge source is suspended after the current RHS has been executed. The knowledge source is resumed when the **<attribute>** of the global frame **<name>** takes the given **<value>**. Timeout is optional.

(attribute-request '<name> '<attribute> '<value> [:timeout ...])
Is to **waitattribute** as **timer-request** is to **waittime**.

The above real-time primitives allow a knowledge source to be waiting for several different things simultaneously. They do not allow the knowledge source to express, e.g., that it wants to wait until one out of several database elements has been added to the database, i.e., a disjunctive wait or that it wants to wait until a number of database elements are in the database simultaneously, i.e., a conjunctive wait.

The requirement posed by the implementation that individual knowledge sources may not be interrupted, influences the operation of the real-time primitives. For example, a knowledge source which is waiting a certain time will only be resumed at that time if no other knowledge source is then running. Otherwise the waiting is prolonged.

The basic real-time primitives presented have been used for construction of higher-level real-time primitives. One example of this is testing of rules with a certain time interval. This is arranged with the following rules.

```
(p test-1
   (wakeup test-1)
   (rule test-1 -period)
   <rule patterns>
-->
   (remove 1)
   <actions>
   (timer-request -period test-1))
```

```
(p not-fulfilled
   (wakeup -tag)
   (rule -tag -period)
-->
   (remove 1)
   (timer-request -period -tag))


(p initialize
   (rule -name -period)
   (start -name)
-->
   (remove 2)
   (fact wakeup -name))
```

The rule `test-1` is the actual rule that should be tested with a certain time interval. The LHS side of this rule is possibly satisfied when the `wakeup` element is added. If so, the actions of the rule are evaluated and a request for a new wakeup is made. The use of `timer-request` instead of `waittime` allows other rules, which become satisfied due to the actions, to be evaluated. The `not-fulfilled` rule acts as the `else` part in an `if .. then .. else ..` construct. It is always satisfied when the `wakeup` element arrives. It is, however, only fired if the periodic rule is not satisfied. This is due to the conflict resolution strategy which favors rule complexity. The `initialize` rule starts the timing. The `initialize` and `not-fulfilled` rules are common to all periodic rules. The rule period can be changed by modifying the `rule` database element. With the described constructs it is possible for one forward chaining knowledge source to have multiple periodic rules.

Since the periodic rules wait a certain time interval and not for an absolute time, a certain time sliding will occur. This could have been avoided with similar real-time primitives where instead an absolute time was given.

Another real-time feature that has been implemented is the possibility to associate duration times with database facts and frame attributes. This is done with the following syntax.

```
(global-fact Example of a fact :duration 00:01:00)

(global-modify 1 attribute1 value1 :duration 00:00:45
          attribute2 value2 :duration 00:02:00)
```

In this example, the `(Example of a fact)` fact will be removed after one minute. The values of the frame attributes will only hold for the specified duration. After that they will take their default values, if specified, or else `nil`. An alternative solution for frame attributes would be to use the previous values of the attributes instead of the default values. Duration times can also be automatically associated with certain frame attributes. This is done with the `:duration` keyword in the frame declaration. Durations are not automatically propagated to database elements whose addition have been caused by elements with durations. In the current implementation, duration times may only be associated with global database elements.

Other real-time constructs are easy to implement. One example could be database modification primitives where the actual modifications take place after a given time. This feature is available in the YES/MVS system (Milliken *et al*, 1986) which is built upon OPS5. This feature could also be used to implement periodic rules.

The combination of the real-time primitives described, with numerical algorithms gives additional possibilities. For example, a level crossing detection algorithm can be used to construct primitives for waiting until a certain signal, e.g., the measured signal, **y**, exceeds a certain value.

Real-time primitives implementation

The real-time primitives are implemented using an extra timer process. The call to a real-time primitive usually results in two things. A frame that contains information about the calling knowledge source is added to the scheduler and a message is sent to the timer process. Consider the following example. A knowledge source, `k-s-1`, evaluates the primitive `(waittime "00:00:30" "Start again")`. This creates an instance of the following frame in the scheduler:

```
(defframe waittime-entry

  "Contains information about a pending
   waittime request"

 ((name "The knowledge source that is waiting")
  (receive-time "Absolute time when the waiting is over")
  (tag "Request identification"))
 ())
```

Suppose that the actual time was `14:45:00` at the time of the call. The instance will then look like

```
(waittime-entry-frame-0 waittime-entry
   name k-s-1
   receive-time "14:45:30"
   tag "Start again").
```

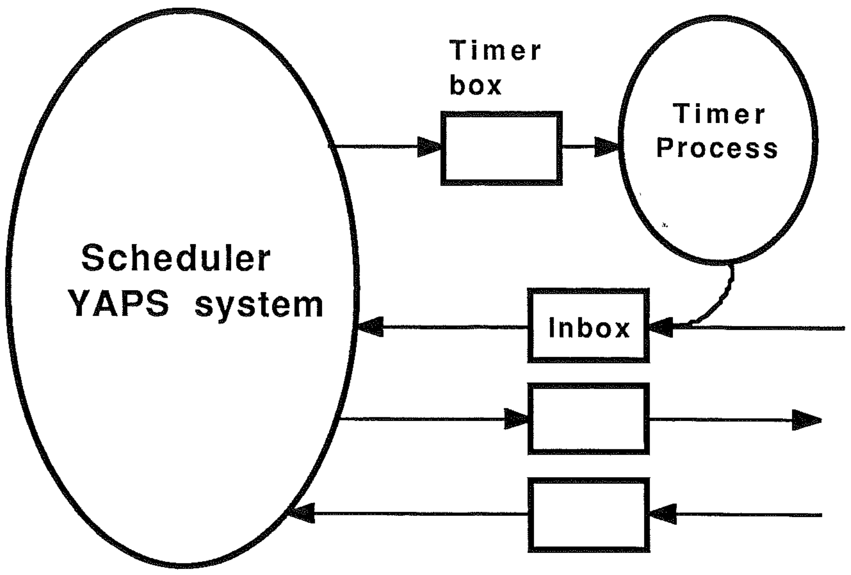

Figure 7.3 Implementation structure

A message that contains the desired wakeup time and the identification tag is sent to the timer process. A high-priority message with the identification tag is returned to the scheduler when the waiting time has elapsed. The fact `(wakeup "Start again")` is added to the knowledge source `k-s-1` and its state is changed to ready. The system together with the timer process is shown in Figure 7.3.

A forward chaining knowledge source is considered to be finished, and thus becomes inactivated, when no matching rules are found and the knowledge source has no pending wait requests.

The timer process is written in Pascal. The basic data structure is a linked list of nodes. A node contains the absolute wakeup time and the request identification tag for a wait request. The nodes are inserted in the linked list in time order. The basic structure of the process is as follows.

```
   while true do
   begin
     if queue_empty then set_timer(infinitely_long_time)
       else set_timer(first_node_time);
w:   wait_for(timer or incoming_message);
     if incoming_message then
     begin
       read_message;
       create_node;
```

```
        insert_node_in_time_order;
        if queue_was_empty or
           (new_time < first_node_time)
         then set_timer(new_time);
        go to w;
      end;
      remove_first_node;
      write_return_message;
    end;
```

The possibility to simultaneously wait for two different events, i.e., the timer and an incoming message, is implemented with the asynchronous system trap (AST) facility in VMS.

7.6 Backward chaining knowledge sources

A limited version of backward chaining knowledge sources has been implemented. It was inspired by a small expert system example in Winston and Horn (1981) which has been extended and embedded into Flavors.

The global database may contain two different types of information, arbitrary lists and frames. The backward chainer internally represents frames as object-attribute-value triplets. The syntax for how a rule is added looks as follows.

```
(<- '<knowledge source> 'buildp
   '<rule name>
   '(<antecedent form> ...
    -->
     <consequent form> ...))
```

```
<antecedent form>   :== <list>|
                        (<frame> <attribute> <value>)
<consequent form>   :== <list>|
                        (<frame> <attribute> <value>)
<frame>             :== <frame identifier>|
                        <matching variable>
<attribute>         :== <symbol>
<value>             :== <lisp value>|
                        <matching variable>
<frame identifier> :== <symbol>
```

The antecedent forms are combined by an implicit ‘and’, i.e., all the antecedents must be fulfilled for the rule to be considered true. The result of an object-attribute-value triplet consequent is the modification of the corresponding frame.

A list consequent is added as a fact. All database modifications are global. Backward chaining knowledge sources may not have any local databases.

Matching variables may be used in the rules. The variables may however only be used in the object place and in the value place of an object-attribute-value triplet. The backward chainer can basically be used in two ways. A value for an empty frame attribute may be searched for. Empty here means that it has the value `nil`. In this case, the query looks like `(frame-1 attribute-3 -x)`. Alternatively stated, the goal of the backward chainer is to find a value for a certain attribute. The second way is for verifying a certain hypothesis. The hypothesis is expressed as a frame attribute having a certain value. The query now looks like `(frame-1 attribute-3 value)`.

Queries which should be verified by a backward chaining knowledge source are added to the scheduler in the following form.

```
(verify <query>)
```

An extra attribute, verify-list, has been added to the knowledge source frame in the scheduler. The list contains the attributes or facts which can be verified by that knowledge source. The scheduler sends the message

```
(<- <knowledge source> 'fact '(verify <query>))
```

to the knowledge source and changes the state of the knowledge source to ready. The knowledge source adds the query to a list of unanswered queries. The verification of the queries begins when the knowledge source receives the `run` message.

Backward chaining systems usually ask the user when a goal cannot be deduced in any other way. The user in this case is the operator. A knowledge based controller is mainly an autonomous system. The relevance of asking the operator could therefore be questioned. For some tasks, however, it is appropriate. One example is the knowledge extraction phase where the operator is asked for a priori process knowledge. Another example is tasks that require information which is difficult to extract by numerical algorithms. The human eye can easily extract qualitative information from, e.g., plotted time series.

Frame attributes may be declared as being allowed to ask for. This is done with the keyword `:askable` in the frame declaration. Facts can also be declared askable. This is done by including them in the global association list `*askable-fact-list*`.

Asking questions to the user also creates real-time problems. Consider the following situation. A backward chaining knowledge source asks the operator for information. During the time which elapses before an answer is given, new messages may arrive that require attention. It is therefore not acceptable to halt the entire knowledge-based system subprocess when a question is asked. If instead the individual knowledge sources were implemented as individual subprocesses, it would be possible to suspend only the subprocess which asked the question.

This does, however, not solve the problem completely. A situation can occur where, during the question, new information arrives that change the conditions which generated the question.

The backward chaining knowledge sources solve the problem as follows. The actual question is written on the terminal. Each question has an associated unique number. The goal which raised the question is saved and the knowledge source is suspended. The reason for the suspension is that the knowledge source is waiting for a database element that looks like

```
    (answer <unique number> <answer>).
```

When an answer with the corresponding number has been given, it is sent to the knowledge source. The answer is saved in the knowledge source. When a run message is received, the knowledge source starts to verify the original top-level query again. If nothing has changed, the need to answer the same question will arise again. Before actually asking, the knowledge source checks if it already has an answer for the question. In that case, this answer is used. If something does have changed, the backward chainer will notice it and alter its line of reasoning. The unique number associated with each question allows multiple backward chaining knowledge sources to be active and ask questions without risk of intermixing the answers. This solution, where the partial verification of a goal may be repeated is simple but inefficient. It is, however, correct from the real-time aspect.

It is not obvious when a backward chaining knowledge source should be considered finished and thus become inactivated. The solution taken here considers a backward chaining knowledge source to be active as long as it is not explicitly inactivated by another knowledge source. Explicit activation and inactivation are described in Section 7.8. An extra state value, **pending**, has been added to take care of the situation when a backward chaining knowledge source is active and ready to receive verification requests.

The questions that are posed to the user has the following forms.

```
Goal: (<frame> <attribute> <value>)

Question: 7 Is the attribute <attribute> of the frame <frame>
equal to <value>?

Legal answers: yes,no,?,(why <number> ...)


Goal: (<frame> <attribute> -x)

Question: 8 What is the <attribute> of <frame>?
```

```
Legal answers: <value>,unknown,?,(why <number> ...)


Goal: <fact>

Question: 9 Is this true: <fact>?

Legal answers: yes,no,?,(why <number> ...)
```

All answers should be preceded by the corresponding question number. Negative answers to any of the questions are not recorded in the database. This means that the current system basically works according to the negation as failure principle. A frame attribute that has the value nil is considered to have no value and may thus be asked for.

Answering with a question mark gives an explanation about the attribute or fact in question. The explanation text used is the attribute description string of the frame declaration. In the case of an ordinary fact, the text is taken from the association list `*askable-fact-list*`.

"Why?" explanation facilities are provided. Answering a question with, e.g., the list `(why 1)`, will print out the rule which caused the question to be asked. Subsequent numbers refer to consequents in the first rule and so on. Consider the following example.

```
(defframe control-loop ""
  ((changed-sign? "Yes or no depending on if the
                  derivative changed sign at
                  the relay switching time" :askable)
   (system-type "SPR, Dominating-time-delay
                or Second-order" :askable)
   (noise-level "low, normal or high" :askable))
   (design-principle "high-gain-feedback,
                      dead-time-compensation or
                      PID design" :askable))
  ())

(<- backward-1 'buildp
   'rule-1
   '((-x system-type SPR)
     (-x noise-level low)
   -->
     (-x design-principle high-gain-feedback)))

(<- backward-1 'buildp
   'rule-2
   '((-x changed-sign? yes)
   -->
     (-x system-type SPR)))

(global-make loop-3 control-loop)
```

The query `(loop-3 design-principle -x)` will result in the following interaction.

```
Question: 0 Is the attribute changed-sign? of the frame loop-3
equal to yes?

->(answer 0 ?)

Yes or no depending on if the derivate changed sign at the
relay switching time

Question: 1 Is the attribute changed-sign? of the frame loop-3
equal to yes?

->(answer 1 (why 1))

It is needed during the evaluation of
Rule rule-2
(loop-3 changed-sign? yes)
-->
(loop-3 system-type SPR)

Question: 2 Is the attribute changed-sign? of the frame loop-3
equal to yes?

->(answer 2 (why 1 1))

It is needed during the evaluation of
Rule rule-1
(loop-3 system-type SPR)
(loop-3 noise-level low)
   -->
(loop-3 design-principle high-gain-feedback)))

Question: 3 Is the attribute changed-sign? of the frame loop-3
equal to yes?

->(answer 3 (why 1 1 1))

(loop-3 design-principle -x) is the top-level goal
```

```
Question: 4 Is the attribute changed-sign? of the frame loop-3
equal to yes?

->(answer 4 yes)

.
.
.
```

The current implementation has semantic restrictions on the usage of variables. The value in an object-attribute-value query may, e.g., not be a frame. This has as an effect that it is impossible to express antecedents referring to frames which are the values of other frames. For example, the following rule is not allowed.

```
(rule-x
  (-x subframe -y)
  (-y interesting-value OK)
 -->
  (-x situation checked))
```

In order to circumvent this problem, all attributes belonging to a frame and to the subframes of this frame may be accessed as if they all belonged directly to the top-level frame. This is applied recursively on the subframes. The above example now looks as

```
rule-x
  (-x interesting-value OK)
 -->
  (-x situation checked).
```

This solution is similar to what is allowed in EMYCIN (van Melle *et al*, 1981).

The backtracking that occurs when a subgoal fails does not handle variable assignments, etc. This limits the power and expressability of the rules. The reason for the short-comings is ease of implementation. An better solution would be to use Prolog for backward chaining. The unification type of pattern matching in Prolog would allow variables to be used freely. Prolog would also allow the use of predicates in the backward chaining rules. Small Lisp implementations of Prolog are available, e.g., (Nilsson, 1983; Carlsson, 1983; Abelson and Sussman, 1985).

The only way for backward chaining knowledge sources to obtain knowledge which cannot be concluded from rules is currently to ask the operator. An interesting modification would be to allow the backward chainer to ask questions to other knowledge sources, i.e., to activate knowledge sources that extracts the needed knowledge, and await their result. One building block of such a modification would probably be the possibility to use rules as methods described in Section 7.3.

7.7 Procedural knowledge sources

Certain subtasks are best represented procedurally. Since the knowledge-based system is implemented in Lisp, the most straightforward choice would be to use Lisp. A knowledge source would then consist of a Lisp function that was evaluated when the knowledge source was activated.

This would suffice if the knowledge source could immediately compute its result and become inactivated. If, however, the knowledge source needs to be suspended, e.g., in order to wait for an incoming message, standard Lisp is insufficient. What is needed is a possibility to suspend the execution of a Lisp function and to later resume the execution according to the following:

```
    .
    .
    .
(Lisp function)
(Suspend)
(Lisp function)    ; Evaluated after a resume
    .
    .
```

What is basically needed is a way to save and restore the state of the computation. The `catch` and `throw` primitives of standard Lisp implements this in a limited way. They can, however, only be used as an escape mechanism to jump out of computations. What is needed is a way to also jump into computations. The Lisp dialect Scheme, (Sussman and Steele, 1975; 1978), allows for this. It is done with continuations which are first-order data objects that represent the future of a computation. The approach taken in this system is to use Lisp to implement an interpreter for a procedural Lisp-like language that allows the computation to be suspended. The language will in the sequel be referred to as SLisp (Suspendable Lisp).

The interpreter

The interpreter is based on the interpreter for the language SCHUM (Charniak *et al*, 1980). A similar interpreter is described in Abelson and Sussman (1985). The interpreter is constructed as a register machine with six registers and a stack. The interpreter is frame-based, i.e., during a recursive call to a function, all registers are pushed onto the stack instead of just the ones that are affected by the call. This simplifies the interpreter.

The syntax of SLisp is similar to Lisp. The SLisp language is lexically scoped and functions are first order data types. All elements, also the first, are evaluated in a function application form. The value of the first element is then applied to the values of the rest of the arguments. SLisp data types are ordinary S-expressions. Ordinary Lisp functions and symbols may be used inside SLisp. The basic SLisp functions are the following:

`(s-defun <name> (<arg> ...) <expression> ...)`
Defines a new SLisp function.

`(s-let (<var1> <val1> <var2> <val2> ...) <expression> ...)`
Binds the variables in the variable list to the corresponding values and evaluates the expressions. Returns the value of the last expression. The variable values can be lambda-expressions which have the syntax `(lam (<arg> ...) <expression> ...)`.

`(s-setq <variable> '<value>)`
Assigns the value to the variable. The SLisp equivalent of `setq`.

`(s-cond (<expr> [<expr> ...]) [(<expr> [<expr> ...])])`
The SLisp equivalent of `cond`.

`(s-do ...)`
The SLisp equivalent of `do` with the same syntax.

The SLisp interpreter is implemented as a flavor. Methods that handle the message set necessary for a knowledge source have been implemented. Each procedural knowledge source is connected to one SLisp function. A procedural knowledge source returns the value of the last expression in the body of the associated SLisp function when finished. This value is usually not used since the knowledge sources normally are called from the scheduler. It is, however, also possible for a procedural knowledge source to start another procedural knowledge source and await its returned value.

The SLisp function connected to a procedural knowledge source may have arguments. This is usually not the case when the knowledge source is called from the scheduler. The arguments are used when the knowledge source is explicitly started from another procedural knowledge source. This is described in more detail in the next section. The stack machine registers and the global database are stored as instance variables of the procedural knowledge source flavor. SLisp variables can refer to the contents of the global database. This is done with the following functions.

`(the-frame <type> <arg> <val> <arg> <val> ...)`
Returns the unique identifier of a frame that matches the given pattern.

Evaluation of the arguments can be forced with a ^. If no matching frame is found the value `nil` is returned. The function can be used in the following way.

```
(s-setq x
       (the-frame control-loop
                  atttribute4 a-value))
```

`(all-frames <type> <arg> <val> <arg> <val> ...)`
Returns a list of the unique identifiers of all the frames that match the given pattern. If no matching frame is found `nil` is returned.

`(the-attribute '<attribute> '<frame-identifier>)`
Returns the value of the attribute `<attribute>` in the frame identified by `<frame-identifier>`.

`(fact-match  <pattern>)`
Returns `t` if a fact that matches pattern belongs to the database. Otherwise, it returns `nil`.

SLisp can also modify the contents of the global database. This is done with the functions `global-fact`, `fact-in`, `scheduler-fact`, `global-make`, `make-in`, `scheduler-make`, `global-modi`, `modi-in`, `global-remove-fact`, and `remove-fact-in`. They have the same meaning and syntax as in forward chaining knowledge sources.

Real-time primitives

Real-time primitives have been added to the procedural knowledge sources in approximately the same way as for forward chaining knowledge sources. They are

`(waittime '<time>)`
Returns `t` when the time is over.

`(waitentry '<symbol1> ... [:timeout '<time>])`
Returns `t` when a matching fact has been added. Returns `nil` if a timeout has occurred.

`(waitmessage '<symbol1> ... [:timeout '<time>])`
The same as `waitentry` with the matching fact restricted to an incoming message.

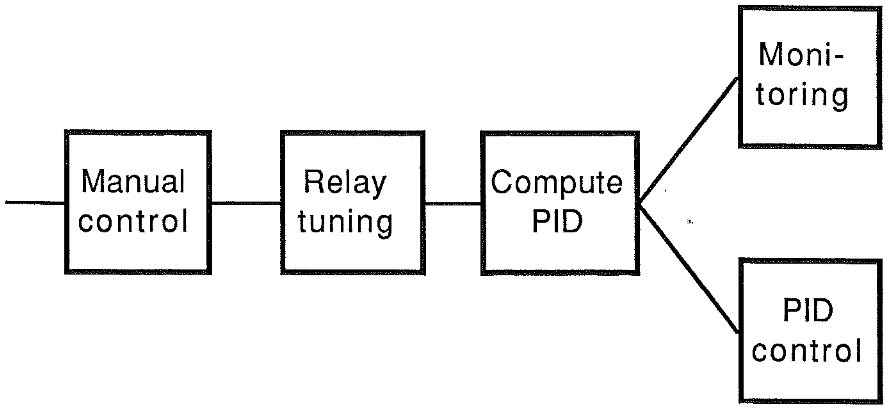

Figure 7.4 Knowledge source sequences

`(waitattribute '<name> '<attribute> '<value> [:timeout '<time>])`
: The same as `waitentry` for the frame case.

The stack machine interpreter simplifies the implementation of the real-time primitives. A suspend operation is basically equivalent to an extra push operation on the stack.

7.8 Combination of knowledge sources

The operation of the knowledge-based controller typically consists of a sequence, with parallel parts, of knowledge source activations. Figure 7.4 describes a typical startup situation. The rightmost part of the figure reflects the monitoring phase during steady-state control. Several knowledge sources are then active simultaneously. One knowledge source takes care of the control algorithm currently executing while the other knowledge sources implement different monitoring aspects. Three different methods for combining knowledge sources into sequences have been implemented.

1. Knowledge source controlled sequencing

The most straightforward way to combine knowledge sources is to let knowledge sources activate and deactivate each other. This is done with the following constructs.

`(activate '<knowledge source>)`
: Activates a knowledge source, i.e., changes its status to active and its state to ready.

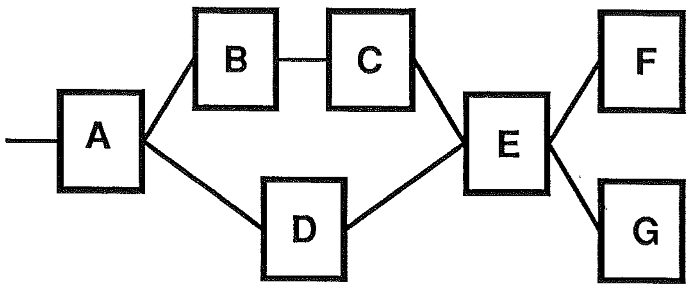

Figure 7.5 Knowledge source sequence

`(deactivate '<knowledge source>)`

Deactivates a knowledge source, i.e., changes its status and state to inactive. Pending waittime requests, if any, are removed.

A knowledge source also has the possibility to wait until another knowledge source is finished. This is done by waiting for the database element `(inactivated <knowledge source>)`.

Procedural knowledge sources also have the possibility to call other procedural knowledge sources and await their result. This is done with the function

`(call '<knowledge source> [<argument> ...])`

The current knowledge source is suspended and the specified knowledge source is activated. The optional arguments are passed to the SLisp function associated with the activated knowledge source. The original knowledge source is resumed when the called knowledge source becomes inactivated. The function returns the value returned by the called knowledge source.

