A Framework for Dynamically Configurable Embedded Controllers

Eker, Johan

1997

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

A Framework for Dynamically Configurable Embedded Controllers

Johan Eker

Department of Automatic Control
Lund Institute of Technology
Lund, October 1997

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

Document name
LICENTIATE THESIS

Date of issue
November 1997

Document Number
ISRN LUTFD2/TFRT--3218--SE

Author(s)
Johan Eker

Supervisor
Karl Johan Åström and Karl Erik Årzén

Sponsoring organisation
Swedish National Board for Industrial and Technical Development (NUTEK)

Title and subtitle
A Framework for Dynamically Configurable Embedded Controllers

Abstract

Embedded control systems are today created using tools that give insufficient support for rapid prototyping and code reuse. New tools for implementation of embedded controllers are necessary. This thesis presents a software framework for implementation of embedded control systems. A new language PAL is presented. PAL is designed to support implementation of control algorithms in particular. A run-time system called Pålsjö is also introduced. Algorithms written in PAL may be executed in the Pålsjö system. The Pålsjö system is designed to allow on-line system configuration and reconfiguration.

Three new software patterns are identified and documented. One pattern describes a way to execute control algorithms on block diagram form. Another pattern deals with the problem of assigning parameters to controllers. A third pattern describes a way to add new objects to a pre-compiled framework.

The thesis contains five case studies in which different aspects of the PAL language and the Pålsjö system are demonstrated.

Key words
Embedded control, Real-time, Framework, Software patterns, Programming languages

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
193

Recipient's notes

Security classification

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

ISSN 0280–5316
ISRN LUTFD2/TFRT–3218–SE

Printed in Sweden by Reprocentralen, Lunds Universitet.
Lund 1997

Contents

Acknowledgments

The work in this thesis has been performed in collaboration with a number of people. My supervisor Professor Karl Johan Åström has been a never ending source of inspiration. His enthusiasm and great knowledge has been invaluable for all parts in the thesis, and I thank him for that. Karl-Erik Årzén, my co-supervisor, guided me through the troublesome process of writing this thesis. His comments and suggestions have been invaluable. The Pålsjö framework has been designed and developed with great help from Anders Blomdell. It has been a very enlightening experience and I have been taught everything I do not know about programming. The work on the hybrid tank controller was done together with Jörgen Malmborg. It has been fun and I am looking forward to the continuation. I am also thankful to Klas Nilsson for taking me on board the embedded systems project.

Finally, I would like to thank my roommate Erik Möllerstedt for always letting me choose music.

1

Introduction

The use of embedded control systems is growing rapidly. Nowadays we can find advanced control systems in consumer products such as washing machines and home stereo equipment. In our everyday life we more and more depend on computers for assistance. If a word processor or the email software crashes it is usually not a big problem, one can just reboot the system and carry on. This is certainly not the case for more critical computer controlled tasks. For example the maneuvering systems in airplanes are more and more relying on computers. It is unlikely that a pilot can press Ctrl-Alt-Delete if the autopilot malfunctions.

There are two important aspects of critical control systems; The control law must be designed to react correctly to all possible input information and the hardware/software must be designed to tolerate faults. There are many types of faults that may occur in embedded systems. There may be control design errors, timing errors, and system design errors. Some very complicated faults may occur due to man-machine interaction . Some notable incidents are the crash of the Swedish JAS-39 Gripen fighter aircraft and the failure of the Ariane rocket [Le Lann, 1996].

Control systems in cars are introduced at different levels. There are low-level tasks such as engine control, anti-locking brake systems, and adaptive suspension control. At higher levels there are cruise control systems and climate control. More advanced tasks are discussed in the automated highway projects, which aim to support the driver in the actual driving of the car. The goal here is to design a control system

which will drive the car itself.

The examples above stress the issue of safe and reliable software. Software that is assigned to handle safety-critical tasks cannot be allowed to fail, it must have ways to handle faults that occur during execution. Such software is called *fault-tolerant*.

In the manufacturing industry there are many automated tasks that are critical. If the computers fail there may be damages to humans, equipment, or the environment. The software and hardware in manufacturing cells need to be upgraded periodically. Since the cost for shutting down a manufacturing cell is high, the upgrade might not be feasible just for changing control algorithms, even if the new algorithm would give better performance and reduce costs in the long run. There is a need for embedded control systems which allow on-line changes, where software modules can be replaced without shutting down the whole system. An on-line upgrade can of course be very dangerous. Old trusted algorithms are replaced by new ones. In order to avoid potentially dangerous situations a system that allows on-line configuration must also provide safety mechanisms so that the controller never fails. This can be done by introducing redundant structures, where several controllers run in parallel and the control signal to be used is determined by a trusted supervision algorithm.

The difficulty to design reusable code modules is a common problem when general purpose languages are used to implement embedded controllers. It is not straightforward to reuse code from one application to another, even though the implementation structures resemble each other a lot. Tools such as code generators may be used in order to speed up development, but usually they have drawbacks, they may give large and inefficient programs.

Many control systems have a similar internal structure. There are functions for user interaction, task management, data logging, network interaction, etc. It would be possible to reuse the code if these functions were implemented in a flexible, modular fashion with well defined interface. Modules for a number of these basic common activities could then be arranged in a framework. The user of such a framework only needs to add the code that is specific for a certain application, i.e., only the control algorithm. The framework should provide management for tasks such as network communication and user interaction.

The PÅLSJÖ environment is an attempt to provide such a frame-

work. The project was outlined in 1994 [Gustavsson, 1994] as a part of the project "Autonomous Control" [Åström, 1993]. The goal was to create a flexible and powerful environment for experiments in automatic control. Initially a C++ class library was proposed, but the library soon became very large and cumbersome to use. Particularly for a control engineer who is not an experienced programmer. A set of C pre-processor macros were created in order to support use of the framework. These worked reasonably well, but to further ease the use of the framework a new language with a compiler was created. The language is called PAL (Pålsjö Algorithmic Language) [Blomdell, 1997] and was designed to support implementation of control algorithms. Control algorithms can often be described as a combination of periodic tasks and finite state machines. PAL supports those types of algorithms, the finite state machine in form of Grafcet [David and Alla, 1992]. Furthermore, the language supports data types such as polynomials and matrices, which are extensively used in control theory.

The Pålsjö system consists of two main parts; a compiler and a framework. The compiler reads algorithms specified in PAL and generates C++ code which fits into the framework. The framework has classes for real-time scheduling, network interface and user interaction. The control algorithm coding is made off-line and the system configuration is made on-line. The system may also be reconfigured on-line without stopping the system.

Outline of the Thesis

This thesis presents the PÅLSJÖ prototyping environment for embedded control systems. The PAL language for describing controllers and the PÅLSJÖ run-time system are presented. Some important implementation issues are discussed in detail. One important goal for the design of the run-time system has been to create an environment which allows controller structures that change with time. Hybrid controllers consisting of a set of sub-controllers, which are switched in and out depending on the working conditions, is an example of such controllers. On-line updating of control algorithms is another example.

This thesis is intended to serve both as a manual to the PÅLSJÖ programming environment and as a thesis demonstrating the engineering principles behind the system. Chapter 2 gives an introduction to embedded controllers and real-time systems. A short discussion on other

prototyping systems is presented. Chapter 3 gives an overview of the PÅLSJÖ environment. The controller description language PAL is presented in Chapter 4, and the PCL configuration language is described in Chapter 5. Chapter 6 presents the internal structure of the system. Some of the used engineering principles are documented as patterns in Chapter 7. Chapter 8 presents a number of case studies. Analysis and design of the control systems are also presented for the inverted pendulum and double tank cases. Some possible future directions are presented in Chapter 9.

1.1 Contributions

The contributions of this thesis are:

- Design and implementation of the PÅLSJÖ software environment for dynamically configurable embedded control systems.
- Three generic software patterns from the design of PÅLSJÖ have been identified and documented
- A novel nonlinear observer for the inverted pendulum has been designed and implemented.
- A new hybrid controller for a double tank system has been designed and implemented.

1.2 Published Papers

This thesis is based on the following papers and reports:

- Johan Eker and Karl Johan Åström (1996): "A Nonlinear Observer for the Inverted Pendulum." In proceedings of the 6th IEEE Conference on Control Applications, Dearborn, Michigan.
- Johan Eker and Anders Blomdell (1996):"A Structured Interactive Approach to Embedded Control", In preprints to the SIRS symposium, Lisbon, Portugal.

- Jörgen Malmborg and Johan Eker (1997): "Hybrid Control of a Double Tank System", In proceedings of the 7th IEEE Conference on Control Applications, Hartford, CT.
- Johan Eker and Anders Blomdell (1997): "Patterns in Embedded Control Systems", Report ISRN LUTFD/2 TFRT- - 7567 - - SE, Department of Automatic Control, Lund Institute of Technology, Sweden.
- Johan Eker and Karl Johan Åström (1995): "A C++ Class for Polynomial Operation", Report ISRN LUTFD/2 TFRT- - 7541 - - SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

The work presented in this thesis has been performed in collaboration with several people. The PÅLSJÖ run-time system has been developed and designed in collaboration with Anders Blomdell. He also wrote the PAL compiler. The work on the inverted pendulum presented in Section 8.2 was done together with Professor Karl Johan Åström. The second case study which deals with the design and implementation of a hybrid tank controller was performed in collaboration with Jörgen Malmborg.

2

Embedded Controllers

An embedded computer system is a system where a computer is directly interfaced with its environment. Computer controlled systems are typical examples. The computer obtains information about the process through sensors and influences the process by actuators. The term embedded comes from the fact that the controller is a part of a larger system, and usually located close to the process. The computing power may also be distributed.

An example that have many of the characteristic features of an embedded system is shown in Figure 2.1, where a computer is used to control a batch reactor. The reactor is used for mixing and heating fluids. Typically when a batch is started the valve is opened and the tank is filled. When the tank level reaches a level $L1$, the heater and the mixer is started. The valve stays open until the upper level $L2$ is reached. When the tank is filled and the content has reached the desired temperature, the tank is emptied and a new batch can be started.

There are three sensors which gives the computer information about the process. Two sensors measure the tank level and one sensor measures the temperature of the fluid. There are also three actuators that the computer uses to control the process: the valve, the heater and the mixer. The controller calculates the control signals based on the sensor readings. The computer must also interact with the human operator, display data and receive commands.

An embedded system can typically be organized as a set of concurrent tasks. In the batch example above there are process oriented task

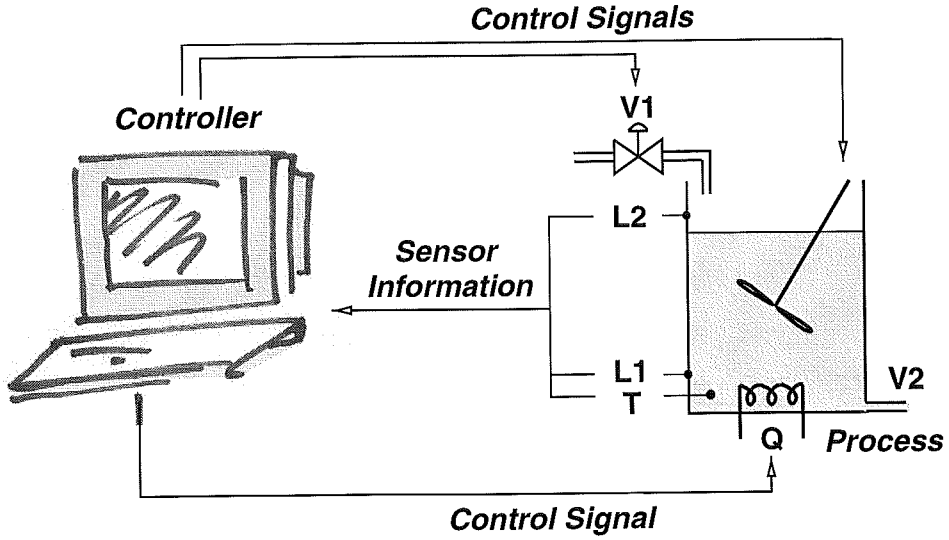


Figure 2.1 A computer controlled batch reactor.

such as heating, mixing and filling. There are also computer oriented tasks such as calculating control signals, logging data and handling the man-machine.

A straightforward way of modeling and programming such systems is by using concurrent languages, in which parallel activities can be expressed explicitly.

Since the computer controls a real physical process it must react to inputs from the process at a certain speed. The system is then a *real-time system*. In [Burns and Wellings, 1997] a real-time system is defined as

> "Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to some movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."

The terms embedded systems and real-time systems are often used interchangeably in the literature.

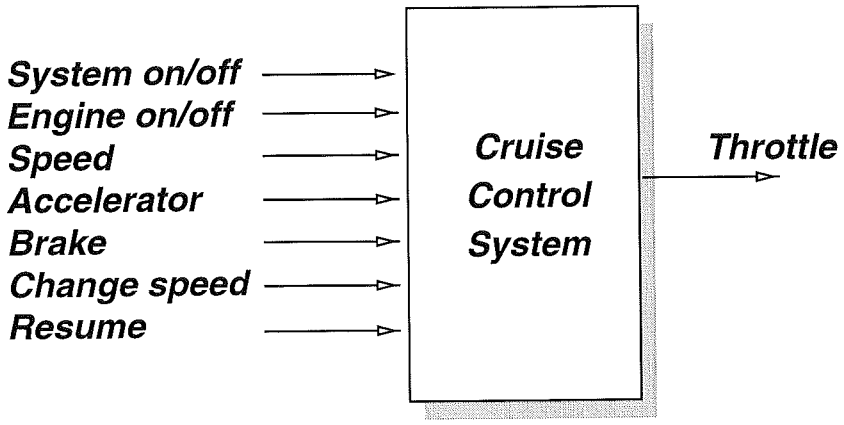


Figure 2.2 The input signals to the cruise controller. This example is taken from [Shaw, 1994]

Real-time

Real-time systems are defined as computational systems where the correctness of the result is dependent on when the result is delivered. For some cases the timing is extremely important, and those systems are labeled *hard real-time systems*. In hard real-time systems responses must occur within the deadline [Burns and Wellings, 1997]. If a deadline is missed the system may fail. In a *soft real-time system* deadlines may be missed occasionally without causing the system to fail. Timing issues are crucial in control system. While time delays may be frustrating in a ticket booking system, they are mission critical for control. Most control systems become unstable with increasing communication delays. Control systems usually consist of tasks with both hard and soft deadlines.

The Cruise Control Problem

A standard example of an embedded system is the car cruise controller. This example is taken from [Shaw, 1994]. The goal here is to build a system for controlling the speed of a car. The controller can measure the speed of the car and uses the throttle to change it. Cruise controllers do normally have access to the brake. The input signals to the cruise control system in Figure 2.2 are:

- System on/off – Indicating if the controller should be running or not.

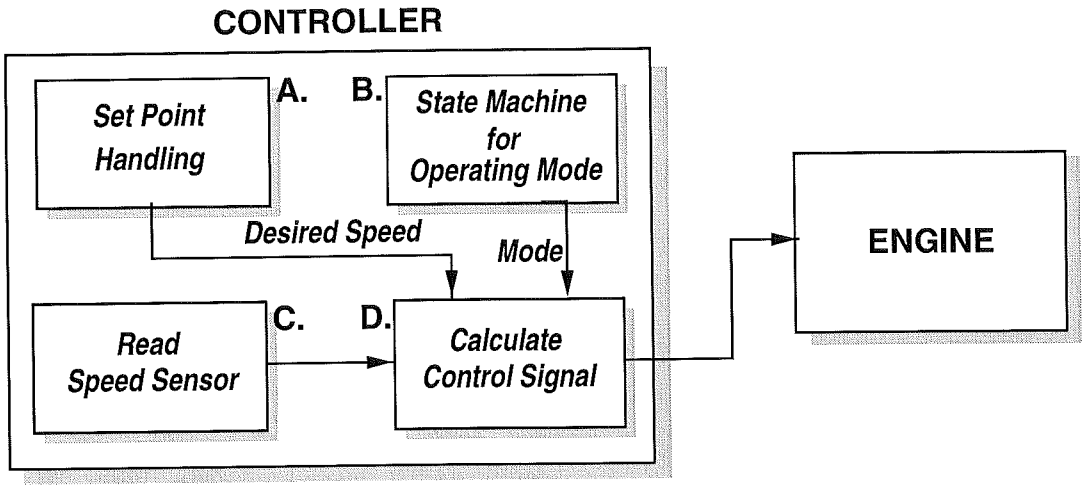


Figure 2.3 The block diagram for the cruise controller.

- Engine on/off – This signal is true if the engine is turned on.
- Speed – The velocity of the vehicle.
- Accelerator – The current position of the accelerator.
- Brake – This signal is true when the brake is pressed
- Change Speed – A signal indicating a change in set point speed
- Resume – Resume the last maintained speed. (increase/decrease).

Cruise control is an interesting problem since it involves ordinary control feedback loops combined with sequential logic. It is a safety critical application, since a failure could have drastic consequences and can not be accepted. Figure 2.4 shows the structure of the system. The controller itself is divided into four different blocks. Block D handles the actual feedback loop. It calculates a new control signal based on the set point, the measured speed, and the internal states of the control algorithm. Block D operates in two modes; on and off. The operating mode is determined by block B, and depends on commands from the user and signals from sensors. First of all the engine must be on in order for the system to be active. Furthermore the user must switch on the system to activate it. If the system is active it is deactivated when the brake is pressed, and is should not be reactivated until the user commands the system to resume. If the accelerator is pressed

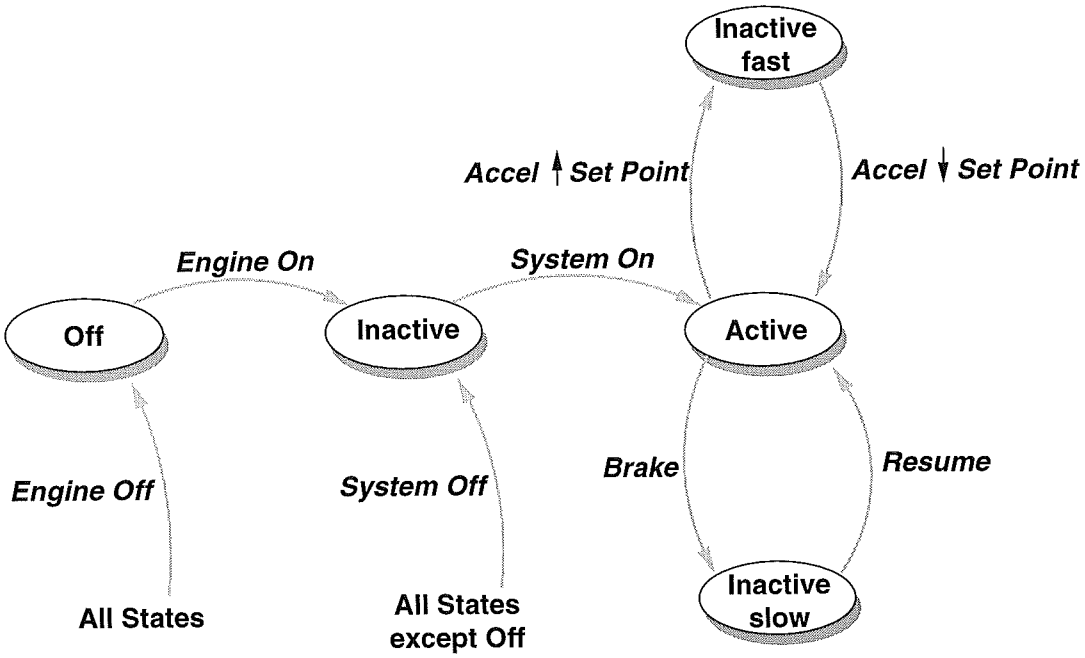


Figure 2.4 State machine for operating mode.

when the system is active, the system will temporarily become inactive for as long as the actual speed is higher than the desired. The state machine for block B is shown in Figure 2.4. Block A handles the set point value. When the controller goes from **inactive** to **active**, i.e. at the event **System On**, the set point is taken as the current speed. The driver may change the set point by requesting increase speed or decrease speed. When the user requests change of speed, the set point is changed by a constant value.

This small example gives a flavor of the different types of components in control systems. Some parts are best described using state machines while others best described using periodic algorithms.

2.1 Programming Embedded Controllers

Embedded controllers are usually large and complex systems, that consist of several concurrent tasks such as low-level feedback control, supervision logic, data logging and user communication. The systems must be efficiently implemented. The execution is constrained by deadlines that should not be missed. This means that many large and com-

plex systems need to be implemented using low-level techniques in order to get the desired performance. Many control applications today are implemented in assembler or low-level languages such as Forth or C. The reason for this is the requirement for fast execution and small programs. Another choice is to use a language with built-in support for concurrency, e.g. Modula-2 or Ada. Those languages provide a higher abstraction level, but will give larger and slower programs. No matter which approach that is used, the implementation of real-time controllers becomes time-consuming and error prone. The languages C and C++ are both general purpose languages. They are not designed to support the implementation of embedded controllers in particular. Usually there is a trade off between performance and higher abstraction levels; the more support the language gives for structured programming, the larger and more inefficient the resulting programs usually become.

General Purpose Languages

The general purpose languages used to implement real-time systems can be divided into a number of categories, the two major ones are

- Concurrent languages, such as ADA or Modula-2
- C/C++ in combination with a real-time kernel or a real-time OS.

Implementing embedded control system with general purpose languages is an error prone and complicated task. The nondeterministic nature of concurrent languages makes it hard to test and verify programs. A program that works in one environment may fail in another. The reason for this is that the resource allocation for each process is done dynamically. If the input stream is changed the execution order of the processes is changed and a new run-time scenario is created.

The implementation of concurrent tasks, and their communication requires firm programmer discipline. An incorrect implementation of a real-time task may cause errors that are very difficult to trace.

Code for embedded control systems implemented with general purpose languages are usually difficult to reuse. One reason for this is the difficulty to separate timing specifications from logical specifications. When timing and logic must be mixed in the source code it is hard to write reusable code.

Several new languages has been proposed for implementation of real-time systems. One interesting language is Erlang [Armstrong *et al.*, 1993], developed by the Swedish telecom industry. It is a functional language with built-in support for concurrency and error recovery. Erlang is aimed for use in soft real-time systems, and it also has mechanisms for on-line code replacement.

Another approach is synchronous languages [Halbwachs, 1993] [Benviniste and Berry, 1991] which have quite recently emerged as a paradigm better suited for designing safety critical real-time systems. At programmer level the synchronous languages use a task model which is similar to ADA and Modula-2, but after compilation the program is completely sequential. This gives programs that are deterministic and possible to analyze. There are currently a number of synchronous languages available, among which Esterel [Bounssinot and de Simone, 1991], Lustre [Halbwacs *et al.*, 1991], and Signal [Halbwachs, 1993] are the most well known.

Related Systems

A different way of generating code has emerged in the control community. Control algorithms usually are simulated and tested before they are implemented. The controller descriptions in the simulation tools are often block based. A controller description then consists of a set of blocks and a signal flow. From this description it is possible to generate real-time code. There are several advantages of working this way. The same code is used for both simulation and implementation. This means that the control algorithm does not have to be rewritten in another programming language, and hence no new programming errors are introduced. Furthermore, the control designer creates the real-time controller without dealing with real-time programming issues. The main drawback with this approach is that the simulation code seldom describes the full operation of the system. Issues such as resource allocation and process scheduling are usually not dealt with. Below a short overview of some of the tools is given.

There are several widely used prototyping systems today. One is Autocode [Int, 1996a] from Integrated Systems, another is Real-time Workshop [Mat, 1997] from MathWorks. Both these system are code generators in combination with a simulation environment. Autocode is used to generate code from SystemBuild [Int, 1996b], which is another

product from Integrated Systems. Here controllers are described with block diagrams. From SystemBuild it is possible to either simulate the system or generate C or Ada code for it. Math Works has a similar approach where controllers are described graphically in Simulink, where it also is possible to either simulate or generate code. In both SystemBuild and Simulink the user has a palette of pre-defined blocks for defining controllers. To create user-defined blocks in SystemBuild the language BlockScript is used, which is a simple block description language with support for basic data types such as floats, integers and booleans. Autocode is then used to generate C or Ada from BlockScript blocks. Autocode uses a template file for customizing the generate code. The template description is made in the template programming language (tpl). This feature makes it possible to generate code that is tailor made for a special system.

So called s-functions are in Simulink used to create user-defined blocks. It is, however, not possible to generate C or Ada code from s-functions using Real-time Workshop. This constraints the programmer to use predefined blocks for code generation. Both Real-time Workshop and Autocode generate static systems, i.e. it is not possible to make any changes to the system without re-compilation. They both lack support for varying control structures.

A similar tool, which generates real-time code from a simulation environment is Sim2DCC [Dahl, 1990], which was developed at the Department of Automatic Control in Lund. Control algorithms are coded and simulated in the Simnon simulation environment [Elmqvist *et al.*, 1990]. The Simnon code may then be translated to Modula-2 code using Sim2DCC. Sequential algorithms are implemented using the graphical GrafEdit interface. Another tool developed at the Department in 1981 is REGULA [Magnusson *et al.*, 1981], which was an interactive environment for implementation of control systems. Control systems were configured by combining a set of computational nodes. The algorithm of a node was interpreted and could be replaced on-line. Another interesting tool for building real-time control applications is ControlShell [Rea, 1995] from Real-time Innovations Inc. ControlShell consists of a C++-class library in combination with graphical editors. A graphical editor is used for creating control applications by connecting blocks. The blocks are implemented in C++, and it is possible for the user to add new block classes. ControlShell is a flexible and ad-

vanced tool which gives the programmer great freedom. It also allows re-configuration with re-compilation. The ControlShell classes are designed for use with the VxWorks real-time kernel.

A commercial system that has some basic ideas in common with the PÅLSJÖ system is SattLine [Johannesson, 1994], from Alfa-Laval Automation. It is a system for programming PLC systems. It consists of a graphical environment for implementing and supervising control applications. It supports sequential algorithms expressed using Grafcet notation, or algebraic algorithms expressed with implicit equations. The SattLine language is based on LICS [Elmqvist, 1985], which also has inspired the work on PÅLSJÖ.

The IEC 1131–3 [Lewis, 1995] programming standard consists of five different ways of representing control algorithms. The five different languages are Structured Text, Function Block Diagram, Ladder Diagram, Instruction List and Sequential Function Charts. Structured Text is a language which resembles Pascal a lot, and has a structure similar to the language used in PÅLSJÖ. Function Block Diagram is a graphical language for specifying data flows through function blocks. Ladder Diagram is another graphical language and is used for expressing ladder logic. The fourth language is Instruction List which is a low-level language similar to assembler. Sequential Function Chart is a graphical language for implementing sequential algorithms. It resembles Grafcet a lot. The textual representation of Sequential Function Chart was used as a starting point when implementing support for sequential algorithms in PÅLSJÖ.

A research project which deals with on-line update of controller code is Simplex [Sha *et al.*, 1995]. Simplex is a proposed architecture for handling real-time replacement of code modules. The focus here is on the problem of handling software faults and algorithm errors that may arise when replacing code. The Simplex architecture suggest mechanisms for detecting faults and switching out incorrect code.

Implementing Control Algorithms

We believe that in order to implement control algorithms in an easy and convenient way, some special support is needed from the language and the software environment. Control laws are usually best described using polynomials or matrices. Support for these data types is a must. Further it is common that an algorithm can be described by both pe-

riodic behavior and sequential logics. A good system should support both these descriptions. Finally there should be an easy way to reuse algorithms from one application to another. One of the problems when reusing real-time code is the fact that code describing the algorithm usually is mixed with code describing the real-time behavior. We believe that in order support reuse there must be a separation between the actual algorithm code and real-time specific information.

2.2 Summary

In this chapter a short introduction to embedded controller and real-time system has been given. Some definitions and two examples were presented. Finally a number of other systems for creating embedded controllers were briefly discussed.

3

An Introduction to the Pålsjö Environment

3.1 Introduction

In this chapter the PÅLSJÖ environment will be introduced. The basic ideas behind the framework are presented, and a number of different design goals are discussed. A small example application is also given.

3.2 A Framework for Real-time Control

In this section the PÅLSJÖ environment is introduced. The design goals and specifications are discussed.

Block diagrams

Block diagrams are used to schematically express the functional entities and their interconnections in a control system. Figure 3.1 shows a block diagram with a control block, a process block and a negative feedback loop. A block diagram basically consists of connections and blocks that produce output values that reflect their internal states and their input values.

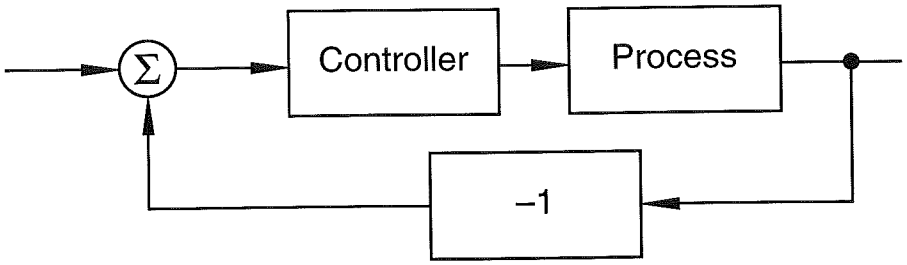


Figure 3.1 Controllers are usually described using block diagram. A block is defined by a set of input and output signals, parameters and states. A block diagram defines how data flow between a set of blocks.

Motivation

The internal structures for many controller implementations are very similar. The same kind of building blocks are used and their internal communication follows certain patterns. A typical embedded controller, see Figure 3.2, contains modules such as

- *OperatorCommunication*, which handles the interaction with the user, i.e. setting and reading parameters.
- *Reference Generator*, which is used to calculate the set point for the controller.
- *Controller*, which implements the control algorithms.
- *Logger*, which handles data for display or analysis.

The main idea with the PÅLSJÖ project is to capture the common behavior of typical control systems. If those common features can be encapsulated in a framework, then a large percentage of the total code that has to be written, can be avoided. Further it is possible to introduce a suitable abstraction level, that will support control algorithms in particular. The goal is to give such a high degree of support so that the programmer can focus only on implementing the actual control algorithms, and have the framework take care of user interaction and network interaction.

Figure 3.3 shows the structure of a PÅLSJÖ application.

In the example shown in Figure 3.2, which was written using a more classical approach, all processes were created by the application programmer. When an application instead is written for the PÅLSJÖ

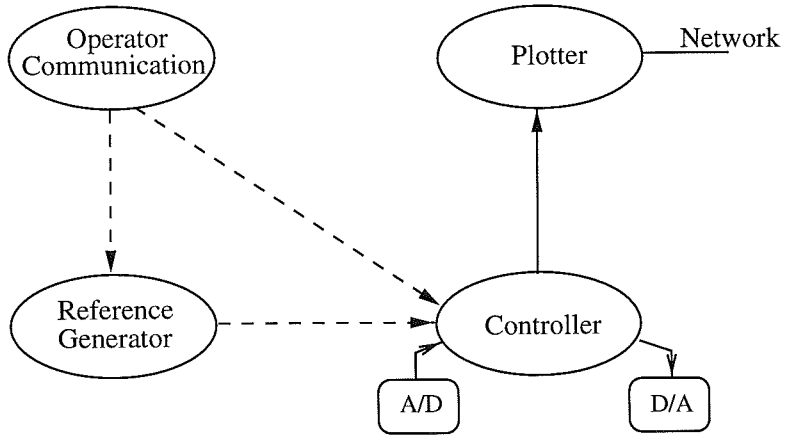


Figure 3.2 A typical control application which here consists of four main modules. *Operator Communication* handles the interaction with the user, i.e. setting and reading parameters. *Reference Generator* is used to calculate the desired position for the controller. *Controller* implements the control algorithms and *Plotter* handles displaying data to the user. The dashed lines mark asynchronous communication while the solid lines mark synchronous communication

system only the processes specific to the application must be handled by the programmer In the PÅLSJÖ run-time system there exist predefined processes for network management and operator communication. In the example in Figure 3.3 there are n processes that represent the actual control application. These processes are initiated by the user through the user interface. Each of these processes consists of a set of algorithmic blocks, which form the algorithms of the processes. In Figure 3.4, a user defined process is shown. Three algorithmic blocks are connected together, and form the algorithm of the process, i.e. each process represents a block-diagram of its own. Each block has a set of functions which are called by the process in order to execute the block. Each time a block is called it calculate new output signals based on its input signal, states, and parameters. Communication between blocks that reside in the same process is synchronous, while communication between different processes is asynchronous. In Figure 3.3 and Figure 3.4 the dashed lines mark asynchronous communication while the solid lines mark synchronous communication. Communication with the plant is of course synchronous while the communication with the user is asynchronous.

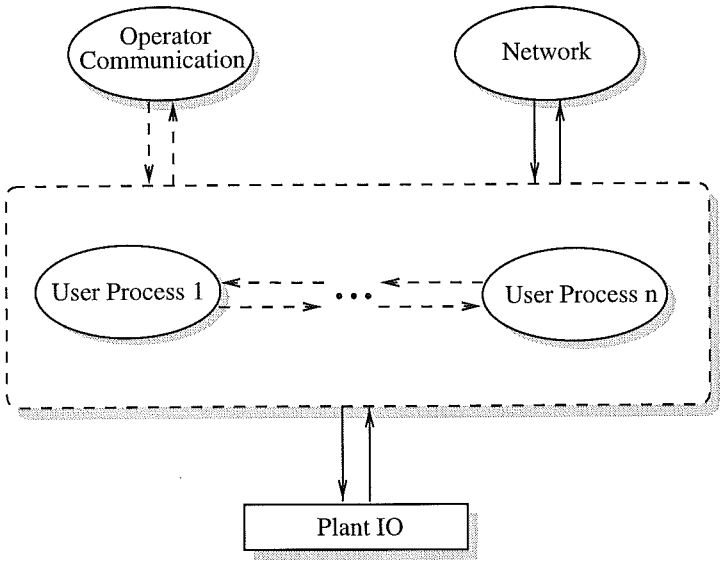


Figure 3.3 A view of the different processes in a PÅLSJÖ application. The processes for network management and operator communication are initiated automatically. The processes marked *User Process 1–n* are user defined process which contains the actual control algorithms. The dashed lines mark asynchronous communication while the solid lines mark synchronous communication.

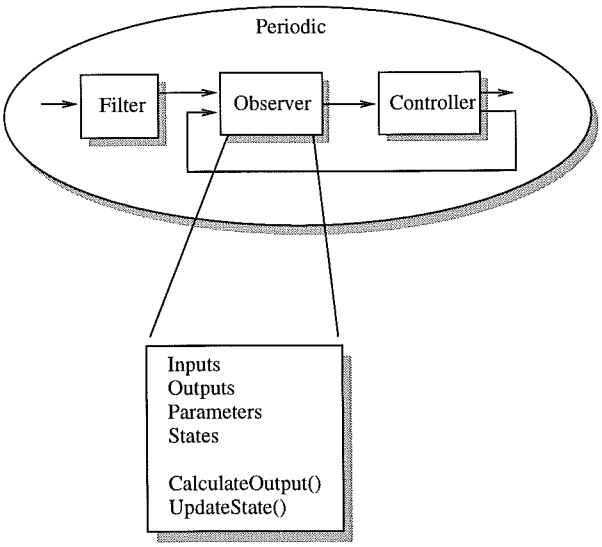


Figure 3.4 The structure of a user defined process in PÅLSJÖ. Several algorithmic blocks are connected and form the algorithm of the process. The blocks are the basic building blocks of PÅLSJÖ. They are either predefined or written by the user.

Requirements

There are a number of important issues that were taken into account when designing the framework. The framework should support

- *Rapid prototyping*
  One of the main reasons to use a framework is to decrease development time. The framework is not intended to be used in the creation of end-user products, but instead as a flexible lab tool.
- *Code Reuse*
  In order to support rapid prototyping there must be good support for reuse of algorithms.
- *Expandable*
  The framework must be expandable so that new features easily can be introduced. For example it should be possible to use datatypes that were not available in the original setup.
- *On-line configurable*
  The system should be configured on-line, and not at compile-time. Changes in running setups should be allowed without stops. One way to handle this could be by interpreting the algorithms, another way would be to use dynamic linking.
- *Portable*
  The framework should be possible to port to a new platform without too many changes to the source code. This means that platform dependent features should be avoided or isolated.
- *Efficient*
  As little run-time overhead as possible should be introduced. An application built using the framework should not differ from an application built from scratch in code size or execution speed.

  For the framework to become really useful it must be efficient and allow fast sampling rates. The timing must also be accurate.

3.3 A Small Example

In this section a small example will be given on how a control algorithm is coded, compiled and executed using the PÅLSJÖ framework. A PI-

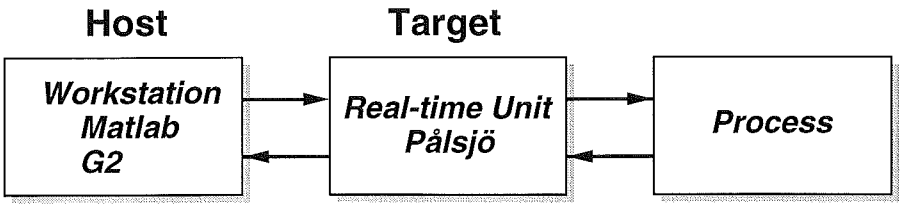


Figure 3.5 The hardware setup, with one host machine running non-real-time software, and a target machine running the control algorithms.

controller is first coded in PAL. This control algorithm is then compiled and linked with the framework. When the run-time system is started, the PI-controller is connected with blocks from standard block libraries, to form a complete control system. The PÅLSJÖ framework is designed to be used in a host-target setup, see Figure 3.5. All hard-real time algorithms are executing on the real-time unit, which typically is a VME-bus or a PC. On the host machine soft deadline algorithms, such as data display, are executed.

Designing the Controller

A common control algorithm is the PI-controller. The control law is described by the following equations, where $e(t)$ is the error and $u(t)$ is the control signal.

$$u(t) = K\left[e(t) + \frac{1}{T_i}\int^t e(s)ds\right] = P + I \tag{3.1}$$

Before it can be implemented, the algorithm must be discretized. The proportional term P in Equation(3.1) is replaced by

$$P(t_k) = Ke(t_k) \tag{3.2}$$

and the integral part is replaced by the following recursive expression which is extended with a tracking term for handling actuator saturations [Åström and Hägglund, 1995].

$$I(t_{k+1}) = I(t_k) + \frac{Kh}{T_i}e(t_k) + \frac{h}{T_r}(u - v) \tag{3.3}$$

Coding the Controller

The execution cycle for the PI-controller can be divided into two parts. One that reads the input signals and calculates a new output signal and another one that updates the internal states of the controller. The pseudo-code for this look like this:

```
loop
   read input signals
   calculate output signals
   write output signal
   update internal state
end loop
```

When writing this in PAL only the calculate output section and the update state section need to be included. All writing and reading of input and output signals is automatically taken care of. In this PI-controller there are two input signals y and y_r, one output signal u and two states e and I. There are also two parameters K and T_i. Below is an example of how this PI-controller is coded in PAL. The block is basically built in three sections; first all signals and parameters are declared, then the output signal is calculated and finally the state is updated. For a more in-depth discussion on why the block algorithm is divided see Section 7.1.

EXAMPLE 3.1

module MyBlocks;

block PI

r, y, u : **input real**;
$v := 0.0$: **output real**;
$I := 0.0, e := 0.0$: **real**;
$K := 0.5, Ti := 10000.0, Tr := 10000.0$: **parameter real**;
h : **sampling interval**;
$bi = K * h/Ti$;
$br = h/Tr$;

calculate
begin

```
    e := r − y;
    v := K ∗ e + I;
  end calculate;

  update
  begin
    I := I + bi ∗ e + br ∗ (u − v);
  end update;

 end PI;

end Controller.
```

□

The parameters K and T_i and the sampling interval h are given values at run-time. It is possible to give default values to both parameters and signals, as is done in the example above for the state variable I. The language PAL will be described in detail in Chapter 4. The PI-controller is compiled with the following command:

```
pal -palsjo MyBlocks.pal
```

The module name and the filename must always be the same. The output from the PAL compiler is the following two files: MyBlocks.palsjo.c and MyBlocks.palsjo.h. These files are then compiled with C compiler and linked with the PÅLSJÖ run-time system.

An easier way to compiled and link a PÅLSJÖ application is to use the makefile that comes with the system. The following command is then used:

```
make PAL=MyBlocks
```

After compilation the next step is to start the PÅLSJÖ run-time environment, where instances of blocks can be created and connected together. To configure and manage blocks in Pålsjö a language called PCL is used. PCL stands for Pålsjö Command Language and will be explained in Chapter 5. The run-time system also supports the export of signals using the network. Signals can be received either in a dedicated plot utility or in MATLAB.

Running the controller

The run-time system works like a shell where PAL-blocks can be allocated, configured and executed. When starting Pålsjö the following interface will appear on the screen.

```
Including module 'built-in'
Including module 'MyBlocks'
                          P Å L S J Ö

          Copyright  1995 Department of Automatic Control
                  Lund Institute of Technology
                           version 1.0


pcl>
```

Blocks for analog input-output and reference value are needed. They are available from the library `StandardBlocks`, see Appendix A.

EXAMPLE 3.2

```
pcl> use MyBlocks
pcl> use StandardBlocks
pcl>{
pcl*> process = new Periodic
pcl*> process.adin = new AnalogIn
pcl*> process.refgen = new RefGen
pcl*> process.control = new PI
pcl*> process.daout = new AnalogOut
pcl*>
pcl*> process.adin.out -> process.control.y
pcl*> process.refgen.out -> process.control.yr
pcl*> process.control.u -> process.daout.in
pcl*>
pcl*> process.tsamp = 0.010
pcl*> process.control.K = 2
pcl*> process.control.Ti = 0.5
pcl*> }
```

```
--> Tree checked and seems to be OK.
Message from 'process' : clearing blocklist...
--> New blocks are now ready to run.
--> Swapping...
Message from 'process' : Analyzing block sequence...
--> Waiting for old processes to terminate ...
pcl>
pcl> process ! start
pcl>
```

□

In the example above, edit mode is started by typing a curly bracket. An instance of the block type `Periodic` is created and given the name `process`. `Periodic` is a built-in block which is used to manage the execution of other blocks. It executes its child blocks in a sequence determined by the data flow between the blocks. After `process` is created, all the other blocks are created as its children. The block type `AnalogIn` has an output signal called `out` which is connected to the input signal `y` of the control block with the connect operator `->`. When all the blocks are connected, parameters are given values. First the sampling period of `process` is set to 10 ms. The two parameters in the PI-algorithm are assigned values. The operation is then closed by a curly bracket. The process is then started with the command

```
process ! start
```

This means that the message `start` is sent to the block `process`. When the controller is started the operator may want to monitor some of the variables in the system. This is simply done by giving the `show` command, as follows

```
pcl> show process.control.u
pcl> show process.control.I
```

The variables `u` and `I` of the PI-controller are now logged and available for export. The next step is to setup up a network connection for transmitting the values of the variables. On the host machine software for receiving data must be started. On the target machine a network connection is created by simply typing

```
pcl>process ! connect
```

This command sends the message `connect` to the block `process` which then tries to establish a connection.

3.4 Summary

In this chapter the PÅLSJÖ framework and the PAL-language were introduced. The major issues for the framework design were presented. Finally a small example was given.

4

PAL – Pålsjö Algorithmic Language

4.1 Introduction

PAL is a block based imperative language for implementation of embedded controllers. It is designed to support and simplify the coding of control algorithms. Language constructs for both periodic algorithms and sequential algorithms are introduced. Furthermore complex data types such as polynomials and matrixes are fully supported.

In this chapter a brief introduction will be given to the PAL language. For a more in depth description see [Blomdell, 1997], which this text is based upon.

4.2 Blocks and Modules

Algorithms are described as input-output blocks. New values for the output signals are calculated based on the values of the inputs, the internals states and the parameters. A PAL block has the following structure:

EXAMPLE 4.1
module example1;

```
block block1
  in : input real;
  out : output real;
  p : parameter real;
  i := 1.0 : real;

  calculate
  begin
    out := i + in;
  end calculate;

  update
  begin
    i := i * p;
  end update;

end block1;
end example1.
```

□

The top level concept of PAL is the module. Several blocks may be grouped together in a module. In the example above the block **block1** is declared within the module **example1**. The first section in the block defines the interface. In this example it consists of one input signal *in*, one output signal *out*, and one parameter *p*. Furthermore there are one state variable *i*. The syntax for defining variables is:

```
name : [interface modifier] data type;
```

The possible data types and interface modifiers are described below. After the interface section comes the algorithm section. The two sections **calculate** and **update**, define the periodic algorithm of the block.

4.3 Interface Modifiers

The interface modifiers define how the variable interacts with the environment, i.e. other blocks and the user.

Input

The input modifier is used in variable declarations and in procedure heads, indicating that the variable is an input. An input variable may not be assigned a value.

Output

The output modifier is used in variable declarations and in procedure heads, indicating that the variable is an output.

Parameter

The parameter modifier is used in variable declarations. A parameter variable may not be assigned a value. Parameters can only be set by the user or the run-time system. There are two types of parameters: direct and indirect. A direct parameter is declared with data type and interface modifier. An indirect parameter is a function of other parameters, and may not be assigned directly in the PAL code. For indirect parameters the data type is given implicitly from the relation with other direct parameters. In Example 4.2 the indirect parameter c is declared as a function of the direct parameters a and b. The data type of the c parameter will be real. Notice that indirect parameters are declared using equality.

EXAMPLE 4.2

a : **parameter real**;
b : **parameter real**;
$c = a * b$;

□

State

The state modifier indicates that the variable is a state. A state variable is visible to the user and the run-time system, but can only be assigned within the block,i.e in a PAL statement. If the interface modifier is omitted, state is used as default.

4.4 Data Types

In this section the available data types in PAL are presented with name and supported operations. Examples are also given.

Scalar types

Real

A real valued variable.

- r : **real**;
- Operators: +, −, *, /.

Integer

A integer valued variable.

- i : **integer**;
- Operators: +, −, *, **div**, **mod**.

String

A text string.

- str = "initial value" : **string**;
- String concatenation available using +.

Dimension

An integer variable used to specify the dimension for aggregate data types.

- n : **dimension**;
- A dimension variable can be used as an integer in expressions and statements. It gets its value upon block creation, and may not be assigned within the algorithm section. The value of a dimension variable can only be set by the runtime system. A dimension variable may be given a default value.

Boolean

A boolean valued variable being either true or false.

- bool := true : boolean;
- Predefined constants: **true** or **false**.
- Three logical operators; **and**, **or** and **not**.

Sampling Interval

The period time for the block given in seconds. The sampling time is set by the user or the run-time system. The sampling time is real valued.

Aggregate types

Array

An array is a fixed sequence of elements of some scalar type. The size of the array is given, as the upper and the lower limits, when defining the array instance. The array size cannot be changed from within the PAL code. It may however be parameterized using dimension variables.

- n : dimension;
  in1 : **input array**[1..n] of **real**;
- m : dimension;
  in2 : **array**[0..m] of **input real**;
- Accessing elements:
  in1[1] := 3.14;
  tmp := in1[2];

Matrix

A matrix is a fixed size two-dimensional array of real valued elements. The size of the matrix is given as the upper and the lower limits, when declaring the matrix instance. Dimension parameters may be used to parameterize the size of a matrix instance.

- out : **input matrix**[1..n,1..m] of **real**;

- Accessing elements:
  out[2,2] := 3.14;
  tmp := out[3,3];
- Operators: $+, -, *$: applied to matrix operands.
  $*$: applied to a scalar real and matrix.
- The matrix data type maps down on the Newmat [Davis, 1997] matrix class. The Newmat function library may thus be used for matrix manipulation.

Polynomial

A polynomial is a fixed one-dimensional array of real valued elements. The degree of the polynomial is given when declaring the polynomial instance. The degree of a polynomial may be parameterized using dimension parameters. A polynomial with degree n has $n+1$ coefficients, starting with index 0. The internal representation of a polynomial is a vector where the coefficient of the highest power is stored at the first position, i.e. position zero. A polynomial

$$A = a_0 z^n + a_1 z^{n-1} + \cdots + a_n$$

will thus be represented in the following way

$$A = [a_0, a_1, \cdots, a_n]$$

- par : **parameter polynomial**[n] of **real**;
- Accessing elements:
  par[2] := 3.14;
  tmp := par[3];
- Operators: $+, -, *, div, mod$: applied to polynomial operands.
  $*, /$: applied to a scalar real and a polynomial.
- The polynomial data type maps down on an external polynomial package [Eker and Åström, 1995]. Functions from this library, which is described in Appendix B are available for polynomial manipulation.

4.5 Expressions and Statements

An expression yields a value. The value domain of an expression is determined by the operation and the operands. Statements are used to describe the execution of an algorithm. A statement may consist of expressions, declarations, and statements. Below is the expressions and statements available in PAL.

Unary Expressions

A unary expression is an expression which only involves one operand. There a three types of unary expressions in PAL:

- Referencing a variable
- Arithmetic negation (e.g. -pi)
- Logical negation (e.g. **not** bad)

Binary Expressions

A binary expression is an expression which involves two operands. There a three types of binary expressions in PAL:

- Arithmetic expressions (i.e. $+, -, *, /$,**mod** or **div**)
- Relational expressions (i.e. $<, <=, <>, >=,$ or $>$)
- Polynomial evaluation, that returns the value of the polynomial in a specific point.

Other Expressions

In PAL there are two other types of expressions:

- Indexing (e.g. $x[i]$), that returns a specific part of an array, matrix, or polynomial.
- Function calls. A function reads input parameters and returns a value.

Statements

Below the available statements in PAL are presented. First the simple statements, assignment and procedure call, are presented, and after that more complex statements **if** and **for** statements are discussed.

Assignment

A variable is assigned using the assignment operator :=.

```
in : input real;
out : output real;
K : parameter real;
...
out := K * in;
```

Procedure call

A procedure call consists of a statement that has the name of the procedure.

```
P();
```

The If statement

Conditional execution is in PAL constructed using the **if** statement.

```
if a = true then
    i := i + 1;
elsif b = true
    i := i + 2;
else
    i := i + 3;
end if;
```

The For statement

In PAL it is possible to repeat one or more statements using the **for** statement.

```
for i = 1 to N do
    sum := vec[i];
end for;
```

4.6 Procedures, Functions and Events

Procedures

A procedure is a subprogram which consists of a sequence of statements. A procedure has a head and a body. The head defines the name

of the procedure, and the interface used for calling the procedure. The body of the procedure contains the statements that are executed when the procedure is called. Parameters to a procedure can either be input or output. Input parameter are passed by value, while output parameters are passed by reference. Parameters of type array are however always passed by reference. It is not allowed to assign values to input parameters inside the function body. In Example 4.3 the implementation of a function which performs dyadic decomposition is shown. The algorithm is taken from [Åström and Wittenmark, 1995]. Three functions *RowAsRealArray*, *RealArrayAsRow*, and *DyadicReduction* are called in the code, but not defined there. The first two are built-in functions for converting data from matrices to an array of reals, and back. The definition of the third function is omitted for simplicity.

EXAMPLE 4.3

```
procedure LDFilter(
  theta : output array [0..n] of real;
  d : output array [0..n] of real;
  l : output matrix [0..n, 0..n] of real;
  phi : array [0..n] of real;
  lambda : input real
);
  i, j : integer;
  e, w : real;
begin
  d[0] := lambda;
  e := phi[0];
  for i := 1 to n do
    e := e − theta[i] ∗ phi[i];
    w := phi[i];
    for j := i + 1 to n do
      w := w + phi[j] ∗ l[i, j];
    end for;
    l[0, i] := 0.0;
    l[i, 0] := w;
  end for;
  for i := n downto 1 do
```

```
    RowAsRealArray(l, 0, tmp1);
    RowAsRealArray(l, i, tmp2);
    DyadicReduction(tmp1, tmp2, d[0], d[i], 0, i, n);
    RealArrayAsRow(tmp1, 0, l);
    RealArrayAsRow(tmp2, i, l);
  end for;
  for i := 1 to n do
    theta[i] := theta[i] + l[0, i] * e;
    d[i] := d[i]/lambda;
  end for;
end LDFilter;
```

□

Functions

A function is a subprogram which consists of a sequence of statements. A value is returned after finishing execution. A function has a head and a body. The head defines the name of the function, the interface used for calling the function, and the type of the return data. Parameters are passed as inputs or outputs in the same fashion as for procedures above.

```
function Limiter(
  max : real;
  min : real;
  value : real
) : real;
begin
  if value < min then
    result := min;
  elsif value > max then
    result := max;
  else
    result := value;
  end if;
end Limiter;
```

Events

All procedures in PAL when executed in the PÅLSJÖ environment are registered as events. Let the procedure Reset below be defined in the block RST.

```
procedure Reset();
  i : integer;
begin
  for i := 0 to m do
    U[i] := 0.0;
    Y[i] := 0.0;
    Uc[i] := 0.0;
  end for;
end Reset;
```

The Reset procedure may then be called from the **PCL** command line by the following command

```
pcl*> process.block = new RST
pcl*> process.block ! Reset
```

Calculate and Update

There are two predefined block procedures **calculate**, and **update**. They are used to implement periodic algorithms. The **calculate** calculates an output signal, while **update** updates the state variables. A PI controller divided like this is found in Example 3.1. The reason for dividing the algorithm like this is discussed in detail in Section 7.1.

Local and Global Variables

Variables declared in the block scope, are global at the block level. They are visible for all procedures and functions in the block. Global variables are also visible from outside the block, i.e. it is possible to monitor and log those variables. Local variables are declared before the **begin** statement in procedures and functions. They are only visible inside the scope of the function or the procedure, and may not be monitored or logged.

4.7 Grafcet

Grafcet [David and Alla, 1992] is a convenient and powerful way of implementing sequential algorithms. Consider the boiler process in Figure 2.1. The Grafcet for this process is found in Figure 4.1. There exist constructs in PAL for expressing Grafcet algorithms. These constructs are steps, actions, and transitions. Grafcet statements are expressed in PAL with a textual representation, that is based on Sequential Function Charts in IEC-1131 [Lewis, 1995].

Steps

Each state in a sequential algorithm is defined as a step in Grafcet. A state may be active or inactive. Several states may be active at the same time. A Grafcet must have an initial step. The syntax for defining a step is illustrated below.

```
initial step Init;
  pulse activate CloseV2;
end Init;

step StartHeating;
  activate Heat;
end StartHeating;

step StopHeating;
  pulse activate NoHeat;
end StopHeating;
```

Actions

To each step an action can be attached. The action contains the statements to be executed while the step is active.

```
action OpenV1;
begin
  V1 := true;
end OpenV1;

action CloseV1;
begin
  V1 := false;
```

end CloseV1;

An action can be activated in a number of different ways:

- **activate** ⟨*Action*⟩ – the action is executed as long as the step is active.
- **pulse activate** ⟨*Action*⟩ – the action is executed one time only, when the step gets active.
- **limit** ⟨*Expression*⟩ **activate** ⟨*Action*⟩ : the action is executed while the step is active, or until the time limit expires.
- **delay** ⟨*Expression*⟩ **activate** ⟨*Action*⟩ – the execution of the action is delayed until the specified time has elapsed. The action must still be active in order for the action to be executed.
- **store activate** ⟨*Action*⟩ – the action will be executed until stated otherwise.
- **reset** ⟨*Action*⟩ – an action that has been stored previously is reset.
- **store limit** ⟨*Expression*⟩ **activate** ⟨*Action*⟩ – the action will be executed until stated otherwise, or the time limit expires.
- **store delay** ⟨*Expression*⟩ **activate** ⟨*Action*⟩ – the execution of the action is delayed with the specified time expression, and will the continue until reset explicitly.
- **delay** ⟨*Expression*⟩ **store activate** ⟨*Action*⟩ – if the step is still active after the specified time the action is stored.

Transitions

A transition has a set of input steps, a set of output steps and a condition. All input steps must be active and the condition must be true for the transition to be fireable. When a transition is fired all output steps become active.

transition from $Init$ **to** $StartHeating$, $StartFilling$ **when** $Start$;

transition from $StartHeating$ **to** $StopHeating$ **when** $T >= Tref$;

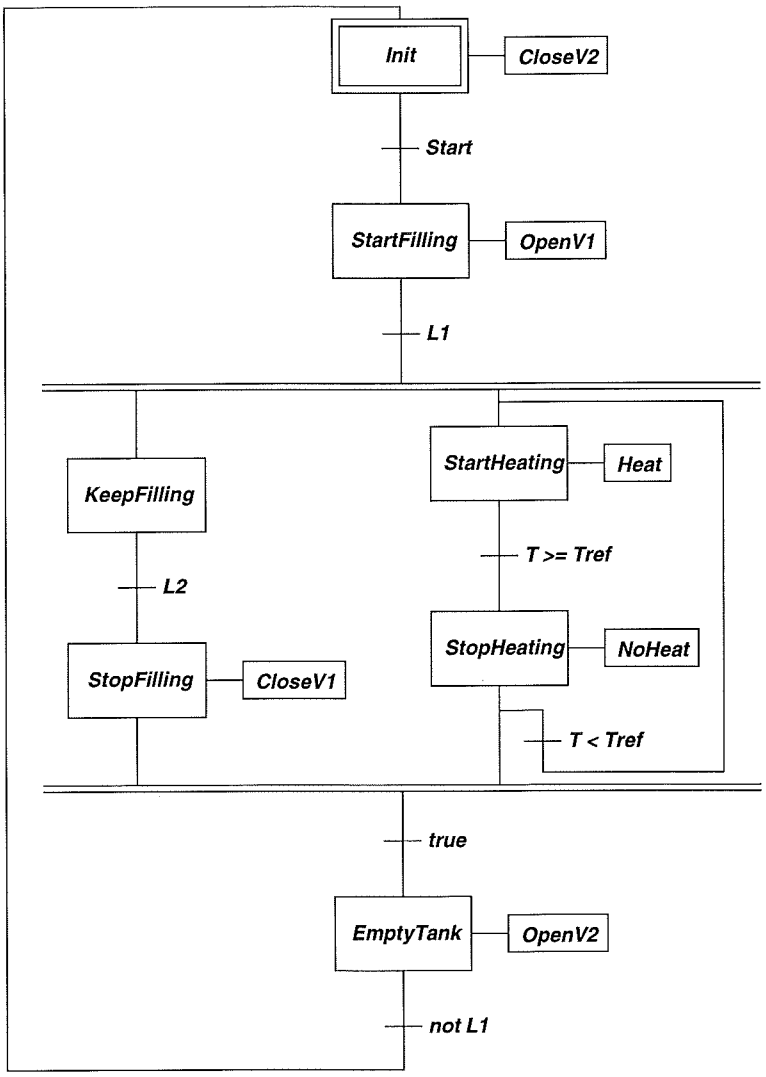


Figure 4.1 Grafcet for the controller of the boiler process. This figure is generated by the PAL compiler using the following command: `pal -fig grafcet.pal`.