A possible extension to the system would be to allow backward chaining knowledge sources to ask questions to the controlled process using the described construct.

2. Pre-stored sequences

Another alternative is to have a number of pre-stored sequences. Each sequence transforms the state of the system from some initial state to a desired final state. The state concept in this sense contains both the internal state of the knowledge-based system and the state of the controlled process. One example of a sequence could be the initial tuning sequence. Other sequences could be used to return to steady-state control when different alarm conditions have been detected.

A small example of a sequence is shown in Figure 7.5. The capital letters denote the knowledge source names. A sequence can be stored as a nested list structure together with its initial and final states. The sequences are expressed in the following EBNF syntax where **p** and **s** denote parallel and sequential, respectively.

```
PLAN     ::= (OPERATOR ARGUMENT [ARGUMENT..])
OPERATOR ::= <p> | <s>
ARGUMENT ::= <knowledge source> | PLAN
```

The example in Figure 7.5 looks like `(s A (p (s B C) D) E (p F G))` in this notation. The sequence notation has no possibility to express that certain parts of the sequence should be executed conditionally or that a sequence conditionally splits into different branches. Instead, the pre-stored sequence represents the most probable sequence for each initial and final state. Conditional execution is solved by having several pre-stored sequences. Each knowledge source has a set of preconditions that must be fulfilled for the knowledge source to be applicable. Before a knowledge source is started, the preconditions are checked. If they are not fulfilled a new pre-stored sequence must be chosen.

Combination of knowledge sources into sequences is basically a procedural operation. It is therefore natural to express it with procedural knowledge sources. This is not possible with the procedural knowledge source primitives described hitherto. What is needed is wait primitives that allows waiting for multiple events. This is shown in the following example were the sequence from Figure 7.5 is implemented in a procedural knowledge source.

```
(s-defun example-sequence ()
 (s-let (x nil)
  (activate A)
  (waitentry inactivated A)
  (activate B)
  (activate D)
  (s-setq x (waitentry (or (inactivated B)
                           (inactivated D))))
  (s-cond ((equal x '(inactivated B))
           (activate C)
           (waitentry (and (inactivated C)
                           (inactivated D))))
          ((equal x '(inactivated D))
           (waitentry inactivated B)
           (activate C)
           (waitentry inactivated C)))
  (activate E)
  (waitentry inactivated E)
  (activate F)
  (activate G)))
```

The changes that have been made are that **waitentry** also allows conjunctions and disjunctions of fact patterns. The conjunctive form of waitentry returns **t** when all the separate patterns have been matched. This does not necessarily mean that they all still remain in the global database when the knowledge source is resumed. The disjunctive form returns the pattern that first is matched.

This solution is attractive in that it relieves the scheduler from much of the knowledge source combination task. It also easily allows conditional knowledge source execution and sequences with different branches. The sequence selection, i.e., the choice of which procedural knowledge source that should be active, can also be handled by a separate knowledge source. This knowledge source could be viewed as a 'meta-knowledge source' which contains knowledge about other knowledge sources.

3. Dynamic sequence generation

The last and most complex method is to dynamically generate sequences. This can principally be accomplished by associating goal states, i.e., post-conditions, and initial states, i.e., pre-conditions, with each knowledge source. Each knowledge source can be viewed as an operator that transforms the state of the system from its initial state to its goal state. A sequence is recursively generated by comparing the desired goal and the current state with the pre- and post-conditions of the operators.

This formulation turns the problem into a planning problem. The scheduler generates a plan which then is executed. Planning is an AI area where much work has been done. An overview of planning from an AI perspective is given in Appendix E.

Planning possibilities

The current planning possibilities of the system are limited. Each knowledge source has associated sets of preconditions and goals. The planning information is contained in a planning frame in the scheduler.

```
(defframe plan-module
  "Planning information for a certain control loop"
((actual-plan "The plan that the system is currently
               following")
 (control-loop "Name of the control loop")
 (actual-goal "The goal which the actual-plan tries
               to fulfill")
 (goal-stack  "Stack of goals to be fulfilled")
 ())
```

The **goal-stack** consists of a stack of goal expressions. The planning process begins with that the top element of the **goal-stack** is being popped to **actual-goal**. The syntax of a goal expression is the following:

```
(or (<state-description> ...)
    (<state-description> ...)
    ...)
```

The elements in each disjunction term are implicitly combined by conjunctions. The planning system tries to generate a plan for the first term of the disjunction. If that does not succeed, then next disjunction term is tried. The disjunctive goal expressions make it possible to express that either one of two goals are allowed but that one goal is preferred before the other. When a plan has been generated the execution of it starts. The plan execution continues until all the knowledge sources in the last part of the plan are finished. Then the next goal element on the **goal-stack** becomes the actual goal.

The preconditions of the individual knowledge sources has the same syntax as the global goal expressions. The different disjunctive terms describe the different situations under which the knowledge source is applicable. The goals of a knowledge source consists of a conjunction of state descriptions.

A plan is generated in reverse order by comparing the final goal with the goals of the individual knowledge sources. Each knowledge source that is needed to fulfill a goal generates a parallel branch in the plan. The preconditions of the chosen knowledge source now become new goals, etc. This results in a plan that is represented in reverse order and which branches every time multiple knowledge sources are needed. When a complete plan has been generated, the plan representation is reversed and rearranged so that parallel plan parts with equal partial knowledge source orderings are combined. Plans are stored using the prestored sequence syntax. The plan in Figure 7.5 has the following form before it is rearranged:

```
(s (p (s F E (p (s D A)
                (s C B A)))
      (s G E (p (s D A)
                (s C B A)))))
```

The planning is performed in a dynamic environment. The dynamic environment may generate events that affect the plan execution. For example, load disturbances may affect a tuning experiment and thus prevent the corresponding knowledge source from fulfilling its goal. To avoid such problems, it is necessary to check that a knowledge source actually has attained its expected goal after its execution. This is implicitly done by checking that the preconditions of each knowledge source are fulfilled before it is applied. If they are not fulfilled, replanning is performed. Replanning is also the means to handle plans with conditional branches. The original plan represents one branch.

The syntax of the state descriptions in the goals and preconditions are as follows:

```
<state-description> :== <symbol>|
                         (<attribute> <value>)
<attribute>         :== <symbol>
<value>             :== <lisp value>
```

All information concerning a control loop is stored in the global database as an instance of the control-loop frame. The information could be stored either directly as attribute values or as subframes. All state information concerning the planning is stored in a list as the value of the attribute `planning-facts`. A state description in the form of a symbol is satisfied when the symbol is a member of the `planning-facts` attribute of the control-loop frame or of any of its subframes. A state description in the form of an attribute-value list is satisfied if the attribute `<attribute>` in the control-loop frame or in any of its subframes has the value `<value>`.

An example of a planning situation is the initial tuning phase. A plan is generated which ends in a steady state situation where several knowledge sources are active in parallel. One of these knowledge sources typically takes care of the control algorithm while the others implement different monitoring aspects. The goal expression which generated this plan could, e.g., be

```
(steady-state-control stability-monitoring ...).
```

The knowledge sources in the final phase of this plan will under normal circumstances never finish. This means that the execution of this plan will never finish.

During steady-state control, changing control conditions may cause the control performance to deteriorate. This should hopefully be detected by a monitoring knowledge source which then pushes a new goal onto the goal stack. This new goal would cause a replanning. The new plan could, e.g., involve a re-tuning of the controller followed by the original goal.

The knowledge sources can typically be divided into two groups: those who have a defined termination and those who continues forever. The latter group causes problems considering when the goals of these knowledge sources should be considered fulfilled. Normally, goals are considered to be fulfilled when the knowledge source is finished. Intuitively, the goals of the never-ending knowledge sources can be considered fulfilled as soon as the knowledge source has started. The planning system does currently not do that. For example, consider the above steady-state control situation. A natural precondition for the stability monitoring knowledge source would be steady-state-control. This is, however, not possible to express in the described planning language. The reason for this is that steady-state-control will never be fulfilled as the result of a finished knowledge source.

What is needed is a way to express that the stability monitoring could start as soon as the steady-state control knowledge source has been started.

A possible solution would be to increase the granularity of the knowledge sources. This would mean that the steady-state control knowledge source was divided into two. One which initialized the controller and one which merely monitored it. This is however not a tempting solution since it would force the decomposition of already well-defined knowledge sources. After all, the main motivation for the knowledge source structure is an attempt to divide the system into natural, well-defined modules. In the current system, the individual state descriptions of the goals are explicitly ordered in a way that ensures that the corresponding knowledge sources will be activated in correct order. In the above example, the steady-state control knowledge source would be started before the stability monitoring knowledge source.

7.9 Discussion

A real-time expert system framework has been developed along the lines of an ordinary operating system. Different knowledge representation methods could be used. The intentional use for the system has been as a framework for development of knowledge-based controllers. The real-time expert system framework *per se* has probably a much wider applicability, e.g., in signal processing or process monitoring.

The description of the system has mainly touched upon functionality. The user interface is another very important part of an expert system framework. In our case two groups of users can be seen: the process operator who is the end user and the control engineer who is responsible for entering the control knowledge. The reason why the user interface has been ignored in the presentation is mainly that this part of the system is not as well-developed as the other parts.

The major reason for the poor user interface is the VAX implementation with alpha-numerical terminals which, e.g., prevents window-based interfaces with mouse interaction. Another reason is that the framework is mainly intended for autonomous applications where the user support is not as important as in a standard, consultative expert system application. Section 8.4 gives an example of the interaction with the system.

The framework also has features which have not been described. The "How?" explanation facility has been carried through in all different knowledge source types. This means that a database element will be explained differently depending on what type of knowledge source that added it. The system also has functions that allow on-line editing and definition of new forward or backward chaining rules and SLisp procedures.

The individual knowledge sources of the current system always run to completion before the control is transfered to the scheduler. With small modifications, the system can be changed to allow interrupts among knowledge sources. The events that may cause a knowledge source to become ready for execution are either external, i.e., the arrival of a message from either the timer process, the algorithm process, or the user interface process, or internal, i.e., the addition or modification of a database element. These events can easily be detected from the knowledge sources. The arrival of an external event is detected by checking the number of messages in the incoming mailbox, **Inbox**. Internal events are detected by examining the number of elements in the scheduler's conflict set immediately before and after a global database modification has been performed. If the number of elements has increased, the database modification caused a match in the scheduler, i.e., the conditions for some waiting knowledge source have been met.

Procedures which performs these types of checks can easily be added to the forward chaining and the procedural knowledge sources. In the forward chaining case the rule interpreter is extended in order to check the number of messages in **Inbox** between each rule execution. If a message has arrived, the knowledge source is suspended. Similarly, the knowledge source is suspended if a global database modification caused an increase of the conflict set of the scheduler. For procedural knowledge sources, the mailbox test can be performed each time a new expression is evaluated. In the backward chaining case, it is not as clear which modifications that are needed.

The current system can only handle one single control loop. The expansion to a multi-loop, DDC-type of system is, however, straightforward. This is basically due to the pattern matching facilities of the system.

The scheduler of the system currently consists of around 25 rules. The dynamic sequence generation feature requires around 20 additional rules.

8

Examples of Knowledge Based Controllers

The purpose of this chapter is twofold. Firstly, to provide examples that show how the real-time expert system framework described in the previous chapter may be used for implementing a knowledge-based controller. This involves examples that show how algorithms and logic can be separated, what knowledge sources may contain and how they can be combined, etc. Secondly, to illustrate various types of "intelligent" behavior that may be exhibited by a knowledge-based controller. The vision behind the knowledge based controller is a controller that is able to reason about process dynamics and specifications. The examples illustrate these ideas in simple settings. The examples are focussed on the initial tuning phase of the controller. Tuning procedures and heuristics for different control design techniques are presented. It should be emphasized that the examples presented are mere examples and thus have limitations.

Two examples are described. The first is an elaborated version of relay auto-tuning. Knowledge about the three points on the open-loop Nyquist curve where the phase is $0°$, $-90°$ and $-180°$ is extracted and used as the basis for the control design. The knowledge-based controller has possibilities for P, PI, PD, PID, and discrete pole-placement design. The knowledge source combination of the first example is done with a pre-stored sequence implemented as a procedural knowledge source as described in Section 7.8.

In the first example it is assumed that the process is linear. The second example considers the initial tuning for nonlinear processes. A less sophisticated tuning procedure is used than in the first example. The controller estimates the degree of linearity of the process and if required automatically builds up a gain-scheduling table. The knowledge source combination is in this example instead

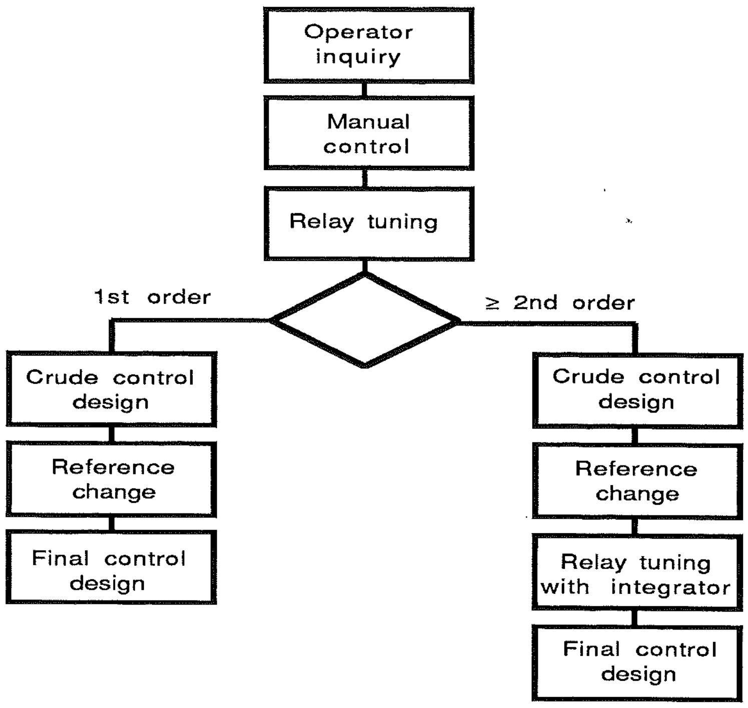

Figure 8.1 Three-point based tuning procedure

implemented using the planning mechanisms described in Section 7.8.

A detailed description of the tuning procedure of the first example is given in Section 8.1. The next sections describe the tools. The numerical algorithms used are given in Section 8.2 and the knowledge sources are described in Section 8.3. A session with the system is described in Section 8.4. Section 8.5. outlines how the first example may be extended. This section further shows some of the power of the knowledge-based control framework. The second example is described in Section 8.6. Less details are given since the tools used are basically the same as in the first example.

8.1 The tuning procedure

The tuning procedure that will be described in this example is relay based. It uses three points on the Nyquist curve: the "$-180°$ point", the "$-90°$ point" and the steady state gain. The standard relay auto-tuning method, (Åström and Hägglund, 1984b), only uses the first of these points as the basis for the control design. This does not work well for all types of processes, (Åström and Hägglund, 1984c).

The tuning procedure follows the flow-diagram of Figure 8.1. The system starts by asking the operator a few questions. The operator is then required to manually control the process until it is in rest at the desired operating point. A relay experiment is performed and based on its results the process is classified either as being of second or higher order or as having dynamics that well may be approximated by a first order system. In the first case, a second relay experiment with an integrator inserted into the feedback loop is performed to acquire the "$-90°$ degree" point on the Nyquist curve. A change in reference value is performed to find out the static gain of the process.

The tuning procedure contains considerable amounts of heuristics. For example, at several stages of the tuning procedure, approximating process models are obtained. The choice of model structure depends on different measurements which may be contradictory. It is often the case that, given a model structure, the parameters in the model can be calculated from different principles with different result. Which model to choose is determined heuristically. The tuning procedure is also a good example of the large complexity of the problems. The following description illustrates the main ideas behind the tuning procedure as well as some of the heuristics used. Space requirements prevent a complete listing of the contents of the different knowledge sources.

Operator inquiry

Three different types of questions may be posed to the operator during the initial phase. The first type concerns information that may be useful for the controller to determine the process dynamics. Currently, only a rough estimate of the dominating time-constant of the process is asked for.

The second type of questions concerns closed loop control specifications. The questions that are currently asked is the value of the maximum allowable relative steady state error and the control objective. The control objective indicates what is most important: fast set-point response or good load disturbance rejection. Other types of specification information that could be requested are closed loop bandwidth, closed loop overshoot or maximum variations in the control signal due to measurement noise during steady state control. Currently, the overshoot and the allowable variations in the control signal are given default values and a reasonable bandwidth is determined by the controller.

The third type of information that is required by the controller regards the configuration. This involves, e.g., information about allowable signal ranges. These are currently given default values.

Manual control

A precondition for most auto-tuning experiments is that the process is in steady state at the desired tuning point. To achieve this, the human operator is required to manually control the process until the system is at rest at the desired operating point. It may be argued that involving the operator in the tuning procedure is in contrast to the spirit of auto-tuning. During the initial phase of the tuning when no process knowledge has been acquired, it is, however, the only sensible thing to do. It assures that the system works and that the process can be controlled around the desired operating point. The argument is supported by extensive industrial experiments with simple auto-tuners (Bååth, 1987).

When the operator has reported that the process is in steady state, the controller attempts to verify this by comparing the mean value of the process output over two subsequent time intervals with the length taken as the time-constant estimation. The measurement noise is determined as the difference of the maximum and minimum values of the process output.

Adding an integrator to the relay feedback loop in order to find out the $-90°$ point is only reasonable for processes without pure integrators. Indications whether this is the case can also be found during manual control. In this experiment, it is assumed that the process has no integral action.

Relay initializations

The value of the relay hysteresis, ε, is determined from the measurement noise. The more measurement noise that is present, the larger hysteresis is needed in order to avoid erroneous relay switchings. A minimal value of the hysteresis is prescribed to avoid problems with too fast relay oscillations. For example, the oscillation frequency for first order system goes to infinity when the hysteresis approaches zero. Since the relay is implemented as an algorithm in the computer, it has a smallest allowable execution rate which may not be exceeded by the relay oscillation frequency. A value of the desired oscillation amplitude is also determined from the noise range. The relay bias is set as the current steady state control signal. During the first half-period of the oscillation the relay amplitude is increased lineary from zero until either it reaches a default amplitude or the error signal exceeds the desired oscillation amplitude. These initializations are similar to the what is used in the regular auto-tuner.

Oscillation measurements

During the relay experiment, the amplitude and hysteresis of the relay are adjusted in an attempt to obtain the desired oscillation amplitude. When the oscillations have stabilized, the oscillations are measured. Several quantities are measured as shown in Figure 8.2. The lengths of two subsequent half periods, $T_c/2$ are measured by counting the time between the relay switching. The oscillation amplitude, A for two subsequent half periods are measured as the peak values of the amplitudes during the half periods. The time, τ that it takes for the error signal to reach its maximum value is measured for two half periods. Six equidistant values of the error signal during one oscillation period, e_0, e_1, e_2, e_3, e_4, and e_5, are measured in order to fit a discrete model to the relay oscillations. The error signal is low-pass filtered to decrease the influence of measurement noise. Three values on a half period are sufficient to fit the second order pulse transfer function $H(z) = (b_1 z + b_2)/(z^2 - az)$, corresponding to the continuous model $G(s) = k_p e^{-sL}/(1 + sT)$, to the oscillations. The reason for taking all measure-

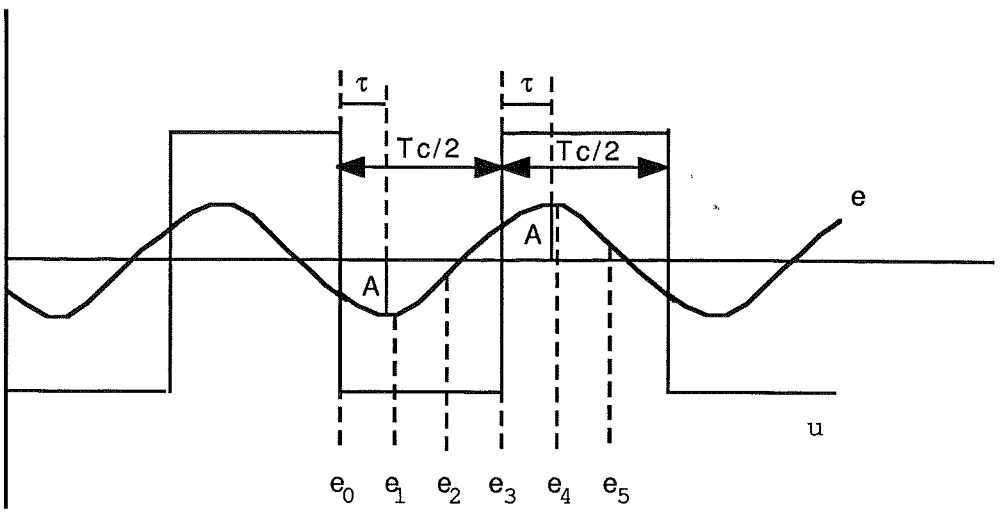

Figure 8.2 Relay oscillation measurements. The value A stands for the peak error amplitude during one half-period.

ments at two subsequent half periods is twofold. It decreases the influence of noise and it gives an estimate of how symmetric the oscillations are.

Oscillation analysis I

The magnitude and phase of the transfer function at the oscillation frequency can be estimated. Using describing function arguments, the phase at this point would be slightly less (since the hysteresis is positive) than $-180°$. This estimate is often poor. It is particularly poor for systems with low order dynamics. For example, first order systems have a phase which always is larger than $-90°$ degrees.

A better estimation of the phase can be made from the value of τ. The estimate is computed as

$$\arg G(i\omega_c) = -\frac{\pi}{2} - \frac{2\pi\tau}{T_c}$$

where ω_c is the oscillation frequency and T_c is the oscillation period. Processes where the process output is sinusoidal will have $\tau = T_c/4$, i.e., a phase of $-\pi$. Processes with $\tau = 0$, i.e., the class of PI-controllable systems defined in Section 3.1, will all have the estimated phase $-\pi/2$ even if the correct phase is larger. A better estimate of the phase would be attained if a FFT analysis was performed during the relay experiment. This would also give an accurate measurement of how sinusoidal the output is. This has not been implemented.

The magnitude of transfer function is determined using the describing function method. The relay has the equivalent gain

$$k_c = \frac{4d}{\pi A}$$

where A is the measured peak error amplitude. Hence,

$$|G(i\omega_c)| = \frac{\pi A}{4d}.$$

The quality of this estimate depends of the actual phase and will be best for processes with $-180°$ phase. For PI-controllable systems the estimate is entirely wrong. Here, the measured oscillation amplitude is determined only by the relay hysteresis. The lack of information is no problem since for these processes, the control design is not based on measured points on the Nyquist curve.

At this stage in the tuning procedure the process is classified either as a PI controllable system, i.e., it has dynamics that well may be approximated by a first order system, or as a system of second or higher degree. The criterion $\tau/T_c < 0.05$ is used to test if the process can be approximated with a first order system, $G(s) = k_p/(1+sT)$.

PI-controllable systems

For PI-controllable systems it is not worthwhile to proceed with a new tuning experiment with an integrator. Instead, a discrete model fitting is performed. The proper discrete model would be $H(z) = b_1/(z-a)$. Using the arguments behind the discrete model fitting method of Appendix A, it can be shown that a model fitting here only requires 2 samples per half period. It is however possible to use the previous second order discrete time model. For a first order system the coefficient b_2 is small compared to b_1 and may be set to zero. Comparing the magnitudes of b_1 and b_2 gives an additional possibility to test if a first order model is appropriate. In the current system this is not done.

A continuous model is computed from the discrete model and and validated. Several alternative methods can be used. In the system, the phase and magnitude of the model at the oscillation frequency is compared against those calculated from the measurements. The measured oscillation period is also compared against the theoretically computed value. For a first order system, $G(s) = k_p/(1+sT)$, under relay feedback with relay amplitude d and hysteresis ε, the exact value of the oscillation period is

$$T_c = -2T \ln \frac{k_p d - \varepsilon}{k_p d + \varepsilon} \approx \frac{4\varepsilon T}{k_p d} \tag{8.1}$$

In the current version the result of validation is merely printed on the terminal. An alternative would be to reject the model, if the result of the validation is not good enough. A third validation method which not has been implemented is to compare the measured values of the error signal against the theoretical ones given by Theorem 3.2.

The measured discrete model does not always has realistic parameter values, e.g., $0 < a < 1$. Problems arise in particular when the relay hysteresis is small, i.e., the oscillation frequency is high. In that case, the oscillation curve becomes piecewise linear, i.e., the system has the dynamics of a pure integrator. The measured values then approach the values

$$e_0 = \varepsilon \qquad e_1 = \frac{\varepsilon}{3} \qquad e_2 = -\frac{\varepsilon}{3}$$

and hence, according to Appendix A, $a = 1$. In this case, noise disturbances may easily cause a to become larger than one. This means that the continuous model parameters k_p and T cannot be computed. The ratio between them can, however, be computed through Equation (8.1).

In order to complete the continuous model one of the coefficients k_p and T must be determined. This can be done in several ways. The relay experiment can be repeated with a larger hysteresis value. This takes some time. The approach taken in the experiment is to design a crude controller from the available knowledge and to make a small change in the reference value. The steady state gain k_p can then by computed by measuring the difference in steady state control signals.

A PI controller is designed as if the process had integrator dynamics, i.e., $G(s) = k_i/s$, where k_i is determined from the equation for the exact oscillation period for an integrator

$$T_c = \frac{4\varepsilon}{dk_i}$$

The gain K of the PI controller is determined as

$$K = \frac{\Delta u}{\Delta n}$$

where Δu is the specified allowed steady state control signal deviations and Δn is the measured noise range. The integration time T_i is determined from the characteristic equation

$$s^2 + 2\zeta\omega_0 s + \omega_0^2 = s^2 + Kk_i s + \frac{Kk_i}{T_i}$$

Hence

$$T_i = \frac{4\zeta^2}{Kk_i}$$

When an estimate of the steady state gain has been achieved, T_i is adjusted to instead fulfill the characteristic equation

$$s^2 + 2\zeta\omega_0 s + \omega_0^2 = s^2 + \frac{(1 + Kk_p)}{T}s + \frac{Kk_p}{TT_i}$$

This design is also used if a complete first order model is obtained initially. The integral part may be omitted if the steady state error with proportional control, $1/(1+Kk_p)$, is smaller than the specified allowed steady state error.

Systems of second or higher order

If the system cannot be approximated by a first order model, a discrete-time process model is computed. Three alternative models can be computed depending on the relation between the time delay of the process and the sampling interval of the discrete model. The models are

$$H_1(z) = \frac{b_1 z + b_2}{z(z-a)}$$
$$H_2(z) = \frac{b_1 z + b_2}{z^2(z-a)}$$
$$H_3(z) = \frac{b_1 z + b_2}{z^3(z-a)}$$

Guidelines for selecting one of these models are found from τ and from the model coefficients in the three cases. The model that has a realistic value of a, if one exist, and where the time delay is larger than τ, is considered correct and the corresponding continuous process model is computed. This model coincides with the true process dynamics for frequencies around the oscillation frequency.

The model is validated by comparing the phase and magnitude against those calculated from the measurements and by comparing the measured oscillation period with the theoretical oscillation period. For the model $G(s) = k_p e^{-sL}/(1+sT)$ the period is

$$T_c = -2T \ln \left| \frac{k_p d - \varepsilon}{k_p d(2e^{L/T}-1)+\varepsilon} \right|$$

As before, the results of the validation are only printed on the terminal.

A crude PI controller is designed in order to determine the true steady state gain. Since it is not certain that a process model is available, the design is based only on the oscillation frequency and amplitude. The Ziegler-Nichols rules are currently used to determine the controller parameters. A better solution would be to use a modified version with more conservative behavior. A small change in the reference value is performed and the steady state gain is computed.

Relay tuning with an integrator

A second relay experiment is performed with an integrator inserted in the feedback loop. For most physical processes, the amplitude and frequency curves of the Bode plot decreases with increasing frequency. The oscillation frequency will then typically decrease when an integrator is used. For the same reason, the oscillation amplitude will be larger.

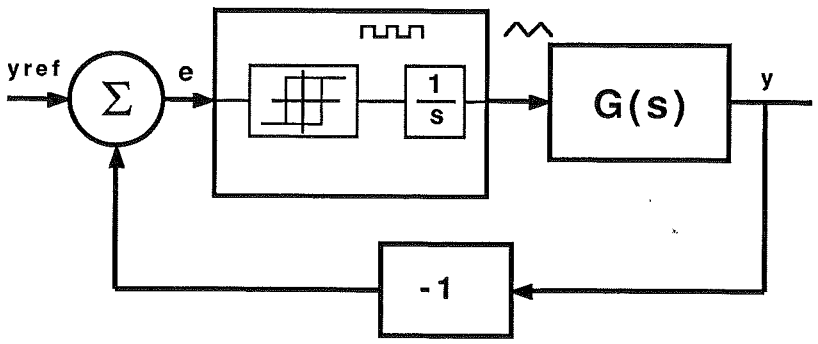

Figure 8.3 Relay experiment with an integrator after the relay.

The experiment can be performed in two different ways. The integrator can either be inserted after the relay or before the relay. An integrator after the relay according to Figure 8.3 give rise to a triangle wave being fed to the process. This signal contains less high frequencies than a square-wave signal and is thus better suited as input signal for the discrete model fitting at a lower frequency. Also, a relay with an integrator like this does not need any bias value. This means that the step where a crude controller is designed in order to find out the steady state gain may be omitted. The new relay experiment could instead be started immediately at a slightly different operating point and the steady state gain could be computed through comparing the mean values of process input and output during the two experiments.

The disadvantages mainly concern the robustness of the experiment. Since the process gain is usually much higher at the $-90°$ point than at the $-180°$ point, the relay amplitude must be much smaller than before to maintain reasonable signal amplitudes during the experiment. This implies that the relay hysteresis must also be decreased to obtain an oscillation point close to $-90°$. This is, however, not possible due to the measurement noise. Putting the integrator after the relay also causes problem for the discrete model fitting. The equations of Appendix A can no longer be used.

Inserting the integrator before the relay, as shown in Figure 8.4, gives a more robust experiment. The signal fed to the relay is internal to the computer and may thus take values that would be far too high in the real process. This means that the old values of the relay amplitude and hysteresis can be used. The integrator also has a low-pass filtering effect on the measurement noise. In the discrete model fitting the same equations can be used as before. Consequently, this way of inserting the integrator is used in the experiment.

It is important to choose a correct initial value for the integrator. The error signal, $y_{ref} - y$, is close to zero when the experiment is started in steady state. During steady state oscillations, the integrated error reaches its peak value when the true error signal is zero. Because of this, the integrator should

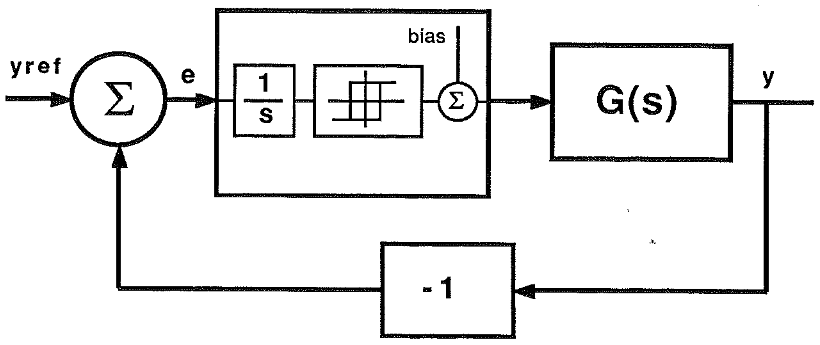

Figure 8.4 Relay experiment with an integrator before the relay

be initialized to the peak value of the integrated error to get fast convergence to periodic oscillations. This value is not known in advance. In the experiment, it is estimated based on the previously measured oscillation amplitude when no integrator was used.

Oscillation analysis II

When the relay oscillation is stable, the same measurements as before are made with the difference that the curve form measurements $e_0..e_5$ are made on the true error signal. A new point on the open loop Nyquist curve is extracted using the same technique as before. A discrete time model that approximates the true process dynamics around the oscillation frequency might also be obtained. This process model is validated in the same way as before.

A third process model that approximates the true process dynamics for frequencies close to zero is obtained from the measured steady state gain, $G(0)$, and the $-90°$ degree oscillation point. The model is computed from the following equations

$$\begin{aligned} k_p &= G(0) \\ |G(i\omega_{-90})| &= \frac{k_p}{\sqrt{1+(\omega_{-90}T)^2}} \\ \arg G(i\omega_{-90}) &= -\arctan \omega_{-90}T - \omega_{-90}L \end{aligned}$$

Knowledge of three different points on the Nyquist curve have now been extracted. At best, three different models have also been computed that each fits the true process at a certain frequency. The knowledge can be used to classify the process in different ways. For example, a dominating time delay can be indicated in several ways. One way is to look at τ in the first experiment. If $\tau > T_c/4$, Conjecture 2 of Section 3.1 indicates that the process has a dominating time delay. If $|G(i\omega_{-90})| \approx |G(i\omega_{-180})|$ and $\omega_{-180} \approx 2\omega_{-90}$, the same thing is true.

The coefficients of the three different process model need not be the same. On the contrary, they are often not even close to each other. If they are close, it is probable that the true process dynamics consists of a first order system with a time delay.

The choice of final model

Based on the information available, the system selects the final process model which will be used for the controller design. The system currently has two choices, either a first order system with time delay, $G(s) = k_p e^{-sL}/(1 + sT)$ or a second order system, $G(s) = k_p/(s^2 + a_1 s + a_2)$. The first order system with delay is intended for processes with a dominating time delay and for processes with high order dynamics, e.g. $G(s) = 1/(1 + sT)^n$ for large values of n. The second order model is intended for processes with second order dynamics.

The rules for selecting between the final models are as follows.

1. If $\tau > 0.8 * T_c/4$ then a first order model with delay is chosen. This rule catches processes with a dominating time delay and processes with high order dynamics.

2. If the magnitudes of the frequency curve at the $-90°$ point and the $-180°$ point are close and the frequency at $-180°$ is about twice the frequency at $-90°$ point then a first order model with delay is chosen. This rule catches processes with a dominating time delay.

3. If the frequency at $-180°$ is about twice the frequency at $-90°$ point then a first order model with delay is chosen. This rule catches processes with high order dynamics.

4. If the parameter values of the three available first order models with delay are close then a first order model with delay is chosen. This rule catches processes with first order dynamics and time delay.

5. Otherwise, a second order model is chosen.

The calculation of the final model

Depending on what type of final model that has been selected, different methods are used to compute the parameters in the model.

If the final model is a first order model with delay and this choice has been based on Rule 4 above, the parameters are calculated as the mean values of the parameters of the models. If the final model is of the same type but the choice has been based on some other rule, the calculation of the model parameters is not as straightforward. In this case, the models are ambiguous and it is not clear which model that can be trusted. Due to this, the calculation of the final parameters is not based on the models. Instead, a first order model with delay is fitted to the three measured frequency points.

One method is to approximate the three known points of the Nyquist curve to a first order model with delay using least squares fitting, (Lilja, 1987). The calculations involved for this are, however, substantial. Instead, a model is fitted to the $-180°$ point and the steady state gain. The final parameters are computed as the mean values of the parameters in this model and the parameters in the model previously obtained by fitting to the $-90°$ point and the steady state gain.

If the final model is a second order system, the parameters are obtained by fitting the model to the $-180°$ point and the steady state gain. The reason why the $-90°$ point is not used is that for second order systems, a PID controller will be selected as the final controller. The regulator design is sensitive to the closed loop behavior at the cross-over frequency. For PI control, the cross-over occurs when the open-loop dynamics has a phase of $-60° - -90°$. For PID control, it occurs closer to $-180°$ degree. Therefore it is in this case important to have a good model around the $-180°$ point.

The final control design

Different control design methods are used depending on the chosen model. A PID design is performed using the oscillation measurements of the first relay experiment and the modified Ziegler-Nichols methods of Appendix A. This is basically the design method used in the standard relay auto-tuner (Åström and Hägglund, 1984b). This controller is calculated only for comparison and is not selected as the final controller.

If the final model is a first order system with a time delay and $L > 1.5 * T$, a discrete pole-placement design is performed. The sampling interval is chosen as L and the two discrete poles are placed at 0.2. A digital low-pass filter is used on the process output y. This discrete design has its limitations. For example, the sampling interval is quite long. This means that disturbances will not be detected immediately.

If the final model is of second order, a PID controller is designed to give the closed loop characteristic equation

$$(s^2 + 2\zeta\omega_0 s + \omega_0^2)(s + \omega_0) = 0$$

The value of ζ is by default taken as 0.707. The value of ω_0 is implicitly determined from the choice of K in the PID controller. This choice is based on the steady state gain of the PID controller according to

$$K(1 + N) = \frac{\Delta u}{\Delta n}$$

The value of N is always chosen as 5.

If the final model is of first order with a time delay such that $L \leq 1.5 * T$, a dominant pole design is performed. The poles can be placed using either PI, PD, or PID control.

In the PI case, the controller parameters are chosen so that the closed loop poles are dominant. This means that the controller parameters k and k_i in the PI controller, $G_R(s) = (k + k_i/s)$, are chosen so that

$$1 + (k + \frac{k_i}{-\zeta\omega_0 \pm i\omega_0\sqrt{1-\zeta^2}})G(-\zeta\omega_0 \pm i\omega_0\sqrt{1-\zeta^2}) = 0$$

where ζ is specified and ω_0 is a design parameter. Solving this equation for different values of ω_0 gives a set of controller parameters. Similar equations are obtained for PD control.

In the PID case the poles are chosen as $s = -\zeta\omega_0 \pm i\omega_0\sqrt{1-\zeta^2}$ and $s = -\omega_0$. This gives the equations

$$1 + (k + \frac{k_i}{s} + k_d s)G(s) = 0 \quad , \quad s = -\zeta\omega_0 \pm i\omega_0\sqrt{1-\zeta^2}$$

and

$$1 + (k + \frac{k_i}{-\omega_0} - k_d\omega_0)G(-\omega_0) = 0$$

Scanning the design parameter ω_0 for the three different controllers gives three sets of controller parameters. For each set, the best controller is selected. For the PI and PID controllers, the choice is the controller parameters where the integral gain k_i takes it maximum value under the additional constraints that all of the controller parameters in that case should be positive and that not ω_0 is so large that additional closed loop poles on the real axis move in and destroy the selected pole dominance. This is checked by investigating the solutions to the equation

$$1 + G_R(-p)G(-p) = 0$$

with $G_R(s)$ as either the PI or the PID transfer function (Åström and Hägglund, 1988).

For the PD case, the best choice is the one where the proportional gain k is maximized under the same constraints as before.

Which controller type that should be selected is determined differently depending on what is most important: fast set point response or good load disturbance rejection. It the set point response is most important, the controller with the largest value of ω_0 should be selected. If this happens to be the PD controller, it must be ensured that the steady state error is acceptable. If so, the PD controller is selected as the final controller. If not, the values of ω_0 for the best PID and the best PI controller are compared. The PID controller is selected if $0.7 * \omega_{0PID} > \omega_{0PI}$. Otherwise, the PI controller is selected. The reason for the factor 0.7 is a desire to not choose the more complex PID controller if the return in performance does not motivate it.

When good load disturbance rejection is most important, the PD controller is first investigated. If the steady state error is acceptable, the PD controller is selected. If not, the values of k_i for the best PID controller and the best PD controller are compared. The PID controller is selected if $0.7 * k_{i_{PID}} > k_{i_{PI}}$. Otherwise the PI controller is selected. The reason for the factor 0.7 is the same as before.

At this point a controller has been selected and started. The process has been approximated with either a first order system, a first order system with a time delay, or a second order system. Depending on the model, an appropriate design method has been used.

Limitations of the tuning procedure

The implemented tuning procedure has limitations. It is assumed that the process has a finite, positive steady state gain. The tuning procedure is sensitive to load disturbances. Disturbances that occur during the relay experiment might be detected by monitoring sudden changes in the relay frequency. In that case the tuning procedure could be interrupted or the relay bias adjusted. This is currently not done. Since the tuning procedure is based on relay experiments, it cannot handle all types of processes. One example is processes with poorly damped oscillatory modes.

8.2 The numerical algorithms

The numerical algorithms needed for the described tuning procedure are the following.

The PID algorithm

The PID control law used is the following due to Åström (1987),

$$u(t) = K(\beta y_{ref}(t) - y(t) + \frac{1}{T_i p}e(t) - \frac{pT_d}{1 + pT_d/N}y(t))$$

where $p = d/dt$ is the differential operator. The parameter β gives a possibility to position the zero introduced by the regulator and to thus reduce the overshoot to step changes in the reference signal. The PID controller introduces a closed loop zero at $-1/\beta T_i$. By choosing β as $\beta = 1/T_i\omega$ where ω is the real, closed loop pole closest to origin, the overshoot will be reduced. The control law with anti-reset windup is discretely implemented as

```
e := yref - y;
z := ad * z - bd * (y - yold);
v := k * (beta * yref - y) + ipart + z;
u := v;
if u < ulow then u := ulow;
```

```
    if u > uhigh then u := uhigh;
    analog_out(u);
    if ti > 0 then ipart := ipart + aw * (u - v) + ai*e;
    yold := y;
```

With ai, ad, bd, and aw defined as

```
    ai := k * h / ti;
    ad := (2 * td - n * h) / (2 * td + n * h);
    bd := 2 * k * n * td / (2 * td + n * h);
    aw := h / t0;
```

with appropriate safeguards when ti and td are small. The parameter h is the sampling interval. The parameter t0 can be interpreted as a time constant that determines how quickly the integral is reset when u has saturated. The parameters that may be set are: h, k, ti, td, n, beta, t0 and u0. The parameter u0 is the correct steady state control and is used to set the integral part of the controller, i.e:

```
     ipart := u0 + K*(1 - beta)*yref.
```

The message (Pid Stop Ustat *value*) with the steady state control signal is sent when the algorithm is stopped.

The relay algorithm

The relay algorithm implements a relay with hysteresis where an integrator can be inserted before the relay. The relay algorithm contains the logic needed for adjusting the hysteresis and the amplitude and for determining when the relay oscillations are in steady state. The relay typically needs somewhere between 4 and 7 half periods to reach steady state oscillations depending on whether the relay parameters need to be adjusted or not. When the relay is initialized, the amplitude increases linearly as a precaution against processes with very high gain. In the OPS4 prototype system, the adjustment logic was implemented as rules in the knowledge based system. As more experience was gained and the logic became more and more stable, it has been natural to move it to the algorithm. The integrator could also be implemented as a separate algorithm. The interaction between the relay and the integrator is however so substantial that it is easier to implement it together with the relay.

The parameters which may be set are the relay amplitude d, the hysteresis ε, the desired oscillation amplitude, the bias value, and a parameter that determines whether the integrator should be used or not and sets the initial value of the integrator.

The relay algorithm sends the following messages to the knowledge based system.

`(relay d changed` *value* `eps` *value*`)` The relay amplitude and hysteresis have been changed.

`(relay d decreased` *value*`)` The relay amplitude has been decreased.

`(relay ready half-periods` *value*`)` The relay oscillation has become stable after *value* half periods.

`(relay load disturbance)` A load disturbance has been detected during the experiment. This is the case if a relay switching still has not taken place twice the time after it was expected, i.e., twice the previous half period time.

The oscillation analyzer

The oscillation analyzer measures the steady state relay oscillations. The measurements take four half periods and they start when the relay switches from negative to positive. During the first two half periods, the peak amplitudes, oscillation half periods, and τ values are measured. Six equidistant samples of the error signal are measured during the two last half periods. The error signal is low-pass filtered with a first order filter with a time constant of 1/10'th of an oscillation half period. The algorithm has no parameters.

When the measurements are ready, two messages are sent to the knowledge based system:

`(osc-analyzer periods` *val val* `amplitudes` *val val* `tau` *val val*`)`

`(osc-analyzer sample-values` *e0 e1 e2 e3 e4 e5*`)`

The algorithm for computing the statistics of y and e

This algorithm gathers statistical information about the process output and the error signal. It contains two arrays where the actual y and e values are stored each time the algorithm is executed. When the arrays have been filled with values, the mean values, variances, and maximum and minimum values are reported to the knowledge based system.

The parameters which can be set are sampling time of the algorithm and the length of the arrays. The product of these parameters determine the rate at which the statistics are reported.

The messages which are sent back are

`(y-statistics meany` *val* `vary` *val* `maxy` *val* `miny` *val*`)`

and

`(y-statistics meane` *val* `vare` *val* `maxe` *val* `mine` *val*`)`

The algorithm can respond to the following questions:

`(y-value` *time*`)` returns the answer message `(yval` *val*`)` where *val* is the value of y, as specified by *time*. If *time* is zero, the last measured value is returned.

`(y-stat` *time*`)` returns statistics information about y measured over the time specified. If *time* is negative, the statistics are measured over the time period beginning when a statistics message last was sent.

Similar questions are available for the error signal.

The algorithm for computing the statistics of u

The statistics algorithm for u is identical to the previously described algorithm but instead operates on the control signal u. The reason for the separation is that the control signal typically changes much slower than the process output and error.

The level crossing detector

The level crossing detection algorithm detects level crossings for the signals y, e or u. When a signal crosses a specified level in a specified direction, the message `(level-crossing` *signal level* `time` *time*`)` is sent to the knowledge based system. The message contains the time that the signal has taken to reach the level.

The parameters that can be set are which signal that is concerned, the level, and information about up or down crossings. Several signals and levels can be monitored simultaneosly.

When a signal has crossed a level, that signal is no longer monitored around that level. A useful extension to the algorithm would be to send return messages each time a signal crosses a level. The algorithm should have hysteresis and it should return the time since the last crossing.

The digital filter

The second order filter

$$G(s) = \frac{k(b_0 s^2 + b_1 2\zeta\omega_0 s + b_2\omega_0^2)}{s^2 + 2\zeta\omega_0 s + \omega_0^2}$$

is implemented using Tustin's approximation. The b_i parameters are used to give different filter characteristics. The values $b_0 = b_1 = 0, \quad b_2 = 1$ gives a low-pass filter, $b_0 = b_2 = 0, \quad b_1 = 1$ gives a band-pass filter, and $b_1 = b_2 = 0, \quad b_0 = 1$ gives a high-pass filter. A notch filter can also be obtained. The parameters that can be set are the sampling interval, $\omega_0, \zeta, k, b_0, b_1$ and b_2.

The filter is currently used only as a low-pass filter on the process output y. In a more elaborate system, the discrete filter would be a standard component with many usages.

The linear discrete controller

The algorithm implements the RST-control law

$$R(q)u(k) = T(q)y_{ref}(k) - S(q)y(k)$$

with the order of the R, S and T polynomials limited to 2. The control law is implemented with anti windup compensation using an observer polynomial according to Åström and Wittenmark (1984, p. 372).

$$\begin{aligned} A_o v &= T y_{ref} - S y + (A_o - R)u \\ u &= \text{sat} \quad v \end{aligned}$$

The parameters are the sampling interval, the coefficients in the R, S, T and A_o polynomials, and the initial steady state value of the controller.

The execution order

The algorithms are executed in the following predetermined order:

```
while true do
begin
  Set_sampling_eventflag(main_sampling_time);
  y := AnalogIn(inchannel);
  if active[y_stat_algorithm] then Y_statistics(execute);
  if active[dig_filter_algorithm] then Dig_filter(execute);
  if active[pid_algorithm] then Pid(execute);
  if active[rst_algorithm] then Rst(execute);
  if active[relay_algorithm] then Relay(execute);
  if active[osc_analyzer_algorithm] then Osc_analyzer(execute);
  if active[u_stat_algorithm] then U_statistics(execute);
  if active[level_algorithm] then Level_cross(execute);
  if message_in_Outbox then read_message;
  wait_for_sampling_eventflag;
end;
```

This ordering ensures that the statistics is measured on the non-filtered process output. The oscillation analyzer accesses a few variables that belongs to the relay algorithm, e.g., the current value and the last value of the relay output. These values are needed by the oscillation analyzer to detect relay switchings.