An Example

The PAL code for the Grafcet in Figure 4.1 is the following.

module grafcet;

block boiler

$L1, L2, Start$: **input boolean**;
T : **input real**;
$Tref$: **parameter real**;
$V1, V2, Q$: **output boolean**;

initial step Init;
  pulse activate CloseV2;
end Init;

step StartHeating;
  activate Heat;
end StartHeating;

step StopHeating;
  pulse activate NoHeat;
end StopHeating;

step StartFilling;
  pulse activate OpenV1;
end StartFilling;

step KeepFilling;
end KeepFilling;

step StopFilling;
  pulse activate CloseV1;
end StopFilling;

step EmptyTank;
  pulse activate OpenV2;
end EmptyTank;

transition from *Init* **to** *StartFilling* **when** *Start*;

transition from *StartFilling* **to** *KeepFilling*, *StartHeating*
  when *L*1;

transition from *StartHeating* **to** *StopHeating* **when** $T >= Tref$;

transition from *StopHeating* **to** *StartHeating* **when** $T < Tref$;

transition from *KeepFilling* **to** *StopFilling* **when** *L*2;

transition from *StopFilling*,*StopHeating* **to** *EmptyTank*
  when true;

transition from *EmptyTank* **to** *Init* **when not** *L*1;

action OpenV1;
begin
  $V1 :=$ **false**;

```
    end OpenV2;

    action CloseV1;
    begin
      V1 := false;
    end CloseV1;

    action OpenV2;
    begin
      V2 := false;
    end OpenV2;

    action CloseV2;
    begin
      V2 := false;
    end CloseV2;

    action Heat;
    begin
      Q := L1;
    end Heat;

    action NoHeat;
    begin
      Q := false;
    end NoHeat;

  end boiler;

end grafcet.
```

Mixed Controller Structures

A block algorithm may be described using the **calculate** and the **update** procedures in combination with one or several Grafcets. This is useful when one part of the algorithm is periodic, and another is sequential. An example is a system where the controller is switching between different modes but the observer is not. All the controller modes are using the values from the observer. The observer is implemented in the **calculate** and the **update** sections, while the controller is implemented using Grafcet.

Execution Semantics

The different functions and procedures of a PAL block are at run-time executed in a very well defined way. The block is executed periodically, in two phases. The first phase is used to calculate a new output signal, while the second one is for updating states. At each sample when the block is activated the *eventhandler* is executed. All events that have arrived since the last activation time are handled. The next thing that is done is checking if the parameters have changed, and if so start using the new parameter set. Next the calculate procedure is run, and finally the actions belonging to active Grafcet steps are executed. Now the output signal from the blocks is made available to connected blocks.

The first thing that is done in the second phase is to update the states by running the **update** procedure. After that the Grafcets are updated, steps are activated or deactivated. The different steps of the algorithm are the following:

Phase one:

EventHandler – Execute the events that have been called by the user, since previous sample.

GetUpdatedSet – If the parameters have changed, then switch to the new parameter set.

CalculateOutput – Execute the **calculate** section.

RunGrafcet – Execute all active Grafcet actions.

Phase two:

UpdateState – Execute the **update** section.

CheckParameterInput – If any of the parameters gets its value from the output signal of another block, this must be dealt with in a special way, see Section 6.6 for more details on output–to–parameter connections.

UpdateGrafcet – Update the Grafcet, by activating and deactivating steps and actions, when transitions are fired.

5

PCL– Pålsjö Configuration Language

5.1 Introduction

User defined blocks for the PÅLSJÖ environment are programmed in PAL and instantiated and connected on-line using PCL, the PÅLSJÖ Configuration Language. PCL is a simple language for administrating blocks and assigning variables. In this chapter all keywords and operators in PCL will be presented. Examples are given to illustrate the use of every command.

5.2 Keywords

The predefined keywords in PCL are the following;

`new`	`delete`	`with`	`help`
`reset`	`quit`	`use`	`enduse`
`dim`	`show`	`hide`	`break`

new

An instance of a specified block type is created using the `new` statement. The following syntax is used

```
pcl> <block name> = new <block type name>
```

The `<block name>` must contain the full block name. Blocks in PÅLSJÖ may be organized in a hierarchical fashion, and by the full block name all the blocks above in the hierarchy is meant. In the example below the full name for the second block is `S.Input`.

```
pcl*>S = new Periodic
pcl*>S.Input = new AnalogIn
```

`Periodic` is a predefined block type. There are three predefined block types, `Periodic`, `Aperiodic` and `RemoteIO`. All other block types must be imported from either block libraries supplied by the user at linkage time or from the Pålsjö standard library, see the `use` statement below.

delete

A block is deleted using the `delete` command.

```
pcl> delete <block name>
```

with

To avoid repeating the block path name, the `with` command is available. It works similarly to the with statement in Modula-2 and Pascal.

EXAMPLE 5.1

```
pcl> use MyBlocks
pcl>{
pcl*>S = new Periodic
pcl*>with S
pcl (S) *>adin = new AnalogIn
```

```
pcl (S) *>control = new PI
pcl (S) *>daout = new AnalogOut
pcl (S) *>
pcl (S) *>refgen.out -> control.yr
pcl (S) *>adin.out -> control.y
pcl (S) *>control.u -> daout.in
pcl (S) *>endwith
pcl *>
```

□

reset

This command is used to remove all blocks from the workspace and clears all system variables. All processes must be stopped before this operation is possible.

```
pcl>reset
```

quit

Stops the run-time system and exits to the surrounding shell. All processes must be stopped before exiting.

```
pcl> quit
```

use

User defined PAL modules are not by default visible to the run-time shell. They must be imported to make the the block types in the modules visible. To import a module the `use` command is used. The syntax is:

```
pcl> use < module name >
```

After this command has been issued , all blocks in the module are visible to the run-time shell and may be accessed by the user. When there exists several blocks with the same name, but in different modules, these modules may not be in use at the same time. A solution to this problem is to use the module name when instantiating the block as shown below.

```
s.control = new MyBlocks.PI
```

enduse

When a PAL module is no longer needed, it is possible to remove it from the list of available modules. This is done using the command `enduse`, as shown below.

```
pcl>enduse < module name >
```

dim

Complex variables such as string, arrays, polynomials and matrices may in PAL be created with a dynamical sizes. The sizes of block variables such as arrays etc may be linked to a dimension parameter. Several variables in the same block can be linked to the same dimension parameter. Dimension variables in different blocks can be connected via global dimension variables in the workspace, which are defined as follows.

```
pcl> dim A
```

In Example 5.2 there are two blocks `BlockA` and `BlockB` which both have a dimension parameter, `dimA` and `dimB`, respectively. The dimension parameter `dimA` determines the size of the output signal `out` and `dimB` determines the size of the input signal `in`. The output signal from `BlockA` is connected to the input signal from `BlockB`. A global

dimension variable DIM is defined, and it is is connected to the dimension variables in the blocks. Finally a value is given to the dimension variable

EXAMPLE 5.2

```
pcl> {
pcl> s = new Periodic
pcl> s.BlockA = new BlockTypeA
pcl> s.BlockB = new BlockTypeB
pcl> s.BlockA.out -> s.BlockB.in
pcl> dim DIM
pcl> s.BlockA.dimA = DIM
pcl> s.BlockB.dimB = DIM
pcl> DIM = 5
pcl> }
```

□

When a dimension variable is changed, all dimension parameters and all variables that in turn are connected to them, are changed. The change of variable sizes is synchronized throughout the whole system, so that problems with incompatible variables are avoided.

show

Pålsjö provides a text interface for entering PCL commands. There is no built-in facility for presenting data in graphical form. Instead it is possible to export data to other programs via the network. The command `show` is used to make data visible on the net. The syntax is shown below.

```
pcl> show < block >.< block >.< signal >
```

Before the signal of a block can be exported, the `Periodic` blocks, that owns the block must be connected over the network to a plot or datalog utility. This is done by sending the event `connect` to the `Periodic`. Below is an example of how this is done.

```
pcl> show process.control.u
pcl> show process.I
pcl> process ! connect
pcl>
```

After the event `connect` is sent to `process` a socket is opened on the network, and the data transmission will start as soon as any client program on the host machine will connect.

hide

The opposite of `show` is the command `hide` which removes a signal from the list of exported signal. `hide` is used in the same way as `show`

```
pcl> hide < block >.< block >.< signal >
```

break

Several commands may be grouped together using curly brackets, and such a set of commands is called an atomic operation, i.e. all commands within the brackets will be interpreted as one complex command. If the atomic operation is incorrect it will not be accepted by the system. The command `break` can be used to leave an incorrect atomic operation and discard the edits. In the example below the atomic operation is not correct since all input signals are not connected. When the command `break` is used all edits made in the atomic operations are reversed, and the running system remains unchanged.

```
pcl>use StandardBlocks
pcl>{
pcl*>s = new Periodic
pcl*>use StandardBlocks
pcl*>s.b = new PI
pcl*>}
Error in 'b' : input signal 'r' not connected!
--> Configuration invalid.
pcl*>
pcl*>
```

```
-->Now discarding all edits!
  Removing invalid nodes...
    Block 's ' is deleted.
    Block 'b ' is deleted.
pcl>
```

5.3 Operators

The predefined PCL operators are the following:

```
=          !          ?          @
->         <-         #          $
=>
```

The Assignment Operator =

All parameters in a PAL-block can be set by the user from the command line. Below is an example of how this is done. The first command sets the parameter `Ti` to 20. In this case `Ti` is a real valued variable, but the same syntax is valid for integers.

```
pcl>s.b.Ti = 20
pcl>s.b1.par1 = false
pcl>s.b1.par2 = {1.0, 2, 3.14}
pcl>s.b1.par3 = {1, 2.1, 3}
pcl>s.b1.par4 = {1, 2, 3.0: 4.1, 5, 6.28}
```

`par1` is a boolean parameter and may be assigned the values `true` or `false`, `par2` is an array with three elements, `par3` is a polynomial of degreee 2, and `par4` is a two by three matrix.

The Event Operator !

It is possible to manually trig the execution of block procedures. An event requesting the execution is sent to the block. The event simply

consists of the name of the proceure. Below a block procedure called on are executed.

```
pcl> s.b2 ! on
```

Events makes it possible to construct blocks and systems that directly interacts with the user.

The Information Operator ?

The information operator retrieves information about a system object. A system object may be a single variable, a block or the whole system. The syntax is straightforward, simply the desired system object followod by a question mark. Below are three examples of the available information, when using the information operator.

EXAMPLE 5.3

```
Including module 'built-in'
Including module 'StandardBlocks'


                                P Å L S J Ö

                Copyright  1995-97 Department of Automatic Control
                     Written by Johan Eker & Anders Blomdell
                          Lund Institute of Technology
                             version Beta-Sep  3 1997

                      bug report: johane@control.lth.se

pcl>use StandardBlocks
pcl>{
pcl*>s = new Periodic
pcl*>s.adin = new ADIn
pcl*>s.refgen = new RefGen
pcl*>s.regul = new PI
pcl*>s.daout = new DAOut
pcl*>?
```

```
W O R K S P A C E_____________________________________________
 Blocks:
 workspace of type WorkspaceBlock, id :0
    s of type Periodic, id :1
       adin of type ADIn, id :2
       refgen of type RefGen, id :3
       regul of type PI, id :4
       daout of type DAOut, id :5

DIMENSIONS:
Block libraries____________________
built-in* : InSocket, OutSocket, Sporadic, Periodic
StandardBlocks* : ADIn, DAOut, RefGen, PI, SimplePI, PID, Filter
```

Information about the `Periodic` block `process` is available in the same way. If no object is specified, then information about the run-time system is retrived.

```
pcl*>s ?
-------------------------------------------------------------
  BlockType: Periodic,   Block Name: s, Block ID: 1
  SIGNALS:_____________________________________________
    state: running(boolean)  = false
  PARAMETERS:__________________________________________
    : tsamp (int) = 2000
    : skip (int) = 5
    : prio (int) = 5
  EVENTS:______________________________________________
    connect[asynchronous]
    disconnect[asynchronous]
    restartplot[asynchronous]
    pauseplot[asynchronous]
    start[synchronous]
    stop[synchronous]
  DIMENSIONS:__________________________________________
  GRAFCET:_____________________________________________
Block list:
```

```
Input Buffers: { }
Output Buffers: { }
Export Buffers: { }
Execution order:
```

In a similair fashion, information about the block `regul` can be retrieved.

```
pcl*>s.regul ?
-------------------------------------------------------------
  BlockType: PI,   Block Name: regul, Block ID: 4
  SIGNALS:_____________________________________________
    input: r (double) [not connected]
    input: y (double) [not connected]
    input: u (double) [not connected]
    output: v (double) =  0.000000
    state: I (double) =  0.000000
    state: e (double) =  0.000000
  PARAMETERS:__________________________________________
    : tsamp (int) = 100
    : offset (int) = 0
    : slave (int) = 1
    : K (double) =  0.500000
    : Ti (double) =  10000.00
    : Tr (double) =  10000.00
    : bi (double) =  5.000E-6
    : br (double) =  10.00E-6
  EVENTS:______________________________________________
  DIMENSIONS:__________________________________________
  GRAFCET:_____________________________________________
pcl*>
```

□

For more information about the signals and parameters of the blocks, see Section 6.4.

The Connect Operators -> and <-

After blocks have been allocated and assigned to `Periodic` blocks, they must be connected to form an excutable system. Output signals and input signals are connected using the connection operators -> and <-. In the following example, two blocks and a `Periodic` block are allocated. The output signal `out` in `BlockA` is connected to the input signal `in` in `BlockB`. A connection is valid only if the input and the output signals are of the same data types and have the same sizes.

EXAMPLE 5.4

```
pcl> {
pcl> s = new Periodic
pcl> s.BlockA = new BlockTypeA
pcl> s.BlockB = new BlockTypeB
pcl> s.BlockA.out -> s.BlockB.in
```

□

If a connection results in an algebraic loop, the system will give a warning. When this occurs the blocks cannot be sorted according to data flow. The execution order will then be determined from the order the blocks were added to the `Periodic` block.

The Disconnect Operator

To break up a connection the disconnect operator is applied on the input signal. To disconnect the blocks in the previous example, the following command is given:

```
pcl> #s.BlockB.in
```

Macro Operator @

It is possible to use macros to simplify system configuration. A macro file consists of a set of PCL commands. A PCL macro may have arguments. Below a macro with two arguments is shown.

```
$1 = new Periodic
with $1
  $2 = new PI
```

```
endwith
```

All occurrences of $1 are replaced by the first argument, and all occurrences of $2 is replaced by the second argument and so on. A macro is called using the following syntax:

```
@<macro name>(par1, par2)
```

A call to the macro above would thus have the following look:

```
pcl>@macroname(S, A)
```

The default file extension for all macro files is '.pcl'.

The Move Operator =>

The move operator is used to move blocks from one location to another. For example from the workspace to a `Periodic` block, or from one `Periodic` block to another. This is demonstrated by an example. The resulting systems in Example 5.4 and Example 5.5 are equivalent.

EXAMPLE 5.5

```
pcl> {
pcl> s = new Periodic
pcl> BlockA = new BlockTypeA
pcl> BlockB = new BlockTypeB
pcl> BlockA => s
pcl> BlockB => s
pcl> s.BlockA.out -> s.BlockB.in
```

□

5.4 Summary

In this chapter the configuration language PCL for the PÅLSJÖ run-time system was introduced. Example on how to use both the operators and the commands were given.

6

The Pålsjö Framework

In this chapter the internal structure of the PÅLSJÖ framework is discussed. The computational blocks used in PÅLSJÖ are defined. Furthermore, the class hierarchy is presented. System blocks for handling the real-time behavior are presented. The different types of block connections are explained. The management of on-line configurations are presented. Implementation issues regarding the management of PAL blocks in real-time are discussed. All figures describing class hierarchy or object relations are drawn using object diagram notation, see Appendix C.

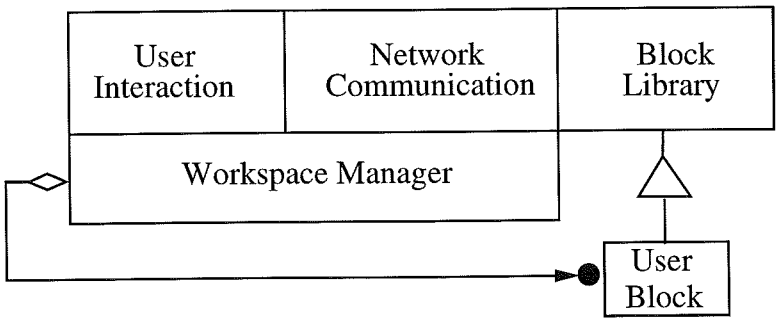


Figure 6.1 The framework handles the user interaction, the network communication and the block administration.

6.1 Structure

The inheritance structure of the framework from an end user's point of view is shown in Figure 6.1. The framework takes care of the interaction with the user, the network communication and manages the block instantiation and the execution. A user defined block class is inherited from a super class in the framework. The control algorithm is coded by implementing some abstract methods.

6.2 Definition of a Block

Block diagram is a tool, suitable for describing algorithms and dataflows. It supports modular programming and is well suited as an implementation model. A block is the smallest programming entity in the PÅLSJÖ environment, and is defined as a seven tuple
$B = \langle I, O, P, S, E, L, A \rangle$.

- A block can have a set of input signals I. An input signal must be connected to an output signal. Input signals may not be assigned values in the PAL code.
- A block can have a set of output signals O. An output signal may be connected to an input signal.
- A block can have a set of parameters P. Parameters can only be set from outside the block by the user or the system. The value of a parameter cannot be changed internally in the block.
- A block can have a set of states S, which describe the internal states of the block. A state can only be assigned internally.
- A block can have a set of events E, which it responds to. An event can be either synchronous or asynchronous. Synchronous events are executed at the next sampling instance. Asynchronous event are executed immediately when they arrive. A synchronous event could be a request for a mode change in the controller. An emergency stop event should be asynchronous.
- A block can contain sequential logic L, which is described by one or several Grafcets.

- A block can contain a periodic algorithm *A*, that describes the periodic behavior of the block. If a block contains periodic algorithms it must be executed periodically.

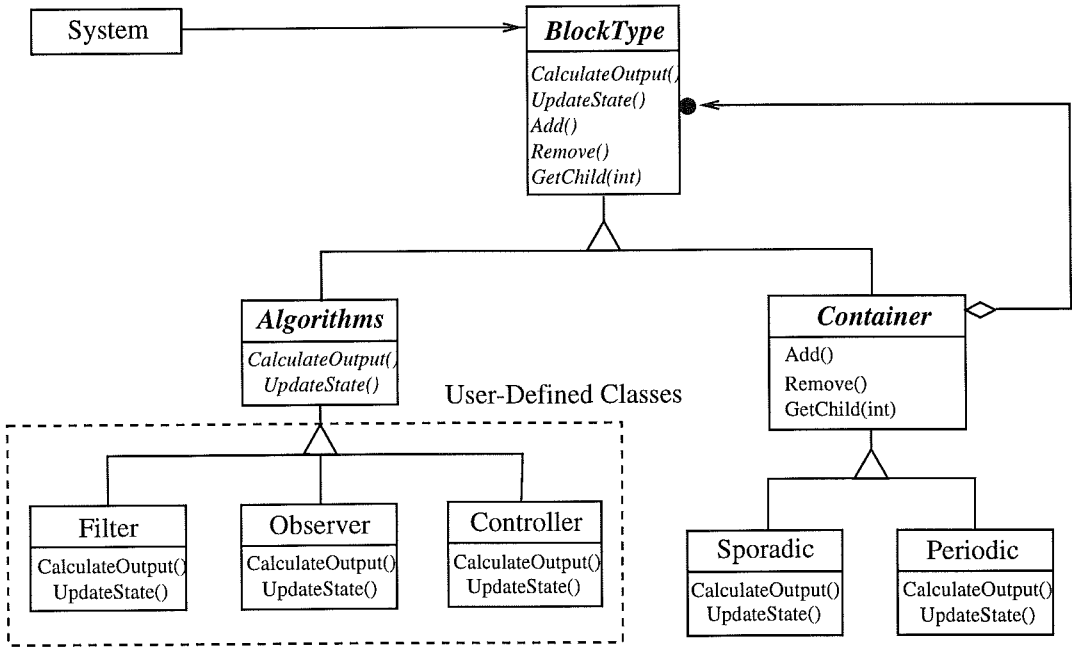


Figure 6.2 A user defined block is inherited from the pre-defined class `Algorithms`. `Container` is a super class designed to administrate and encapsulate a set of blocks. Two subclasses `Periodic` and `Sporadic` are available. They are both used for managing the execution of other blocks.

To support reuse of algorithms a design decision was made to separate temporal and functional specifications. The basic idea is to view PAL blocks as input-output blocks, which only need to know at what frequency they are executed. A PAL block cannot contain any temporal constraints, and neither can it demand synchronization. All temporal functionality is taken care of by designated system-blocks, which handle the actual execution of the PAL blocks. Using this approach the programmer does not have to deal with any real-time programming. Furthermore it is possible for the systems to optimize the execution and prevent problems, for example jitter[1] [Törngren, 1995].

[1] *Jitter* refers to non intentional variations in the sampling period.

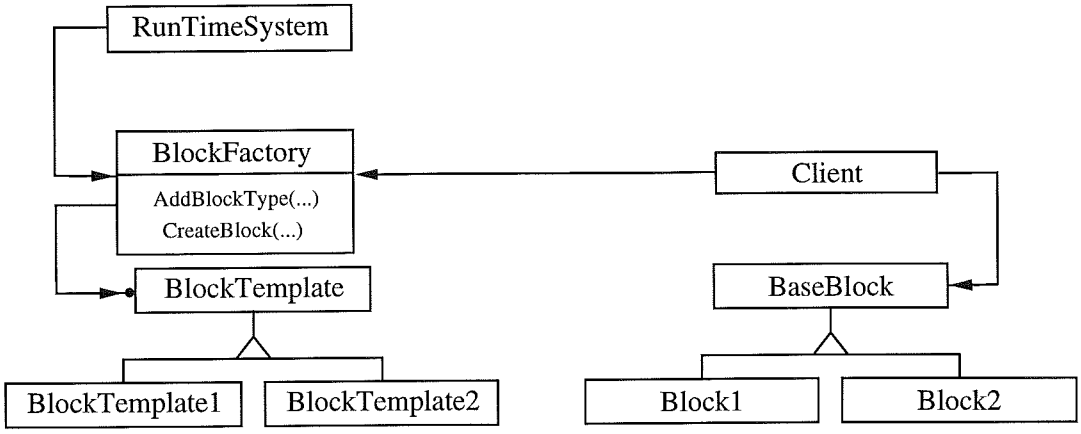


Figure 6.3 All available block types are stored in the `BlockFactory`. Allocation of a block is made using the `BlockFactory`. It is possible to add new blocks to the `BlockFactory` on-line.

6.3 The Framework and its Classes

The inheritance structure of the class library is shown in Figure 6.2. The super block to all block classes is `BlockType`, which contains the basic functionality that a class needs to be integrated in the PÅLSJÖ environment. The subclasses implement the abstract methods `CalculateOutput` and the `UpdateState`. Two subclasses from `BlockType` exist. One is `Container` which is designed to encapsulate other blocks. The other is `Algorithms`, which is the super class for all user defined PAL blocks. Two subclasses from `Container` are available, `Periodic` and `Sporadic`. Those are used to manage the execution of algorithm blocks at run-time. The inheritance structure is know as the Composite pattern [Gamma *et al.*, 1995]. Using `Container` blocks the system supports the requirements for a hierarchical structure that was specified.

Dynamic Creation of Data types and Block types

It must be possible to extend the framework in a simple way. Blocks and data types are in PÅLSJÖ loosely coupled to the rest of the framework through the use of an Abstract Factory pattern, see [Gamma *et al.*, 1995]. The Abstract Factory is designed to *provide an interface for creating families of related or dependent object without specifying their concrete classes.* Figure 6.3 show the class diagram for the block

factory. The run-time system calls the `BlockFactory` upon initialization and registers each available block type by giving a name tag and a constructor function. The `BlockFactory` keeps a table over all registered block types. When the client wants to create a new block it calls the `BlockFactory` with the name of the block. The `BlockFactory` tries to find and run the constructor for the wanted block type, and upon success it returns a handle to the new block instance. It is possible to register new block types and delete old during execution. This means that it is possible to extend the system with new functionality, without having to shut it down.

A similar factory is used for dealing with data-types. Whenever a new block is instantiated the first thing it does is to allocate its variables through the `VariableFactory`.

The available block and data types can thus be changed on-line, without having to stop and restart the system.

6.4 The System Blocks

Block Execution

While PAL is used to describe the algorithm of a block, it cannot be used to specify *how* the block shall be executed. All information that defines real-time behavior is entered through PCL. Instances of the system blocks `Period` and `Sporadic` are created to manage the execution. A block derived from `Algorithm` must have a `Container` block as a parent in order to be executed. The `Periodic` block executes its child blocks periodically according to the data flow between the child blocks. All timing and synchronization between the child blocks is taken care of by the scheduler in `Periodic`. Consider the block diagram in Figure 6.4. It consists of six blocks. An analog input block followed by a pre-filter, two controllers in parallel, a switch and an analog output block. The pre-filter is used to reduce the noise from the measurement signal. To get good control performance the filter is executed at a higher sampling rate than the control blocks. The measurement signal is thus down sampled before it reaches the controllers. The desired real-time behavior are the following:

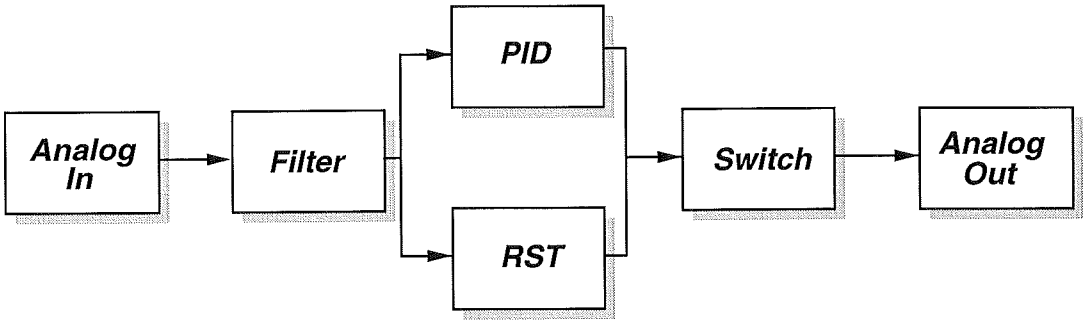


Figure 6.4 The block diagram created in Example 6.1 system.

- Execute **RST**, **PID**, **Switch** and **Analog Out** at a slower sampling rate.
- Execute **Analog In** and **Filter** at a higher sampling rate.

To configure the system to get the this behavior, two instances of the system block `Periodic` are created, one for each sampling rate. The PCL-code to achieve this is the following:

EXAMPLE 6.1

```
pcl>{
pcl*> p1 = new Periodic
pcl*> p2 = new Periodic
pcl*> p1.adin = new AnalogIn
pcl*> p1.filter = new Filter
pcl*> p2.rst = new RST
pcl*> p2.pid = new PID
pcl*> p2.switch = new Switch
pcl*> p2.daout = new AnalogOut
pcl*>
pcl*> p1.adin.y -> p1.filter.u
pcl*> p1.filter.y -> p2.pid.u
pcl*> p1.filter.y -> p2.rst.u
pcl*> p2.pid.y -> p2.switch.u1
pcl*> p2.rst.y -> p2.switch.u2
pcl*> p2.switch.y -> p2.daout.u
pcl*> p1.tsamp = 0.010
```

```
pcl*> p2.tsamp = 0.050
```

□

Blocks that reside within the same `Periodic` is executed synchronously. This means that no data sharing problems will arise for blocks within the same `Periodic` block. The data-flow mechanism is implemented, so that unnecessary copying is avoided. Input signals in blocks are implemented as pointers, that simply point at output signals, see Figure 6.5. This means that there is no copying of data involved, when passing signals from one block to another within the same `Periodic` block. Connected blocks, that belong to different `Periodic` blocks must be handled in a special way. Buffers are automatically created to preserve the consistency of data, shared between several `Periodic` blocks. Mutual exclusion is thus handled automatically by the system.

By grouping blocks together, the execution gets more efficient and problems with jitter is avoided. Consider, for example, if instead each block was running as a separate process. First, there would be a problem with data latency, since the processes are running without synchronization. If all blocks execute with the same sampling interval, it would be possible that data is delayed by one sample for each block, from input to the output. Another disadvantage would be the overhead introduced due to the context switching, when processes are stopped and restarted.

Periodic

The `Periodic` block is a block for scheduling periodic algorithm blocks. Algorithm blocks are added to the `Periodic` block on-line. When the `Periodic` starts to execute it creates a schedule for the execution of its child blocks. This schedule is based on the attributes presented below, which all algorithm blocks have.

- **tsamp** is the sampling period in milliseconds.
- **offset**, defines an offset in milliseconds. This is used for blocks having the same sampling rates, but executing at different time instances. The offset parameter defines the time skew between the block and time zero.