The algorithm handler

A separate module, the algorithm handler, provides a common interface between the knowledge sources and the algorithms. Its main task is to convert the messages sent from the knowledge sources to the algorithms into the correct syntactic form and to keep an up-to-date record of the status of all the numerical algorithms.

The algorithm handler is implemented as a YAPS system. The algorithm information is stored in the following frames.

```
(defframe algorithm
  "Information about one algorithm"
 (name
  (shortname "The name used in the communication")
  (description "Brief  description of the algorithm")
  (state "Inactive or active depending
          if the algorithm is executing."
         :default inactive)
  (parameters "List of parameter-description frames
               describing each parameter.")
  (questions "List of question-description frames
              describing each question that the
              algorithm can answer")
  (return-messages "List of return-message-description
                    frames describing the messages that may
                    come from the algorithm"))
  ())


(defframe parameter-description
  "Information about an algorithm parameter"
 (name
  shortname
  algorithm
  description
  (actual-value "Current value")
  default-value
  (argument "Used or not-used depending on if the
             argument is a dummy variable or not"
             :default used))
  ())


(defframe question-description
  "Information about a question to an algorithm"
 (name
  shortname
  algorithm
  description
  (argument :default used)
  (answer "Description of the returned answer"))
 ())


(defframe return-message-description
  "Information about a return message"
 (algorithm
  (description "Description of the syntax and semantics
               of the returned message"))
 ())
```

Global parameters such as the main sampling period, the set point and the control signal during manual control, are stored as **`parameter-description frames`** where the algorithm attribute has the value **`global`**.

When a message is sent to the algorithms, the algorithm handler converts it to the correct syntax, e.g., substitutes shortnames against parameter names, and changes the attribute values of the different frames.

The knowledge sources may access the information about the algorithm and, e.g., acquire parameter values. This is implemented through the emulation of object-oriented programming described in Section 7.3. For example, when a knowledge source adds the fact **`(parameter-value`** *algorithm parameter*) to the algorithm-handler, a rule is triggered which returns the actual value of this parameter to the knowledge source. This rule looks as:

```
(p parameter-value

  (parameter-value -algorithm -parameter)
  (frame parameter-description
    name -parameter
    algorithm -algorithm
    actual-value -value)
-->
  (remove 1)
  -value)
```

The algorithm handler also contains rules that print out different descriptions of the algorithm status.

8.3 The knowledge sources

The following knowledge sources are used in the first example:

Operator-inquirer: Asks the operator a few questions about the process characteristics and the control specifications.

Manual-control-supervisor: Supervises the manual control phase. Checks that the process is in steady state and measures the noise range.

Relay-supervisor: Initializes and supervises the relay experiments. Performs preliminary analysis of the relay measurements.

Modeller: Contains knowledge about continuous model building and validation. This involves, e.g., fitting models to frequency measurements, converting between discrete time and continuous time models, choice of final model, etc.

Designer: Contains knowledge about control design, e.g., characterization of the control problem, selection of controller structure, computing controller parameters, etc.

Control-supervisor: Takes care of manual parameter changes commands to the different controllers.

Explainer: Generates explanations for questions about the process and the controller.

Y-statistics: Gathers statistics on the process output and error.

U-statistics: Gathers statistics on the control signal.

All of these knowledge sources are of the forward chaining type except the Operator-inquirer which uses backward chaining. The combination of the knowledge sources is performed by a procedural knowledge source, the Initial-relay-tuner. It contains the following S-Lisp procedure that controls the sequencing of the knowledge sources.

```
(s-defun initial-relay-tuning ()
  (s-let (control-loop 'process)
    (write-welcoming-information)
    (activate Operator-inquirer)
    (fact-in scheduler
      verify ^control-loop time-constant-estimation -x)
    (fact-in scheduler
      verify ^control-loop allowed-steady-state-error -x)
    (fact-in scheduler
      verify ^control-loop control-objective -x)
    (deactivate Operator-inquirer)
    (waitentry
      inactivated Operator-inquirer)
    (set-up-specifications-frame)
    (activate Y-statistics)
    (activate Explainer)
    (start-and-wait-for 'Manual-control-supervisor)
    (fact-in Relay-supervisor
      tune at 180)
    (start-and-wait-for 'Relay-supervisor)
    (start-and-wait-for 'Modeller)
    (s-cond
      ((and (fact-match first order system)
            (fact-match complete model))
       (activate U-statistics))
      ((fact-match first-order-system)
       (fact-in Designer
         design and start crude PI for first order system)
       (activate Designer)
       (activate U-statistics)
       (waitentry
         inactivated Designer)
       (fact-in Modeller
         check steady state gain)
       (start-and-wait-for 'Modeller))
      (t
       (fact-in Designer
         design and start crude PI for higher order system)
       (activate Designer)
       (activate U-statistics)
       (waitentry
         inactivated Designer)
       (fact-in Modeller
         check steady state gain)
       (activate Modeller)
       (fact-in Relay-supervisor
         tune at 90)
       (start-and-wait-for 'Relay-supervisor)
       (start-and-wait-for 'Modeller)
       (fact-in Designer
         design and start-up good controller)))
    (start-and-wait-for 'Designer)
    (activate Control-supervisor)))


(s-defun start-and-wait-for (knowledge-source)
  (activate (s-eval knowledge-source))
  (waitentry inactivated ^knowledge-source))
```

The procedure starts by writing some initial information. It then activates the Operator-inquirer and poses three questions. Based on the answer of the questions, the closed loop specifications are set up. When the process is in steady state, i.e., the Manual-control-supervisor is finished, the first relay experiment is started. Three different situations may arise when the Modeller has analyzed the measurements. Firstly, a complete first order model has been obtained. Secondly, an incomplete first order model has been obtained. In that case, a crude controller is designed and a small set point change is made in order to measure the steady state gain. In the third case, the relay measurements indicate a higher order process model and a new relay experiment with an integrator is initiated.

The sequence is finished when an acceptable controller has been designed and with the knowledge sources Control-supervisor, Y-statistics, U-statistics, and Explainer being active.

Frame definitions

To obtain a clean structure of the rules, it is important to group associated information into frames.

The information about an entire control loop is gathered in the control-loop frame.

```
(defframe control-loop
  "Contains information about a control loop"
 (name
  (a-priori-information "A-priori-info. frame containing
                         the information obtained by the
                         Operator-inquirer")
  (specifications "Specifications frame")
  (noise-range "Measurement noise")
  (tuning-at-180 "Relay-experiment frame containing info.
                  about the tuning without an integrator")
  (tuning-at-90 "Relay-experiment frame containing info.
                 about the tuning with an integrator")
  (nyquist-points "List of frequency-point frames each
                   describing one point on the open-loop
                   Nyquist curve")
  (discretely-fitted-models "List of first-order-with-delay
                             frames containing the model
                             achieved through  discrete
                             model fitting")
  (final-model "Frame describing the finally chosen model")
  (controller-alternatives "List of frames describing the
                             different controller alternatives
                             computed.")
  (final-controller "Frame describing the finally chosen
                     controller"))
 ())
```

In the general case, this frame would also contain configuration information, e.g., where in the process the control loop is. The **a-priori-information frame** contains the information that is acquired by the Operator-inquirer.

```
(defframe a-priori-information
 "A-priori information from the Operator-inquirer"
 ((time-constant-estimation "Crude estimate of the dominating
                             time constant " :askable)
  (allowed-steady-state-error "Allowed steady state error in
                               percent" :askable)
  (control-objective "Allowed values are load-disturbance
                      or set-point depending on what is
                      most important"
                      :askable))
 ())
```

This frame would probably be extended significantly in a more general application. The closed loop specifications looks as follows.

```
(defframe specifications
  "Closed loop control specifications"
 ((relative-damping :default 0.707)
  (u-range "Allowed variations in the steady state
           control signal due to measurement noise"
           :default 0.1)
  (amplitude-margin :default 1.5)
  (phase-margin :default 45)
  (steady-state-error :default 0.05))
 ())
```

Information regarding a relay experiment are grouped together in the `relay-experiment frame`.

```
(defframe relay-experiment
 "Information about a relay experiment"
 ((yref "Actual set point")
  (u0 "Steady state control signal")
  (point "180 or 90 depending on if an integrator
         is used or not")
  (d "Relay amplitude")
  (hysteresis "Relay hysteresis")
  (h "Oscillation period")
  (amp "Peak-to-peak oscillation amplitude")
  (wc "Oscillation frequency")
  (kc "Ultimate gain, 1/|G(iwc)|")
  (tau "Time between relay switching and change of
       derivative sign")
  (approx-order "1 or 2 depending on the value of tau,
                1 means that the system is PI-controllable,
                  2 means that the order is >=2")
  (delay "Number of pure time delays in the discrete model")
  (samp "Sampling time for the discrete model")
  a1 b11 b21 a2 b12 b22 a3 b13 b23)
  ())
```

The a and b attributes are the value of the a, b_1, and b_2 parameters for the three discrete models.

Information about one point on the open-loop Nyquist curve is stored in a frequency-point frame.

```
(defframe frequency-point
  "Information about one frequency point"
 ((w "Frequency")
  gain
  phase-shift)
 ())
```

Information about different models are stored in the following frames.

```
(defframe continuous-model
  "Information about a continuous model"
 ((frequency "Frequency range where the model is valid")
  (status "Takes the values not-validated, validated, or
           in-complete"))
 ())

(defframe first-order
  "First order model k/(1 + sT)"
 (k
  T
  (T/k "Ratio for incomplete model"))
 (continuous-model))

(defframe first-order-with-delay
  "First order model with delay, kexp(-sL) / (1 + sT)"
 (L)
 (first-order))

(defframe second-order
  "Second order model, k / (s2 + a1 s + a2)"
 (k a1 a2)
 (continuous-model))
```

Notice the use of inheritance. Information about the different controllers are stored in the following frames.

```
(defframe controller
  "Controller information"
 ((h "Sampling interval")
  (active "Has the value t if it is used")
  (status "A symbol denoting from which
          principles the controller was
          designed"))
 ())

(defframe Pid-controller
  "PID controller"
 (k Ti Td beta N)
 (controller))

(defframe discrete-controller
  "Discrete RST-controller"
 (r1 r2 t0 t1 t2 s0 s1 s2 a1 a2)
 (controller))
```

The attributes **a1** and **a2** in the discrete-controller frame are the coefficients in the observer polynomial A_o.

The Operator-inquirer knowledge source

The operator questioning phase has been kept very simple in the examples. Due to this, the Operator-inquirer knowledge source does not contain any rules at all. For example, when the fact

```
    (verify process time-constant-estimation -x)
```

is added to the scheduler, it becomes the current goal of the Operator-inquirer knowledge source. Since it has no rules for how to derive a value for the time-constant-estimation attribute and this attribute has been declared askable, a question is posed to the operator. The operator has the possibility to ask for further explanation of the question, in which case the description string for this attribute is written.

The Manual-control-supervisor knowledge source

The Manual-control-supervisor knowledge source contains 5 forward chaining rules. During manual control the operator has the possibility to change the control signal and the reference value until the process is in steady state at the desired operating point. The operator then checks that the process is in steady state and measures the noise-range. The rule which measures the noise range looks as follows.

```
(p measure-noise-range

 "Measures the noise-range in steady state manual control.
  It is computed as the difference between the maximum value
  and the minimum value of y during a time given by the
  estimated time constant"

 (frame control-loop
   a-priori-information -frame
   noise-range nil)
 (-frame a-priori-information
   time-constant-estimation -time)
 (steady-state)
-->
 (remove 3)
 (let ((max (max-of y -time))
       (min (min-of y -time)))
   (global-modify 1 noise-range ^(difference max min)))
 (deactivate self))
```

The Relay-supervisor knowledge source

The Relay-supervisor knowledge source contains 8 forward chaining rules. They initialize the relay experiment, awaits messages from the relay algorithm, starts the oscillation analysis and makes an initial analysis of the relay measurements. The rule which starts the relay experiment without integrator looks as follows

```
(p initialize-relay-without-integrator

 "Starts a relay experiment without integrator"

 (tune at 180)
 (frame control-loop
   tuning-at-180 nil
   noise-range -noise)
-->
 (let ((desired-error
         (cond ((>= -noise 0.02) 0.06)
               ((>= -noise 0.01) (times 3 -noise))
               (t 0.03)))
       (hysteresis
         (cond ((>= -noise 0.005) (times 2 -noise))
               (t 0.01)))
        (bias (parameter-value global u)))
    (global-modify 2
      tuning-at-180 ^(global-make relay-experiment
                        yref ^(parameter-value global yref)
                        u0 ^bias
                        point 180
                        hysteresis ^hysteresis
                        d 0.2))
    (send start relay
       hysteresis ^hysteresis
       d 0.2
       desired-error ^desired-error
       bias ^bias)
    (message-request relay)))
```

In the `let` statement, the bias variable gets its value through an access to the algorithm-handler. The `send` function inserts the message into the algorithm-handler which converts it and sends it to the algorithms. The rule ends by setting up a request for incoming messages from the relay.

The Modeller knowledge source

The Modeller knowledge source contains about 15 rules for model building and model validation. The following rule checks if the initial relay experiment has resulted in a reasonable discrete model corresponding to a continuous first order model and in that case computes the continuous model and stores it in a first-order frame.

```
(p compute-first-order-model

  "Computes a first order model"

 (frame control-loop
   tuning-at-180 -frame)
 (-frame relay-experiment
   samp -h
   delay 1
   a1 -a
   b11 -b
   b21 0)
(~ (frame first-order
     frequency 180))
test
  (and (numberp -a)
       (> -a 0)
```

```
         (< -a 1)
         (> -b 0))
-->
 (let* ((time-constant
             (calc (- -h) / (log -a)))
          (k (calc -b / (1 - -a)))
          (frame (global-make first-order
                      k ^k
                      T ^time-constant
                      frequency 180
                      status not-validated)))
    (global-fact first order system)
    (global-modify 1
      final-model ^frame)))
```

The calc function allows for infix notation in mathematical expressions. The rule choose-final-model-3 chooses the final models as a first order model with delay if the oscillation frequency at the $-180°$ point is about twice the frequency of the $-90°$ point.

```
(p choose-final-model-3

"If the oscillation frequency at the -180 point is about twice
the frequency of the -90 point then choose a first order model
with delay as the final model"

(frame control-loop
  final-model nil)
(frame frequency-point
  w -w180)
(frame frequency-point
  w -w90)
(~ (final-model . -))
test
  (and (> -w180 -w90)
       (> -w90 0)
       (< (calc -w180 / -w90) 2.1))
-->
(fact final-model first-order-with-delay high-order-dynamics))
```

The Designer knowledge source

The Designer knowledge source contains about 15 rules for design of different controllers. The information needed for the accurate dominant pole design of PID controllers are stored in a separate frame.

```
(defframe pid-design-table-entry
  "Contains the controller parameters for one value
   of the natural frequency w0"
 (w0
  (k-pi "Prop. gain in PI design")
  (ki-pi "Int. gain in PI design")
  (Ti-pi "Integration time in PI design")
  (p10-pi "Normalized real pole in PI design")
  (wcross-pi "Cross-over frequency with PI design")
  (k-pid "Prop. gain in PID design")
  (ki-pid "Int. gain in PID design")
  (kd-pid "Der. gain in PID design")
  (Ti-pid "Integration time in PID design")
  (Td-pid "Derivation time in PID design")
  (p10-pid "Normalized real pole in PID design")
  (w-cross-pid "Cross-over frequency with PID design")
  (k-pd "Prop. gain in PD design")
  (kd-pd "Der. gain in PD design")
  (Td-pd "Der. time in PD design")
  (p10-pd "Normalized real pole in PD design")
  (w-cross-pd "Cross-over frequency with PD design"))
  ())
```

During the design stage, the database contains one instance of this frame for each frequency ω_0 under consideration. The instances may be viewed as rows in a table with different controller parameters for different values of ω_0. The design is performed by searching for rows where certain parameters, e.g. **ki-pi**, reaches it maximum value. This can easily be expressed through pattern matching. For example, the following patterns which will match the row where **ki-pi** has its maximum.

```
  (frame pid-design-table-entry
    ki-pid -ki-1)
(~ (frame pid-design-table-entry
     ki-pid -ki-2)
   with (> -ki-2 -ki-1))
```

The Control-supervisor knowledge source

The Control-supervisor knowledge source contains around 5 rules which allows manual set point changes and parameter changes for the different controllers.

The Explainer knowledge source

The Explainer knowledge source contains rules for answering questions about the current process model and controller. Some questions that can be answered are:

Which controller is running?

How is the controller designed?

Which process model has been derived?

Why was this model derived?

How was this model derived?

What is happening now?

The statistics knowledge sources

The statistics knowledge sources, Y-statistics and U-statistics, collect statistical information about the process output signal and error signal, and the control signal, using their associated numerical algorithms. The algorithms can be viewed as consisting of shift registers where the signal values are stored. When the shift register is full, the statistics is computed and sent to the knowledge-based system. Similarly, the statistics knowledge sources store the incoming information in shift registers. When these registers are full, statistics are again computed and these are stored in new shift registers. The result is that the statistics information is stored at different time scales.

The information about a signal is stored in the following frame.

```
(defframe statistics
  "Statistics information about a signal"
 ((name "Signal name: y e or u")
  (sampling-time "Sampling interval for the shift register
                  in the algorithm" :default 1)
  (first-length "Length of the shift register in the
                 algorithm" :default 120)
  (counter "Counter for the second shift register")
  (first-time "The time interval at which the first
               shift register is filled i.e. the rate
               at which the second shift register
               gets a new value,
               sampling-time * first-length" :default 120)
  (second-length "Length of second shift register" :default 5)
  (second-time "The time interval at which the second
                register is filled" :default 600)
  (third-length "Length of third shift register" :default 6)
  (third-time "The time interval at which the third register
                is filled" :default 3600)
  mean-second var-second min-second max-second
  mean-third var-third min-third max-third)
 ())
```

The last attributes are the actual shift registers which are stored as lists. If the above default values are used, the statistics information are stored on a 1-second, 2-minute, and 10-minute basis over the last hour.

The statistics knowledge sources are designed to run as soon as a new fact is added to them according to Section 7.3. They implement the following functions, which may be called by the other knowledge sources.

`(mean-of` *signal time*`)` Returns the mean value of the signal over the last *time* seconds. If the time is negative the mean value is taken around *time* seconds ago with the resolution available. For example, with the above defaults the function `(mean-of y 300)` returns the mean value over the last 5 minutes, whereas `(mean-of y -300)` returns the best approximation of the mean value as it was around 5 minutes ago.

`(variance-of` *signal time*`)` The same as above for the variance.

`(max-of` *signal time*`)` The same as above for the maximum value.

`(min-of` *signal time*`)` The same as above for the minimum value.

`(value-of` *signal time*`)` Returns the signal value *time* seconds ago. If the time is larger than what is stored in the first shift register, the mean value around that point is returned.

Each knowledge source contains 6 rules. Five of the rules are shared among the two knowledge sources.

8.4 A session with the system

This section contains an example session with the system as well as a performance evaluation of the finally designed controller for different types of processes. The simulation program Simnon (Elmqvist, 1975) running in real-time mode is used to represent different physical processes. During the session two terminals are used. On the alpha-numerical terminal, commands are given to the system and results are written out. The graphics terminal shows the y, $yref$ and u signals according to Figure 8.5.

During the example session, the process which is being simulated has the transfer function $G(s) = 1/(1+15s)^2$. The output messages from the system will

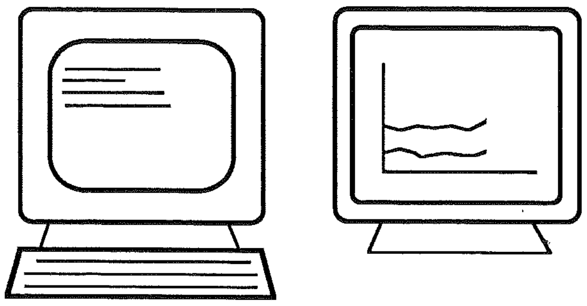

Figure 8.5 Terminal set-up

be typeset in `typewriter type` and the input will be typeset in *slanted type*. The signals y, u and $yref$ from the experiment are shown in Figure 8.6. To better visualize the interaction with the system, the Simnon times when commands are given are written in the example. These times correspond to Figure 8.6. The example session will be run in a verbose mode with large amounts of information being printed on the terminal.

The system starts by asking the user a few questions about the process.

```
      ** WELCOME **

   I will start by asking different questions about the process.
   The answer to the questions are entered with  .
   IO>>answer a-<question-number> <ans> The values that <ans>
   can take are:

   y, yes             --- confirmation
   n, no              --- negation
   u                  --- unknown
   ?                  --- explains the question
   (why <nr> .. )     --- gives WHY? explanations
                          according to Section 7.6 of the
                          thesis


   Question 0: What is the control-objective of process?

   IO>>answer a-0 ?
   t
   IO>>

   Allowed values are load-disturbance or set-point depending
   on what is most important
```

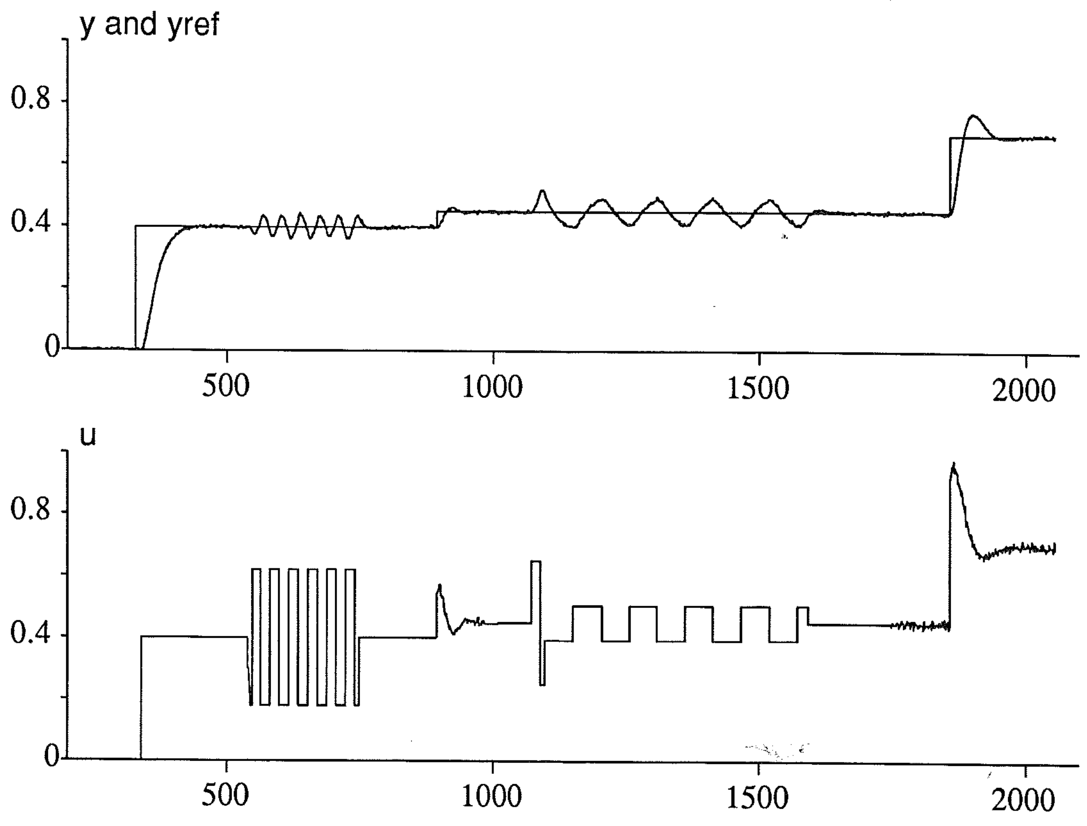

Figure 8.6 The signals $y, yref$ and u during the example session

```
Question 0: What is the control-objective of process?
```

answer a-0 set-point

```
t
IO>>

Question 1: What is the allowed-steady-state-error
of process?
```

answer a-1 ?

```
t
IO>>

Allowed steady state error in percent

Question 1: What is the allowed-steady-state-error
of process?
```

answer a-1 10

```
t
IO>>

Question 2: What is the time-constant-estimation of
process?
```

```
answer a-2 10
t
IO>>
```