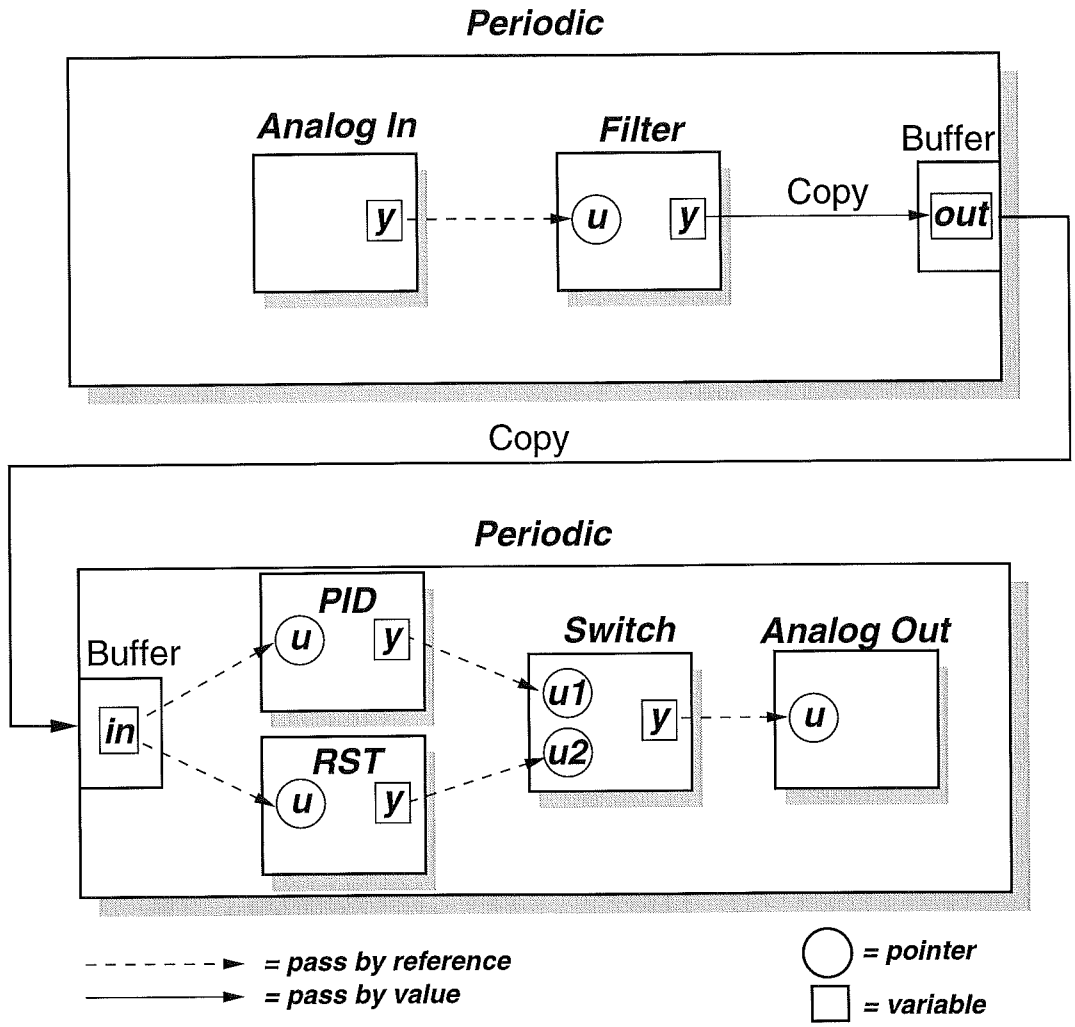


Figure 6.5 A system with one process for down sampling the measurement signal, and one process for the control algorithms.

- **slave** This is a boolean parameter, which defines whether the sampling rate shall be determined from the parameters above, or if it shall be set by the `Periodic` block.

The `Periodic` handles the real-time management of its child blocks by creating an execution schedule. This schedule contains lists over the child blocks that are to be executed at each sample. The execution array contains information on how the blocks shall be executed, and it is created when the `Periodic` block is started. The creation of the execution array consists of three parts; First the sampling period for the `Periodic` process is calculated, then the length of the execution

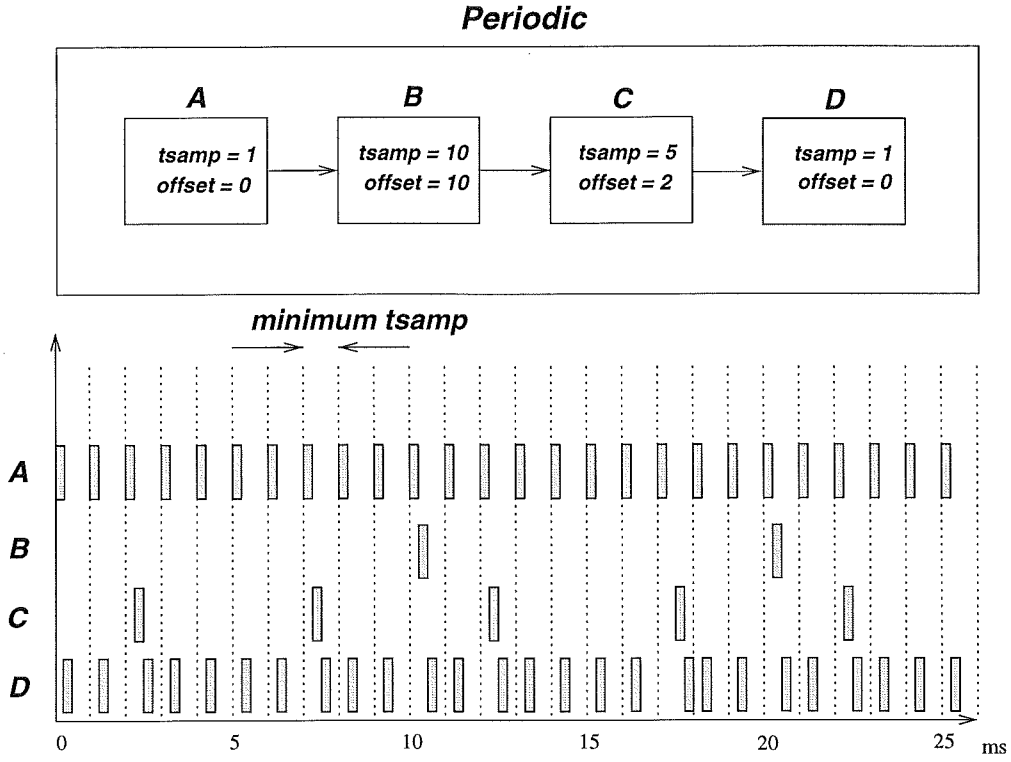


Figure 6.6 To create a schedule, the `Periodic` block uses the parameters `tsamp` and `offset` in each block. The lower figure shows the resulting schedule for the four blocks A, B, C, and D in the block diagram at the top.

array is determined. Finally the execution order of the blocks with respect to data-flow is calculated. This last step will be ignored if there are any algebraic loops in the configuration. The execution order will then be the order the blocks were added to the `Periodic` block.

The algorithm for creating the schedule is the following:

- Let *list* be formed as the sampling times and offsets of all non slave blocks.

$$list = \{tsamp_1, offset_1, tsamp_2, offset_2, \cdots, offset_n\} \qquad (6.1)$$

- The sampling time for the `Periodic` block is given as the greatest common divisor of the elements of *list*. The greatest common divisor is found using Euclid's algorithm [Knuth, 1969], and the identity

$$gcd(u_1, u_2, \cdots, u_n) = gcd(u_1, gcd(u_2, \cdots, u_n)) \qquad (6.2)$$

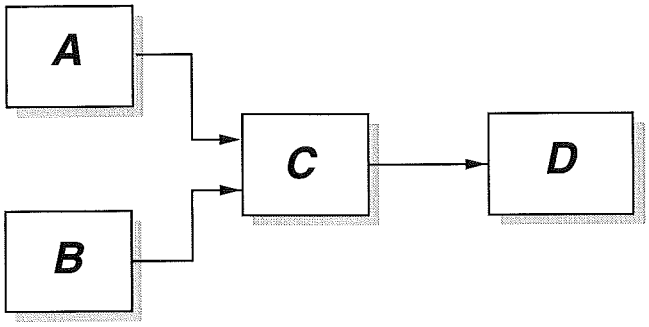


Figure 6.7 The block diagram used in Example 6.2, 6.3, and 6.4.

- The schedule length is found as the *least common multiplier, lcm*, which is calculated using the following identity:

$$u \cdot v = gcd(u,v) \cdot lcm(u,v) \tag{6.3}$$

  The number of samples necessary to form a schedule cycle can now be expressed as

$$array\_length = \frac{lcm(list)}{gcd(list)} \tag{6.4}$$

- Find the execution order of the blocks based on the data-flow.
- Allocate the execution schedule and fill it with lists of blocks that should be executed at each sample.

If a `Periodic` block has child blocks and some blocks are slaves, and other are not, the sampling rate for the slave blocks are calculated based on the values of the non slave blocks. The slave blocks will execute every time the scheduler is invoked.

Schedule Examples

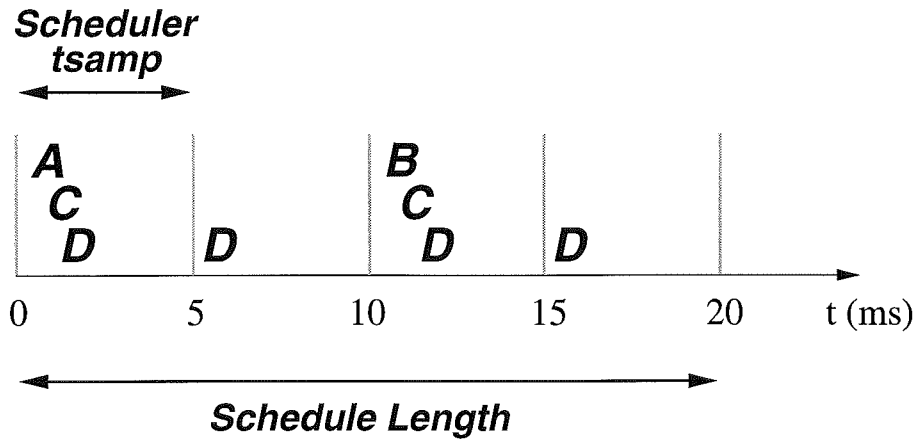
In this section three scheduling examples are given. Three different schedules are calculated for the system in Figure 6.7. All four blocks in Figure 6.7 belong to the same `Periodic` block. In the examples, the scheduling parameters for each block are given on the following format: $\langle tsamp, offset, slave \rangle$. The *tsamp* and *offset* parameters are only used when *slave* is equal to $false$. Unused parameter values are marked by "–". The scheduling parameters for each block is ordered in the *input* vector, according to data flow.

EXAMPLE 6.2

Let the input vector be

$$input = \langle P_A = \langle 20, 0, false \rangle, P_B = \langle 20, 10, false \rangle, \\ P_C = \langle 10, 0, false \rangle, P_D = \langle 5, 0, false \rangle \rangle$$

The resulting schedule is then



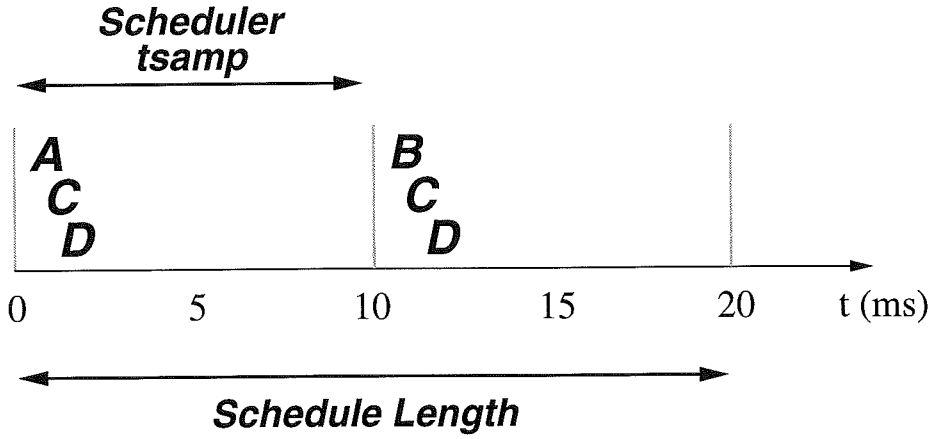
□

EXAMPLE 6.3

Let the input vector be

$$input = \langle P_A = \langle 20, 0, false \rangle, P_B = \langle 20, 10, false \rangle, \\ P_C = \langle -, -, true \rangle, P_D = \langle -, -, true \rangle \rangle$$

The resulting schedule is the following



□

EXAMPLE 6.4

Let the input vector be

$$input = \langle P_A = \langle -, -, true \rangle, P_B = \langle -, -, true \rangle, \\ P_C = \langle -, -, true \rangle, P_D = \langle -, -, true \rangle \rangle$$

In this example, the **slave** parameter is true for all blocks. All blocks will then execute with the same sampling period, and this sampling period will be the `tsamp` of the `Periodic` block. □

Running a `Periodic` Block

A `Periodic` is started by the event `start`. If all the child blocks are slaves, the sampling time used is determined by the `tsamp` parameter in the `Periodic` block. Besides handling the execution of the child blocks, the `Periodic` block also handles the data exporting, i.e. making data available outside PÅLSJÖ. A block variable can be made ready for export using the PCL command `show`. Each `Container` block has its own network socket. The default name of the socket is `palsjo!<block name>-<block id>`. The socket is opened by the event `connect` and closed by the event `disconnect`. How frequently data will be collected is determined by the `skip` parameter. A `Periodic` block will log data at every `skip`:th sample.

Sporadic

The `Sporadic` block is simpler than the `Periodic` block. It just executes its child blocks cyclicly without any timing concerns. This means that the algorithms that are executed by a `Sporadic` block must contain at least one blocking statement or a time delay in order not to starve out other, low priority processes. An example of a blocking statement is a function which waits for a signal to arrive, i.e. the function will block execution until the signal arrives.

The `Sporadic` block is designed to hold blocks that are communicating asynchronously with other blocks or the environment. An example is a block which sends and receives data over the network. The data export facilities are similar to those of `Periodic`.

RemoteIO

The third type of system blocks are the remote input-output blocks. The `InputSocket` is used for reading from the network, and the `OutputSocket` is used for writing to the network. Both blocks have a parameter named `server`, which specifies the name of the network socket. The `InputSocket` has a `timeout` parameter, which sets the timeout in milliseconds. If the timeout is set to zero, `InputSocket` is blocking. In Example 6.5 a data matrix is first read from the network, and then written back to the network.

EXAMPLE 6.5

```
pcl>{
pcl*>  s = new Sporadic
pcl*>  s.a = new InSocket
pcl*>  s.b = new OutSocket
pcl*>  dim cols
pcl*>  dim rows
pcl*>  s.a.cols = cols
pcl*>  s.a.rows = rows
pcl*>  s.b.cols = cols
pcl*>  s.b.rows = rows
pcl*>  rows = 4
pcl*>  cols = 4
pcl*>  s.a.data->s.b.data
```

```
pcl*>  s.a.server = "inputserver"
pcl*>  s.b.server = "outputserver"
pcl*>  s ! start
pcl*>}
```

□

6.5 Dimensions

Dimension is a special parameter type used for defining the sizes of compound data types, such as matrices and polynomials. The need of a special data type for handling sizes of compound data types is motivated by the need for synchronized size changes. In a situation where, for example an output polynomial is connected to an input polynomial, and the degrees are changed, it is necessary that this change is done simultaneously throughout the system. Dimension variables in PAL blocks may be connected to global dimension variables, created by the user in PCL. When the user assigns a value to a global dimension variable, all dimension variables, that are connected to it, are updated synchronously.

6.6 Connections

There are three types of connections in PÅLSJÖ:

1. output signal to input signal
2. output signal to parameter
3. dimension to dimension

There is no restriction in what data types that may be used in connections. For aggregate types such as matrices, arrays and polynomials the dimensions must match.

The first type of connection is the most common. It is used to simply transfer data from one block to another during the execution of the `calculate` and `update` procedures.

The second type of connection is intended for adaptive systems, and is more expensive regarding computation time.

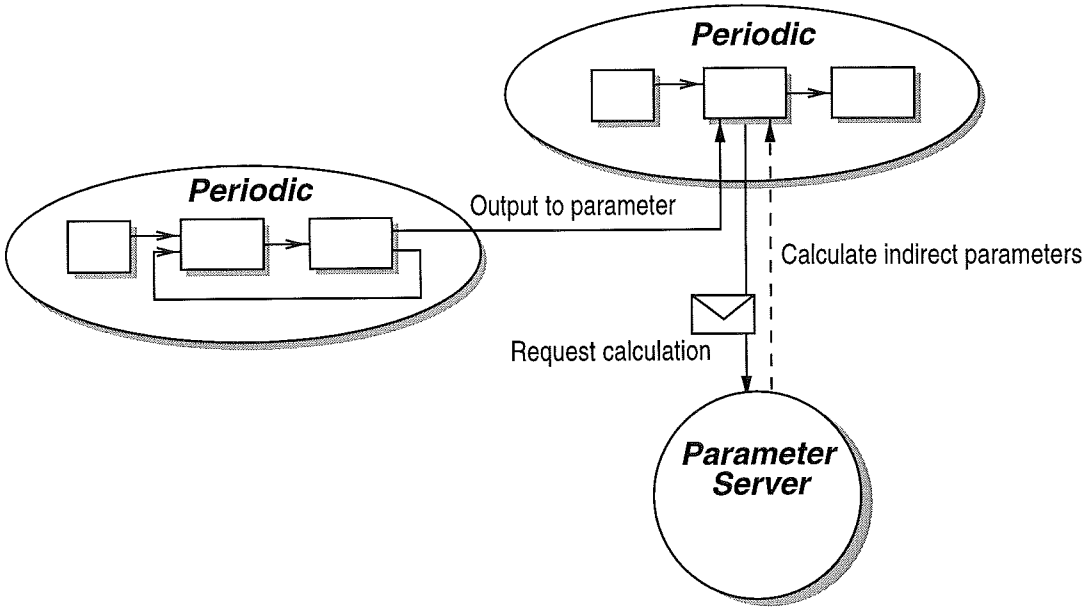


Figure 6.8 To handle connections of the type "output to parameter", special care must be taken. In PÅLSJÖ a special `ParameterServer` process is used to handle this.

The reason for this is that it is possible to define parameters as functions of other parameters. When the output signals of the source block change, the parameters in the target blocks must be updated. However, before the new parameters can be used by the target block, all indirect parameters must be calculated. This may be a time consuming operation and cannot be performed by the block itself, since the extra computation time could lead to time-delays. In PÅSLJÖ, when a parameter receives a new value from an output signal, it sends a request to the **ParameterServer**. The **ParameterServer** process then takes care the of the calculation of the indirect parameters. Figure 6.8 shows the structure with a **ParameterServer**. In a sampled system, for example, the system matrices may be defined as functions of the sampling interval. If the sampling interval is changed, the system must be re-sampled. The calculation for this can be quite extensive, and must be done in such a way that it does not interfere with the execution of the block. All signals are marked with time-stamps every time they are assigned. This information is used by the system, which

only transfers values from output signals to parameters when the output signals have changed. The **ParameterServer** then handles the calculation of the indirect parameters. When it is finished, it sends a message to the target block, that new parameters are available.

In Figure 6.9 there are two `Periodic` blocks. The `Periodic` blocks are marked by dashed lines. The estimator, located in the upper `Periodic` block, receives the process value and the control signal from the RST-controller. The outputs of the design block are connected to the parameters of the RST-controller. The estimator recursively calculates a new process model, and passes this further to the design block. The outputs of the design block are then connected to the parameters of the RST-controller. The grey boxes on the edges of the `Periodic` blocks mark the buffers, used for exchanging data between `Periodic` blocks. New controller parameters are calculated by the design block. When new values are assigned to the outputs, they get new time-marks. This is detected by the system, which now calculates the new parameters values for the RST-controller. When the calculation of the new parameters is completed, the RST-controller is notified.

The dimensions of the input signal and the output signal must of course be the same. For example, an output signal of type polynomial must be of the same degree as the input signal to which it is connected.

The third type of connection is dimension to dimension. The dimension parameters of one or several PAL blocks, may be connected to a global dimension variable. When the global dimension variable changes, all local dimension variables connected to it are updated. This update is made synchronously throughout the system.

An example of a system with dimension to dimension connections are shown in Figure 6.10 The gray dashed line is the dimension connection. This connection ensures that the dimensions of the estimated system in the estimator, and the dimensions of the polynomials in the controller, always are correct.

6.7 On-Line Configurations

It is possible to edit the block configuration on the fly, i.e. replace one or more blocks without having to stop and restart the system. This could be a very dangerous maneuver. Before the old blocks are

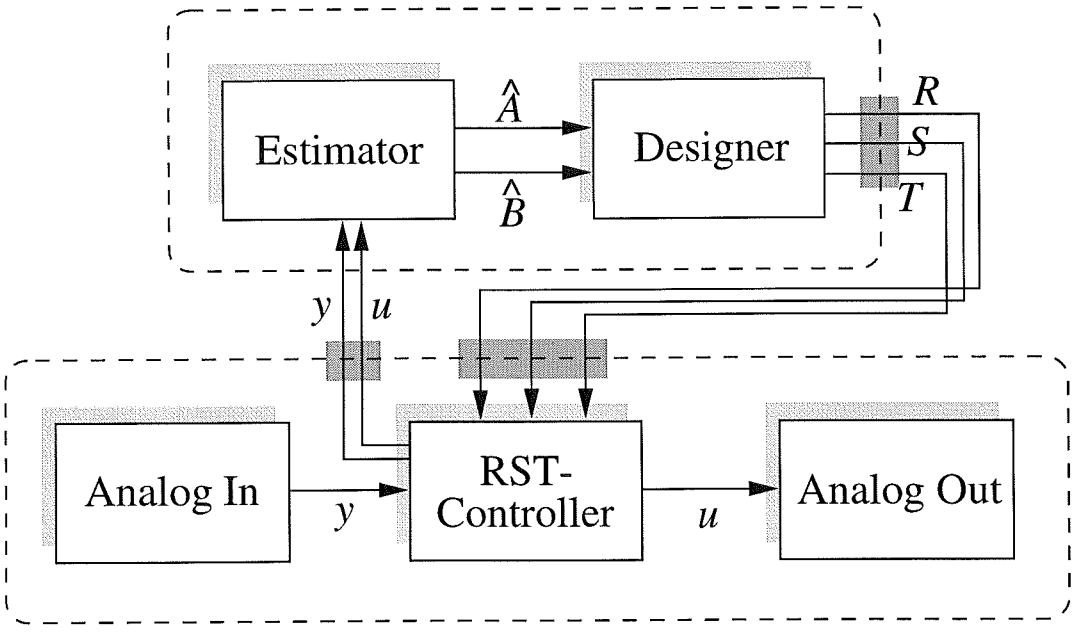


Figure 6.9 An adaptive system with two processes. One which handles the feedback control loop, and one that handles the slower adaption loop. The output signals from the Designer-block are connected to the parameters of the RST-block. The RST-parameters are only updated when the outputs from the Designer-block are changed.

replaced, the new block configuration must be analyzed and checked for errors. In Pålsjö it is possible to treat a set of primitive operations as one atomic operation. An atomic operation must always transform the system from one valid configuration to another. An atomic operation can either consist of one PCL command or of several PCL commands grouped together by curly brackets.

When an atomic operation is ended, the run-time system checks if the new configuration is correct. If everything seems to be OK, the current configuration is replaced by the new one. If instead an error is found, the user will get an error message. The user can choose to either correct the operation or discard it. Internally there exist two copies of each block, one that is used for executing in real-time, and another which is used for editing. The set of blocks which is executed is called the *Running* configuration, and the other set is called the *Shadow* configuration. This setup is shown in Figure 6.11. All commands that deal with creating blocks, moving blocks, connecting blocks, etc., are done on the shadow configuration. When the edit session is completed

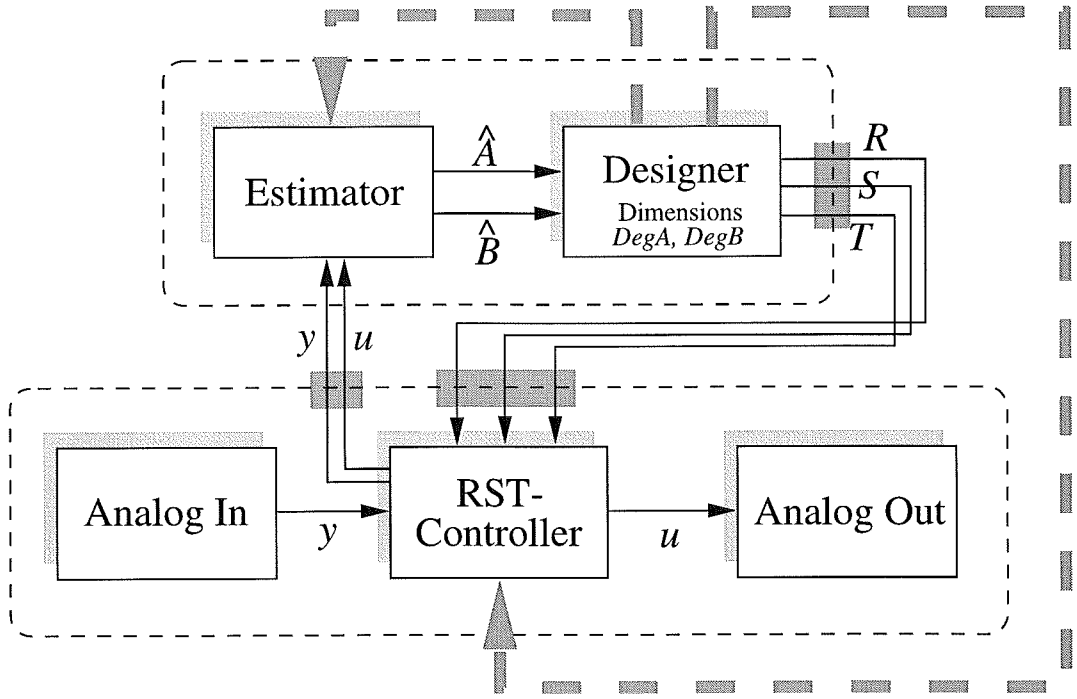


Figure 6.10 An adaptive controller where the degree of the RST-polynomials and the processes model polynomials are connected via dimension connections.

the shadow configuration becomes the running configuration. What happens when a block is replaced is shown in Figure 6.12.

In Example 3.2, a controller consisting of an analog-in block, a reference generator, an analog-out block and a controller was created. This session is continued in Example 6.6, where the controller block now is being replaced.

EXAMPLE 6.6

```
pcl>{
pcl*> delete s.control
pcl*> s.control2 = new RST
pcl*>
pcl*> process.adin.out -> process.control2.y
pcl*> process.refgen.out -> process.control2.yr
pcl*> process.control2.u -> process.daout.in
pcl*> }
--> Tree checked and seems to be OK.
```

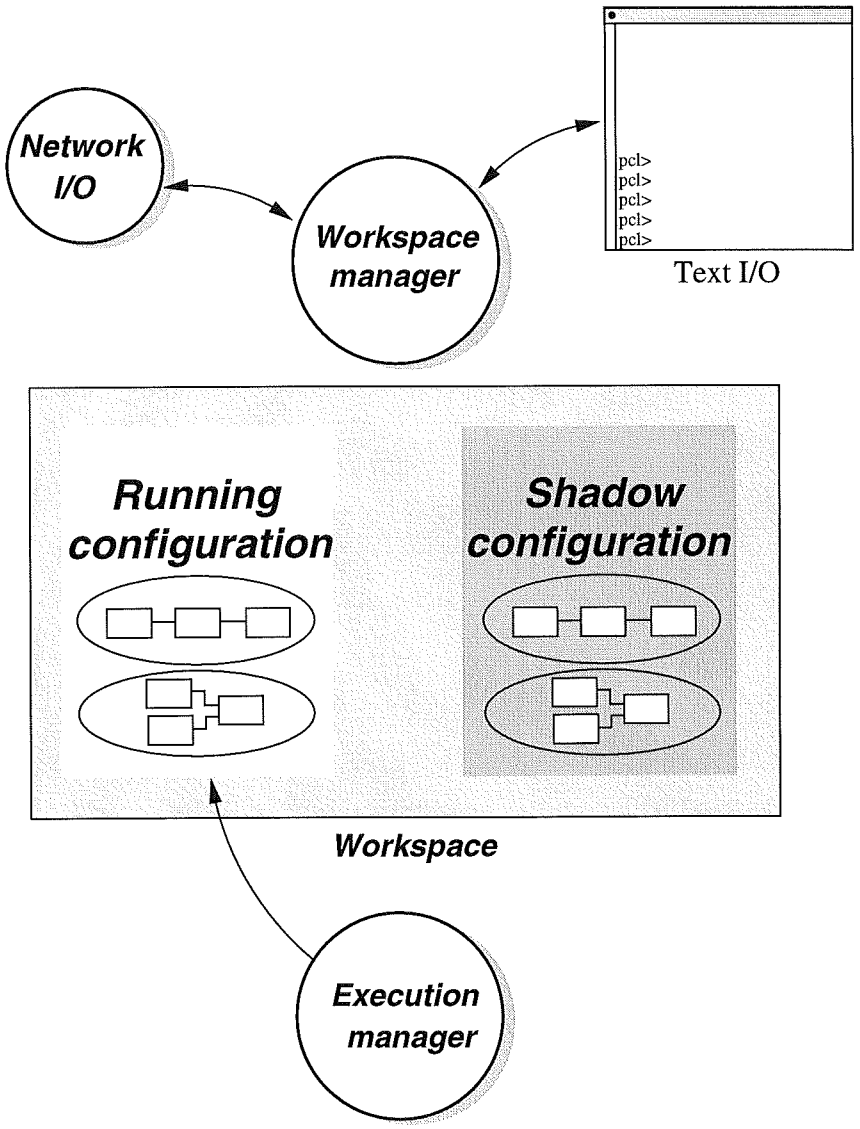


Figure 6.11 The internal structure of the run-time system. There are processes for network communication, user interaction and for task management. The system configuration is stored on the workspace. *Running configuration* is the system configuration in use, while *Shadow configuration* is used for editing.

```
Message from 'process' : clearing blocklist...
--> New blocks are now ready to run.
--> Swapping...
Message from 'process' : Analyzing block sequence...
--> Waiting for old processes to terminate ...
pcl>
```

□

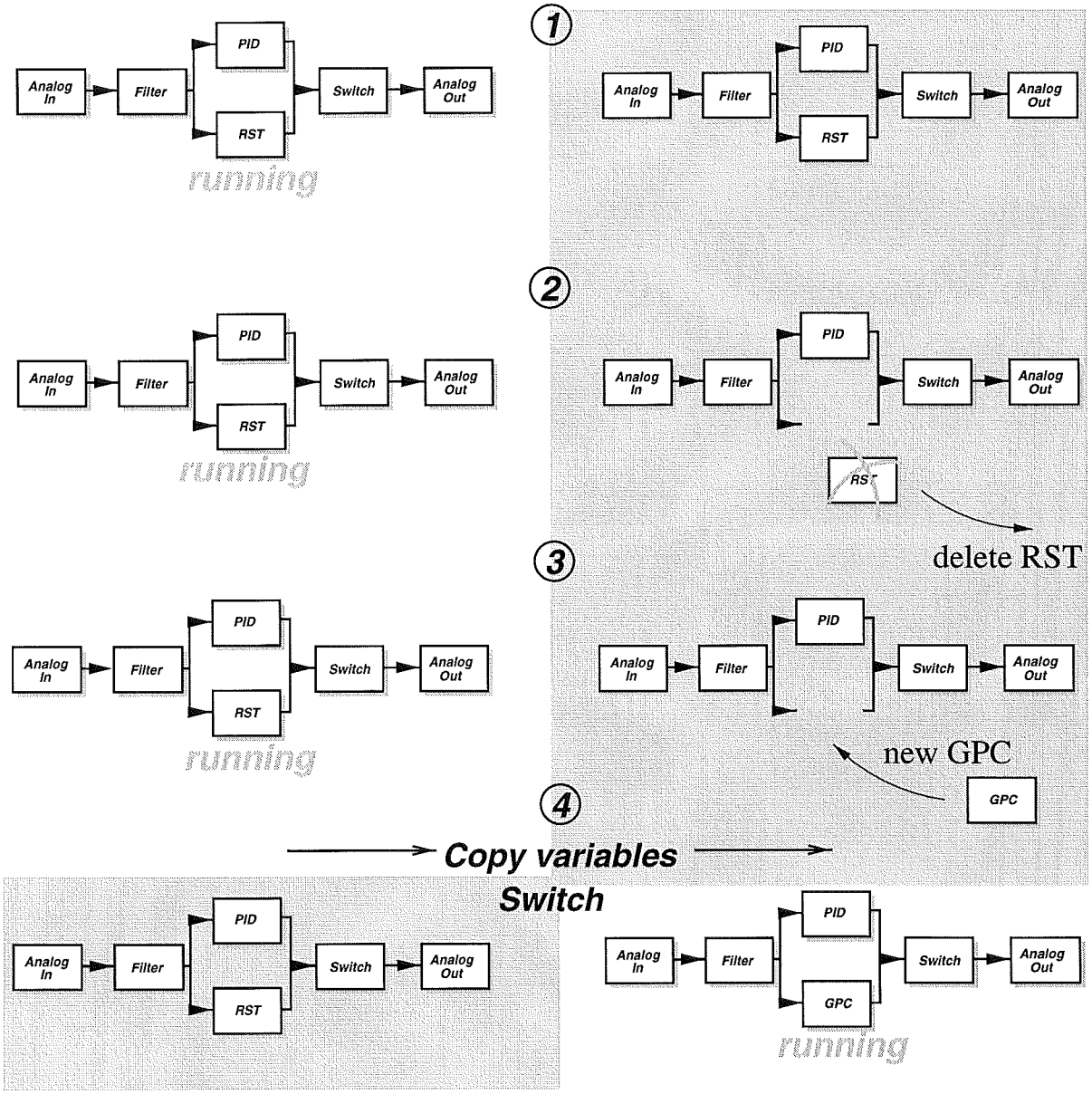


Figure 6.12 The internal structure during an edit session where one block is replaced by another. All edits are made on the shadow configuration to the right, which, when the edit session is completed, becomes the active one. The edit session is shown in four steps. In step two the RST block is removed by the user. A new block is then created and connected in step three. In step four the shadow configuration becomes the running configuration. Before the switch is performed the values of all variables are copied from the running configuration to the shadow configuration.

In a valid configuration all input signals in every block must be connected. In Example 6.7, which is an alternate continuation of Example 3.2, the input signal yr is not connected and at the end of the atomic operation an error message is displayed. Since the atomic operation cannot be concluded, the system remains in edit mode. The command break is then used to leave the faulty atomic operation. All commands in the atomic operation are then reversed, and the system configuration is not changed.

EXAMPLE 6.7

```
pcl>{
pcl*> delete s.control
pcl*> s.control2 = new RST
pcl*>
pcl*> process.adin.out -> process.control2.y
pcl*>
pcl*> process.control2.u -> process.daout.in
pcl*> }
ERROR in block 'control': input signal 'yr' not connected!
--> Configuration invalid.
pcl*> break
-->Reverse commands
   Block 'control' is restored.
-->Removing invalid nodes...
    Block 'control2 ' is deleted.
pcl>
```

□

6.8 Summary

In this chapter the internal structure of the framework has been discussed. The class hierarchy was presented. The principles for different kinds of communication between blocks was discussed. System blocks

for scheduling and network communication have been presented. The structure for handling on-line configurations was shown together with a sample session.

7

Patterns

In this chapter three patterns used in the PÅLSJÖ framework are presented. All figures describing the class hierarchy and object relations are drawn with the object diagram notation. For a short introduction to patterns and object diagrams, see Appendix C.

7.1 The Calculate-Update Pattern

Intent

This pattern addresses the problem where several interconnected function blocks are used to perform calculations. Function blocks consist of input signals, output signals and internal states. When a function block is executed new output values are calculated based on the values of the input signals and the states. The values of the output signals are used as inputs to the next block in the signal flow path. The problem is to execute the blocks so that their internal states are updated correctly. In a control system the output signals from blocks later in the signal flow path are often needed in order to update the states correctly. Consider a system with three blocks. The output from the first block is used as input to the second, and so on. Here in order to update the states of the first block, the output signal from the third block must be known.

Another use of this pattern is if the computational result from a function block can be divided into two parts. Again consider a control

system. It is important to finish the calculation of the new control signal as fast as possible, so that the time delay is minimized. The update of the internal state is on the other hand allowed to be more time consuming.

The two issues described above are addressed by the Calculate-Update pattern.

Variations - Also known as

Forward – Backward
Forward sweep – Backward sweep

Motivation

Consider the control system in Figure 7.1. The controller internally consists of a set of blocks. First the value of the process is sampled and the signal propagates from the first block to the second one. The output of the second block is then passed on to the third block and so on. The input signal is sampled and a new output signal is calculated periodically at a specified sampling rate. In a control system the stability of the closed loop system depends on the time it takes for a process measurement to show up in the actuator signals to the process. If there are multiple blocks between the input and output, special care has to be taken to ensure that no unnecessary delays are introduced due to improper ordering of calculations. Consider the following two different execution orders for the blocks in the controller in Figure 7.1.

Backward order	**Forward order**
AnalogOut(out3);	in1 := AnalogIn();
out3 := in3;	out1 := in1;
in3 := out2;	in2 := out1;
in2 := out1;	in3 := out2;
out1 := in1;	out3 := in3;
in1 := AnalogIn();	AnalogOut(out3);

If the calculations are done in backward order, the input values reach the output after 6 sampling intervals, while in forward order they reach the output in a fraction of a sampling interval, and this seems to be the obvious choice since we want to minimize the time delay.

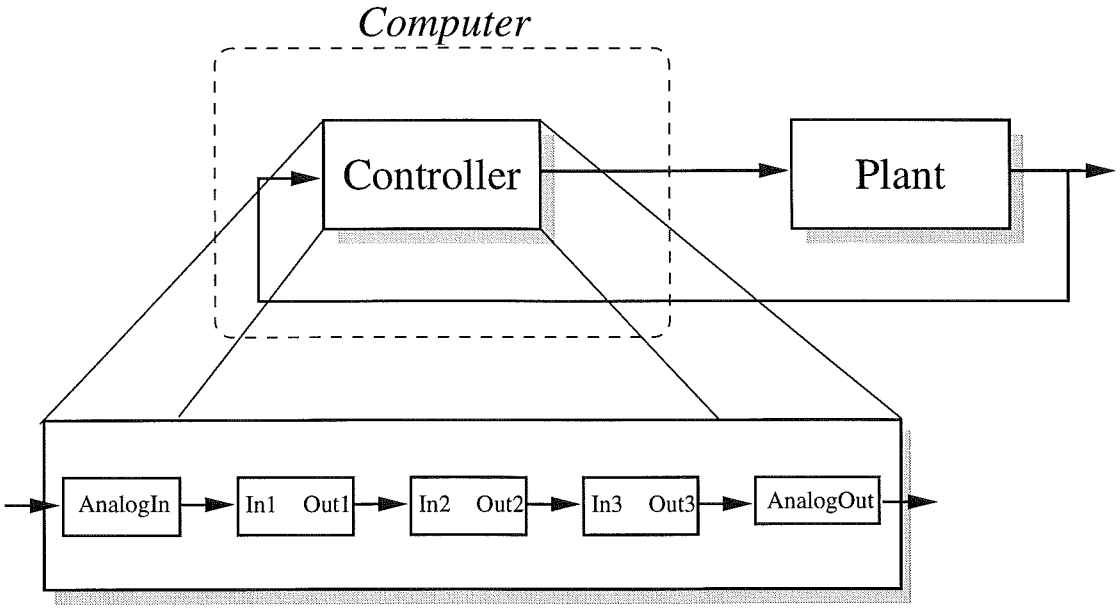


Figure 7.1 A schematic view of a controller which consists of a number of sub–blocks. The output from the plant is sampled by *AnalogIn*, and then the signal is propagated from the left to the right until it reaches *AnalogOut*, which sends the control signal to the actuator. The Calculate-Update pattern addresses the problem of minimizing the time delay between process measurement and actuation.

On the other hand, as mentioned earlier, when updating the internal state of block i, information about the output signal of block $i + 1$ could be needed, and this would motivate backward order.

Structure

The discussion above suggests a division of the calculations made in a block into two parts as shown below.

Calculate (Forward)	Read inputs Do calculations needed for outputs Write outputs
Update (Backward)	Do all other calculations

Again consider the controller in Figure 7.1, and let each of the sub-

blocks have two functions *Calculate()* and *Update()*. The execution of the blocks would now be the following:

AnalogIn.Calculate()
Block2.Calculate()
Block3.Calculate()
AnalogOut.Calculate()
AnalogOut.Update()
Block3.Update()
Block2.Update()
AnalogIn.Update()

Participants

In the PÅLSJÖ Framework the execution of a block diagram is taken care of by an object inherited from the class `ContainerBlock`, see Figure 6.2.

Sample Code

The algorithm for the PI-controller in Example 3.1 is divided into two parts. One part for calculating the output signal and one part for updating the integral state.

A control system usually consists of a set of sub-blocks. Assume that all blocks have a **calculate** procedure and an **update** procedure. The execution of the system with five sub-blocks would then be

```
block Periodic
  blocks : array [1..5] of block;
  n := 5 : integer;

  calculate
    i : integer;
  begin
    for i := 1 to n do
      blocks[i].calculate();
    end for;
  end calculate;
```

```
  update
    i : integer;
  begin
    for i := n downto 1 do
      blocks[i].update;
    end for;
  end update;

end Periodic;
```

Consider the implementation of a cascade PI-controller. Two PI-blocks are connected in series. The first PI-controller generates the reference value to the second PI-controller. In order to handle actuator limitations correctly, the integral states must be updated with care. The PAL code for the PI-controller in this example is similar to the code in Example 3.1, but extended with an extra output signal $r2$. The $r2$ signal is calculated in the **update** procedure and is used to propagate actuator limitations backwards, against the data flow. The input signal r is the desired reference signal and $r2$ is the reference signal modified with respect to actuator limitations.

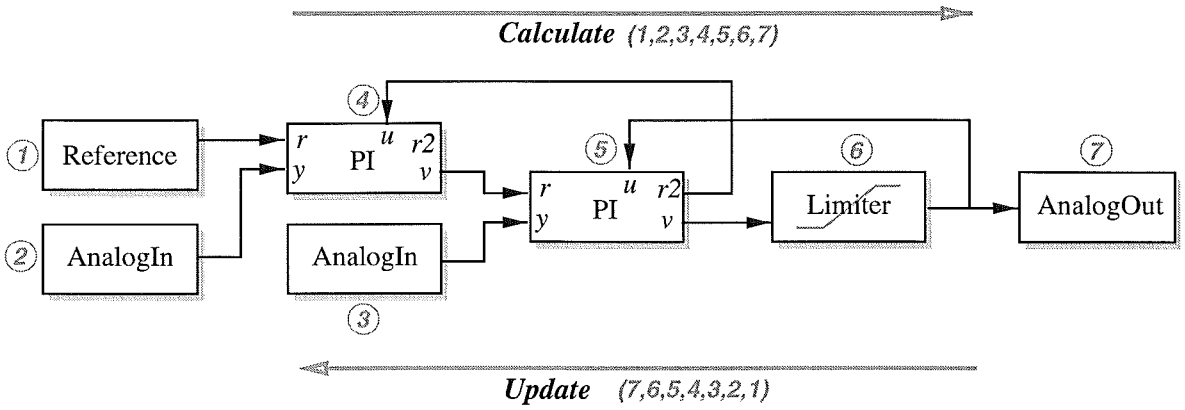
```
block PI
  r, y, u : input real;
  v, r2 : output real;
  I := 0.0, e := 0.0 : real;
  K := 0.5, Ti := 10000.0, Tr := 10000.0 : parameter real;
  h : sampling interval;
  bi = K * h/Ti; br = h/Tr;

  calculate begin
    e := r - y;
  end calculate;

  update begin
    r2 := y + (u - I)/K;
    I := I + bi * e + br * (u - v);
  end update;

end PI;
```

This execution order of the blocks is illustrated in the figure below.



Known Uses

This is a well known pattern in the control community. Use of this is suggested in many control textbooks. One known industrial application is SattLine from Alfa Lava Automation [Johannesson, 1994]. Use of this pattern is also discussed in [Elmqvist, 1985].

7.2 The Parameter-Swap Pattern

Intent

The Parameter-Swap pattern is a real-time pattern which deals with the problem of updating, a possibly large, set of controller parameters in a fast and consistent way.

Motivation

Consider the following piece of code showing an algorithm expressed in PAL. The block has one input signal *in*, one output signal *out*, one state s, and three parameters a, b and c.

```
block par
  in : input real;
  out : output real;
  s : state real;
  a : parameter real;
```

```
    b : parameter real;
    c = a * b;

    calculate
    begin
      out := ...
    end calculate;

    update
    begin
      s := ...
    end update;

  end par;
```

Parameters are special variables that can only be assigned from outside the block. When the parameters a and b are assigned new values, a new value for c must also be calculated. The parameters a and b are so called *direct* parameters while c is an *indirect* parameter. Indirect parameters may not be assigned directly by the user.

Assume that the algorithm is executing and using all three parameters. At the same time the user wants to change the value of one or several of the parameters. If the user is allowed to directly assign the parameters, the risk for a non-consistent parameter set is evident, since a change in one direct parameter must propagate to all indirect parameters before the new values safely can be used.

Structure

The solution suggested by this pattern is to have two parameter sets, one that is used by the block algorithm, and another that is used for assigning direct parameters and for calculating indirect parameters. This setup is shown in Figure 7.2. When an assignment operation is finished and the values of all indirect parameters are calculated, the block gets access to the new parameter set through a pointer swap. The involved classes and their relations are shown in Figure 7.3.

Sample Code

The class `AbstractBlock` constitutes a logical block that can be accessed from the surrounding framework. It has two public methods,

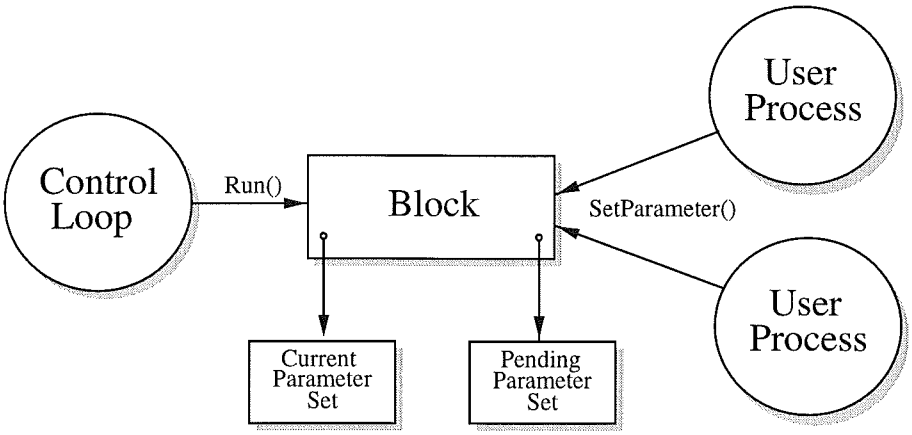


Figure 7.2 The block algorithm is executed by the Control Loop process. Here the algorithm uses the Current Parameter Set. For assigning values to parameters the Pending Parameter Set is used. When an assignment operation is finished and all necessary calculations are made the two sets are swapped.

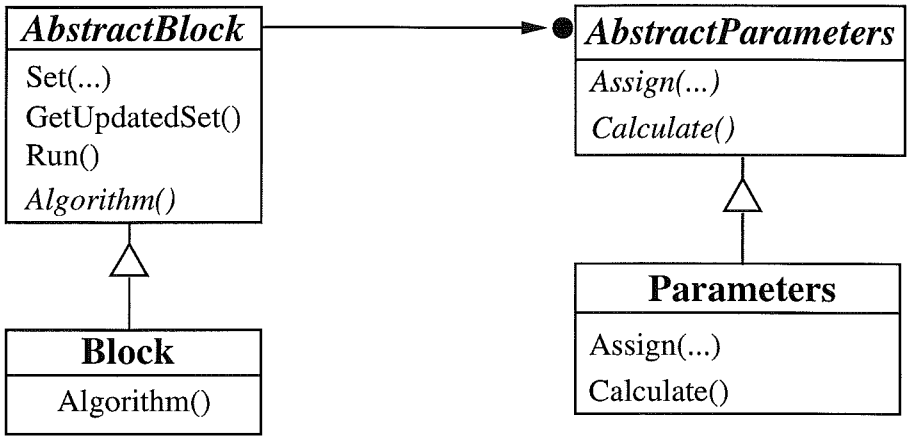


Figure 7.3 The interaction between the user, the framework and the parameters is encapsulated by the super classes `AbstractBlock` and `AbstractParameters`.

`Run` for executing the block algorithm, and `Set` for assigning block parameters. Usually these two methods are accessed from different real-time processes, which are not synchronized. The `Run` method is executed by a high priority process, while the `Set` method is called from low priority processes. It is necessary that the blocking time for the `Run` method is minimized.

```
class AbstractBlock {
 public:
```

```
  int Set(int nrOfPars, ParID *parID, Value *value);
  void GetUpdatedSet();
  void Run();
  virtual void Algorithm() {}

 private:
  int parChanged;
  Event event;
  AbstractParameters *current, *pending;
};
```

The class Parameters is used to encapsulate the parameters of a block. AbstractParameters is an abstract class which simply provides the interface between the block class and its parameters. Each block has two instances of the AbstractParameters class, current, that is used by the Run method and pending, that is used by the Set method. The method Calculate is used for calculating the values of the indirect parameters, based on the new values of the direct parameters.

```
class AbstractParameters {
  Monitor innerMon, outerMon;
 public:
  virtual void Assign(int nrOfPars, ParID *parID, Value *value);
  virtual void Calculate() {};
};
```

The parameters are shared by several processes and must thus be protected. There are two monitors for this. The monitor outerMon is used so that only one low priority process at the time can assign new values. When new values are assigned, the flag parChanged is set, to signal to the Run method that new parameters are available. The monitor innerMon is used to protect this flag. The high priority process executing the Run method may be blocked waiting for innerMon to become free, but since innerMon only contains one assignment statement, the blocking time will be short.

```
void AbstractBlock::GetUpdatedSet()
{
  InterruptMask mask;
  AbstractParameters *tmp;
```

```
  innerMon.Enter();
  if ( parChanged ) {
    tmp = pending;
    pending = current;
    current = tmp;
    parChanged = FALSE;
  }
  innerMon.Leave();
}
```

The algorithm of a block is executed by calling the Run method. Run first calls GetUpdatedSet to get the latest parameter set, and then it calls Algorithm.

```
void AbstractBlock::Run()
{
  GetUpdatedSet();
  Algorithm();
}
```

The class AbstractBlock provides the Set method for the assigning parameters. The arguments are the number of parameters to be assigned, a list of parameter identifiers and a list of values. The monitor outerMon is needed since there may be several user processes trying to assign parameters. If the flag parChanged is true when the Set method is entered, the flag is temporarily set to false. It is thus possible to make several calls in a row to Set.

```
int AbstractBlock::Set(int nrOfPars, ParID *parID, Value *value)
{
  int result;

  outerMon.Enter();
  innerMon.Enter();
  parChanged := FALSE;  // Postpone the use of parameter changes
  innerMon.Leave();     // made previously.

  pending->Assign(nrOfPars, parID, value);
  pending->Calculate();
  innerMon.Enter();
  parChanged = TRUE;
```

```
  outerMon.Leave();

  return result;
}
```

The two super classes AbstractBlock and AbtractParameters handle all the interaction with the environment. An algorithm is added to AbstractBlock in the sub-class Block. The Algorithm method uses the data in the Parameters class.

```
class Block : public AbstractBlock {
 public:
  virtual void Algorithm();
};


void Block::Algorithm()
{
  Parameters *myCurrent = (Parameters *) current;

  // Here comes the algorithm code
  ...
}
```

In the sub-class Parameters the actual parameter variables are added. Two functions for managing them are implemented. The Assign method implements the assignment of one or several parameters. The Calculate method implements the calculation of indirect parameters.

```
class Parameters : public AbstractParameters {
 friend class Block;
 public:
  virtual void Assign(int nrOfPars, ParID *parID, Value *value);
  virtual void Calculate();
 private:
  // Here the parameters are defined
  double par1, par2, par3;
};


void Parameters::Assign(int NrOfPars, ParID *parID, Value *value)
{
```

```
  ...
}

void Parameters::Calculate()
{
  par3 = par1 + par2;
}
```

Known Uses

The basic ideas behind this is pattern are well known [Årzén, 1996]. The sample code here is based on [Nilsson, 1996].

7.3 The Register Idiom

Intent

The register idiom provides a method to linking new code to an existing framework without recompiling the main program.

Motivation

When working with a framework such as the PÅLSJÖ environment users need to add new classes. When the new classes are compiled they need to be integrated with the rest of the code. This could be done by adding the appropriate code in the main program so that the new code will be included during linkage. This approach requires that the user has access to the source code.

When designing PÅLSJÖ we wanted to avoid this. Instead the user should only need to add the names of the new files that should be linked in. To solve this the Register idiom is used.

Structure

The way the Register idiom works is similar to dynamic linking of code. The basic idea is that when a new class is added to the framework it notifies the main program that a new class is available and it also tells the main program how to create an instance of this new class. The class is said to register, when it notifies the framework. The registration

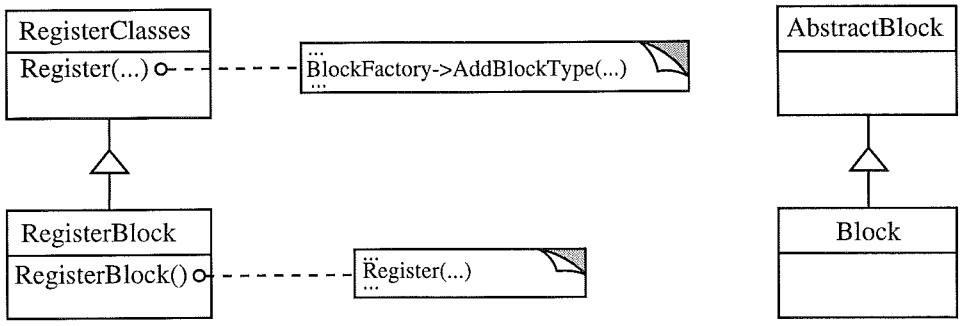


Figure 7.4 The register idiom is implemented with two super classes `RegisterBlock` and `AbstractBlock`.

can be done either on the startup by using static objects, see below, or during run-time. When a class has been registered with the framework, the client may create instances of it through a factory [Gamma *et al.*, 1995].

Sample Code

In the example below the class `Block`, which is inherited from the super class `AbstractBlock` is discussed. The goal is to register the `Block` class with the framework in a simple and convenient way. This is solved here by creating a sub class, called `RegisterBlock`, from `RegisterClasses`. The purpose of this class is to notify the framework of the new sub-class `Block`. This is done by defining a static instance of `RegisterBlock` in the header file. When `RegisterBlock` is allocated the constructor will register the `Block` class with the `BlockFactory`. The `RegisterClasses` class is defined below

```
#include "AbstractBlock.h"

class RegisterClasses {
 public:
  void Register(char *block, AbstractBlock* (*f)());
};
```

In this example there exists one global instance of BlockFactory.

```
#include "RegisterClasses.h"
#include "BlockFactory.h"
#include "AbstractBlock.h"
```

```
extern BlockFactory *blockfactory;

void RegisterClasses::Register(char *block, AbstractBlock* (*f)())
{
  blockfactory->RegisterBlockType(block, f);
}
```

The user defined sub-classes are presented below. The first class is the `Block` class, which should be integrated with the framework. The second class is `RegisterBlock`, of which a static instance is defined.

```
#include "AbstractBlock.h"
#include "RegisterClasses.h"

class Block : public AbstractBlock
{
  public:
  // Here the block methods and attributes are defined
};

static class RegisterBlock : public RegisterClass
{
  public:
    RegisterBlock();
} registerblock_Block;
```

When `registerblock_Block` is allocated, its constructor is executed and calls the `BlockFactory` object. The argument in the call to the `BlockFactory` object is a function pointer to `Create_Block` and the name of the block class. The `BlockFactory` stores the function pointer and the name in a list.

```
 #include "Block.h"

// Here the body of the class Block should be implemented

static BaseBlock *Create_Block() {
  BaseBlock *result = new Block();
  return result;
}
```

```
RegisterBlocks::RegisterBlocks() {
  Register("Block", Create_Block);
}
```

Participants

The Factory [Gamma *et al.*, 1995] pattern is used for creating objects of the new classes.

Related Patterns

The idea behind Singleton [Gamma *et al.*, 1995] is similar to this.

8

Case Studies

8.1 Introduction

In this chapter five case studies are presented. Three of these examples include control design and analysis in combination with PAL code and PCL commands. The idea is to give the reader a full motivation why the controller implementation is structured the way it is. The first example is an implementation of a controller for the inverted pendulum. The inverted pendulum is a good test bench, both from a real-time, and from a control theory point of view. The next example shows how the PÅLSJÖ system is used to control a industrial robot. Case study number three deals with the implementation of a hybrid controller for a double tank system. Both control design and analysis are given, together with PAL code. The examples with the pendulum and the double tanks demonstrate how PAL is used for implementing algorithms that are partially sequential and partially periodic. An adaptive RST-controller for a mechanical servo system is designed and implemented in Section 8.5. The use of dimension variables, polynomials, local and external procedures and functions are the main objectives with this example. In the last section the implementation of a fault tolerant controller scheme is discussed. Only the outline for the PAL and PCL code is given here.

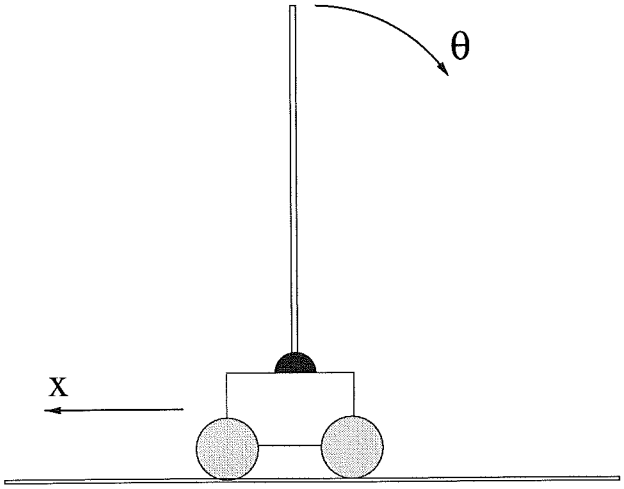


Figure 8.1 The pendulum on a cart.

8.2 Inverted Pendulum

Controlling the inverted pendulum is a classical problem in control laboratories. The pendulum process provides a suitable test-bench for control algorithms as well as for real-time control applications. Since the process is both nonlinear and unstable it gives rise to a number of interesting problems. Its unstable nature also makes it a good process for testing real-time controllers, since if the timing fails, the pendulum is likely to fall down. The goal with the control is to bring the pendulum from the downward position to the upward position and keep it stabilized.

This section describes the design and implementation of a controller for the pendulum. For estimating the angular velocity a nonlinear observer is proposed. First the process model and the controller structure are presented. In the next section the the nonlinear observer for the angular velocity is introduced. Design, analysis and simulations for the observer are presented. Finally the PAL code for the controller is shown. This implementation shows how a periodic algorithm, like the observer, may be combined with a sequential algorithm, in this case the controller.

The Process

The inverted pendulum is shown in 8.1. The equation of motion of the

inverted pendulum is given by

$$\frac{d^2\theta}{dt^2} = \omega_0^2 \sin\theta + u/g\omega_0^2 \cos\theta$$

where ω_0 is the natural frequency for small oscillations around the stable equilibrium, g is the natural gravity, and u is the acceleration of the pivot. Introduce the state variables

$$\begin{aligned} x_1 &= \theta \\ x_2 &= \frac{1}{\omega_0}\frac{d\theta}{dt} \end{aligned}$$

The equation of motion can then be written as

$$\begin{aligned} \frac{dx_1}{dt} &= \omega_0 x_2 \\ \frac{dx_2}{dt} &= \omega_0 \sin x_1 + u/g\omega_0 \cos x_1 \end{aligned} \tag{8.1}$$

Since the input signal to the process is the acceleration, the equation for the cart simply becomes an integrator. If the position of the cart is denoted x_3 then

$$\ddot{x}_3 = u \tag{8.2}$$

The Controller

The controller is designed to first swing up the pendulum and then stabilize it in the upright position. This is done using a hybrid controller which consists of three sub-controllers: one for the swing-up phase, one for the catching of the pendulum, and one for stabilizing the pendulum. The switching logic for this hybrid controller is shown as a Grafcet in Figure 8.2.