When the question phase is finished, the operator is instructed to manually control the process.

```
Control the system manually until the controlled value is in
steady state at your desired operating point.
The control signal is changed
with the command u *value* and the reference value with yref
*value*. When you are finished type ready.            (** 300 s.**)


u 0.4                                                 (** 310 s.**)
t
IO>>yref 0.4                                          (** 320 s.**)
t
IO>>ready                                             (** 470 s.**)
t
```

Now, the first relay experiment is performed.

```
IO>>what is happening?                                (** 700 s.**)

A relay experiment is being performed to determine the
process dynamics around the -180 degree phase
point.

t
IO>>

The process is classified as being                    (** 800 s.**)
of second order or higher.

The  first order model with delay computed
from the discrete time model fitted around
the -180 point is
kp = 0.317
T = 13.63
```

```
L = 8.451

The oscillation period from the model = 35.7
The measured oscillation period = 34.5

The phase from the model = -2.73
The estimated phase = -2.53

The magnitude from the model = 0.119
The estimated magnitude = 0.123
```

A crude controller is designed and a small set-point change is made in order to determine the steady state gain.

```
The measured steady-state gain = 1.056
```
(** 1000 s.**)

Since the process has been classified as a high order system, a second relay experiment is performed.

```
The first order model with delay computed
from the discrete time model fitted around
the -90 point is
kp = 4.37
T = 120.2
L = 6.52
```
(** 1700 s.**)

```
The phase from the model = -1.84
The estimated phase = -1.49

The magnitude from the model = 0.58
The estimated magnitude = 0.54


The first order model with delay fitted
to the steady state gain and the -90 point is
kp = 1.06
T = 27.3
L = 7.3

Cross-validation of the -90 model at
the -180 point

The phase from the model = -2.71
The estimated phase = -2.53
The magnitude from the model = 0.20
The estimated magnitude = 0.123


Cross-validation of the -180 model at
the -90 point

The phase from the model = -1.23
The estimated phase = -1.49

The magnitude from the model = 0.24
The estimated magnitude = 0.54
```

After the second relay experiment, a process model has been derived and a controller has been designed. The user has now the possibility to ask questions about the process model and controller.

```
IO>>which process model has been derived?
t

The derived process model is
G(s) = k / (s^2 + a1 s + a2)
with
k = 0.00462
a1 = 0.119
a2 = 0.00437

IO>>why was this model selected?
t

A second order model is chosen  as default when no
first order model with or without delay is appropriate


IO>>how was the model computed?
t

The model was derived by fitting to the steady state gain and
the -180 degree point.

IO>>why was the model computed like this?
t

A good process model is required at the closed loop
cross-over frequency. A second order model will be controlled
```

```
by a PID controller. For PID control, the cross-over occurs
close to -180 degrees.

IO>>which controller is running?
t

The controller is a PID controller with the parameters
Sampl. interval =  1
K = 1.14
Ti = 20.87
Td = 6.38
n = 5
beta = 1.52

IO>>how is the controller designed?
t

K is determined from the allowable steady state control
signal range and the measured noise range as
Delta-u / (Delta-n * (1 + n)) where n has the default
value 5. The controller parameters are chosen to give
the closed loop characteristic equation

(s^2 + 2 zeta w0 s + w0^2)(s + w0) = 0

with
zeta = 0.707 and w0 = 0.063

IO>>yref 0.7                                         (** 1840 s.**)
t
```

Several switches can be set in order to get more information about the internal behavior of the system. The `**print-messages**` switch will cause messages between the numerical algorithms and the knowledge-based system to be printed out. The `**print-k-s-state**` switch will cause messages to be printed when state of the knowledge sources is changed.

```
IO>>@(setq **print-messages** t)
t
** Mess. in**
  (y-statistics ymean 0.701 yvar 0.00013 ymax 0.712 ymin 0.695)
** Mess. in**
  (y-statistics emean 0.001 evar 0.00013 emax 0.05 emin -0.012)
** Mess. in**
  (u-statistics umean 0.699 uvar 0.00054 umax 0.73 umin 0.66)
** Mess. in**
  (y-statistics ymean 0.697 yvar 0.00022 ymax 0.709 ymin 0.697)
```

```
** Mess. in**
  (y-statistics emean 0.003 evar 0.00022 emax 0.03 emin -0.09)
** Mess. in**
  (u-statistics umean 0.701 uvar 0.00077 umax 0.77 umin 0.67)
IO>>@(setq **print-k-s-state** t)
t
IO>>yref 0.6
Knowledge source Control-supervisor resumed
** Mess. out**
  (glob yr 0.6)
Knowledge source Control-supervisor suspended
t
IO>>@(setq **print-messages** nil)
nil
IO>>@(setq **print-k-s-state** nil)
nil
```

The @ character preceding a Lisp-expression written on the terminal will cause the expression to be remotely evaluated in the knowledge-based system process and the resulting value to be sent back. Tracing can be turned on to various degree. For example, the function **(yaps-trace 'short)** will cause the name of all YAPS rules to be printed as they are executed.

```
IO>>@(yaps-trace 'short)
t
Running rule: main-restart

IO>>yref 0.4
t
Running rule: Schedule-16
Running-rule: Schedule-1
Knowledge source Control-supervisor resumed
Running-rule: yref-change
Running-rule: algo10
Knowledge source Control-supervisor suspended
Running rule: main-restart

IO>>@(yaps-untrace 'short)
t
```

The possibility to evaluate arbitrary Lisp expressions in the knowledge-based system process simplifies development and debugging. Functions which examine or alter the internal state of the knowledge-based system process can be evaluated on-line. For example, the command **(global-db)** prints out the contents of the global database.

```
IO>>@(global-db)

Cycle nr   Fact
    .         .
    .         .
    .         .
   56.       (second-order-frame-0 k 0.00462 a1 0.009
              a2 0.00437 frequency all status final)
    .         .
    .         .
    .         .
```

"How" explanations can be generated on arbitrary database entries if the switch ****how-explanations**** was set when the database element was added. For example, the frame shown above that represents the final second order model can be explained by

```
IO>>@(expl 56)
t
```

This generates a lengthy explanation according to Section 7.3. Various debugging commands have been implemented. The current status of the numerical algorithms can be shown with the command **describe-algorithm-status**. Similar commands exist that show the status of the different knowledge sources. Rules can be added, edited and deleted on-line. In the following small example a rule that prints out the variance of the control error every two minutes, is added to the Control-supervisor knowledge source.

```
IO>>@(add-new-rule Control-supervisor
'(periodic-variance-check
(wakeup periodic-variance-check)
(rule periodic-variance-check -period)
—>
(remove 1)
(patom "The variance of e is ")
(patom (variance-of e 120)) (terpr)
(timer-request -period periodic-variance-check)))
t
IO>>@(fact-in Control-supervisor
rule periodic-variance-check "00:02:00")
t
```

```
IO>>@(fact-in Control-supervisor
wakeup periodic-variance-check)
t
IO>>
The variance of e is 0.00021
The variance of e is 0.00031                        (** 2 min. later **)
      .
      .
      .
```

Control performance evaluation

The performance of the derived controller will be examined through some examples. Measurement noise is added to the process output in the examples. The control objective is fast set-point response.

EXAMPLE 8.1 — First order system
In this example, the simulated process is $G(s) = 1/(1+50s)$. Based on the curve form of the first relay experiment, the process is classified as a first order system and the estimated model is computed as $G(s) = 0.89/(1+66.3s)$. The resulting controller is a PI controller with $K = 5.822$, $T_i = 17.97$ and $\beta = 0.84$. The step response of the closed loop system is shown in Figure 8.7.

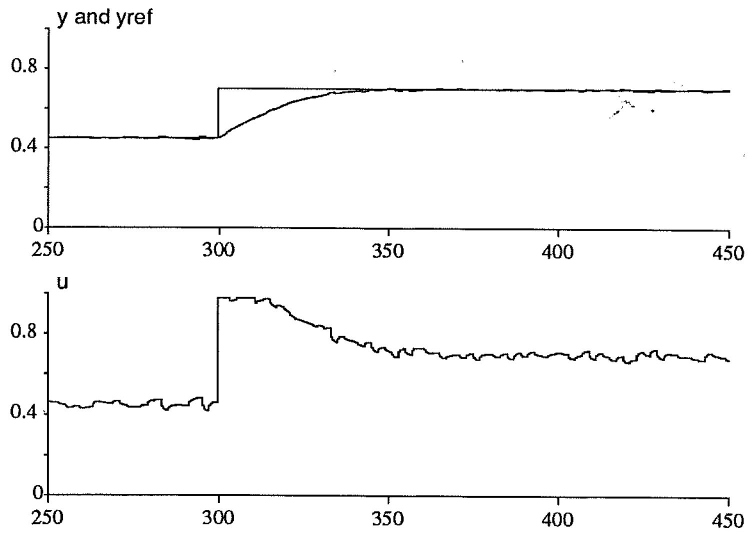

Figure 8.7 Step response for Example 8.1.

Since the process is classified as being of first order, an explicit estimation of a point on the Nyquist curve is not performed. □

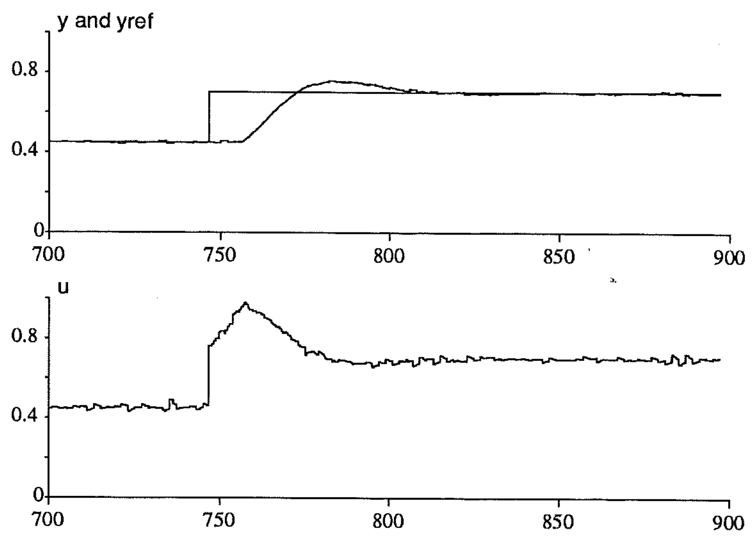

Figure 8.8 Step response for Example 8.2.

EXAMPLE 8.2 — First order system with short delay
In this example the simulated process is $G(s) = e^{-10s}/(1+20s)$. Based on the first relay experiment, the process is classified as having dynamics of second order or higher. The three estimated points on the Nyquist curve are:

	0° point	−90° point	−180° point
Measured ω	0	0.058	0.138
Estimated gain	1.005 (1)	0.659 (0.650)	0.429 (0.341)
Estimated phase	0 (0)	−1.286 (−1.447)	−3.020 (−2.603)

The true values are shown in parentheses. The three first order models with delay are:

	−180° model	−90° model	Steady state gain model
k_p	0.700	1.353	1.005
T	13.62	30.03	19.71
L	13.04	12.52	7.36

The finally computed model is $G(s) = 1.005e^{-10.53s}/(1+17.52s)$. The choice of a first order model with delay is based on the curve form of the first relay experiment. The discretely fitted models varies too much to be reliable. Instead, the model parameters are computed based on the estimated frequency points as described in Section 8.1. The resulting controller is a PID controller with $k = 1.44$, $T_i = 16.75$, $T_d = 2.45$, $\beta = 0.85$ and $N = 5$. The step response of the closed loop system is shown in Figure 8.8. □

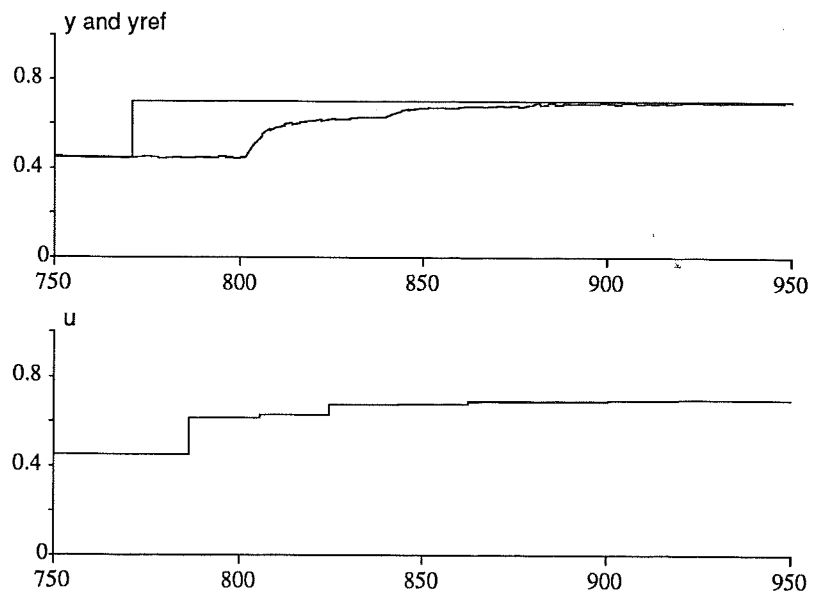

Figure 8.9 Step response for Example 8.3.

EXAMPLE 8.3 — First order system with long delay
In this example the simulated process is $G(s) = e^{-15s}/(1+5s)$. Based on the first relay experiment, the process is classified as having dynamics of second order or higher. The three estimated points on the Nyquist curve are:

	$0°$ point	$-90°$ point	$-180°$ point
Measured ω	0	0.0714	0.136
Estimated gain	0.9995 (1)	1.015 (0.942)	0.833 (0.826)
Estimated phase	0 (0)	−1.392 (−1.413)	−3.210 (−2.646)

The three first order models with delay are:

	$-180°$ model	$-90°$ model	Steady state gain model
k_p	1.102	1.029	0.9995
T	7.33	9.43	0
L	14.95	14.55	19.51

The finally computed model is $G(s) = 0.9995e^{-19.23s}/(1 + 4.85s)$. The choice is based on the curve form of the first relay experiment and on the facts that the magnitudes of the frequency curve at the $-90°$ point and the $-180°$ point are close and that the frequency at the $-180°$ point is about twice the frequency at the $-90°$ point. The parameters of the final model are calculated by fitting to the estimated frequency points. Since $L > 1.5 * T$, the resulting controller is a discrete pole-placement controller which will give dead-time compensation. The parameters are $h = 19$, $r_1 = -0.3810$, $r_2 = -0.6190$, $t_0 = 0.6527$, $s_0 = 0.6647$

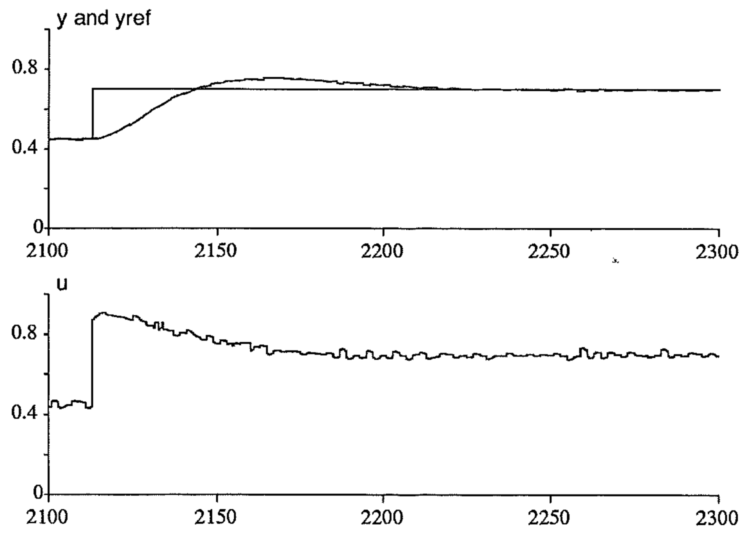

Figure 8.10 Step response for Example 8.4.

and $s_1 = -0.012$. The step response of the closed loop system is shown in Figure 8.9. The value 0 for the time constant in the third model is explained by the estimated gain at the $-90°$ point which is larger than the estimated steady state gain. □

EXAMPLE 8.4 — Second order system
In this example, the simulated process is $G(s) = 1/(1+15s)^2$. Based on the first relay experiment, the process is classified as having dynamics of second or higher order. The three estimated points on the Nyquist curve are:

	$0°$ point	$-90°$ point	$-180°$ point
Measured ω	0	0.0622	0.182
Estimated gain	1.056 (1)	0.536 (0.535)	0.123 (0.118)
Estimated phase	0 (0)	−1.493 (−1.501)	−2.527 (−2.439)

The three first order models with delay are:

	$-180°$ model	$-90°$ model	Steady state gain model
k_p	0.317	4.366	1.056
T	13.63	120.2	27.31
L	8.45	6.52	7.31

The finally computed model is $G(s) = 1.067\omega^2/(s^2 + 2\zeta\omega s + \omega^2)$ with $\zeta = 0.9$ and $\omega = 1/15.2$. A second order model was chosen as the default since no rule for selecting a first order model with delay was fulfilled. The parameters were obtained by fitting to the $-180°$ point and the steady state gain. The resulting controller is a PID controller with $K = 1.14$, $T_i = 20.87$, $T_d = 6.38$, $\beta = 1.52$ and $N = 5$. The step response of the closed loop system is shown in Figure 8.10. □

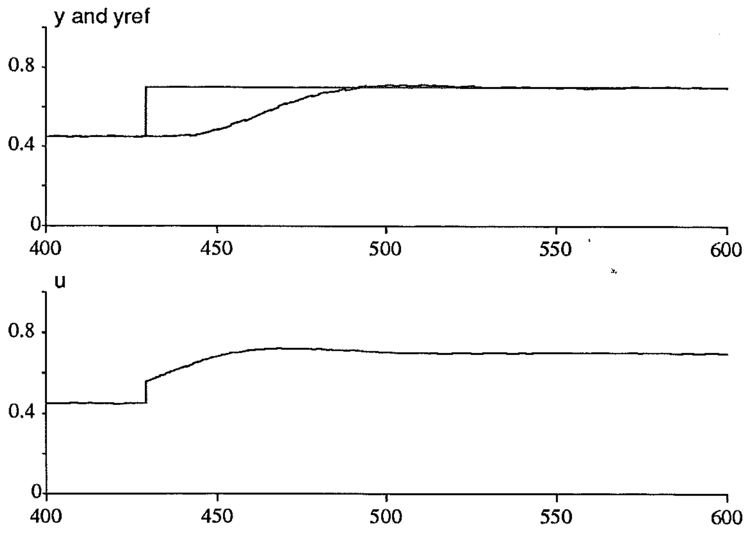

Figure 8.11 Step response for Example 8.5.

EXAMPLE 8.5 — High order system
In this example, the simulated process is $G(s) = 1/(1+5s)^6$. Based on the first relay experiment, the process is being classified as having dynamics of second or higher order. The three estimated points on the Nyquist curve are:

	0° point	−90° point	−180° point
Measured ω	0	0.0496	0.101
Estimated gain	1.002 (1)	0.858 (0.836)	0.537 (0.509)
Estimated phase	0 (0)	−1.42 (−1.46)	−3.00 (−2.79)

The three first order models with delay are:

	−180° model	−90° model	Steady state gain model
k_p	1.26	1.54	1.002
T	22.18	30.12	12.16
L	18.21	15.45	17.57

The finally computed model is $G(s) = 1.002e^{-18.7s}/(1+13.9s)$. A first order model with delay was chosen based on the oscillation curve form of the first relay experiment and since the frequency at the −180° point is about twice the frequency at the −90° point. The model parameters were obtained by fitting to the estimated frequency points. The resulting controller is a PI controller with $K = 0.39$, $T_i = 14.43$ and $\beta = 1.11$. The reason why not a PID controller was chosen was that the performance improvement was neglectable. The reason for this is the large time delay. The step response of the system is shown in Figure 8.11. □

The examples show that the tuning procedure is able to come up with reasonable controller designs for a wide range of processes. They also give a feeling for how contradictory the estimated frequency information and process models may be. To base the control design upon only one of estimated frequencies or process models may give poor results.

8.5 Extensions to the example

The current example does not show all the features of the real-time expert system framework developed. It also has limitations, e.g., it cannot handle load disturbances during the relay experiments. This section outlines how the example could be modified in order to overcome the limitations as well as give some general examples of how different knowledge sources may interact.

Other algorithms

The described tuning procedure estimates three points on the open-loop Nyquist curve. A better estimate could be obtained if an FFT algorithm was added. The tuning is based entirely on relay experiments. This is of course not the only method one may use. Alternatives would, e.g., be to use a parameter estimator at some stage in the tuning procedure or to instead use step or pulse response methods. This would require different kinds of numerical algorithms.

A few of these have, fully or partially, been implemented. One such example is an explicit recursive least-squares identification algorithm. The input parameters to this are, e.g., the sampling interval, the number of A and B parameters, the forgetting factor, initial values for the parameter estimates, initial values for the diagonal elements of the covariance matrix, and limits on the residual for when the adaptation should be interrupted. The algorithm can be asked questions about the estimated parameters, the covariance matrix, and about the residual. Messages are sent to the knowledge based system when the adaptation is interrupted and restarted.

Another algorithm that has been implemented solves the Diophantine equation. These two algorithms could be used to implement a pole-placement self-tuner in the knowledge-based controller. If a spectral factorization algorithm was added, LQG self-tuners could also be implemented.

For pulse response based tuning methods, a pulse response algorithm is needed. It is designed to measure the different integrals in the method of moments of Appendix B. Zero-crossings and the time period between zero-crossings are also measured.

Error recovery

In the current example, the Initial-relay-tuner knowledge source is by default activated when the controller is started. If an error occurs during the tuning experiment that is not anticipated by the Initial-relay-tuner, the tuning procedure and the entire knowledge-based controller stops. One solution to this problem is to have a separate source that contains knowledge of what actions to take if an error occurs. This knowledge source could be viewed as a 'meta-knowledge' source, i.e, a knowledge source which contains knowledge about other knowledge sources. The following rule could be used for starting the Initial-relay-tuner knowledge source.

```
(p initial-start

  (start-up)
-->
  (remove 1)
  (activate Initial-relay-tuner)
  (waitentry Initial-relay-tuner))
```

A load disturbance can be detected by the relay algorithm. A rule may be added to the Relay-supervisor that adds the global-fact `(relay succeeded)` if the relay experiment is successfully completed and the global-fact `(relay failed load disturbance)` if a load disturbance is detected. In the Initial-relay-tuner, the statement

```
(start-and-wait-for 'Relay-supervisor)
```

is replaced by

```
(start-wait-for 'Relay-supervisor)
(s-cond ((fact-match relay succeeded)
         (global-remove relay succeeded))
        ((fact-match relay failed load disturbance)
         (global-remove relay failed load disturbance)
         (global-fact
           Initial-relay-tuner failed
           load disturbance in first experiment)
         (halt-procedural-knowledge-source)))
```

The statement `(global-fact Initial-relay-tuner succeeded)` is added as the final statement in the knowledge source. The same modifications are performed at the second relay experiment but with a different failure message. It is now easy to write rules in the 'meta-knowledge' source that takes care of the errors.

```
(p load-disturbance-1

 (Initial-relay-tuner failed
  load disturbance in first experiment)
-->
 (global-remove 1)
 (activate Initial-relay-tuning-2)
 (waitentry Initial-relay-tuning-2))
```

The Initial-relay-tuning-2 is another procedural knowledge source that does not contain the query phase and which may make use of the partial information that already has been acquired. In the same way, a rule may be written to take care of the situation where the load disturbance occured during the second relay experiment.

The example only considers the initial tuning phase. Monitoring of steady state control may also be implemented using the 'meta-knowledge' source. Suppose that steady state monitoring was implemented. The initial-relay-tuning knowledge source would then typically end as

```
   .
   .
(activate Control-supervisor)
(activate Monitoring-k-s-1)
(activate Monitoring-k-s-2)
(global-fact Initial-relay-tuning succeeded)))
```

where `Monitoring-k-s-1` and `Monitoring-k-s-2` are knowledge sources that implements different monitoring aspects.

The monitoring could typically result in two different things. Either the controller parameters are slightly adjusted or a new tuning experiment is initiated. Parameter adjustments may be implemented by rules in the existing knowledge sources. A new tuning experiment, however, typically involves a sequence of knowledge source activations. The start of a procedural knowledge source for this could be handled by the 'meta-knowledge' source as in the following

```
(p re-tuning

  (Monitoring-k-s-1 stopped retuning needed)
-->
  (deactivate Control-supervisor)
  (deactivate Monitoring-k-s-2)
  (activate Re-tuning-k-s)
  (waitentry Re-tuning-k-s))
```

where `Re-tuning-k-s` is the procedural knowledge source that performs the retuning experiment. The same arrangements could be made to handle, e.g., mode changes initiated by the operator.