Swinging up the Pendulum

To swing up the pendulum an energy approach is used. The idea is to pump the right amount of energy into the pendulum, so that it will

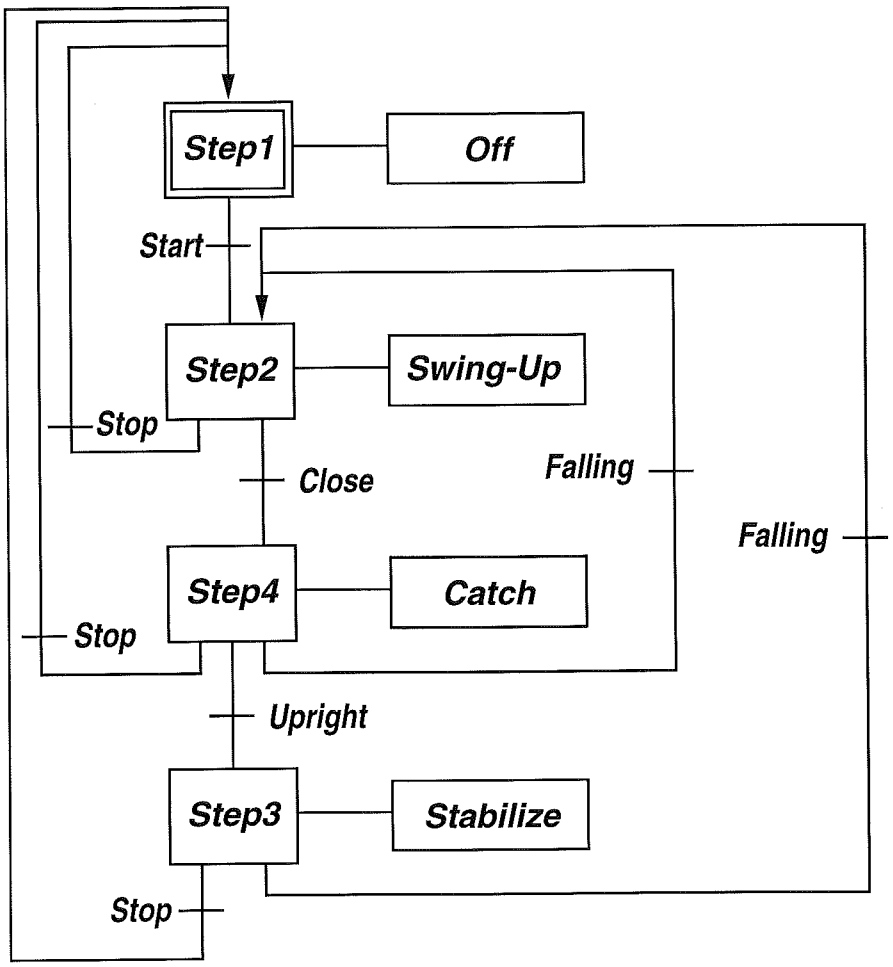


Figure 8.2 The Grafcet which describes the switching rules for the hybrid pendulum controller.

move to the upward position. The algorithm used is found in [Wiklund *et al.*, 1993], [Åström and Furuta, 1996]. The control law is the following

$$u = sat_{ng} k(E - E_0) sign(\dot{\theta} \cos \theta) \tag{8.3}$$

where E_0 is the desired energy, and k is a design parameter. Here E_0 corresponds to the pendulum being in the upright position.

Catching the Pendulum

When the pendulum is close to the upright equilibrium the catching controller is switched in. The task for this controller is to, as smoothly

as possible, hand over the control from the swing-up mode to the balance mode. The controller used in the experiments is the following state-feedback controller

$$u = -l_1 x_1 - l_2 x_2 \tag{8.4}$$

This controller stabilizes the pendulum, but takes no concern to the position of the cart.

Balancing the Pendulum

To balance the pendulum at the upright equilibrium, another state feedback controller is used. The control law is the following

$$u = -l_1 x_1 - l_2 x_2 - l_3(x_3 - x_r) - l_4 x_4 \tag{8.5}$$

where x_r is the desired position for the cart. The design of the controller is based on the linearized model for the upright position.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & \omega_0 & 0 & 0 \\ \omega_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_0/g \\ 0 \\ 1 \end{bmatrix} u \tag{8.6}$$

The controller parameters are calculated using pole-placement design.

Measurement Values

There are only two measurement values available from the physical plant. These are the position of the cart, x_3, and the angle of the pendulum x_1. This means that the velocity of the cart, x_4 , and the angular velocity x_2 must be estimated. Creating a filter for estimating x_4 is trivial, since it is a linear system. Finding the corresponding filter for estimation of x_2 is harder due to the nonlinearities. In order to implement the control laws discussed above, good values for all the states are necessary. To get a good value of the angular velocity a nonlinear observer is needed. The design of such an observer is discussed in the next section.

Designing an Observer

For the estimation of the angular velocity a nonlinear observer is proposed. The nonlinear observer has the same structure as a linear observer but with the linear model replaced by a nonlinear model. The need for a nonlinear observer for the inverted pendulum becomes evident when implementing the swing-up strategy, since it involves estimation of the angle and the angular velocity at all positions, and hence a linear observer cannot be used to accomplish this task. The pendulum can make a complete revolution which makes the system highly nonlinear. Conditions for stability of the nonlinear observer are given and a design based on linearized analysis is presented.

Since ω_0 in Equation 8.1 is a scaling factor which can be removed by a proper choice of time scale, the case with $\omega_0 = 1$ will be considered. It is trivial to transform back later when the result is obtained. To further simplify calculations, the control signal u is given in normalized acceleration. It is assumed that the angle θ is measured with noise and that it is desired to filter the noisy signal and to generate an estimate of the velocity $d\theta/dt$. Let the output be

$$y = x_1 + n$$

where n is measurement noise. A reasonable structure of a nonlinear observer is

$$\begin{aligned} \frac{d\hat{x}_1}{dt} &= \hat{x}_2 + k_1(x_1 - \hat{x}_1 + n) \\ \frac{d\hat{x}_2}{dt} &= \sin\hat{x}_1 + u\cos\hat{x}_1 + k_2(x_1 - \hat{x}_1 + n) \end{aligned} \tag{8.7}$$

Notice that the observer has a very simple structure. It combines a model for the nonlinear dynamics with a simple feedback from the measured angle.

Introducing the estimation error $\tilde{x} = \hat{x} - x$ gives

$$\begin{aligned} \frac{d\tilde{x}_1}{dt} &= -k_1\tilde{x}_1 + \tilde{x}_2 + k_1 n \\ \frac{d\tilde{x}_2}{dt} &= \sin\hat{x}_1 - \sin x_1 + u(\cos\hat{x}_1 - \cos x_1) - k_2\tilde{x}_1 + k_2 n \end{aligned}$$

This can be written as

$$\begin{aligned} \frac{d\tilde{x}_1}{dt} &= -k_1\tilde{x}_1 + \tilde{x}_2 + k_1 n \\ \frac{d\tilde{x}_2}{dt} &= -k_2\tilde{x}_1 + v + k_2 n \end{aligned} \tag{8.8}$$

where

$$v = 2\sin\frac{\tilde{x}_1}{2}\Big(\cos(x_1 + \frac{\tilde{x}_1}{2}) - u\sin(x_1 + \frac{\tilde{x}_1}{2})\Big) \tag{8.9}$$

Linearization

To get some insight into the properties of the system, the error equation (8.8) will be linearized around the operating condition θ_0. This gives

$$\begin{aligned} \frac{d\tilde{x}_1}{dt} &= -k_1\tilde{x}_1 + \tilde{x}_2 + k_1 n \\ \frac{d\tilde{x}_2}{dt} &= -(k_2 + \alpha)\tilde{x}_1 + k_2 n \end{aligned} \tag{8.10}$$

where $\alpha = -\cos\theta_0$. Notice that $\alpha = -1$ when the pendulum is standing up and that $\alpha = 1$ when it is hanging down.

Robustness

First the sensitivity to parameter variations is considered. This is done by investigating how the roots of the characteristic equation of the filter change over the range of operating conditions. The characteristic equation of the filter becomes

$$s^2 + k_1 s + k_2 + \alpha = 0$$

Assume that the design gives the following the characteristic equation

$$s^2 + 2\zeta_0\omega_n s + \omega_n^2 = 0$$

in the nominal case $\alpha = 0$. The characteristic equations for the extreme cases are

$$\begin{aligned} s^2 + k_1 s + k_2 + 1 &= 0 \\ s^2 + k_1 s + k_2 - 1 &= 0 \end{aligned}$$

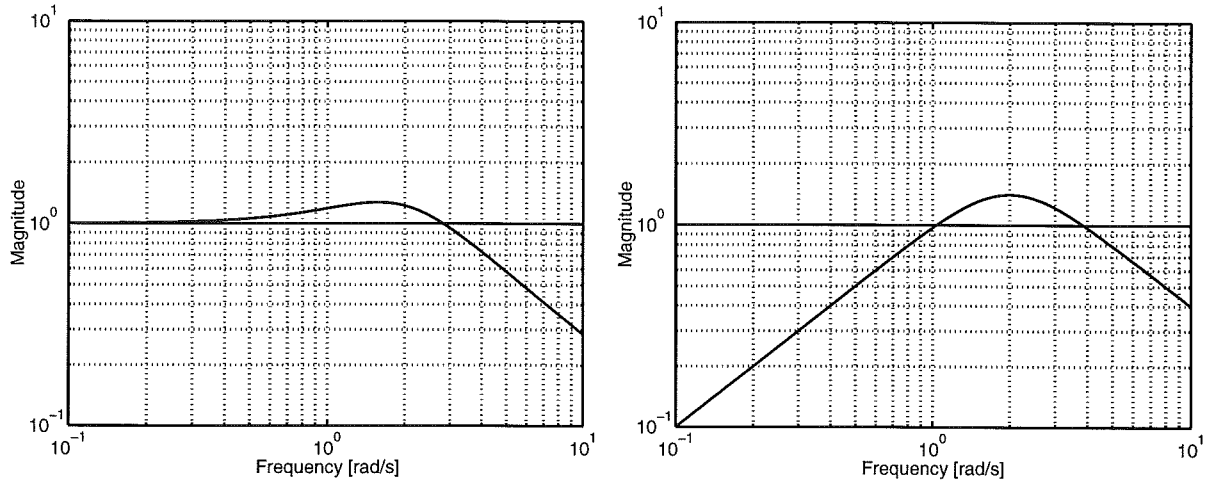


Figure 8.3 Bode diagrams for the transfer functions $G_\theta(s)$ (left) and $G_v(s)$ (right) in Equation (8.11), which describe how measurement noise propagates to the estimates of the angle and the angular velocity. The values of the design parameters are $\zeta = 0.707$ and $\omega_n = 2$. The transfer function $|G_v(s)|$ has a resonance peak of 1.414 at $\omega = \omega_n$.

With a relative damping ζ_0 in the nominal case, the following equation is obtained

$$\zeta = \zeta_0 \frac{\omega_n}{\sqrt{\omega_n^2 \pm 1}}$$

in the extreme cases. To get a system whose behavior is invariant over the operating range it is apparently advantageous to choose the parameter ω_n as large as possible. Choosing $\omega_n = 2$, i.e. the filter twice as fast as the nominal system, implies $\zeta = 0.89\zeta_0$ and $\zeta = 1.15\zeta_0$, respectively.

Performance

A key issue is how efficient the filter is in reducing the noise from the angle sensor. To explore the transfer functions from measurement noise to the errors in the estimates of the angle and the angular velocity, are calculated.

It follows from Equation (8.10) that the transfer functions are

$$G_\theta(s) = \frac{k_1 s + k_2}{s^2 + k_1 s + k_2 + \alpha}$$
$$G_v(s) = \frac{k_2 s - \alpha k_1}{s^2 + k_1 s + k_2 + \alpha}$$

For $\alpha = 0$ these transfer functions become

$$\begin{aligned} G_\theta(s) &= \frac{k_1 s + k_2}{s^2 + k_1 s + k_2} = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \\ G_v(s) &= \frac{k_2 s}{s^2 + k_1 s + k_2} = \frac{\omega_n^2 s}{s^2 + 2\zeta\omega_n s + \omega_n^2} \end{aligned} \tag{8.11}$$

It follows from these equations that there is a roll-off for high frequency measurement noise with a slope of -1 for both transfer functions. The frequency where the roll-off begins is essentially determined by the parameter ω_n. The transfer function G_θ has low pass character and G_v has band pass character. The transfer function G_v has a resonance peak at $\omega = \omega_n$ which is given by

$$|G_v(i\omega_n)| = \frac{\omega_n}{2\zeta}$$

This shows that too high a value of ω_n gives a high noise level in the velocity estimate. A comparison with the robustness analysis shows the traditional trade off, between performance and robustness. To make a specific design it is highly desirable to determine the spectrum of the measurement noise.

Bode diagrams of the transfer functions are shown in Figure 8.3. The figure indicates that the observer gives a good estimate of the derivative for frequencies up to $\omega_0 = 1$.

The angle is typically measured by a potentiometer or an encoder. Additional noise is also generated from slip-rings that are often used. Figure 8.4 shows a realization of the angle measurement from a laboratory process with a potentiometer. The signal is filtered with an analog anti-aliasing filter and sampled at 200 Hz. A spectrum analysis shows that the measurement noise is white over a wide range.

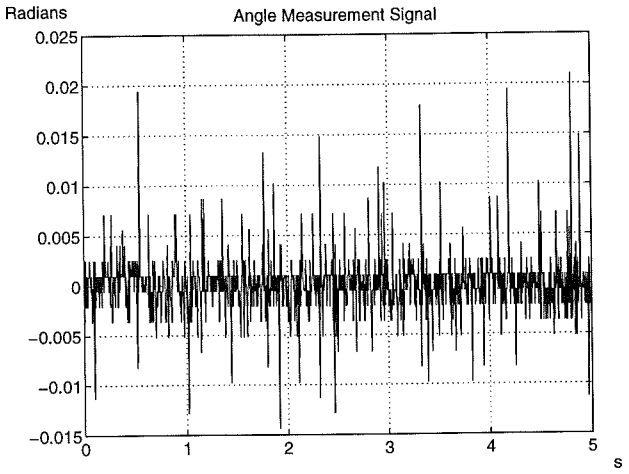


Figure 8.4 A sequence of the measurement noise taken from a laboratory pendulum where the angle is measured with a potentiometer.

Stability

Although the linearized analysis gives insight it does not guarantee stability of the nonlinear system (8.8). To analyze the stability of the nonlinear system the theory of input-output stability of nonlinear systems is applied, see [Desoer and Vidyasagar, 1975] and [Khalil, 1992].

Pure Filtering

First the pure filtering problem, i.e. when there is no control signal, is discussed. The signals x_1 and x_2 can be considered as time functions and the function v in Equation (8.9) becomes

$$v = 2 \sin \frac{\tilde{x}_1}{2} \cos \left(x_1(t) + \frac{\tilde{x}_1}{2} \right) \tag{8.12}$$

The system described by Equations (8.8) and (8.9) can be regarded as an interconnection of a linear system with the transfer function

$$G(s) = \frac{1}{s^2 + k_1 s + k_2}$$

and the nonlinear time-varying function $v(\tilde{x}_1, t)$ given by Equation (8.12). See Figure 8.5. The following result is now established.

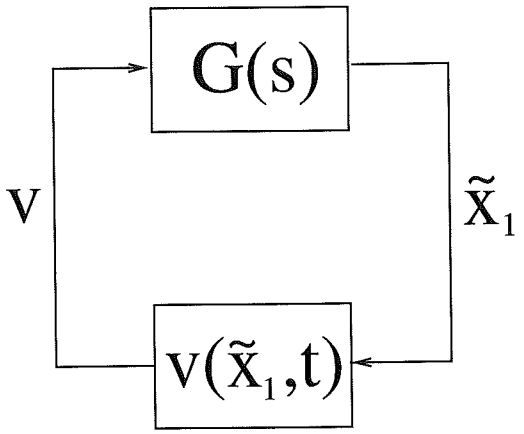


Figure 8.5 Block diagram of the observer. The upper block represents the linear part of the observer and the lower block represents the nonlinear part.

THEOREM 8.1
Assume that the signal space is L_{2e}, and that the control signal is zero, then the system (8.8) is input-output stable if

$$k_2 > \begin{cases} k_1^{-2} + k_1^2/4 & \text{if } k_1 < \sqrt{2}, \\ 1 & \text{if } k_1 \geq \sqrt{2} \end{cases} \tag{8.13}$$

□

Proof

The system in Figure 8.5 is a standard configuration where the small gain theorem can be used, see [Desoer and Vidyasagar, 1975] and [Khalil, 1992]. For this purpose the gains of the linear and the nonlinear subsystems are calculated.

Since the cosine function is bounded by one it follows from Equation (8.12) that

$$|v(\tilde{x}_1, t)| \leq 2|\sin\frac{\tilde{x}_1}{2}| \leq |\tilde{x}_1|$$

The gain of the nonlinear block is thus at most one. The small gain theorem, [Khalil, 1992], then implies that the closed loop system is stable if the gain of the linear system is less than one.

To calculate the gain of the linear block it is observed that

$$\frac{1}{G(i\omega)} = k_2 - \omega^2 + ik_1\omega \tag{8.14}$$

Hence

$$\begin{aligned}\left|\frac{1}{G(i\omega)}\right|^2 &= \omega^4 + (k_1^2 - 2k_2)\omega^2 + k_2^2 \\ &= (\omega^2 - k_2 + k_1^2/2)^2 - k_1^4/4 + k_1^2 k_2\end{aligned}$$

If $k_1^2 < 2k_2$ the maximum of $|G(i\omega)|$ is obtained for $\omega = \sqrt{k_2 - k_1^2/2}$, hence

$$\max G(i\omega) = \frac{1}{\sqrt{k_1^2 k_2 - k_1^4/4}}$$

Requiring that the gain of the linear block is less than one gives

$$k_1^2 k_2 - k_1^4/4 > 1$$

or

$$k_2 > k_1^{-2} + k_1^2/4$$

If $k_1^2 \geq 2k_2$ the maximum is obtained for $\omega = 0$, hence

$$\max G(i\omega) = \frac{1}{k_2}$$

and the condition that the gain of the linear block is less than one becomes $k_2 > 1$, which completes the proof. The values of the observer gains which give a stable closed loop system are shown in Figure 8.6. Note that the stability conditions for the linearized case are $k_1 > 0$, $k_2 > 1$. This shows that the stability conditions for the nonlinear observer are not very conservative and that there is considerable freedom for the designer. This verifies the intuitive insight that it is easy to obtain a low gain of the nonlinear system simply by choosing high values of the observer gains. The linearized analysis in Section 8.2 does however indicate that there are disadvantages in using too high gains.

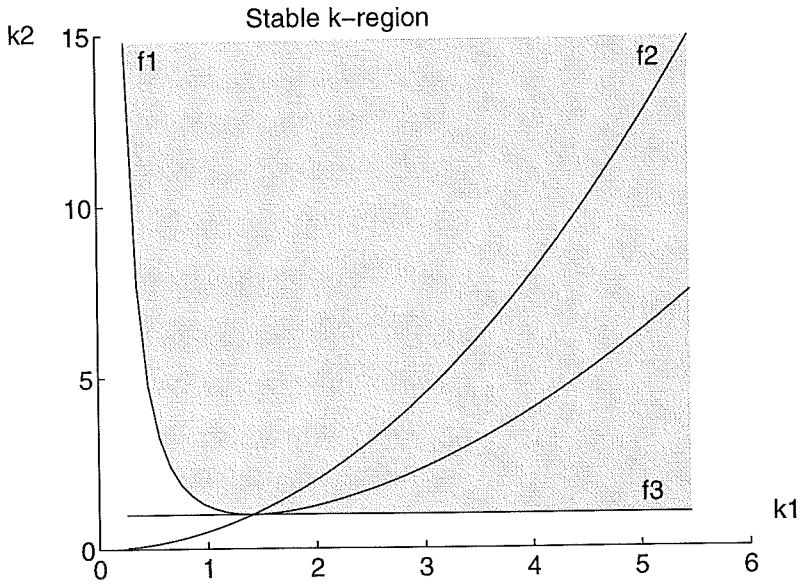


Figure 8.6 The shaded area marks the k-values, for which the resulting filter will be stable. The area is bounded by the following three curves: $f_1(k_1) = k_1^{-2} + k1^2/4$, $f_2(k_1) = k_1^2/2$, and $f_3(k_1) = 1$.

Filtering with Feedback

So far a free observer where the control signal u was zero has been discussed. When a feedback is introduced the control u becomes a function of the state. It is straightforward to extend the analysis to this case. The results will in general depend on the nature of the feedback. It is highly desirable to obtain conditions that do not depend on the detailed structure of the control law. Such results can be obtained if the reasonable assumption that the acceleration of the pivot is bounded is made. This leads to the following theorem

THEOREM 8.2
Assume that the signal space is L_{2e}, and that $|u| \leq u_{max}$. Let $\beta = \sqrt{1 + u_{max}^2}$, then the system (8.8) is input-output stable if

$$k_2 > \begin{cases} \beta^2 k_1^{-2} + k_1^2/4 & \text{if } k_1 < \sqrt{2\beta}, \\ \beta, & \text{if } k_1 \geq \sqrt{2\beta} \end{cases} \tag{8.15}$$

□

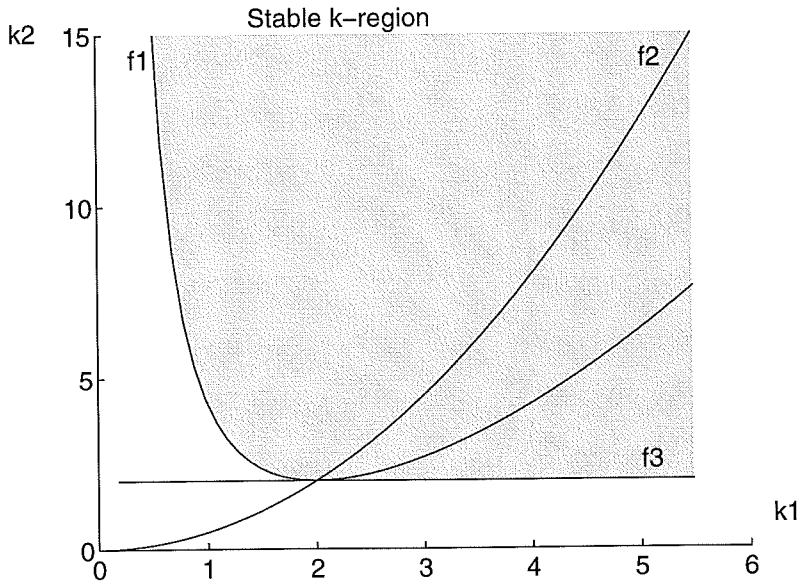


Figure 8.7 The shaded area marks the k-values, for which the resulting observer becomes stable. The area is bounded by the following three curves: $f_1(k_1) = \beta^2/k_1^2 + k1^2/4$, $f_2(k_1) = k_1^2/2$, and $f_3(k_1) = \beta$. β is equal to two in the figure.

Proof

When the control signal is different from zero it follows from Equation (8.9) that the nonlinear block is characterized by

$$v = 2 \sin \frac{\tilde{x}_1}{2} \left(\cos \left(x_1(t) + \frac{\tilde{x}_1}{2} \right) - u \sin \left(x_1(t) + \frac{\tilde{x}_1}{2} \right) \right)$$

Since

$$|\cos \phi + u \sin \phi| \leq \sqrt{1 + u_{max}^2} = \beta$$

it follows that the gain of the nonlinear block is bounded by $\beta = \sqrt{1 + u_{max}^2}$. The calculation of the gain of the linear block given in the proof of Theorem 1 now leads to the results and completes the proof. The observer parameters that give a stable closed loop system are illustrated in Figure 8.7. A comparison with Figure 8.6 shows the additional restrictions on the observer gains that are introduced because of the feedback.

Simulation of the Observer

To illustrate the result two simulations of the system are presented. The normalized system has been used in the simulations. For the nominal case $\alpha = 0$, $\zeta = 0.707$ is chosen. To have guaranteed stability with $\zeta = 0.707$, ω_0 must be larger than 1. To have some margin $\omega_0 = 2$ is chosen. This means that the observer is about twice as fast as the natural frequency of the pendulum. The filter gains are $k_1 = 2.818$ and $k_2 = 4$. Figure 8.8 shows a simulation where the pendulum performs an oscillatory motion starting from an almost upright position. Measurement noise has been simulated by a sinusoid with frequency $\omega = \omega_0$, which is the most sensitive case. The amplitude of the measurement noise in the simulation is one degree. A sample of the real measurement noise is shown in Figure 8.4. The assumption that the noise is approximately one degree is thus reasonable. Notice that the estimates of the angle and the angular velocity settle quickly and that the system is not very sensitive to the noise.

If instead $\omega_0 = 0.5$ is chosen, the condition in Theorem 1 is violated, and the resulting observer will become much more sensitive to the nonlinearities. A simulation of this case shows that the estimator does not converge.

Experiments with Pure Filtering

A number of experiments have been performed on a laboratory process. The system used is a version of the Furuta pendulum. In this pendulum the pivot is rotated in the horizontal plane, see [Furuta *et al.*, 1991]. The position of the pendulum was measured with a potentiometer and the signals were transmitted through slip rings. An anti-aliasing filter was used before the signals were sampled. The pendulum has a natural frequency $\omega_0 = 6.3$ rad/s. The observer was implemented by approximating the differential equations by finite differences. The sampling period used was 5 ms. The Furuta pendulum can be approximated by Equation (8.1) provided that the rotation of the pivot in the horizontal plane is not too large.

Figure 8.9 shows estimates of the angular velocity in an experiment where the pendulum swings freely after it is released from the upper position. The dashed line shows the output of the nonlinear filter and the full line shows the estimate obtained by filtering the position sig-

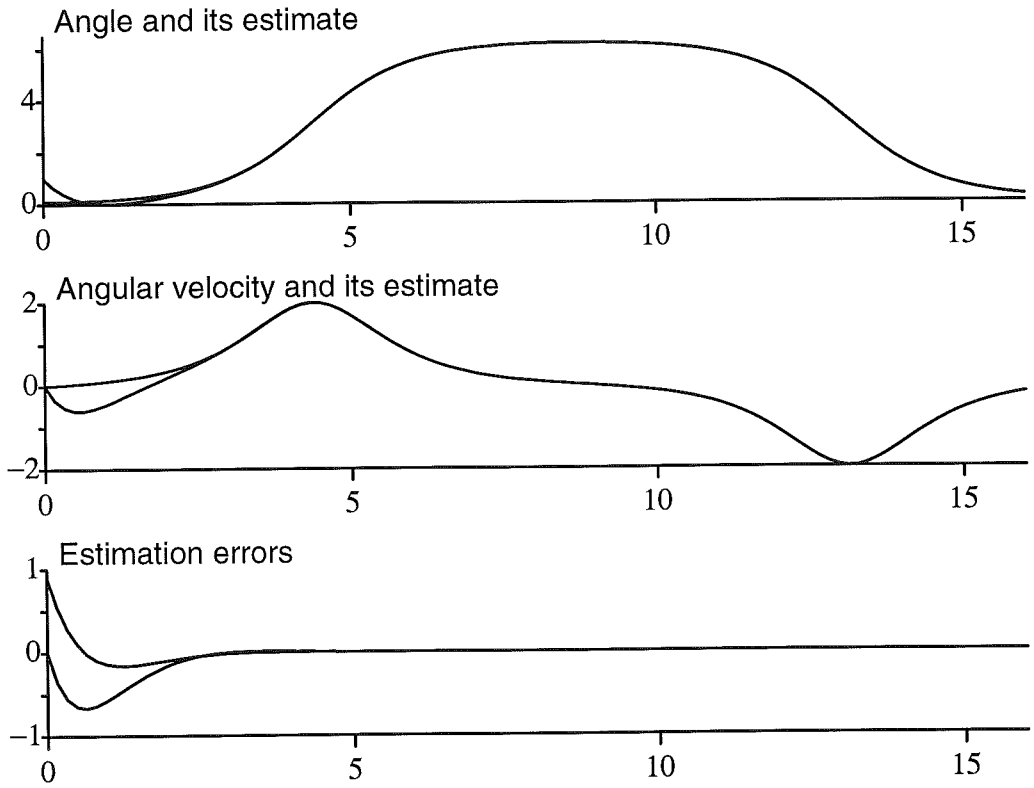


Figure 8.8 A simulation of the observer with $\omega_0 = 2$. This choice of ω_0 will as expected give a stable system.

nal with the filter $G(s) = s/(1 + s/N)$. The filter shows clearly the advantage of using the nonlinear observer. As seen in the figure the low-pass filtered derivative is more sensitive to noise, has longer delay and has a gain that is less than one. The attenuation of the filtered derivative is approximately $\sqrt{1 + (6.3/10)^2} \approx 1.2$.

Experiments with Feedback Control

Finally the results from experiments where the observer is used for feedback control are shown. Figure 8.10 shows a simulation and Figure 8.11 shows real data from a swing-up experiment. The pendulum is started in the downward position. A swing-up controller is used to move the pendulum to its upward equilibrium, where a linear feedback controller is used to stabilize the pendulum. In both the simulation and the real case the observer was initiated to start in the downward position. In the simulations measurement noise is approximated with a high frequency sinusoid. Notice the good agreement between ex-

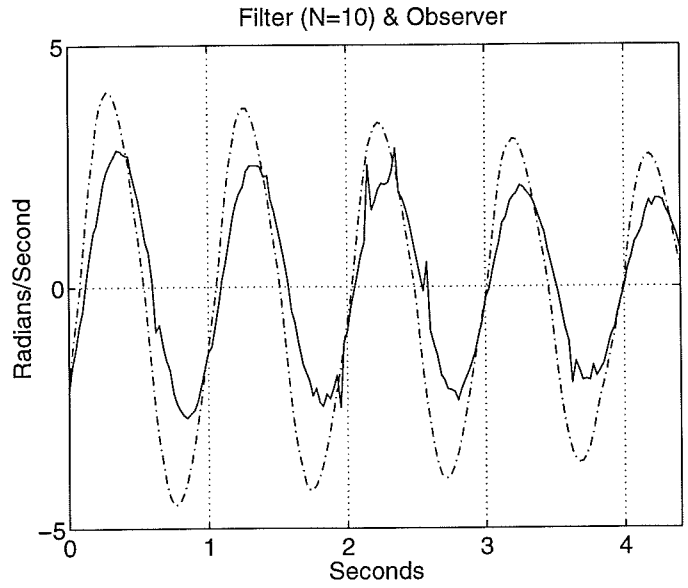


Figure 8.9 Two curves taken from the real process; the dashed line is the angular velocity from the filter described in this section, and the solid line is the theta measurement fed through the filter $G(s) = s/(1+s/N)$, with $N = 10$.

periment and simulation and the good performance of the nonlinear observer.