Control performance monitoring

The implementation of different monitoring alternatives will be outlined.

When the Initial-relay-tuning knowledge source has finished, a 'good' controller has been designed and started. One way to judge the quality of the controller designed is to involve the operator and give him the opportunity to judge the controller performance. It would, e.g., be possible to add commands for changing the bandwidth or overshoot of the system.

Another possibility is to leave it to the knowledge-based system to assess the quality of the designed controller. The performance during set-point changes could be assessed by measuring the overshoot and settling time when set-point changes are made. The overshoot could be estimated by examining the extreme values of the process output during a time determined by the closed loop dynamics. The settling-time could be estimated with the help of the level-crossing algorithm.

An extra algorithm is needed to adjust the controller based on load disturbances. This algorithm should be designed to detect peaks in the error signal. It should also measure the time interval between error peaks. This algorithm, together with a knowledge source that performs the necessary control adjustments could be used to implement the tuning mechanism of the Exact auto-tuner (Kraus and Myron, 1984).

Another possibility is to use the statistics gathered. If, e.g., the variances are slowly increasing, it may be the case that the process dynamics has varied since the tuning and that a re-tuning need to be made.

The possibility to test rules periodically is probably most useful for monitoring. The following rule could be used to check if the process output variance has exceeded some limit value.

```
(p periodic-variance-check

  (wakeup periodic-variance-check)
  (rule periodic-variance-check -period)
  (variance-limit -limit)
  (measurement-time -time)
-->
  (remove 1)
  (cond ((> (variance-of y -time) -limit)
         (global-fact initiate retuning))
        (t
         (timer-request -period periodic-variance-check))))
```

Good use of the possibility to associate duration times to database elements can be made in this case. Suppose that, the variance check only should be performed when the system is not in a transient mode. This could be accomplished by modifying the rule to

```
(p periodic-variance-check

  (wakeup periodic-variance-check)
  (rule periodic-variance-check -period)
  (variance-limit -limit)
  (measurement-time -time)
(~ (transient-mode))
-->
  (remove 1)
  (cond ((> (variance-of y -time) -limit)
         (global-fact initiate retuning))
        (t
         (timer-request -period periodic-variance-check))))
```

and by introducing a rule which adds the fact **(transient-mode)** with a certain duration, when a set-point change is made.

```
(p set-point-change

  (new set-point -yref1)
  (set-point -yref2)
  (settling-time -time)
-->
  (send global yref -yref1)
  (remove 1 2)
  (global-fact set-point -yref1)
  (global-fact transient-mode :duration ^(seconds -time)))
```

One of the actions of the periodic rules could be that it, if something unnormal has been detected, activated a knowledge source that performed further examinations. This construct is similar to the *focusing* facility of PICON (Picon, 1985) and G2 (Gensym, 1987). Periodic rules can, e.g., be used to periodically run procedural tests

```
(p periodic-test-procedure

  (wakeup periodic-test-procedure)
  (rule periodic-test-procedure -period)
   <pattern>

-->
  (activate procedural-test-1)
  (activate procedural-test2)
  <concluding actions>
  (timer-request -period periodic-test-procedure))
```

`Procedural-test-1` and `procedural-test-2` would in this case be procedural knowledge sources that performed the tests. Another example is where periodic rules are used to regularly perform hypothesis verifications using the following construct:

```
(p periodic-hypothesis-verification

  (wakeup periodic-hypothesis-verification)
  (rule periodic-hypothesis-verification -period)
   <patterns>

-->
  (fact-in scheduler
    verify process <attribute> <value>)
  <concluding actions>
  (timer-request -period periodic-hypothesis-verification))))
```

This assumes that a backward-chaining knowledge source is active which can verify the given hypothesis.

8.6 Automatic generation of gain schedules

The second example considers the initial tuning of processes where it can be suspected that gain-scheduling is necessary. The tuning procedure used is less sophisticated than the previously described. It performs a relay experiment without an integrator and based on that classifies the process as either PI-controllable or being of second or higher order. In the PI-controllable case, a PI design is made which assumes that the process can be approximated by a pure integrator. If the process is of higher order, a modified Ziegler-Nichols PID design is made. The reason why a less elaborated tuning procedure is used is mainly chronological. This example was developed earlier than the previous one.

When the tuning has been performed at one set-point, the controller performance around that set point is assessed. This is done by measuring the steady state variances and by performing small set-point changes around the set-point and measuring the overshoot and settling-time.

The need for gain-scheduling is judged based on the linearity of the static gain characteristic. The static gain at five equally spread set-points is recorded by measuring the steady state control signal needed at every point. The control performance around each new set-point is compared with the originally assessed performance. If the performance has deteriorated significantly, a tuning experiment is performed around the new set-point and a new controller is designed. When the static characteristic is known at five points, the linearity of the characteristic is roughly estimated. If the process is considered to be nonlinear, tuning experiments are performed around each set-point where previously no tuning has been performed. The controller parameters are stored in a gain-scheduling table.

In this experiment, the frame definitions are somewhat different than before. For example, the controller frames also has associated what static point they were designed for.

The knowledge sources used are roughly the same as before, although many of the rules are different. The modelling knowledge is contained in the Relay-supervisor. The Start knowledge source is new and is used only for initiating the Operator-inquirer. The sweeping between different set-points is performed by the procedural knowledge source Linearity-checker.

The knowledge sources are in this example combined through dynamic sequence generation. The different knowledge sources have the following associated preconditions and goals:

Knowledge source	**Preconditions**	**Goals**
Operator-inquirer	nil	(a-priori-collected)
Start	nil	(started)
Manual-control-supervisor	(started a-priori-collected)	(steady-state noise-known)
Relay-supervisor	(noise-known steady-state)	(relay-ready)
Designer	(relay-ready)	(pid-control)
Linearity-checker	(pid-control)	(linearity-info-available)
Control-supervisor	(linearity-info-available)	(pid-monitoring-on)
Y-statistics	nil	(y-statistics-on)
U-statistics	(pid-control)	(u-statistics-on)

When the controller is started the goal-stack has the value

```
goalstack ((u-statistics-on
            pid-monitoring-on
            y-statistics-on))
```

The plan in Figure 8.12 is generated from this. Planning is also used when a new relay experiment needs to be done. In that case, the goal **`(pid-control)`** is pushed onto the goal stack. This will cause the plan

```
    (s Relay-supervisor Designer)
```

to be created and executed.

The possibility to combine knowledge sources using planning ideas can be used as an alternative to the pre-stored sequences used in the first example. It is, however, the author's experience that, at least for the knowledge-based control

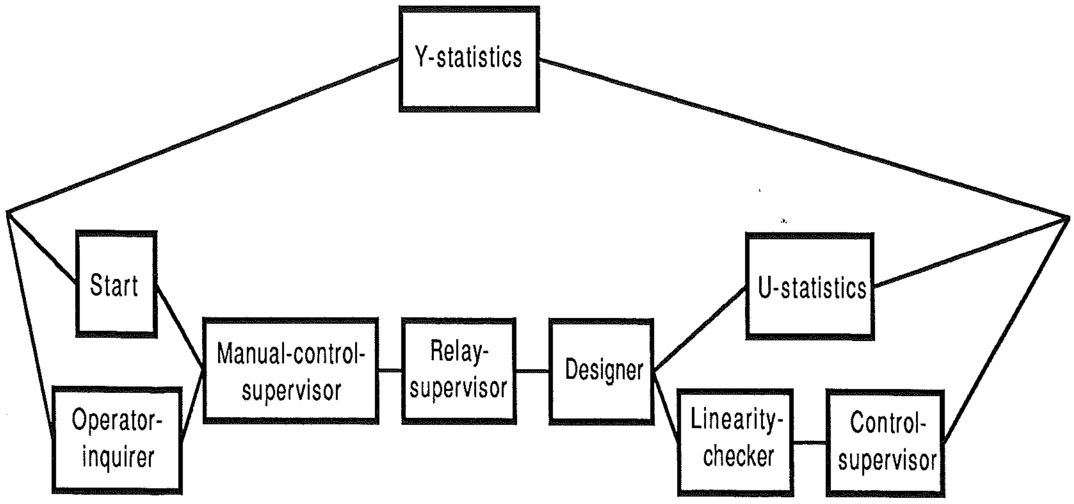

Figure 8.12 Tuning plan.

problem, the pre-stored sequences gives a better understanding of how the system operates. This could of course depend on the implementation of this particular planning system. Similar experiences have also been found by others (Trankle, 1986). A motivation for the planning approach is that it gives the system the possibility to generate new solutions when confronted with situations that weren't anticipated from the beginning. The new solutions would consist of new combinations of knowledge sources. To make this possible, the individual knowledge sources must be extremely carefully designed. It is the author's experience that this is very difficult to do. In practice, it would mean that the individual knowledge sources should be designed to handle all different cases that the designer can think of. Using pre-stored sequences seems to be a better alternative.

9

Conclusions and Suggestions for Future Work

There is currently a significant interest in expert system techniques in the process control community. A common application is to use the expert system as an operator aid for process monitoring on top of a conventional, distributed control system. The topic of this thesis has been to explore the possibilities of using expert system techniques in feedback control systems. This application requires a much tighter integration of the expert system with the control system. It also points towards a different structure of process control systems. A problem with using expert systems on-line is that they are slow. Most industrial processes are, however, also slow. The dynamic computer development may also allow the ideas to be applied to faster control loops.

In Chapter 2, it was argued that the need for the expert system approach is just as large on the local, controller level as on the plant-wide level. The local control problem is, however, also simpler and thus a good starting point for research. It is also of interest to see that even conventional feedback loops contain a significant amount of heuristic logic. The need for separation between the numeric algorithms and the heuristic logic as well as the need for a structured implementation of the heuristics was pointed out.

The initial tuning of a regulator is a good example of a problem where heuristic decisions are mixed with theoretical knowledge and numerical algorithms. Chapter 3 treated relay tuning. New theorems which describe the oscillation curve form under relay feedback were given. The theoretical results were used to derive empirical rules for determining different classes of process dynamics. An empirical method was presented which can be used for sorting out processes with dynamics that well may be approximated with a first order system. The method is based on measuring whether the derivative of the process output changes sign at the relay switching time or not.

The main contribution in the thesis concerns the design and architecture of a knowledge based controller. Chapter 4 presented an architecture where the heuristic logic and the numerical algorithms are implemented as separate concurrent processes. The implementation of the architecture on a VAX 11/780 computer was described. A flexible implementation environment for the numerical algorithms where the algorithms can be re-configured on-line and independently of each other was considered important.

Chapter 6 describes a prototype system where the conventional, off-the-shelf expert system framework OPS4 was used to implement the heuristic logic. Several conclusions can be drawn from this prototype. The separation between heuristics and numerics is valuable. It also proved relatively straight-forward to extend the system and add new functionality by introducing new algorithms and logic. The need for structuring facilities was, however, apparent. It is not natural to express all problem types with a single representation technique. It is therefore desirable to have a system that allows different representations. For example, diagnosis may be stated as a rule-based backward chaining problem. The monitoring of incoming events is easily done with forward chaining rules. Other tasks are best represented sequentially. The need for structuring the common database was also clear. The simple list structures of OPS4 are not suited for representing related information. The possibility to have frame structures is important. Another observation was the lack of real-time constructs in the majority of existing expert system frameworks. Real-time constructs are needed in real-time applications. For example, the monitoring task requires that tests may be performed periodically.

To overcome the problems with existing expert system shells a real-time expert system framework has been developed. This was described in Chapter 7. A modular approach, inspired by the blackboard ideas, has been followed. It allows a decomposition of the problems into a number of subtasks which are implemented as separate modules in the form of knowledge sources. The knowledge representation in each individual knowledge source can be chosen to fit the actual problem structure. Currently, rule-based forward or backward chaining can be used as well as a procedural representation. The object oriented implementation makes it easy to extend the system with new methods for knowledge representation. The common database allows both frame structures and lists. The implemented framework can be compared with a conventional real-time operating system and has similar real-time constructs. Knowledge sources can be suspended for a certain time or until some database condition is fulfilled. Rules can be executed periodically and duration times may be associated with database elements.

Examples of how the framework is used were presented in Chapter 8. The examples were focussed on the initial tuning of a regulator. A relay based auto-tuning procedure was presented which makes use of three points on the open-loop

Nyquist curve and can design controllers in many different ways. The separation between algorithms and knowledge sources was shown and examples were given of what rules may look like.

How to encode automatic control knowledge into a program is a research topic in itself. The examples presented have been iterated and re-implemented several times and still have limitations. During this development, the knowledge-based approach has proved very valuable. The modularity of the system has made it possible to develop and test the individual knowledge sources quite independently. The rule-based programming style has supported the structured implementation of separate pieces of knowledge. The "parallel" control structure of forward chaining supports extensions of the system with new rules. All rules are treated equally where the current contents of the database determine which rule that should be fired. The possibility to combine procedural and rule-based representation has shown to be natural way to handle problems with a large sequential element.

It may be questioned if the tuning procedure could not just as well be implemented with standard techniques. It would of course be possible to implement a system with the same functionality or 'input-output' behavior. The internal structure would, however, be entirely different. It is the author's opinion that the knowledge based approach has many advantages, e.g., the ease at which the system can be modified and extended, the explicit, declarative representation of knowledge,the interactive development environment, etc.

Suggestions for future work

This thesis may be continued in several different ways. When the work was started, the focus was more on the control part of it than on the expert system parts. Due to the short-comings of existing expert system tools, the emphasis gradually slided over towards real-time expert systems. As a result of this, many things remains to be done before a true "intelligent" controller has been built. The examples presented can be extended and some of their limitations may be removed. For example, load disturbances that occur during the relay experiment need to be handled, the heuristics for choosing the final process model can be refined, the control design can be improved. It is, however, probably not that much work that remains before a true "smart" PID controller would be available. Such a system would incorporate knowledge about simple control loops at the level of a good process engineer. A further extension would be to add adaptivity to the system. Correctly initialised conventional parameter estimation methods may then be used. Another extension would be to develop tuning procedures that also work for e.g. cascaded control loops.

The examples provided deal only regarded the initial tuning of a regulator. Monitoring is an equally important part of the knowledge based controller. The human is good at extracting symbolic information by looking at plotted signal curves. To develop algorithms that can do the same is an important issue. Nu-

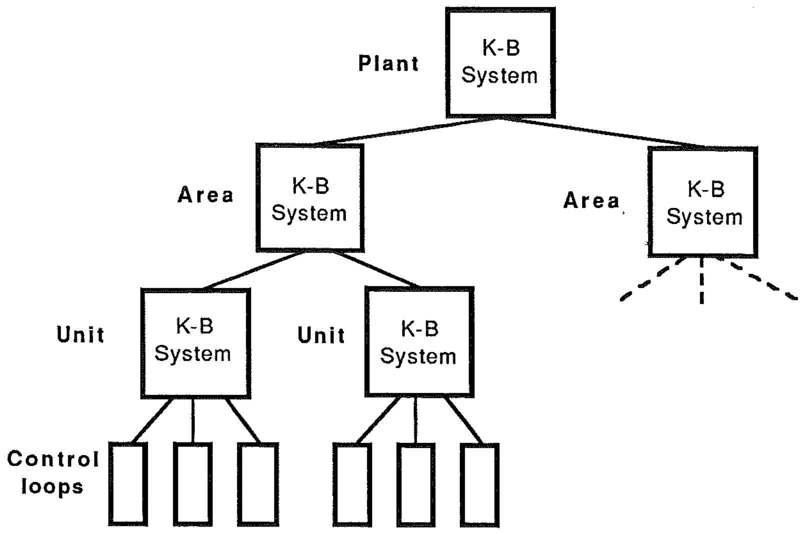

Figure 9.1 Hierarchical knowledge-based control structure

merical algorithms that can extract phase and amplitude-margins for a process under, e.g., PID control would be very useful tools.

The current implementation can only handle one control loop at a time. The pattern matching based system makes it relatively easy to extend it to allow multiple control loops. In the current system, the numerical algorithm and the knowledge based system are implemented as concurrent processes in one processor. An alternative is to represent them in separate processors. The numerical algorithms could then be implemented with standard hardware, e.g. one microprocessor for each control loop, and the knowledge based system could be implemented with special purpose hardware. The communication issues between the separate processors then becomes an important issue. A hierarchical structure with knowledge-based systems at different levels is an attractive solution.. On the lowest level, the knowledge-based system handle the control loops in a certain unit. Higher up another knowledge-based systems may be used to coordinate the different units that belong to a certain area of the plant as seen in Figure 9.1.

A different future direction is to instead focus on the real-time expert system framework. Exploring how and if truth maintenance techniques might be used is one key problem. Others could be to develop the planning mechanisms of the system or to improve the backward chaining capabilities. There is also much that can be done in order to improve the speed of the framework. Some of this is outlined in Chapter 7. Currently, the scheduler is implemented as a rule-based YAPS system. An implementation with this part in plain Lisp will probably run faster. The same thing is true about the algorithm handler.

References

ABELSON, H. and G.J. SUSSMAN (1985): *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, MA.

ALLEN, E.M. (1983): "YAPS: Yet another production system," TR-1146, Department of Computer Science, University of Maryland.

ALLEN, J.F. (1984): "Towards a general theory of action and time," *Artificial Intelligence* **23**, 123–154.

ÅRZÉN, K-E. (1986a): "Expert systems for process control," in D. Sriram and R. Adey (Eds.): *Proc. of First International Conference on Applications of Artificial Intelligence in Engineering Practice*, Springer Verlag, Berlin, pp. 1127–1138.

ÅRZÉN, K-E. (1986b): "Use of expert systems in closed loop feedback control," *Proc. of American Control Conference*, Seattle, WA.

ÅSTRÖM, K.J. (1979): "Simple self-tuners I," Technical report TFRT-7184, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K.J. (1982): "Ziegler-Nichols auto-tuners," Technical report TFRT-3167, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K.J. (1983): "Implementation of an auto-tuner using expert system ideas," Technical report TFRT-7256, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K.J. (1987a): "Adaptive feedback control," *Proc. of IEEE* **75**, 2, 185–217.

ÅSTRÖM, K.J. (1987b): "Implementation of PID regulators," Technical report TFRT-7344, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K.J. and J.J. ANTON (1984): "Expert control," *Proc. 9'th IFAC World Congress*, Budapest, Hungary.

ÅSTRÖM, K.J. and P. EYKHOFF (1971): "System identification — a survey," *Automatica* **7**, 123–167.

ÅSTRÖM, K.J. and T. HÄGGLUND (1984a): "Automatic tuning of simple regulators," *Proc. IFAC 9'th World Congress*, Budapest, Hungary.

ÅSTRÖM, K.J. and T. HÄGGLUND (1984b): "Automatic tuning of simple regulators with specifications on phase and amplitude margins," *Automatica* **20**, 645–651.

ÅSTRÖM, K.J. and T. HÄGGLUND (1984c): "A frequency domain approach to automatic tuning of simple feedback loops," *Proc. 23rd IEEE Conf. on Decision and Control, Las Vegas.*

ÅSTRÖM, K.J. and T. HÄGGLUND (1987): "A new auto-tuning design," Technical report TFRT-7368, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K.J. and T. HÄGGLUND (1988): *Automatic Tuning of PID Regulators,* To appear, Instrument Society of America, Research Triangle Park, North Carolina.

ÅSTRÖM K.J. and B. WITTENMARK (1984): *Computer Controlled Systems,* Prentice-Hall, Englewood Cliffs, New Jersey.

ÅSTRÖM K.J. and B. WITTENMARK (1988): *Adaptive Control,* To appear, Addison-Wesley, Reading, MA.

ÅSTRÖM, K.J., J.J. ANTON and K.-E. ÅRZÉN (1986): "Expert control," *Automatica* **22**, 3, 277–286.

ATHERTON, D.P. (1975): *Nonlinear Control Engineering,* van Nostrand Reinhold Co., London, UK.

BÅÅTH, L (1987): "Personal communication,".

BIRDWELL, J.D., J.R.B. COCKETT, R. HELLER, R.W. ROCHELLE, A.J. LAUB, M. ATHANS and L. HATFIELD (1985): "Expert systems techniques in a computer based control system analysis and design environment," *Proc. 3rd IFAC/IFIP Int. Symp. on Computer Aided Design in Control and Engineering Systems,* Lyngby, Denmark.

BOBROW, D.G. and M. STEFIK (1983): "The LOOPS manual," Xerox Corporation, Palo Alto, CA.

BOBROW, D.G. and T. WINOGRAD (1977): "An overview of KRL, a knowledge representation language," *Cognitive Science* **1**, 1, 3–46.

BOBROW, D.G., K. KAHN, G. KICZALES, L. MASINTER, M. STEFIK and F. ZDYBEL (1985): "COMMONLOOPS: Merging Common Lisp and object-oriented programming," Intelligent Systems Lab. Series ISL-85-8, Xerox Palo Alto Research Center, Palo Alto, California.

BRACHMAN, R.J. (1979): "On the epistemological status of semantic networks," in N.V. Findler (Ed.): *Associative Networks: Representation and Use of Knowledge by Computers,* Academic Press, New York.

BRISTOL, E.H. (1977): "Pattern recognition: An alternative to parameter identification in adaptive control," *Automatica* **13**, 197–202.

BROWNSTON, L., R. FARRELL, E. KANT and N. MARTIN (1985): *An Introduction to Rule-based Programming,* Addison-Wesley, Reading, MA.

BYRNES, C.I. (1985): "Necessary conditions in adaptive control," in C.I. Byrnes and A. Lindquist (Eds.): *Proc. 7th International Symposium on the Mathematical Theory of Networks and Systems*, North Holland, Amsterdam.

CANNON, H.I. (1982): "Flavors: A non-hierarchical approach to object-oriented programming,," unpublished paper.

CARLSSON, M. (1983): "On implementing Prolog in functional programming," Technical Report no. 5, UPMAIL, Uppsala University, Uppsala, Sweden.

CARROLL, J.M. and J. MCKENDREE (1987): "Interface design issues for advice-giving expert systems," *Communications of the ACM* **30**, 1, 14–31.

CHARNIAK, E., C.K. RIESBECK and D.V. MCDERMOTT (1980): *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

CLARK, D.W. (1984): "Implementation of adaptive controllers," in Harris and Bildings (Eds.): *Self-tuning as Adaptive Control*, Peter Peregrinus, U.K..

CLOCKSIN, W.F. and C.S. MELLISH (1984): *Programming in Prolog*, Springer-Verlag, Berlin.

COX, B.J. (1986): *Object Oriented Programming - An Evolutionary Approach*, Addison-Wesley, Reading, MA.

CROSSMAN, E.R.F.W. and J.E. COOKE (1962): "Manual control of slow-response systems," in E. Edwards and F.P. Lees (Eds.): *The Human Operator in Process Control*, Taylor and Francis, Ltd., London, 1974.

DAHL, O.J. and K. NYGAARD (1966): "SIMULA–an algol-based simulation language," *Communications of the ACM* **9**, 671–678.

DE KLEER, J. (1986): "An assumption-based TMS," *Artificial Intelligence* **28**, 127–162.

DIGITAL EQUIPMENT CORPORATION (1984): *Introduction to VAX/VMS System Routines*, VAX/VMS Version 4.0, Digital Equipment Corporation, Maynard, MA.

DOYLE, J. (1979): "A truth maintenance system," *Artificial Intelligence* **20**, 231–272.

DRESCHER, G.L. (1985): "The ObjectLisp user manual (preliminary)," LMI Corp., Cambridge, MA.

DUDA, R.O., P.E. HART and R. REBOH (1977): "A rule-based consultation system for mineral exploration," *Proc. of the Lawrence Symposium on Systems and Decision*, UC Berkeley, California, pp. 306–309.

ELMQVIST, H. (1975): "SIMNON, An interactive simulation program for nonlinear systems," Technical report TFRT-3091, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ENSOR, J.R. and J.D. GABBE (1985): "Transactional blackboards," *Proc. of the 9th International Joint Conference on Artificial Intelligence*, William Kaufmann, Inc., Los Altos, CA, pp. 340–344.

EYKHOFF, P (1974): *System Indentification: Parameter and State Estimation*, Wiley, London.

FAGAN, L.M. (1978): "Ventilator Manager: A program to provide on-line consultative advice in the intensive care unit," Heuristic Programming Project Memo HPP-78-16, Department of Computer Science, Stanford University, California.

FIKES, R.E. and N.J. NILSSON (1971): "STRIPS: A new approach to the application of theorem proving in problem solving," *Artificial Intelligence* **2**, 189–208.

FISHER, E.L. (1986): "An AI-based methodology for factory design," *AI Magazine* **7**, 4, 72–85.

FODERARO, J.K., K.L. SKLOWER and K. LAYER (1983): "The Franz Lisp Manual," UC Berkeley, California.

FORGY, C.L. (1979): "OPS4 User's manual," Technical report CMU-CS-79-132, Department of Computer Science, Carnegie-Mellon University.

FORGY, C.L. (1981): "OPS5 User's manual," Technical report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University.

FORGY, C.L. (1982): "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence* **19**, 1, 17–37.

FRANCIS, J.C. and R.R. LEITCH (1985): "Artifact: A real-time shell for intelligent feedback control," in M.A. Bramer (Ed.): *Research and Developments in Expert Systems*, Cambridge University Press, UK.

FU, K-S. (1970): "Learning control systems – review and outlook," *IEEE Transactions on Automatic Control* **15**, 210–221.

FU, K-S. (1971): "Learning control systems and intelligent control systems: An intersection of artificiall intelligence and automatic control," *IEEE Transactions on Automatic Control* **16**, 70–73.

GALE, W.A. and D. PREGIBON (1982): "An expert system for regression analysis," *Proc. of the 14th Symposium on the Interface, Troy, N.Y.*, Springer Verlag, New York, pp. 110–117.

GENSYM (1987): *G2 User's manual*, Gensym Corp., Cambridge, MA.

GEORGEFF, M.P. and A.L. LANSKY (1986): "Procedural knowledge," *Proceedings of the IEEE* **74**, 10, 1383–1398.

GLATTFELDER, A.H. and W. SCHAUFFELBERGER, "Stability analysis of single-loop control systems with saturation and antireset-windup circuits," *IEEE Transactions on Automatic Control* **28**, 12, 1074–1081.

GOLDBERG, A. and D. ROBSON (1983): *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA.

GREEN, C. (1969): "Application of theorem proving to problem solving," *Proc. IJCAI*, Washington DC, pp. 219–239.

HÄGGLUND, T. (1981): "A PID tuner based on phase margin specification," Technical report TFRT-7224, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HÄGGLUND, T. and K.J. ÅSTRÖM (1985): "Automatic tuning of PID controllers based on dominant pole design," *Proceedings of IFAC Conference on Adaptive Control of Chemical Processes*, Frankfurt, W. Germany.

HAMEL, B. (1949): "Contribution a l'etude mathematique des systemes de reglage par tout-ou-rien," *C.E.M.V. Service Technique Aeronautique* **17**.

HARMON, P. and D. KING (1985): *Expert Systems*, Artificial Intelligence in Business, John Wiley & Sons, Inc., New York.

HAYES P. (1973): "The frame problem and related problems in artificial intelligence," in A. Elithorn and D. Jones (Eds.): *Artificial and Human Thinking*, Jossey-Bass Inc..

HAYES-ROTH, F., D. WATERMAN and D. LENAT (1983): *Building Expert Systems*, Addison-Wesley, Reading, MA.

HEIN, U. (1983): "PAUL–The kernel of a representation and reasoning system designed for knowledge engineering tasks," AILAB Working paper No 16, Linköping University, Sweden.

HOLMBLAD, L.P. and J.J. OSTERGAARD (1981): "Control of a cement kiln by fuzzy logic," F.L. Smidth Review, Copenhagen, Denmark.

HOOPES, H.S., W.M. HAWK JR. and R.C. LEWIS (1983): "A self-tuning controller," *ISA Transactions* **22**, 49–58.

INFERENCE CORP. (1984): *ART – User Manual.*

INTELLICORP (1984): *Knowledge Engineering Environment (KEE) – User Manual*, Menlo Park, CA.

ISERMANN, R. (1982): "Parameter adaptive control algorithms – a tutorial," *Automatica* **18**, 513–528.

JAMES, J.R., D.K. FREDERICK and J.H. TAYLOR (1985): "The use of expert-system programming techniques for the design of lead-lag compensators," *IEE Conference, Control '85*, Cambridge, England.

JURY, E.I. (1964): *Theory and Application of the Z-Transform Method*, John Wiley, New York.

KASHTAN, D.L. (1982): "EUNICE: A system for porting UNIX programs to VAX/VMS," Artificial Intelligence Center, SRI International, Menlo Park, California.

KNUTH, D.E. (1984): *The TEXbook*, Addison-Wesley, Reading, MA.

KRAUS, T.W. and T.J. MYRON (1984): "Self-tuning PID controller uses pattern recognition approach," *Control Engineering*, June, 106–111.

LARSSON, J.E. and P. PERSSON (1987): "An expert system interface for IDPAC," Technical report TFRT-3184, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LILJA, M. (1987): "Least squares fitting to rational transfer function with time delay," Technical report TFRT-7363, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LINDQVIST, T. (1985): "En auto-tuner för PI regulator," Master thesis TFRT-5332, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, In swedish..

LJUNG, L. (1987): *System Identification: Theory for the user*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

LUKAS, M.P. (1986): *Distributed Control Systems*, Their evaluation and design, Van Nostrand Reinhold Company, New York.

MÅRTENSSON, B. (1986): "Adaptive stabilization," PhD thesis, TFRT-1028, Lund, Sweden.

MAMDANI, E.H. and S. ASSILIAN (1975): "A fuzzy logic controller for a dynamic plant," *Int. Journal of Man-Machine Studies* **7**, 1–13.

MCCARTHY, J. and P.J. HAYES (1969): "Some philosophical problems from the standpoint of artificial intelligence," in B. Meltzer and D. Mitchie (Eds.): *Machine Intelligence*, American Elsevier, New York.

MCCARTHY, J. (1980): "Circumscription–A form of non-monotonic resoning," *Artificial Intelligence* **13**, 27–39.

MCDERMOTT, D. and J. DOYLE (1980): "Non-monotonic logic I," *Artificial Intelligence* **13**, 41–72.

MCDERMOTT, D. (1982): "A temporal logic for reasoning about processes and plans," *Cognitive Science* **6**, 2.

MCDERMOTT, J. (1980): "R1: A rule-based configurer of computer systems," Technical report 80–119, Carnegie-Mellon Department of Computer Science.

MICHALSKI, R.S., J. CARBONELL and T. MITCHELL (1983): *Machine Learning : an artificial intelligence approach*, Tioga Press, Palo Alto, CA.

MICHIE, D, and R.A. CHAMBERS (1968): "BOXES: an experiment in adaptive control," in E. Dale and D. Michie (Eds.): *Machine Intelligence 2*, Oliver and Boyd, Edinburgh, pp. 137–152.

MILLIKEN, K.R., A.V. CRUISE, R.L. ENNIS, A.J. FINKEL, J.L. HELLERSTEIN, D.J. LOEB, D.A. KLEIN, M.J. MASULLO, H.M. VAN WOERKORN and N.B. WAITE (1986): "YES/MVS and the automation of operations for large computer complexes," *IBM Systems Journal* **25**, 2, 159–180.

MINSKY, M. (1975): "A framework for representing knowledge," in P.H. Winston (Ed.): *The Psychology of Computer Vision*, McGraw-Hill, New York.

MOORE, R.L., L.B. HAWKINSON, C.G. KNICKERBOCKER and L.M. CHURCHMAN (1984a): "A real-time expert system for process control," *Proc. First Conf. on AI Applications*, IEEE Computer Society, Denver, Colorado, pp. 529–576.

MOORE, R.L., L.B. HAWKINSON, C.G. KNICKERBOCKER and L.M. CHURCHMAN (1984b): "Expert system applications in industry," *Proc. Instrument Society of America, Int. Conf.*, Houston.

MOORE, R.L., L.B. HAWKINSON, M.E. LEVIN and C.G. KNICKERBOCKER (1985): "Expert control," *Proc. American Control Conf.*, Boston, MA, pp. 885–887.

NELSON, W.R. (1982): "REACTOR: An expert system for diagnosis and treatment of nuclear reactor accidents," *Proc. of the National Conference on Artificial Intelligence*, Pittsburgh, PA, pp. 296–301.

NEWELL, A. and H.A. SIMON (1972): *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ.

NEWELL, A., J.C. SHAW and H.A. SIMON (1960): "Report of a general problem-solving program for a computer," *Proc. of an International Conference on Information Processing*, UNESCO, Paris, France, pp. 256–264.

NII, H.P. (1986a): "Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures," *AI Magazine* **7**, 2, 38–53.

NII, H.P. (1986b): "Blackboard systems: Blackboard application systems, blackboard systems from a knowledge engineering perspective," *AI Magazine* **7**, 3, 82–106.

NII, H.P., E.A. FEIGENBAUM, J.J. ANTON and A.J. ROCKMORE (1982): "Signal-to-symbol transformation: HASP/SIAP case study," *AI Magazine* **3**, 2, 23–35.

NIIDA, K. and T. UMEDA (1986): "Process control system synthesis by an expert system," in M. Morari and T.J. McAvoy (Eds.): *Chemical Process Control – CPCIII*, CACHE, Elsevier, Amsterdam.

NILSSON, M. (1983): "FOOLOG – A small and efficient Prolog interpreter," Technical report no. 20, UPMAIL, Computing Science Department, Uppsala University, Sweden.

NILSSON, N.J. (1982): *Principles of Artificial Intelligence*, Springer-Verlag, Berlin.

OHTA, T., N. SANNOMIYA, Y. NISHIKAWA, H. TANAKA and K. TANAKA (1980): "A new optimization method of PID control parameters for automatic tuning by process computer," *Proceedings of the IFAC Symposium Computer Aided Design of Control Systems*, Pergamon Press, Oxford, pp. 133–138.

PICON (1985): "PICON User Guide," LMI, Cambridge, MA.

PALOWITCH JR, B.L. and M.A. KRAMER (1985): "The application of knowledge-based expert system to chemical plant fault diagnosis," *Proc. ACC*, Boston, MA, pp. 646–651.

PANG, G.K.H. and A.G.J. MCFARLANE (1987): *An Expert System Approach to Computer-Aided Design of Multivariable Systems*, Springer Verlag, Berlin.

PELAVIN, R. and J.F. ALLEN (1986): "A formal logic of plans in temporally rich domains," *Proceedings of the IEEE* **74**, 10, 1364–1382.

QUILLIAN, M.R. (1966): "Semantic memory," Report AFCRL-66-189, Bolt, Beranek & Newman, Cambridge, MA.

REITER, R. (1980): "A logic for default reasoning," *Artificial Intelligence* **13**, 81–132.

RITCHIE, D.M. and K. THOMPSON (1978): "The UNIX time-sharing system," *The Bell System Technical Journal* **57**, 6, 1905–1929.

ROSENBLATT, F. (1961): *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington DC.

SACERDOTI, E.D. (1974): "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence* **5**, 115–135.

SACERDOTI, E.D. (1975): "A structure for plans and behavior," Tech. note 109, AI Center, SRI International Inc., Menlo Park, CA.

SAKAGUCHI, T. and K. MATSUMOTO (1983): "Development of a knowledge-based expert system for power system restoration," *IEEE Transactions on Power Apparatus and Systems* **PAS-102**, 2.

SANDEWALL, E. (1972): "An approach to the frame problem, and its implementation," in B. Meltzer and D. Michie (Eds.): *Machine Intelligence*, Wiley, New York, pp. 195–204.

SANDEWALL, E. and R. RÖNNQUIST (1986): "A representation of action structures," *Proc. of the 5th National Conf. on Artificial Intelligence*, AAAI, Philadelphia.

SANOFF, S.P and P.E. WELLSTEAD (1985): "Expert identification and control," *Proc. IFAC Identification and System Parameter Estimation*, York, UK, pp. 1273–1278.

SARIDIS, G.N. (1977): *Self-Organizing Control of Stochastic Systems*, Marcel Dekker, Inc., New York.

SARIDIS, G.N. (1983): "Intelligent robotic control," *IEEE Transactions on Automatic Control* **28**, 5, 547–557.

SCHINSKEY, F.G. (1986): "An expert system for the design of distillation controls," in M. Morari and T.J. McAvoy (Eds.): *Chemical Process Control – CPCIII*, CACHE, Elsevier, Amsterdam.

SHORTLIFFE, E.H. (1976): *Computer Based Medical Consultations: MYCIN*, Elsevier, New York.

SMITH, S.F., M.S. FOX and P.S. OW (1986): "Constructing and maintaining detailed production plans: Investigations into the development of knowledge-based factory scheduling systems," *AI Magazine* **7**, 4, 45–61.

STEELE JR., G.L. (1984): *Common Lisp*, Digital Press, Digital Equipment Corp..

STEFIK, M. and D.G. BOBROW (1986): "Object-oriented programming: Themes and variations," *AI Magazine* **6**, 4, 40–62.

STENERSON, R.O. (1986): "Integrating AI into the avionics engineering environment," *Computer*, Febr, 88–91.

STROUSTRUP, B. (1986): *The C++ Programming Language*, Addison-Wesley, Reading, MA.

SUSSMAN, G.J. and G.L. STEELE (1975): "SCHEME: An interpreter for extended lambda calculus," Memo 349, MIT AI Laboratory, Cambridge, MA.

SUSSMAN, G.J. and G.L. STEELE (1978): "The revised report on SCHEME," Memo 452, MIT AI Laboratory, Cambridge, MA.

SWARTOUT, W.R., "XPLAIN: a system for creating and explaining expert consulting programs," *Artificial Intelligence* **21**, 3, 285–325.

TONG, R.M. (1977): "A control engineering review of fuzzy systems," *Automatica* **13**, 559–569.

TONG, R.M. (1984): "A retrospective view of fuzzy control systems," *Fuzzy Sets and Systems* **14**, 199–210.

TRANKLE, T.L. and L.Z. MARKOSIAN (1985): "An expert system for control system design," *Proc. of IEE Conference Control '85*, Cambridge, UK, pp. 495–499.

TRANKLE, T.L., P. SHEU and U.H. RABIN (1986): "Expert system architecture for control systems design," *Proc. ACC*, Seattle, WA, pp. 1163–1169.

TRANKLE, T.L. (1986): "Personal communication,".

TSYPKIN, Y.Z. (1958): *Theorie der Relais Systeme der automatischen Regelung*, R. Oldenburg, Munich, W. Germany.

TSYPKIN, Y.Z. (1984): *Relay Control Systems*, Cambridge University Press, Cambridge, UK.

TURNER, R. (1984): *Logics for Artificial Intelligence*, Ellis Horwood Limited, Chichester, England.

VAN MELLE, W., A.C. SCOTT, J.S. BENNETT and M. PEAIRS (1981): "The EMYCIN manual," Technical report HPP-81-16, Computer Science Department, Stanford University, California.

VAN VALKENBURG, M.E. (1960): *Introduction to Modern Network Synthesis*, John Wiley & Sons, Inc., New York.

VERE, S.A. (1983): "Planning in time: Windows and durations for activities and goals," *IEEE Transactions on Pattern analysis amd Machine intelligence* **5**, 3, 246–267.

WATERMAN, D.A. (1986): *A Guide to Expert Systems*, Addison-Wesley, Reading, MA.

WEISS, S.M. and C. A. KULIKOWSKI (1981): "Expert consultation systems: The EXPERT and CASNET projects," *Machine Intelligence*, Infotech State of the Art Report, Pergamon Infotech Ltd., Maidenhead Berks, U.K..

WIDROW, B. (1962): "Generalization and information storage in networks of Adaline neurons," in M.C. Yovits, G.T. Jacobi and G.D. Goldstein (Eds.): *Self-Organizing Systems 1962*, Spartan Books, Washington DC.

WILKINS, D. (1984): "Domain independent planning: Representation and plan generation," *Artificial Intelligence* **22**, 269–301.

WINSTON, P.H. and B.K.P. HORN (1981): *Lisp*, 1st edition, Addison-Wesley, Reading, MA.

WINSTON, P.H. and B.K.P. HORN (1984): *Lisp*, Addison-Wesley, Reading, MA.

WITTENMARK, B. and K.J. ÅSTRÖM (1984): "Practical issues in the implementation of self-tuning control," *Automatica* **20**, 5, 595–605.

YARBER, W.H. (1984): "Electromax V Plus, A logical progression," *Proc. Control Expo '84.*

ZADEH, L.A. (1965): "Fuzzy sets," *Inform. Control* **8**, 338–353.

ZIEGLER, J.G. and N.B. NICHOLS (1943): "Optimum settings for automatic controllers," *Trans. ASME* **65**, 433–444.

A

Relay-based control design methods

This appendix contains a survey of different control design techniques based on relay experiments.

The original Ziegler-Nichols method: The Ziegler-Nichols rules, (Ziegler and Nichols, 1943), can be used both for design of P, PI and PID controllers. The recommended parameters are

Controller	K	T_i	T_d
P	$0.5k_c$		
PI	$0.4k_c$	$0.8t_c$	
PID	$0.6k_c$	$0.5t_c$	$0.12t_c$

where k_c and t_c denotes the ultimate gain and ultimate period. The Ziegler-Nichols rules are generally considered to give too small relative damping.

Modified Ziegler-Nichols methods: In Åström and Hägglund (1984b), modified Ziegler-Nichols design techniques are presented. They are based on knowledge of one point on the Nyquist curve. This point can be moved to an arbitrary position in the s-plane using P, I, and D action. This may be used to obtain controllers with a prescribed amplitude margin or phase margin. For example, a PID controller that gives a prescribed phase margin of ϕ_m can be obtained as follows.

$$K = k_c \cos\phi_m$$

$$T_d = \frac{\tan\phi_m + \sqrt{\frac{4}{\alpha} + \tan^2\phi_m}}{2\omega_c}$$

$$T_i = \alpha T_d$$

The choice of α gives an additional degree of freedom.

High-gain PI design: Systems for which the output derivative change sign at the relay switching times may theoretically be controlled arbitrarily well by constant high gain feedback. In practice, control signal saturation and measurement noise limit the maximum allowed feedback gain and thus integral action might be needed. The proportional gain can be chosen as

$$K = \frac{\Delta u}{\Delta n}$$

where

$\Delta n =$ Measured noise variations

$\Delta u =$ Maximum allowable control signal variations

The integration time can be determined by specifying the relative damping of the closed loop system, ζ, under the assumption that the open loop system is only an integrator.

$$G(s) = \frac{k_i}{s}$$

The value k_i may be approximated as

$$k_i = \frac{4\varepsilon}{t_c d}$$

The characteristic equation of the system now looks as

$$s^2 + Kk_i s + \frac{Kk_i}{T_i} = 0$$

This gives

$$T_i = \frac{4\zeta^2}{Kk_i}.$$

Discrete system fitting: The Ziegler-Nichols method and its modifications make only use of the oscillation period and amplitude. And underlying assumptions in these methods is that the process fulfills the describing function conditions and thus have a pure sinusoidal oscillation curve form. As shown in Section 3.1, this is, e.g., not true for simple first order systems.

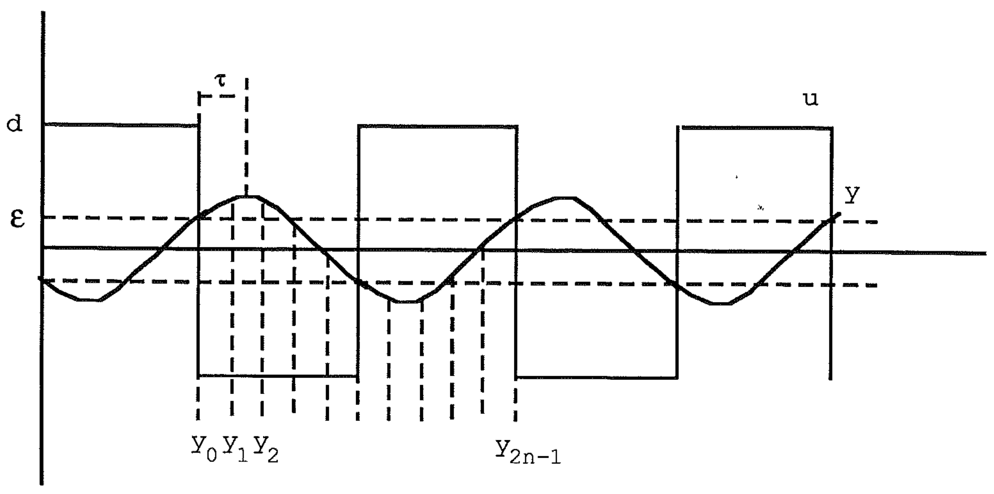

Figure A.1 Discrete model fitting definitions

An alternative method is instead to use the shape of the oscillation curve form as proposed in Åström and Hägglund (1987). The basis for the method is that a relay feedback experiment in stationarity gives periodic input and output signals. $2n$ samplings per oscillation period is chosen as a reasonable sampling rate. By using Z-transform methods, it is shown that the coefficients in the input-output model

$$A(q)y(k) = B(q)u(k) \qquad r = \deg A - \deg B$$

can be determined from the equation

$$A(z)D(z) + z^r B(z)E(z) = z^r(z^n + 1)Q(z) \tag{A.1}$$

where

$$E(z) = d(z^n + z^{n-1} + \ldots + z)$$
$$D(z) = y_r z^n + y_{r+1} z^{n-1} + \ldots + y_{r+n-1} z$$

The $Q(z)$ polynomial corresponds to the initial conditions that give the steady state periodic output. The measurements of the oscillation are defined according to Figure A.1. In the generic case, it is shown that $n \geq 2 \deg A$ is a necessary condition for Equation (A.1) to be solvable. For special cases, however, n may be smaller. This method can principally be used to fit a model of arbitrary complexity to the signal waveform. A first order system with a time delay, $G(s) = k_p e^{-sL}/(1 + sT)$, is a reasonable process model to use. The calculations for this model are contained in Åström and Hägglund (1987) and will be summarized here. Three different cases need to be examined depending on the relation between the sampling period and the time delay. In all three cases, the discrete model contains three coefficients.

$$\begin{aligned} y(kh+h) &= ay(kh) + b_1 u(kh - rh + h) + b_2 u(kh - rh) \\ L &= (r-1)h + L' \end{aligned}$$

From this it follows that $n = 3$ is the smallest allowed value in order to achieve a solution for Equation (A.1).

Equation (A.1) has the following solutions.

$$\begin{aligned} a &= \frac{e_2 - e_3}{e_1 - e_2} \\ b_1 &= \frac{-e_1^2 + e_2^2 + e_3^2 - e_1 e_3 + e_1 e_2 - e_2 e_3}{2d(e_1 - e_2)} \\ b_2 &= \frac{e_1^2 + e_2^2 - e_3^2 - e_1 e_3 - e_1 e_2 + e_2 e_3}{2d(e_1 - e_2)} \\ q_0 &= \frac{e_1^2 + e_2^2 + e_3^2 - e_1 e_3 - e_1 e_2 - e_2 e_3}{2d(e_1 - e_2)} \\ q_1 &= 0 \end{aligned}$$

where e_1, e_2, and e_3 have the following values for different values of r.

$r =$	1	2	3
e_1	y_1	y_2	$-y_0$
e_2	y_2	$-y_0$	$-y_1$
e_3	$-y_0$	$-y_1$	$-y_2$

The equation has a solution provided $e_1 \neq e_2$. This correspond to the case when the waveform is a pure sinusoidal.

The choice of model depends on the actual value of the time-delay which is not known in advance. It can, however, be estimated from the maximum point of the oscillation waveform, τ, defined in Figure A.1. The time delay must be larger than τ. Several relations exist between the coefficients in the different models that can be used for model validation purposes.

The outcome of the model fitting is a discrete process model. From this model, the corresponding continuous model can be calculated.

$$\begin{aligned} T &= -\frac{h}{\log a} \\ L' &= T \log \frac{b_1 + b_2 e^{h/T}}{b_1 + b_2} \\ L &= h(r-1) + L' \\ k_p &= \frac{b_1 + b_2}{1 + e^{-h/T}} \end{aligned}$$

An alternative is to instead base the control design directly on the discrete process model. The coefficients in a RST-controller, $Ru = Ty_{ref} - Sy$, with integral action could, e.g., be computed.