PAL

The PAL code for the pendulum controller is divided into one sequential part and one periodic part. The sequential part is described as a Grafcet with four steps, see Figure 8.2. In the initial step the output signal is set to zero. In step 2 the swing-up algorithm is switched in, and when the pendulum comes close to the upright position step 3 becomes active. Finally when the pendulum has reached the upright position step 4 becomes active.

EXAMPLE 8.1

```
module Pendulum;
  function cos(r : input real) :real; external "cos";
  function sin(r : input real) :real; external "sin";
  block PendulumController
    x, th : input real;
```

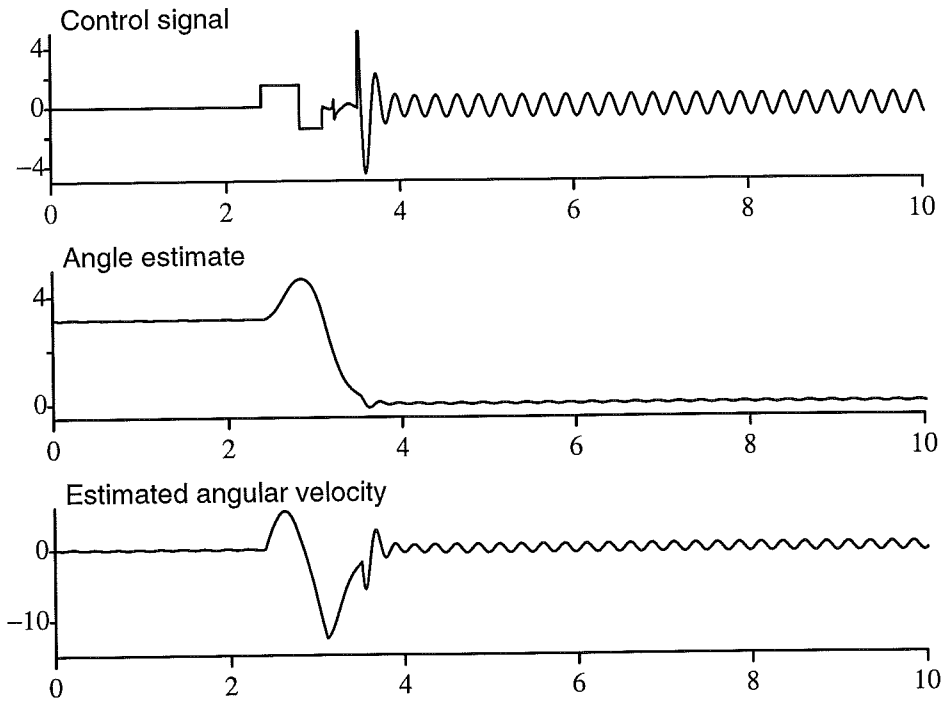


Figure 8.10 A simulation of swinging up and stabilizing the pendulum using the non-linear observer. The simulation starts with the pendulum in the downward position. The observer is initiated to start in this position.

$v := 0.0, u := 0.0$: **output real**;
$xold := 0.0$: **real**;
$xhat := 0.0, dxhat := 0.0, xhatold := 0.0, dxhatold := 0.0$: **real**;
$x1hat := 0.0, x2hat := 0.0, dx1hat := 0.0, dx2hat := 0.0$: **real**;
$thetaHat, dthetaHat$: **real**;
$uold := 0.0, xr, Enorm := -2.0$: **real**;
$on :=$ **false** : **boolean**;
$a1, a2, a3, a4, b1$: **parameter real**;
$k, k1, k2, k3, k4, l1, l2, l3, l4$: **parameter real**;
$w0 := 6.3$: **parameter real**;
$minTh, maxTh, n$: **parameter real**;
h : **sampling interval**;

calculate
begin
$dx1hat := w0 * x2hat + k1 * (th - x1hat)$;
$dx2hat := w0 * sin(x1hat) - w0 * uold/9.81 * cos(x1hat) + k2 * (th - x1hat)$;
$x1hat := x1hat + dx1hat * h$;

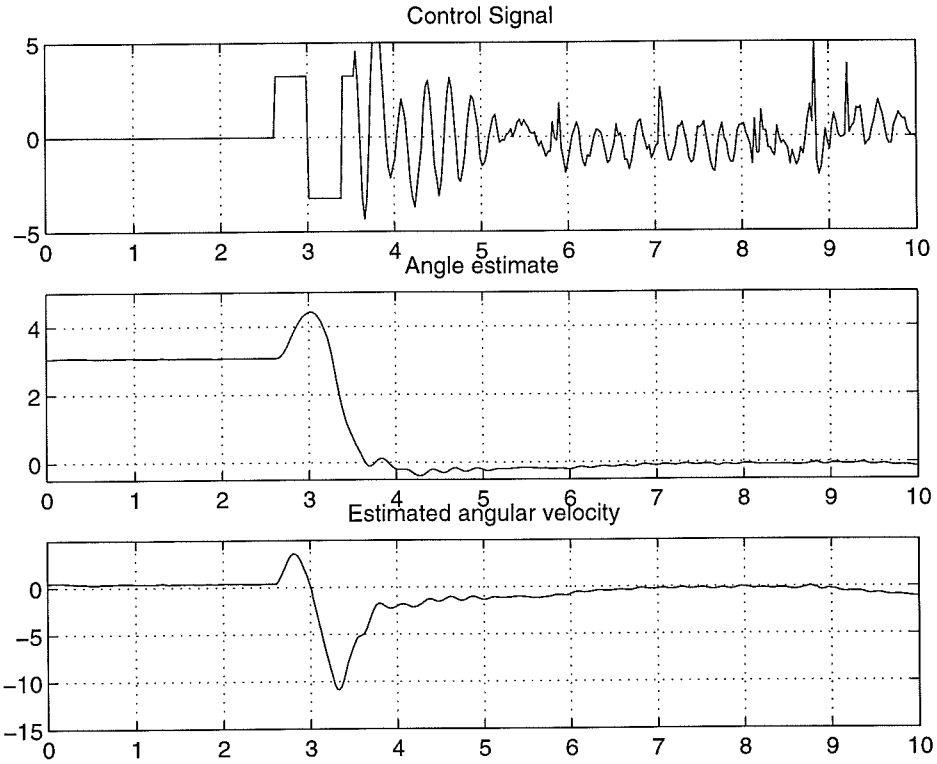


Figure 8.11 Plots from the swing-up and stabilization of a real pendulum process. The pendulum starts in the downward position.

$x2hat := x2hat + dx2hat * h;$
$xhat := a1 * xhatold + a2 * dxhatold + b1 * uold + k3 * (xold - xhatold);$
$dxhat := a3 * dxhatold + a4 * uold + k4 * (xold - xhatold);$
$thetaHat := x1hat;$
$dthetaHat := w0 * x2hat;$
end calculate;

update
begin
$xhatold := xhat;$
$dxhatold := dxhat;$
$xold := x;$
$uold := u;$
end update;

action off;
begin
$u := 0.0;$
end off;

action swing;
begin
$Enorm := 0.5 * dthetaHat * dthetaHat/(w0 * w0) + cos(th) - 1.0;$
$u := -sign(dthetaHat * cos(th)) * sat(n * 9.81, k * Enorm);$
end swing;

action catch;
begin
$xr := x;$
$u := -l1 * th - l2 * dthetaHat;$
end catch;

action stabilize;
begin
$u := -l1 * th - l2 * dthetaHat - l3 * (x - xr) - l4 * dxhat;$
end stabilize;

initial step step1;
activate off;
end step1;

step step2;
activate swing;
end step2;

step step3;
activate catch;
end step3;

step step4;
activate stabilize;
end step4;

transition from *step*1 **to** *step*2 **when** *on*;

transition from *step*2 **to** *step*3 **when**
$th > minTh$ **and** $th < maxTh$ **or** $th < -minTh$ **and** $th > -maxTh$;

transition from *step*3 **to** *step*4 **when** $th > -minTh$ **and** $th < minTh$;

transition from *step*4 **to** *step*2 **when** $th < -maxTh$ **or** $th > maxTh$;

transition from *step*3 **to** *step*2 **when** $th < -maxTh$ **or** $th > maxTh$;

transition from *step*2 **to** *step*1 **when not** *on*;

transition from *step*3 **to** *step*1 **when not** *on*;

transition from *step*4 **to** *step*1 **when not** *on*;

```
    function sat(max : input real; value : input real) : real;
    begin
      if value > max then
        result := max;
      elsif value < -max then
        result := max;
      elsif value < 0.0 then
        result := -value;
      else
        result := value;
      end if;
    end sat;

    procedure start();
    begin
      on := true;
    end start;

    procedure stop();
    begin
      on := false;
    end stop;

    function sign( r : real) : real;
    begin
      if r < 0.0 then
        result := -1.0;
      else
        result := 1.0;
      end if;
    end sign;

  end PendulumController;

end Pendulum.
```

□

PCL

The controller is configured by the PCL-script in Example 8.2. Two modules `StandardBlocks` and `Pendulum` are imported. See Appendix A for an overview of the `StandardBlocks` library. Then all blocks are allocated and connected in one atomic operation. It is important that this is done in an atomic operation, because the run-time system expects to

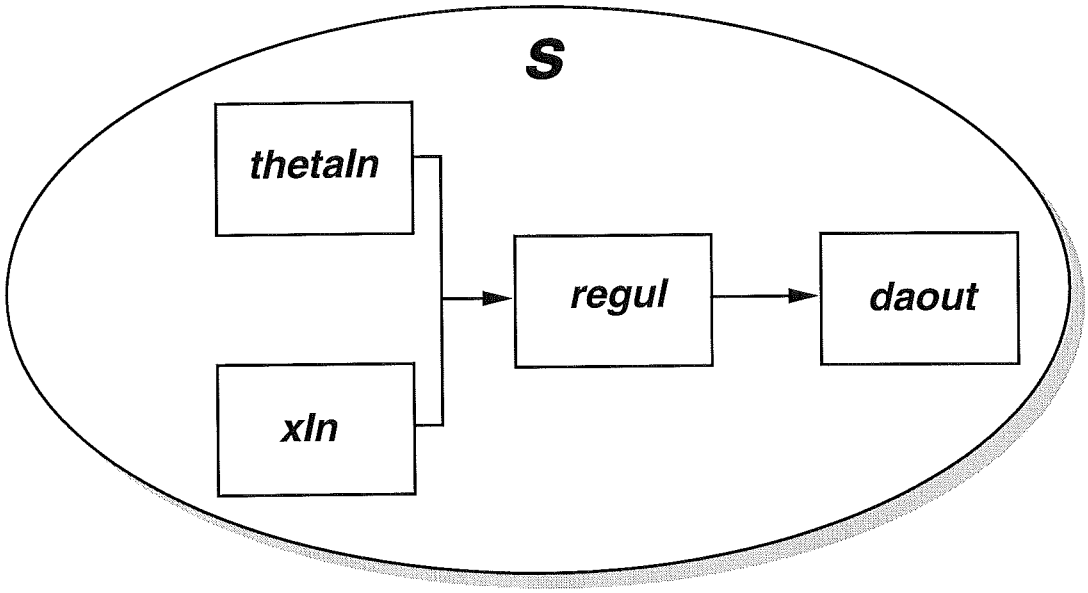


Figure 8.12 The system created by the PCL code in Example 8.2. The four algorithmic blocks `thetaIn`, `xIn`, `regul` and `daout` are executed by the `Periodic` block `s`.

always have valid configurations, and blocks with unconnected input signals are not valid. This means that if for example the `PendulumController` was created outside an atomic operation, the run-time system would give an error message, saying that the input signals were not connected. Finally in the PCL-script, three variables are made available for export using the `show` command.

EXAMPLE 8.2

```
use Pendulum
use StandardBlocks

{ s = new Periodic
  s.xIn = new ADIn
  s.xIn.channel = 0
  s.thetaIn = new ADIn
  s.thetaIn.channel = 1

  s.regul = new PendulumController
  s.daout = new DAOut
  s.xIn.out -> s.regul.xdirect
  s.thetaIn.out -> s.regul.thdirect
```

```
  s.regul.v -> s.daout.in
  s.tsamp = 0.005
}

s ! start
show s.regul.v
show s.regul.th
show s.regul.dthetaHat
s ! connect
```

□

Conclusions

In this section the implementation and design of a controller for the inverted pendulum was presented. First the process model and the controller design was shown. In order to get good estimates of the angular velocities a nonlinear observer was proposed. Design and analysis for this observer was presented together with some simulations. Finally the implantation was presented with PAL and PCL code.

8.3 A Robot Controller

In this section PÅLSJÖ is used for controlling an industrial robot. The robot swings up and stabilizes an inverted pendulum, similar to the one described in the previous section. The robot holds the shaft of the pendulum using a gripper that is attached to the fourth joint, see Figure 8.14. Joint one is then used for swinging up and balancing the pendulum. The ABB Irb-2000 industrial robot used in the experiments is shown in Figure 8.13.

The Controller

The controller is the same as the one described in Section 8.2. In that case the input signal to the process was the acceleration of the cart. In the robot setup the input signal to the process is the acceleration of the pivot point, i.e. the point where the pendulum is attached to the robot. While the cart moves along a straight line, in the robot case the pivot point moves along a circular path. This fact is not considered in

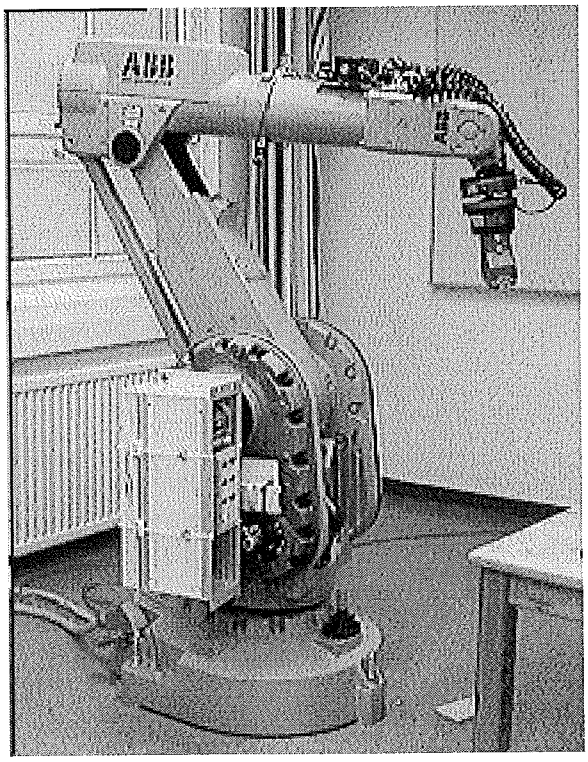


Figure 8.13 The ABB Irb2000 robot used in the pendulum experiment.

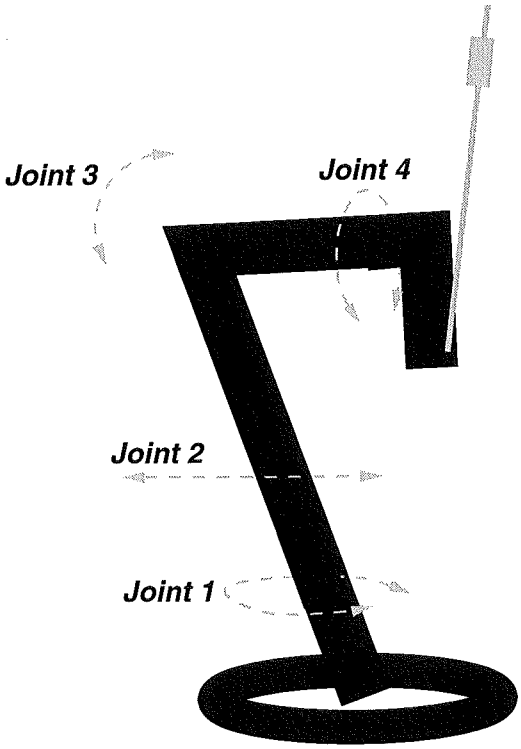


Figure 8.14 The pendulum is held by the gripper on the fourth joint.

the implementation, but as long as the speed of the robot arm is low, the influence from Corioli forces may be neglected. The output from the control block is the desired acceleration for the pivot. This signal may not be sent directly to the robot. Instead it is sent to the PAL block `RobotTrajectory`, which converts the acceleration to reference values for the low level controllers. The low level controllers execute at a speed of several kHz.

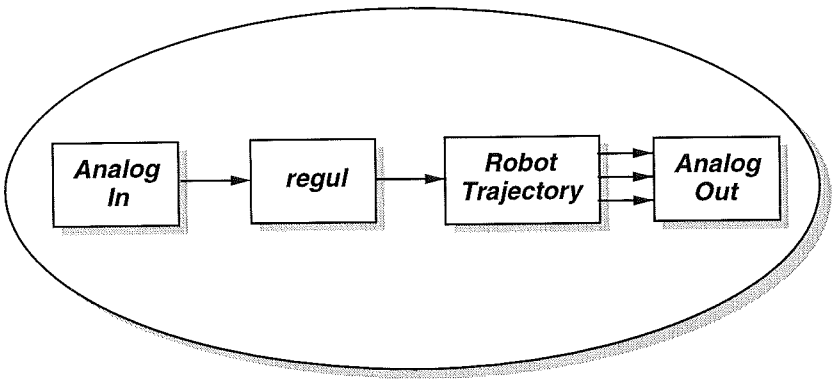


Figure 8.15 The PAL blocks used for controlling the robot. The `regul` is defined in previous section.

In Figure 8.15 the `Periodic` block used for controlling the robot is shown. The actual control algorithm is located in the `regul` block. For this application three new blocks are written. Two blocks `AnalogIn` and `AnalogOut` are simple reading and writing values to the robot. The third block `RobotTrajectory` reads the control signal from the `regul` block, and calculates the corresponding reference values for the low level controllers. How PÅLSJÖ is communicating with the other control units is shown in Figure 8.16.

Summary

This example demonstrates how controller algorithms easily may be reused. The block diagram paradigm is a suitable way for packing code for reuse.

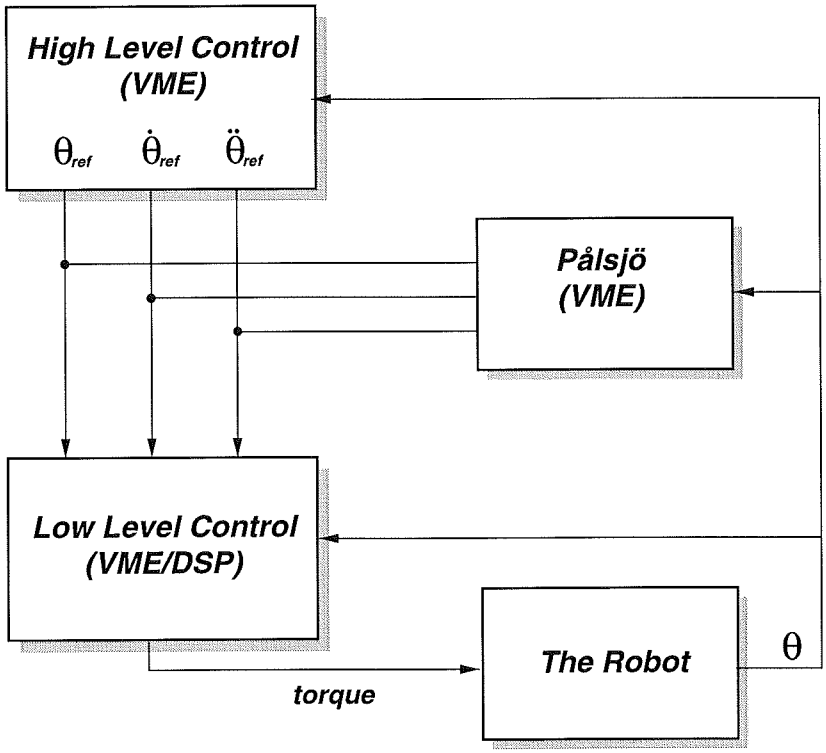


Figure 8.16 The setup where PÅLSJÖ is used for controlling the Irb2000 robot.

8.4 A Hybrid Tank Controller

In this section the design and implementation of a hybrid controller for a double tank system will be presented. In control practice it is quite common to use several different controllers and to switch between them with some type of logical device. One example is systems with selectors which have been used for constraint control for a long time. Systems with gain scheduling, see [Åström and Wittenmark, 1995], is another example. Both selectors and gain scheduling are commonly used for control of chemical processes, power stations and in flight control. Other examples of systems with mode switching are used in robotics. It is well known that hybrid systems are difficult to analyze. Nevertheless they are used more and more. The reason for this is that they give better performance than ordinary systems and that they can solve problems that cannot be dealt with by conventional control.

In process control it is common practice to use PI control for steady state regulation and to use manual control for large changes. In this

case it seems very natural to try to combine the steady state regulation with a minimum time controller for the set point changes. Such a controller is designed and implemented in this section.

First the process and the process model are introduced. Then the hybrid controller will be motivated, and the design for the sub-controllers is presented

The Process

In this section the design of a hybrid controller consisting of two simple controllers, one PID controller and one time-optimal controller is presented. It is shown that the use of this hybrid controller will lead to better performance. Both good response to set-point changes and good disturbance rejection are desired.

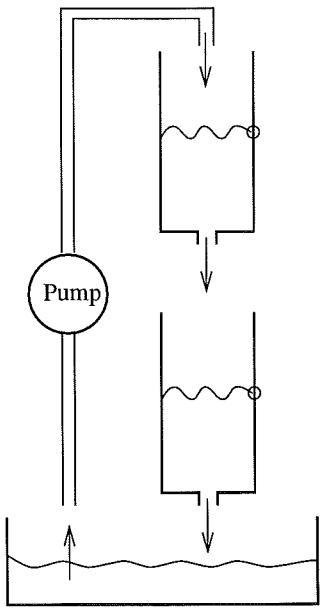


Figure 8.17 The double tank process

The process to be controlled, both in simulation and by the real time system consists of two water tanks in series, see Figure 8.17. The goal is to control the level of the lower tank and indirectly the level of the upper tank. The two tank levels are both measurable. Choosing the level of tank i as state x_i the following state space description is

derived

$$\dot{x} = f(x,u) = \begin{bmatrix} -\alpha_1\sqrt{x_1} + \beta u \\ \alpha_1\sqrt{x_1} - \alpha_2\sqrt{x_2} \end{bmatrix}. \tag{8.16}$$

where the inflow u is our control variable. The inflow can never be below zero and the maximum flow is $\bar{u} = 27 \cdot 10^{-6}\, m^3/s$. Furthermore, in our experimental setting the outflow areas are the same, giving $\alpha_1 = \alpha_2$.

The Controller

As mentioned above a supervisory switching scheme with two sub-controllers will be used. The time-optimal controller is used when the states are far away from the reference point. Coming closer the PID controller will automatically be switched in to replace the time optimal controller. At each different set point the controller is redesigned, keeping the same structure but using reference point dependent parameters. Figure 8.18 describes the algorithm with a Grafcet. The Grafcet for the tank controller consists of four states. Initially the controller is off. This is the Init state. Opt is the state where the time optimal controller is active and PID is the state for the PID controller. The Ref state is an intermediate state used for calculating new controller parameters before switching to a new time optimal controller.

The sub-controller designs are based on a linearized version of Equation (8.16):

$$\dot{x} = \begin{bmatrix} -a & 0 \\ a & -a \end{bmatrix} x + \begin{bmatrix} b \\ 0 \end{bmatrix} u \tag{8.17}$$

In this linearized equation the parameter b has included the factor $27 \cdot 10^{-6}$ and the new control variable u is in $[0,1]$. The parameters a and b are functions of α, β and the linearization level. It is later shown how the neglected nonlinearities will affect the performance. To be able to switch in the PID controller a fairly accurate knowledge of the parameters is needed.

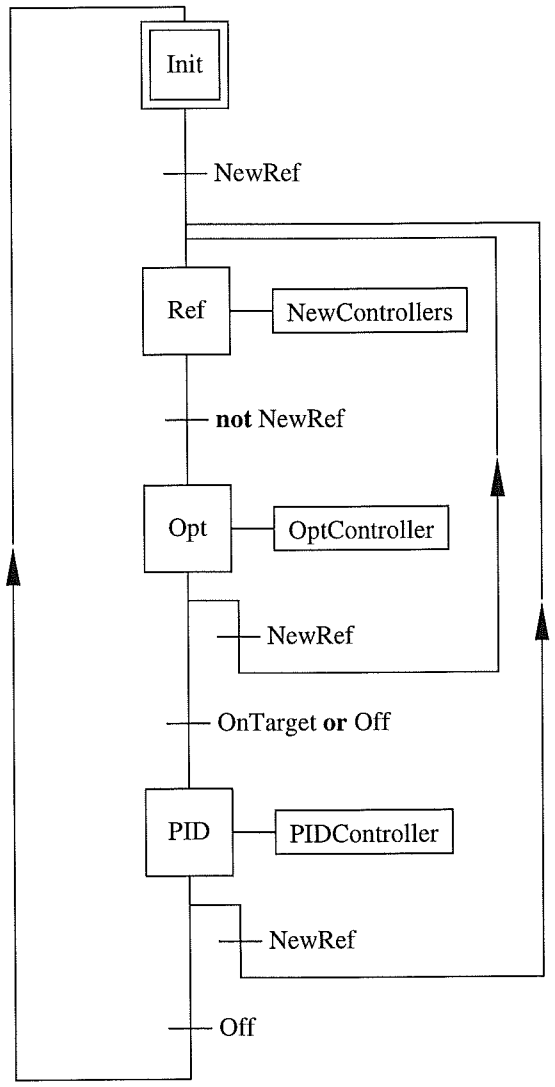


Figure 8.18 A Grafcet describing the control algorithm.

PID controller design

A standard PID controller on the form

$$G_{PID} = K(1 + \frac{1}{sT_I} + sT_d)$$

is used. The design of the PID controller parameters K, T_d and T_I is based on the linear second order transfer function,

$$G(s) = \frac{ab}{(s+a)(s+a)}$$

derived from Equation (8.17). The desired closed loop characteristic equation is

$$(s + \alpha\omega)(s^2 + 2\zeta\omega s + \omega^2).$$

The parameters α, ω and ζ (1.0, 0.06 and 0.7) are chosen for a reasonable behavior, both in case of set point changes and under load disturbances. For some systems it is possible to get a smaller overshoot by set point weighting. Figure 8.20 shows the set point and load disturbance responses for the PID controller. When implementing the real-time version of the PID algorithm a filter is used on the derivative part.

Time optimal controller design

The time optimal controller will bring the system as fast as possible from one set point to another. The Pontryagin maximum principle is used to prove that the time optimal control strategy for the System (8.16) is of bang-bang nature. The time optimal control is the solution to the following optimization problem

$$\max J = \int_0^T -1 \cdot dt \tag{8.18}$$

under the constraints:

$$\begin{aligned} x(0) &= [\,x_1^0 \quad x_2^0\,]^T \\ x(T) &= [\,x_1^R \quad x_2^R\,]^T \\ u &\in [0,1] \end{aligned}$$

The Hamiltonian, $H(x,u,\lambda)$, for this problem is

$$H = -1 + \lambda_1(-a\sqrt{x_1} + bu) + \lambda_2(a\sqrt{x_1} - a\sqrt{x_2}),$$

with the adjoint equations, $\dot{\lambda} = -\frac{\partial H}{\partial x}$,

$$\dot{\lambda} = \begin{bmatrix} -\frac{a}{2\sqrt{x_1}} & \frac{a}{2\sqrt{x_1}} \\ 0 & -\frac{a}{2\sqrt{x_2}} \end{bmatrix} \lambda. \tag{8.19}$$

To derive the control signal the complete solution to these equations is not needed. It is sufficient to note that the solutions to the adjoint equations are monotonous. This together with the switching function

$$\sigma = \lambda_1 bu$$

results in the following possible optimal control sequences

$$\langle 0,1\rangle, \langle 1,0\rangle, \langle 0\rangle, \langle 1\rangle$$

The switching times are determined by the new and the old set points. In practice it is preferable to have a feedback loop instead of precalculated switching times. Hence an analytical solution for the switching curves is needed. For the linear case it is possible to derive this solution:

$$x_2(x_1) = \frac{1}{a}[(ax_1 - b\bar{u})(1 + \ln(\frac{ax_1^R - b\bar{u}}{ax_1 - b\bar{u}})) + b\bar{u}]$$

where $\bar{u}$ takes values in $\{0,1\}$.

The fact that the nonlinear system has the same optimal control structure as the linearized system makes it possible to simulate the nonlinear switching curves and to compare them with the linear switching curves. Note that the linear and the nonlinear switching curves are quite close for our double tank model, see Figure 8.19. The diagonal line is the set of equilibrium points, $x_1^R = x_2^R$. Figure 8.19 shows that the linear switching curves are always below the nonlinear switching curves. This will cause the time optimal controller to switch either too late or too soon.

It is not necessary to use or know the exact nonlinear switching curves since the time optimal controller is only used to bring the system close to the new set point. When sufficiently close the PID controller will take over.