Dominant pole design based on conformal mappings: An alternative design method based on relay feedback is described in Hägglund and Åström (1985). This method uses knowledge of two close points on the Nyquist curve. The points are used to position the dominant poles of the closed loop system.

Many feedback loops have a pole-zero configuration with a pair of complex poles that dominate the transient response. The dominating poles can be estimated from knowledge of the Nyquist curve of the open loop system. The closed loop poles are given by the characteristic equation

$$G(s) + 1 = 0.$$

A Taylor series expansion around $s = i\omega$ and neglecting terms of second and higher order in σ gives

$$1 + G(iw) - i\sigma G'(i\omega) = 0.$$

From this both σ and ω can be determined. The derivate $G'(i\omega)$ can, based on conformal mapping arguments, be approximated by the difference between two close points on the Nyquist curve. The following expression is obtained for determining σ.

$$\frac{G(i\omega_2) - G(i\omega_1)}{\omega_2 - \omega_1} = i\frac{1 + G(i\omega_2)}{\sigma}.$$

With a PID controller the dominant poles can be moved to a desired position. This could be specified in terms of relative damping ζ and frequency ω. Since a PID controller has three adjustable parameters, one extra condition is needed. One possibility is to choose $T_d = \alpha T_i$. The two points on the Nyquist curve close to the ultimate frequency is obtained by relay feedback with different values of the ratio ε/d. The two measurements give a feeling for how fast the amplitude and phase changes around the ultimate frequency. This can e.g. be used to indicate if the dynamics are dominated by a time-delay. In that case the phase decreases substantially more than the amplitude. Hägglund and Åström (1985) contains examples of the design technique.

Dominant pole design through direct calculations: In Åström and Hägglund (1988), the dominant pole design methods are further explored. Here, the open-loop dynamics are assumed to be known. The characteristic equation under PI control becomes

$$1 + (k + \frac{k_i}{s})G(s) = 0.$$

The two dominating poles are specified by their relative damping ζ and natural frequency ω_0. It is shown that this equation is solvable under quite general conditions and that k and k_i are functions of ζ, ω_0, A and B, where

$$\begin{aligned} A &= \operatorname{Re} G(-\sigma + i\omega) \\ B &= \operatorname{Im} G(-\sigma + i\omega) \end{aligned}$$

That an solution exists, does however not imply that the closed loop system is stable or that the chosen poles are dominating. This has to be ensured by other methods. Similar computations can be made for the PD and PID case. This design method requires knowledge of the process dynamics. The discrete model fitting method could, e.g., be used to obtain an approximative process model.

M-circle design: In Åström and Hägglund (1988) a design technique is described that attempts to find a compensator so that the magnitude of the closed loop frequency response has unit gain at low frequencies and a resonance peak M_p which is smaller than a prescribed value. The design method requires knowledge of one point on the Nyquist curve and the derivative of that point. Similar to the first dominating pole design method this can be obtained through relay feedback with different values of the ratio ε/d. Trough PID control this point is moved so that the compensated Nyquist curve is tangential to the M_p circle at that frequency. The value of M_p is typically chosen as $M_p = 1.1 - 1.5$.

B

Step and pulse response based control design methods

This sections contains an overview of different control design methods based on step and pulse response experiments.

Ziegler-Nichols step response method

The Ziegler-Nichols method is perhaps the most well-known tuning method based on step response (Ziegler and Nichols, 1943).

A unit step is entered and the parameters a and b, defined in Figure B.1, are measured. The PID parameters are calculated from the following table.

Controller	K	T_i	T_d
P	$1/a$		
PI	$0.9/a$	$3b$	
PID	$1.2/a$	$2b$	$b/2$

Area calculations

The Ziegler-Nichols method is sensitive to measurement noise since it depends on calculations of the maximum slope tangent of the step response. A less sensitive approach is instead to calculate different areas of the step response. This method was originally proposed in Ohta *et al* (1980). The process model which is treated is the monotone system

$$G(s) = \frac{ke^{-Ls}}{\prod_{j=1}^{n}(1+sT_j)}.$$

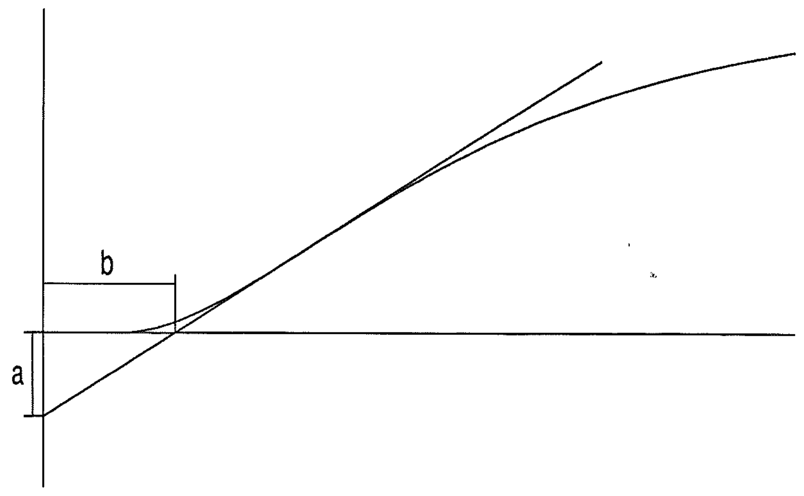

Figure B.1 Ziegler-Nichols step response set-up

A version exists where one pole is allowed to be in the origin. The formulas for $j = 1$ and a unit step input looks as follows.

$$L + T = \frac{A0}{k}$$
$$T = \frac{A1}{k} e^1$$

The areas $A0$ and $A1$ are defined in Figure B.2. The area $A1$ is the area under the step response up to time $T + L$. Notice that in order to calculate $A1$ the values of the entire step response must be saved during the experiment. A pulse with unit amplitude and the width T_p can be used instead of a step. The step response $s(t)$ can be calculated from the pulse response $y(t)$ according to

$$s(t) = \begin{cases} y(t), & 0 \le t \le T_p \\ y(t) + s(t - T_p), & t > T_p \end{cases} .$$

Using a pulse has two advantages. Firstly, the stationary value of the process remains the same after the tuning experiment. Secondly, if the stationary value after the tuning is different from the value before, this may indicate that the process has integral action. If a step was used the response would, in this case, never reach a stationary value.

Similar methods exist for the closed-loop tuning experiment. These typically require the calculation of additional areas and have more complex formulas for calculating the process model. Experimental results on the use of this tuning method are described in Lindqvist (1985).

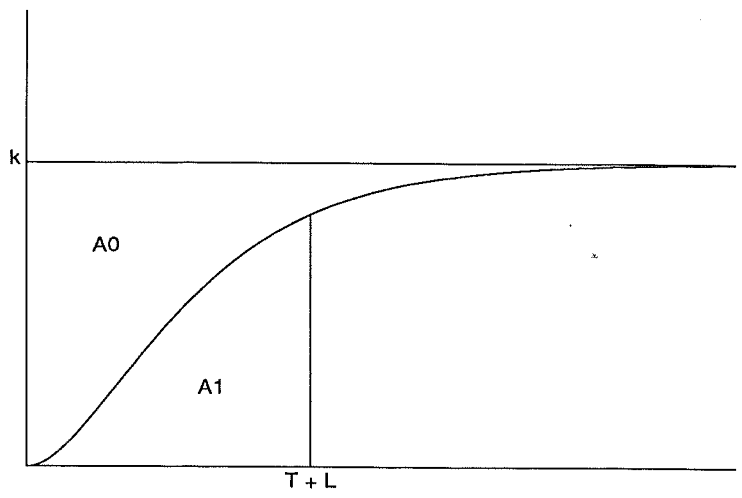

Figure B.2 Area definitions

The method of moments

In Åström and Hägglund (1988), the area calculation methods are generalised. In this method the process model is calculated from the value of the transfer function and its derivatives at $\omega = 0$. For the process model

$$G(s) = \frac{k_p}{1+sT}e^{-sL}$$

it follows that

$$\begin{aligned} G(0) &= k_p \\ G'(0) &= -k_p(T+L) \\ G''(0) &= k_p(2T^2 + 2TL + L^2) \end{aligned}$$

i.e.:

$$\begin{aligned} k_p &= G(0) \\ T + L &= -\frac{G'(0)}{G(0)} \\ T &= \sqrt{\frac{G''(0)}{G(0)} - \left(\frac{G'(0)}{G(0)}\right)^2} \end{aligned}$$

The value of the transfer function and its derivates at $\omega = 0$ can be computed from the following integrals

$$U(0) = \int_0^\infty u(t)dt$$

$$U'(0) = -\int_0^\infty tu(t)dt$$

$$U''(0) = \int_0^\infty t^2u(t)dt$$

$$Y(0) = \int_0^\infty y(t)dt$$

$$Y'(0) = -\int_0^\infty ty(t)dt$$

$$Y''(0) = \int_0^\infty t^2y(t)dt$$

together with the equations

$$\begin{aligned} Y(0) &= G(0)U(0) \\ Y'(0) &= G'(0)U(0) + G(0)U'(0) \\ Y''(0) &= G''(0)U(0) + 2G'(0)U'(0) + G(0)U''(0) \end{aligned}$$

This method can be used with arbitrary input signals.

C

The mailbox interface

Mailboxes are created with the system service **SYS$CREMBX** (Digital Equipment Corp., 1984). This can be done either from VMS or EUNICE. In this project all mailboxes are created during initialization in the knowledge-based system process. The resulting mailbox identifiers are passed to the other subprocesses through the VMS logical name tables. All this is done in a user-written C function.

Franz Lisp allows the inclusion of user-written functions in C, Pascal, and Fortran. These functions may return arbitrary Lisp data objects. The compiled user written functions are dynamically loaded into the Lisp. Using C has one major advantage. Since Franz Lisp is written in C, it is possible to use the Lisp system's internal data types in the user-written functions and thus easily construct valid Lisp objects.

Mailboxes may be accessed from three different layers. The lowest layer is directly through the system services. The system service **SYS$QIO** is used to access a mailbox. The next layer is the VMS Record Management Service (RMS). Here a mailbox is associated with a file. Access is done through get and put operations. RMS internally implements this through system services. The highest layer is to access the mailboxes directly from the high-level languages. Here the mailbox cannot be separated from an ordinary file. All access is done with, in, e.g., the Pascal case, read and write statements.

The data structure used for I/O in Franz Lisp is the port. A port is a stream-oriented I/O channel. The standard input and output port is the terminal, but ports can also be connected to files or pipes. A pipe is a Unix method for stream communication between different processes. The mailbox to Lisp interface is made by connecting the mailboxes to Unix files and then connecting the files to the internal Lisp structure that represents ports.

By default, the read and write primitives of the mailboxes provide synchronous communication, i.e., a process which writes a message in a mailbox is halted until an other process reads the message. This is not sufficient in our case. For example, the time critical numerical algorithm process cannot wait for the knowledge-based system to actually read a message being sent to it in **Inbox**. Asynchronous communication can, however, be achieved by changing a time-out parameter in the VMS Record Access Block (RAB). Each VMS file has an associated RAB that contains information about how the records in the file are accessed.

The numerical algorithms must also have the possibility to check if there are any messages in **Outbox** before it attempts to read them. If not and the mailbox is empty, this process will be halted until a message is sent. The system service `SYS$GETDEV` provides a possibility to check the number of messages in the mailbox. This is used as a check before reading.

The details of the mailbox interface is shown in the following examples. The first example is the C function **crembx** that creates two mailboxes and starts a Pascal program with these mailboxes as standard input and output. The function connects the mailboxes to Franz Lisp ports and returns the ports as a dotted pair.

```
/*  Inclusion of type definition files.
    Franz Lisp's type declarations are contained in
    global.h. */

include </usr/src/cmd/lisp/franz/h/global.h>
include </usr/include/vms/dibdef.h>
include </usr/include/vms/ssdef.h>
include </usr/include/eunice/eunice.h>
include <stdio.h>

/* The crembx function returns an arbitrary Lisp expression. */

lispval
 crembx()

{ struct {int size; char *ptr; } inbox, outbox, imagename,
                                 In_Descr, Out_Descr, logname;
  int Status, pidnr, fd1, fd2;
  short int unitnr1, unitnr2;
  char mbx_name1[9], mbx_name2[9];
  register char *cp1, *cp2;
  register int i, j;
  FILE *inport, *outport;
  lispval list;

define CHECK_STATUS  if(!((Status =
define END_CHECK     ) & 1)) printf("ERROR");
```

```
inbox.ptr = "INBOX";
inbox.size = 5;
outbox.ptr = "OUTBOX";
outbox.size = 6;
imagename.ptr = "NUMERICS";
imagename.size = 8;
logname.ptr = "NUMERICS";
logname.size = 8;

/* Two mailboxes are created. */

fd1 = creat(&inbox,0777,"ipc",256,"tmp",&unitnr1);
fd2 = creat(&outbox,0777,"ipc",256,"tmp",&unitnr2);

/* The mailbox names are created from the unit numbers
   and are entered in the group logical name table.
   The code is only shown for one mailbox. */

i = unitnr1
cp1 = &mbx_name1[9];
*cp1-- = '\0';
*cp1-- = ':';
while (i) {
           *cp1-- = (i % 10) + '0';
           i /= 10;
           }
*cp1-- = 'A'; *cp1-- = 'B'; *cp1-- = 'M'; *cp1 = '_';
In_Descr.ptr = cp1
In_Descr.size = 9;
strcpyn(In_Descr.ptr,cp1,9);

CHECK_STATUS
  sys$crelog(1,&inbox,&In_Descr,0)
END_CHECK

   .
   .

/* The process NUMERICS is started with the mailboxes
   as standard input and output. */

CHECK_STATUS
  sys$creprc(&pidnr,&imagename,&Out_Descr,&In_Descr,&In_Descr,
             0,0,&logname,4,0,0,0)
END_CHECK
```

```
/* The mailboxes are connected to UNIX files and opened. */

inport = fdopen(fd1,"r");
outport = fdopen(fd1,"w");

/* The RAB structure of output is changed to allow
   asynchronous communication. */

FD_FAB_Pointer[fd2]->rab.rab$l_rop |= RAB$M_TMO;

/* The UNIX files are associated with Lisp ports
   and a dotted pair of the ports is returned. */

ioname[PN(inport)] = (lispval) inewstr(In_Descr.ptr);
ioname[PN(outport)] = (lispval) inewstr(Out_Descr.ptr);

list = newdot();
list->d.car = P(inport);
list->d.cdr = P(outport);

return(list);

}
```

The Pascal program is interfaced as shown in the following example.

```
[inherit('SYS$LIBRARY:STARLET')] program numerics(input,output);

type Ptr_to_RAB = ^RAB$TYPE;
     unsafefile = [unsafe] file of char;

var RAB : Ptr_to_RAB;

function PAS$RAB(var f : unsafefile): Ptr_to_RAB; extern;

{* Function MoreInBox returns true if there are any messages
   in the given mailbox. *}

function MoreInBox(name:packed array [integer] of char):boolean;

  var status : integer;
      dbuff : packed array [1..DIB$K_LENGTH] of char;

begin
  status := $GETDEV(DEVNAM := name, PRIBUF := dbuff);
  if not odd(status) then writeln('MoreInBox:',status);
  MoreInBox := ord(dbuff[9]) > 0;
```

```
end;

begin

  {* Change to asynchronous communication. *}

  RAB := PAS$RAB(output);
  RAB^.RAB$V_TMO := true;

  {* Main loop *}

  while true do
  begin
    control;
    if moreinbox('SYS$INPUT') then read_message;
  end;
end.
```

The C written function is compiled separately with the command

```
%cc -c crembx.c
```

This results in an object file that is loaded into Franz Lisp with the command

```
>(cfasl 'crembx.o '_crembx 'crembx "function")
```

A call to crembx will look like

```
>(crembx)
(%_MBA:1256 . %_MBA:1257)
```

The car and cdr of this list are the ports connected to the mailboxes.

D

YAPS commands

This appendix contains an overview of the most important messages and functions available in plain YAPS.

`(use-yaps-db '<database>)`
: Change the currently active database to **<database>**.

`(p <name> <pattern> ... [ test <test> ...] --> <expression> ...)`
: Define the rule **<name>** and install it in the active database.

`(installp '<rule> '<rule> ...)`
`(<- '<database> 'installp '<rule> '<rule> ...)`
: Installs the already defined rules in a database.

`(printp 'p1 'p2 ...)`
: Print the rules whose names are p1, p2 etc.

`(fact <expression> ...)`
: Encapsulates the expressions in a list and adds it to the currently active database. The arguments are by default not evaluated with the exception of pattern matching variables which are substituted by their value. Evaluation can be forced by preceeding an argument with the ^ character.

`(<- '<database> 'fact '<list-of-expressions>)`
: Adds the argument list as a fact in **<database>**. The value of the fact is given by **<expression>**.

`(db)`
`(<- '<database> 'db)`
: Print the values and cycle numbers of of the facts in a database.

`(remove '<number> '<number> ...)`

Removes facts from the current database. May only be used within the RHS of a rule. The numbers correspond to the facts matching the patterns in the LHS of the rules. Negated patterns are not counted.

`(rm '<cycle> '<cycle> ...)`
`(<- '<database> 'rm '<cycle> '<cycle> ...)`

Remove the facts with the given cycle numbers from a database.

`(refresh ['<number> '<number> ...])`

Removes all facts, or only the ones specified, from the current database and adds them again. This causes rules which already have fired to be fired again. May only be used within a RHS.

`(ref ['<cycle> '<cycle> ...])`
`(<- '<database> 'ref ['<cycle> <cycle> ...])`

Like refresh but instead requires the cycle numbers of the facts to be refreshed.

`(run ['<n>])`
`(<- '<database> 'run ['<n>])`

Begins the execution of the rules. If `<n>` is given only n rules are fired. Otherwise it continues until either the conflict set is empty or an explicit halt has been executed.

`(halt)`

The rule firing is stopped when the current rule is finished. May only be used within a RHS.

`(yaps-trace ['<all>])`
`(<- '<database> 'trace)`

Turn tracing on for the specified database.

`(yaps-untrace ['<all>])`
`(<- '<database> 'untrace)`

Turn tracing off.

`(age-only-strategy)`
`(<- '<database> 'age-only-strategy)`

Changes the conflict resolution strategy of a database to age-only-strategy.

`(directed-strategy ['<keyword>])`
`(<- '<database> 'directed-strategy ['<keyword>])`

Change the conflict resolution strategy of a database to give priority to facts which begin `<keyword>`. The default strategy is `(directed-strategy 'goal)`.

E

Planning from an AI perspective

The origin of the work on planning in AI is the General Problem Solver (GPS), e.g., (Newell *et al*, 1960). This was the first problem solving program that separated general problem-solving methods from task-specific knowledge. A task was described as a triple of an initial object (state), a goal object (state) and a set of operators. Operators were chosen on the basis of how much the difference between the initial object and the goal object was reduced. No information was assumed to automatically carry through from one state to the succeeding state. This means that the operator was responsible for generating all information in a succeeding state. The representation format of the states and operators were not predetermined and varied from one domain to another.

Much of the work in state-based planning is based on situation calculus, (McCarthy and Hayes, 1969). The representation format is usually first-order predicate calculus or some extension of it. Resolution is used as the problem solving method. The domain under consideration is assumed to always be in a certain state. A state is described by means of predicates. For example, the fact that an object is at a certain position in certain state can be expressed with the following predicate.

```
at(object1,position6,state7)
```

Events, or actions, are represented as functions that takes a situation, including a state, and returns the resulting state. An axiom that describes that object1 can be pushed from `position6` to `position7` looks as follows.

```
(∀s) [at(object1,position6,s) ⊃
   at(object1,position7,push(object1,position6,position7,s))]
```

The function **push** returns the resulting state. An example of a planning system along these lines is described in Green (1969).

A general problem in planning is the problem of which relations that are affected by an action and which are not. This is referred to as "the frame problem", (Hayes, 1973). Frame here means the frame of reference in which a relation holds. In resolution-based planning systems and also GPS, it is necessary to explicitly state all relations that are left unaltered for each and every actions. For example, in the above scenario with objects at different positions it is, e.g., necessary to have axioms that says pushing an object from one position to another doesn't, hopefully, change the position of other objects. Since most actions have local affect, this leads to numerous trivial so called "frame axioms".

The perhaps most well-known planning system is STRIPS (Fikes and Nilsson, 1971). In STRIPS, each operator has associated a set of preconditions, an add list of clauses, and a delete list of clauses. Applying an operator results in the deletion from the model of all the clauses in the delete list and the addition to the model of all the clauses in the add list. All clauses which are not contained in the add or delete lists are assumed to be unaffected by the operator. This is called "the STRIPS assumption". STRIPS is an example of a nonhierarchical planner. This means that the plan developments consists of one level. The individual pieces of the plan are generated one after another starting at the beginning.

Hierarchical planners generates a hierarchy of plans with with different degrees of details. The highest degree is an abstraction of the plan and the lowest degree is the full detailed plan. An example of a hierarchical planner is AB-STRIPS (Sacerdoti, 1974). Another example is the NOAH system, (Sacerdoti, 1975). NOAH uses procedural nets to represent plans. The procedural nets incorporate both procedural and declarative knowledge.

Problems with interacting subproblems can occur when a problem has conjunctive goals. The order in which the goals are fulfilled are perhaps not specified, but can be critical for a plan to be found. A different problem arise when the conjunctive goals must be fulfilled simultaneously. The majority of the work in planning concerns sequential planning. Planning of parallel activities is a much more difficult problem. The STRIPS assumption also cause problems for planning problems in dynamic environments. The presumption that the world only is changed by the planning agent's actions makes it difficult to handle externally generated events.

Several planning systems have tried to extend the possible class of planning problem beyond what is allowed in a "STRIPS planner". The SIPE system, (Wilkins, 1984), can handle plans with concurrent actions. A plan consists of partially ordered actions. Actions without ordering are considered to be concurrent. Actions that do not share the same resource can be executed in parallel. The DEVISER system, (Vere, 1983), also allows plans with concurrent actions.

External events which are known to occur at a certain future time are allowed. Duration times express how long time actions take. Time windows may be specified for goals, e.g., that a goal should hold between two time points. Deviser models time as nonnegative real numbers where time zero is the time of planning.

An related area is the work on theories of action, i.e., on what constitutes an action. Allen (1984) has developed a temporal logic for reasoning about actions. The driving force behind this works has mainly been problems concerning the meaning of action sentences in natural-language understanding. The temporal logic is based on time intervals rather than time points. The logic is a typed first-order predicate calculus where the types could, e.g., be time intervals, propositions that can hold or not hold during a particular interval, and objects in the domain. Dynamic aspects of the world are captured by the term occurrences. Occurrences are divided into processes, which describes activities not involving a culmination, and events which describes activities that involve a resulting outcome. A similar temporal logic has been developed by McDermott (1982).

The temporal logic of Allen has been extended with two modalities that can be used to support planning problems by Pelavin and Allen (1986). The first modality is the INEV operator. The statement `(INEV i P)` means that at time interval `i`, statement `P` is inevitable, i.e., regardless of what happens after `i`, `P` will be true. Using this operator, the possibility operator, `POS`, can be defined. The second modality is the IFTRIED operator. The statement `(IFTRIED pi P)` means that if plan instance `pi` was to be executed then `P` would be true. The resulting planning system can handle concurrent activities and externally generated events. The frame problem is basically solved through frame axioms. Although this logic system provides a general framework for expressing planning problems it is not obvious how it should be used effectively for practical problems.

A formalism for action structures with partially ordered actions that may occur in parallel has been developed by Sandewall, (Sandewall and Rönnquist, 1986). In this work actions are described with preconditions, postconditions, and prevail conditions. The pre- and postconditions correspond to the delete and add lists of STRIPS. The prevail conditions describe the conditions of the world that must remain while the action is executed.

A more procedural approach to reasoning about actions and planning is taken in the work by Georgeff, (Georgeff and Lansky, 1986). In this work, actions are described by processes that have both a declarative semantics and an operational semantics. The use of processes is motivated by the fact that much expert knowledge is procedural in nature and thus is better represented procedurally than with action sequences.