Stabilizing Switching Schemes

It is well known that switching between stabilizing controllers may lead to an unstable closed loop system. It is therefore necessary to have a switching scheme that guarantees stability. Consider the system

$$\dot{x} = f(x,t,u_i) \tag{8.20}$$

$$u_i = c_i(x,t) \tag{8.21}$$

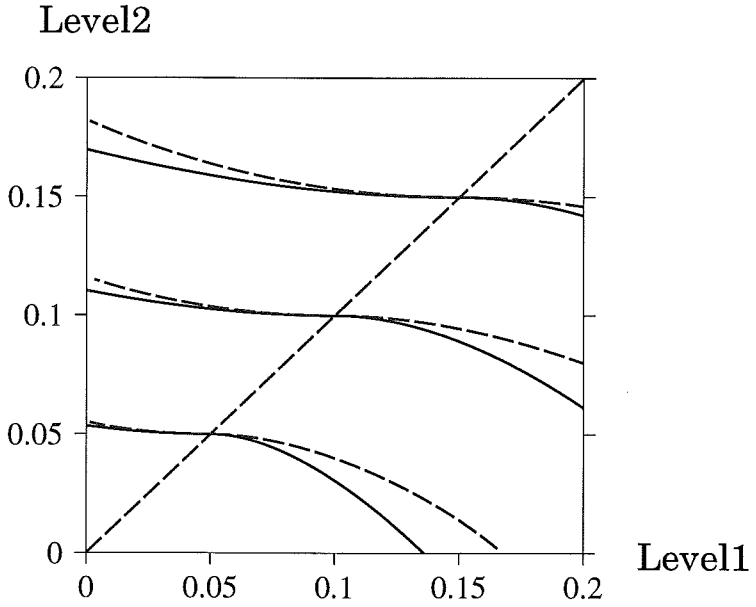


Figure 8.19 Linear (full) and nonlinear (dashed) switching curves

where the $c_i(x,t)$ represent different controllers. In a hybrid control system different controllers are switched in for different regions of the state space or in different operating modes. There exist some switching schemes that guarantee stability. One of these is the min-switch strategy described in [Malmborg *et al.*, 1996]. Here, a number of stabilizing controllers, c_i, are designed for System (8.20). For each controller c_i an operating region Ω_i is defined and a Lyapunov function V_i is derived. At every moment the supervisor selects the controller with the smallest value of its Lyapunov function. The controllers can be of different types and they need not share the same state space.

DEFINITION 8.1—MIN-SWITCHING STRATEGY

Let $f_i(x,t)$ be the right-hand side of Equation (8.20) when control law c_i is used. Use a control signal u^* so that,

$$\dot{x} = f(x,t,u^*) = \sum_{i=1}^{n} \alpha_i f_i(x,t) \tag{8.22}$$

where $\alpha_i \geq 0$ satisfies $\sum \alpha_i = 1$ and where $\alpha_i = 0$ if either $x \notin \Omega_i$ or if $V_i(x,t) > \min_j [V_j(x,t)]$. □

Notice that the α_i's are not unique. The following result is presented in [Malmborg *et al.*, 1996].

THEOREM 8.3—STABILITY OF HYBRID SYSTEMS

Let the system be given by Equation (8.20). Introduce W as

$$W = \min(V_1, V_2, \ldots, V_n)$$

The closed loop system is stable with W as a non-smooth Lyapunov function if the min-switch strategy is used. □

Lyapunov function modifications

From a control designer's point of view the design of a hybrid control scheme using the *min-switching strategy* can be reduced to separate designs of n different control laws and their corresponding Lyapunov functions. To improve performance it is often convenient to change the location of the switching surfaces. This can, to some degree, be achieved by different transformations of the Lyapunov functions. One example is transformations of the form

$$\tilde{V}_i = g_i(V_i) \tag{8.23}$$

where $g_i(\cdot)$ are monotonously increasing functions.

In some cases there can be very fast switching, chattering, between two or more controllers having the same value of their respective Lyapunov function. One way to avoid this is to add a constant Δ to the Lyapunov functions that are switched out and subtract Δ from the Lyapunov functions that are switched in. This works as a hysteresis function.

Simulations

In this section some different switching methods are evaluated. In all simulations a switching surface for the time optimal controller based on the linearized equations is used.

All simulations have been done in the Omola/Omsim environment [Andersson, 1994], which supports the use of hybrid systems.

Pure time optimal and pure PID control

This first simulation set, Figure 8.20, shows control of a linearized system using either a time optimal controller or a PID controller. Note that PID control gives a large overshoot. The time optimal controller works fine until the level of the lower tanks reaches its new set point. Then the control signal starts to chatter between its minimum and maximum value.

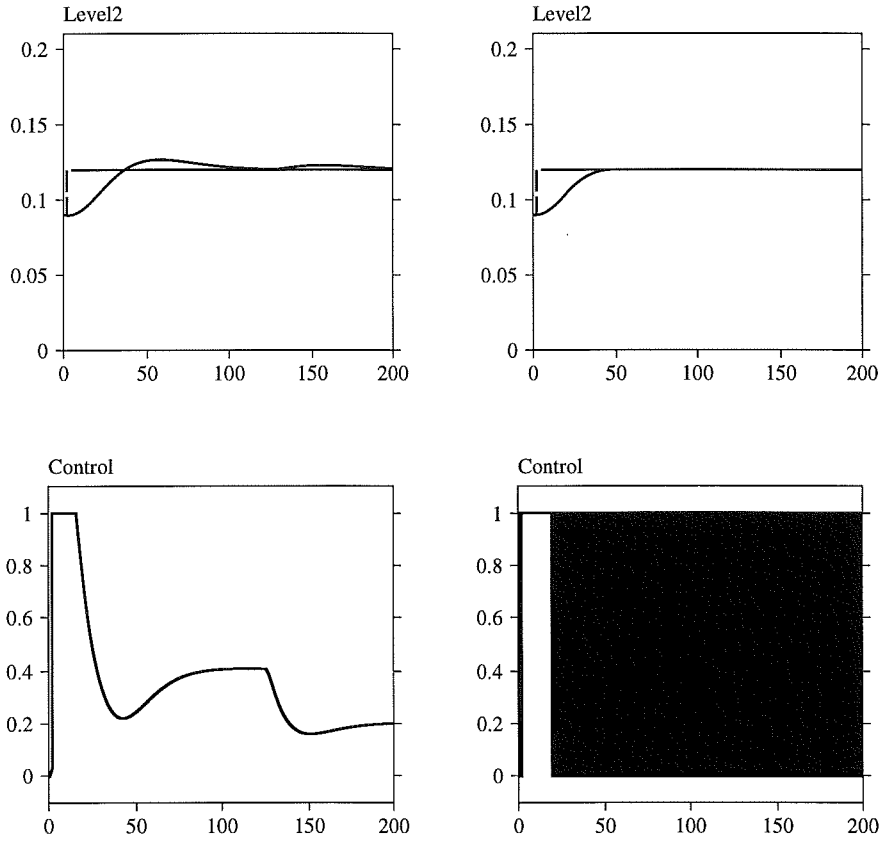


Figure 8.20 Pure PID (left) and pure time optimal control (right)

A natural, simple switching strategy

A natural switching strategy would be to pick the best parts from both PID control and time optimal control. One way to accomplish this is to use the time optimal controller when far away from the equilibrium point and the PID controller when coming closer. As a measure of

closeness the function V_{close} is used.

$$V_{close} = \begin{bmatrix} x_1^R - x_1 \\ x_2^R - x_2 \end{bmatrix}^T P(\theta,\gamma) \begin{bmatrix} x_1^R - x_1 \\ x_2^R - x_2 \end{bmatrix}$$

$$P(\theta,\gamma) = \gamma_1 \begin{bmatrix} \cos^2\theta + \gamma_2 \sin^2\theta & (1-\gamma_2)\sin\theta\cos\theta \\ (1-\gamma_2)\sin\theta\cos\theta & \sin^2\theta + \gamma_2\cos^2\theta \end{bmatrix}$$

The switching strategy here is to start with the time optimal controller and then switch to the PID controller when $V_{close} < \rho$. With the γ and θ parameters the size and shape of the catching region may be changed. In this simulation switching back to the time optimal controller is not allowed until there is a new reference value. See Figure 8.18 for a graphical description of the algorithm. The simulation results, Figure 8.21, show how the best parts from the sub-controllers are used to give very good performance.

Lyapunov based switching

In this third simulation set the *min switching strategy* that guarantees stability for the linearized system is used. The two Lyapunov functions are defined as

$$V_{PID} = \begin{bmatrix} x_1^R - x_1 \\ x_2^R - x_2 \\ x_3^R - x_3 \end{bmatrix}^T P(\theta,\gamma) \begin{bmatrix} x_1^R - x_1 \\ x_2^R - x_2 \\ x_3^R - x_3 \end{bmatrix}$$

$$V_{TO} = \textit{time left to reach new set point}$$

$$P(\theta,\gamma) = \gamma_1 \begin{bmatrix} \cos^2\theta + \gamma_2 \sin^2\theta & (1-\gamma_2)\sin\theta\cos\theta & 0 \\ (1-\gamma_2)\sin\theta\cos\theta & \sin^2\theta + \gamma_2\cos^2\theta & 0 \\ 0 & 0 & \gamma_3 \end{bmatrix}$$

The state x_3 is the integral state in the PID controller. x_3^R is its steady state value. As in the previous simulation set the parameters γ and

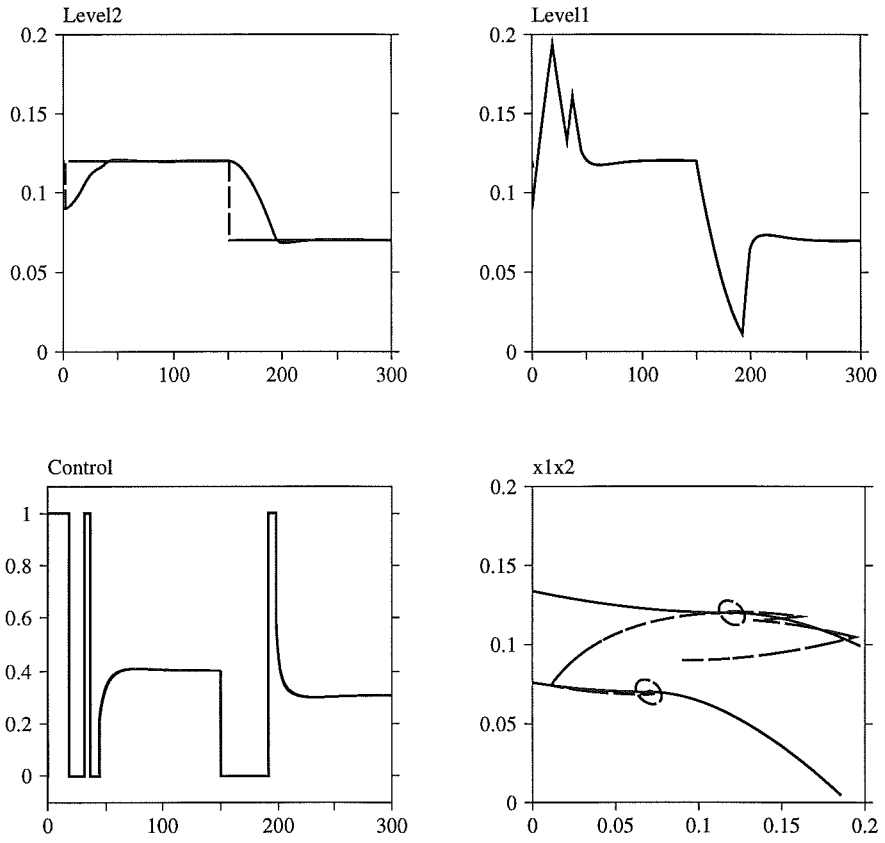


Figure 8.21 Simulation of the natural, simple switching strategy. Catching regions are shown in lower right sub-figure.

θ are used to shape the catching region. The new state x_3 is preset to its value at the new equilibrium point, i.e. x_3^R, any time there is a set point change. This state is updated after the first switch to PID control. Using this method a similar two-dimensional catching region as in the previous simulation set is constructed.

This supervisory scheme may lead to two types of chattering behavior. One is due to the nonlinearities. The nonlinear switching curve lies above the linear, see Figure 8.19. That causes the trajectory of the nonlinear system to cross the linear switching curve. One way to remove this problem is to introduce a hysteresis function for going from minimum to maximum control signal in the time optimal controller. There can also be chattering between the PID and the time optimal controller

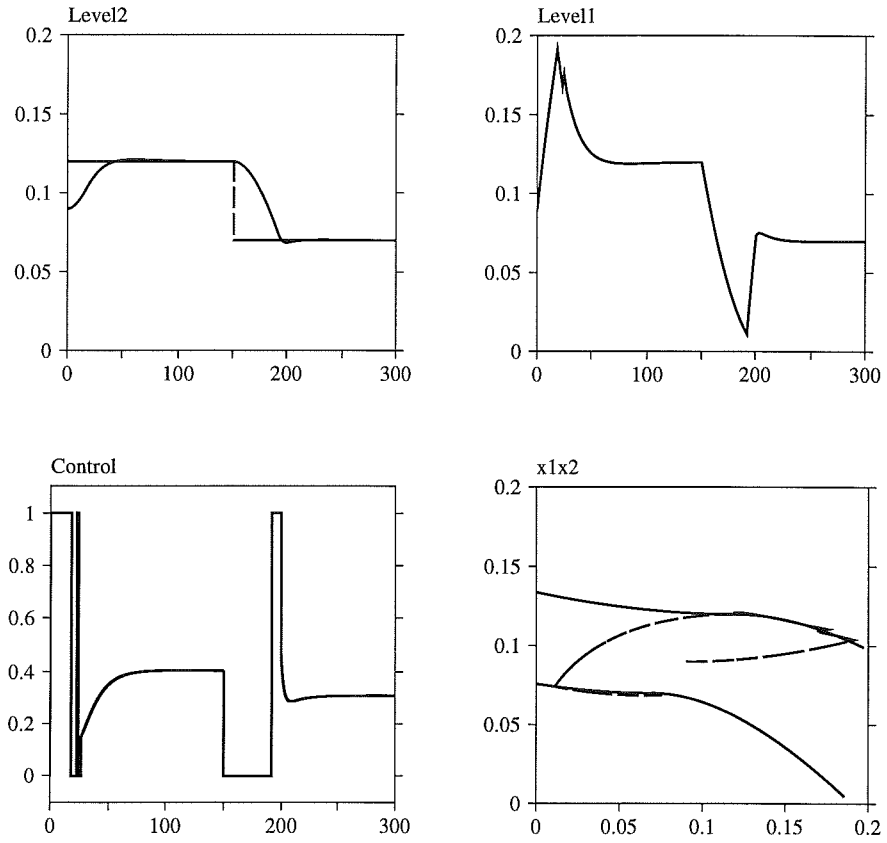


Figure 8.22 Lyapunov based switching

if their Lyapunov functions have the same value. One solution to this problem is to add and remove the constant Δ as discussed in the section on Lyapunov functions modifications. The simulation results can be seen in Figure 8.22.

Experiments

The theory and the simulations are verified by experiments. For simplicity only the switching strategy in Sec. 8.4 is implemented. Figure 8.23 shows the results of that experiment with the double tanks.

The measurements from our lab process have a high noise level as can be seen in Figure 8.23. A first order low-pass is used filter

$$G_f(s) = \frac{1}{s+1}$$

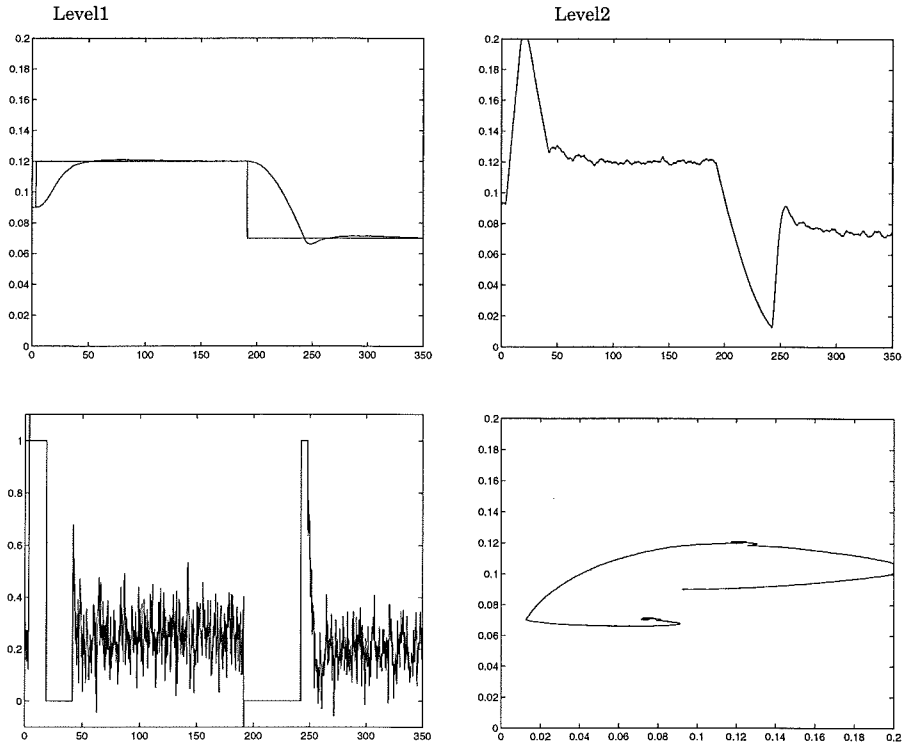


Figure 8.23 Lab experiment

to eliminate some of it. To further reduce the impact of the noise a filter is added to the derivative part of the PID controller in a standard way.

The parameters in the simulation model were chosen to match the parameters of our lab process. It is thus possible to compare the experimental results directly with the simulations. Comparing Figure 8.21 and Figure 8.23 shows the close correspondence between the simulation and experimental results.

During experiments it was found that the difference between the linear and the nonlinear switching curves were not so important. However, a good model of the static gains in the system is needed. If there is a large deviation it cannot be guaranteed that the equilibrium points are within the catching regions.

PAL Code

In this section the PAL code for the controller used in Figure 8.23 is presented. The controller is written as a Grafcet with four states, see Figure 8.18.

module regul;

function ln(r : **input real**) : **real**;**external** "ln";

function sqrt(r : **input real**) : **real**;**external** "sqrt";

block Tank

$y1, y2$: **input real**;
$OnTarget :=$ **false**, $NewRef :=$ **false**, $Off :=$ **false** : **boolean**;
h : **sampling interval**;
$a := 1.0, b := 1.0, yref := 0.50$: **real**;
$K := 5.0, Ti := 60.0, Tr := 60.0, Td := 10.0$: **real**;
$gamma1 := 1.0, gamma2 := 1.0, Vpid := 100.0$: **real**;
$e := 0.0, yold := 0.0, P := 0.0, I := 0.0, D := 0.0$: **real**;
$d1 := 1.0, d2 := 1.0, i1 := 1.0, w1 := 1.0$: **real**;
$umin := 0.0, umax := 1.0$: **real**;
$x1 := 5.0, x2 := 5.0, xref := 5.0, , u := 0.0$: **real**;
$Ku := 0.000027, Kc := 5.0, region := 0.1$: **parameter real**;
$omega := 0.04, zeta := 0.7, alpha := 1.0, N := 10.0$: **parameter real**;
$c := 2.0, aa := 0.00000707, AA := 0.00273$: **parameter real**;
$v := 0.0$: **output real**;

function Switch(z1 : **input real**;z3 : **input real**;ubar : **input real** : **real**;
begin
$result := 1.0/a * ((a * z1 - b * ubar) * (1.0 + ln((a * z3 - b * ubar)/$
$(a * z1 - b * ubar))) + b * ubar)$;
end Switch;

function Sat(min : **input real**;max : **input real**;x : **input real**) : **real**;
begin
if $x < min$ **then**
$result := min$;
elsif $x > max$ **then**
$result := max$;
else
$result := x$;
end if;
end Sat;

calculate
begin
$x1 := (1.0 - c * h) * x1 + c * h * y1$;
$x2 := (1.0 - c * h) * x2 + c * h * y2$;
$xref := yref * Kc$;
end calculate;

```
update
begin
  Vpid := gamma1 * ((x1 - xref) * (x1 - xref)+
          gamma2 * (x2 - xref) * (x2 - xref));
  yold := x2;
end update;

initial step Init;
  activate OffController;
end Init;

step Ref;
  activate NewControllers;
end Ref;

step Opt;
  activate OptController;
end Opt;

step PID;
  activate PIDController;
end PID;

transition from Init to Ref when NewRef;

transition from Ref to Opt when not NewRef;

transition from PID to Init when Off;

transition from Opt to PID when OnTarget or Off;

transition from PID to Ref when NewRef;

transition from Opt to Ref when NewRef;

action OffController;
begin
  v := 0.0;
  Vpid := 100.0;
  Off := false;
end OffController;

action NewControllers;
begin
  a := aa/AA * sqrt(9.81/2.0/xref);
  b := Ku/AA * Kc;
  K := (omega * omega * (1.0 + 2.0 * alpha * zeta) - a * a)/b/a;
  Ti := b * K * a/alpha/omega/omega/omega;
```

$Td := 1.0/a/b/K * (omega * (alpha + 2.0 * zeta) - 2.0 * a);$
$d1 := Td/(Td + N * h);$
$d2 := K * N * d1;$
$i1 := h * K/Ti;$
$w1 := h/Tr;$
$NewRef :=$ **false**;
end NewController;

action OptController;
begin
if $Vpid < region$ **then**
$OnTarget :=$ **true**;
$e := xref - x2;$
$P := K * e;$
$D := d1 * D + d2 * (yold - x2);$
$I := aa/AA * sqrt(2.0 * 9.81 * xref * Kc)/b;$
end if;
$u := 0.0;$
if $x1 > xref$ **and** $x2 < Switch(x1, xref, umin)$ **then**
$u := umax;$
end if;
if $x1 < xref$ **and** $x2 < Switch(x1, xref, umax)$ **then**
$u := umax;$
end if;
$v := u;$
end OptController;

action PIDController;
$usat$: **real**;
begin
$OnTarget :=$ **false**;
$e := xref - x2;$
$P := K * e;$
$D := d1 * D + d2 * (yold - x2);$
$u := P + I + D;$
$usat := Sat(umin, umax, u);$
$v := u;$
$I := I + i1 * e + w1 * (usat - u);$
end PIDController;

procedure Ref05();
begin
$yref := 0.05;$

```
    NewRef := true;
  end Ref10;

  procedure Ref15();
  begin
    yref := 0.11;
    NewRef := true;
  end Ref11;

  procedure Stop();
  begin
    Off := true;
  end Stop;

end Tank;

end regul.
```

Summary

In this case study a hybrid controller for a double tank system has been designed and implemented. Both simulations and real experiments were presented. Finally the PAL code used in the experiments were shown.

8.5 An Adaptive Controller

Introduction

In this section the implementation of an adaptive controller for a servo motor is described. First the process model and the control law are introduced. The PAL code and the PCL script are also presented.

Preliminaries

The servo motor system is described by the transfer function

$$G_\theta(s) = \frac{K_\theta}{s(1+sT)} \tag{8.24}$$

with $K_\theta = 93.5s^{-1}$ and $T = 8.3s$. The angle of the servo wheel is the output signal and the voltage to the motor is the input signal.

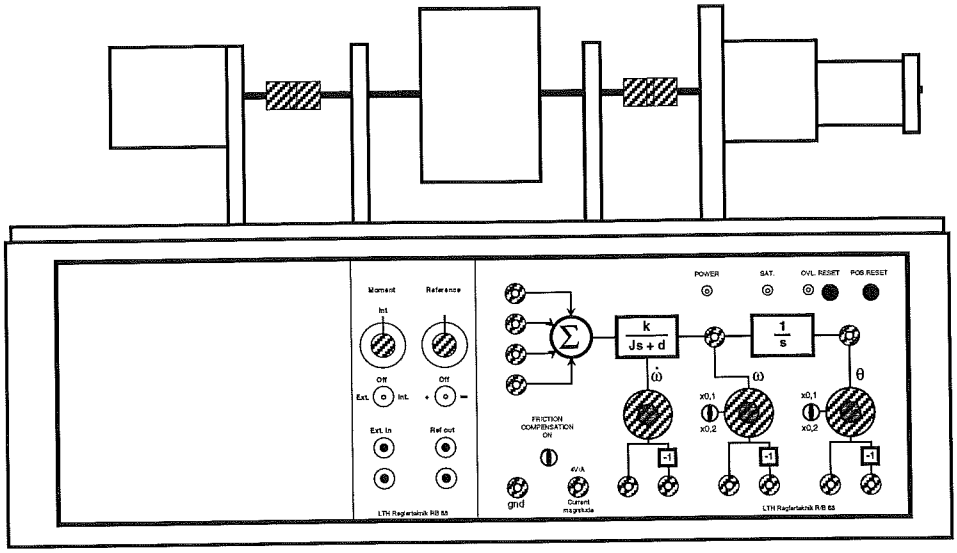


Figure 8.24 The servo motor.

The sampled system is described by the pulse transfer function

$$H(q) = \frac{b_1 q + b_2}{q^2 + a_1 q + a_2} \tag{8.25}$$

The closed loop specification in continuous time is given by

$$(s^2 + 2\omega\zeta s + \omega^2)(s - \alpha) = 0 \tag{8.26}$$

which in discrete time corresponds to

$$(q^2 + a_{m1} q + a_{m2})(q + ao1) = A_m A_o = 0 \tag{8.27}$$

The desired control law is given through solving the Diophantine equation below

$$AR + BS = A_m A_o \tag{8.28}$$

In the adaptive case the process model is estimated on-line, and new R, S, and T polynomials are calculated in real-time. The block diagram for this adaptive controller is shown in Figure 8.25.

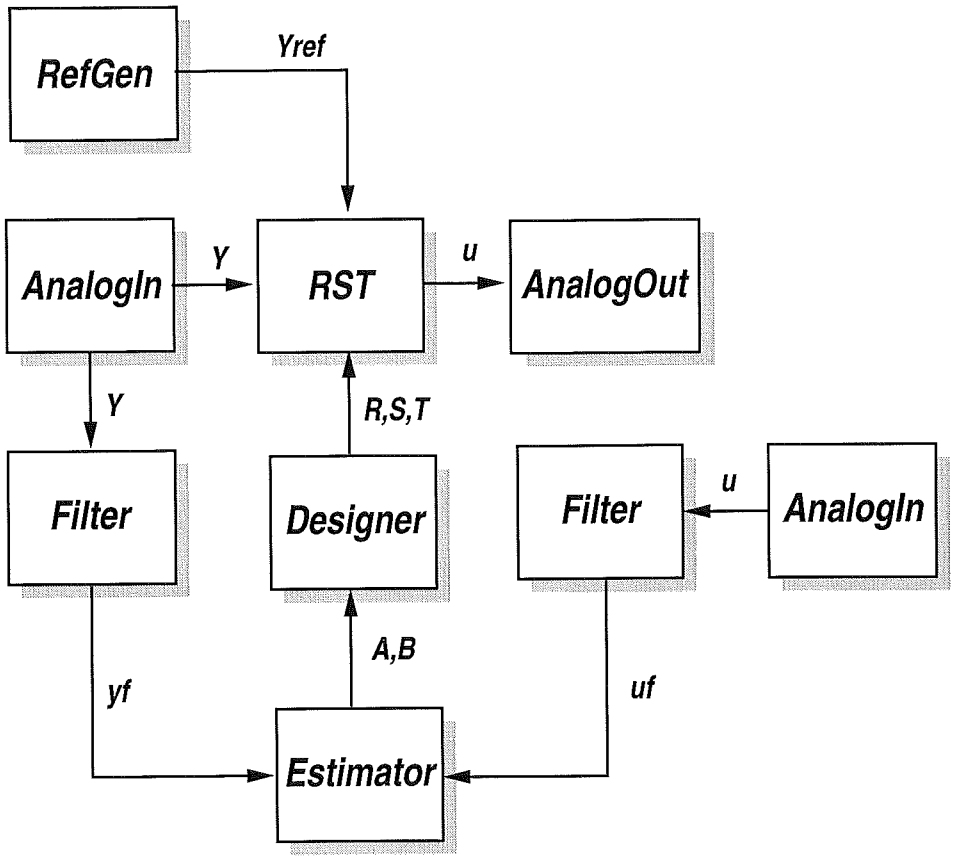


Figure 8.25 The block diagram for the adaptive control for the servo motor.

Running Pålsjö

The adaptive controller has two feedback loops, one fast loop feeding back the value of the process to the controller, and one slow loop which estimates the process model and calculates new controller parameters. When implementing this, it seems natural to use two processes, i.e. two `Periodic` blocks. One process handles the fast loop and the other handles the slow loop. The PCL script below shows how such a system is created. The PAL-blocks needed are described later. For a discussion on the stability of sampled adaptive controllers, where the design process is excuting at a lower frequency than the feedback loop, see [Shimkim and Feuer, 1988].

```
use StandardBlocks
use Adaptive

{ s = new Periodic
```

```
  s2 = new Periodic
  s.tsamp = 0.020;
  s2.tsamp = 0.20;
  s.ib = new ADIn
  s.ib.channel = 2
  s.rg = new RefGen
  s.rst = new RST
  s.ob = new DAOut
  s.ob.channel = 1
  s.ctrl = new ADIn
  s.ctrl.channel = 3
  s.est = new Estimator
  s2.dsg = new Designer
  s.uFilter = new Filter
  s.yFilter = new Filter

  size degA ; degA = 2
  s.rst.degA = degA; s.est.degA = degA; s2.dsg.degA = degA
  size degB ; degB = 1
  s.rst.degB = degB; s.est.degB = degB; s2.dsg.degB = degB
  size degAm; degAm = 2;
  s2.dsg.degAm = degAm; s.rst.degAm = degAm
  size degAo; degAo = 1;
  s2.dsg.degAo = degAo; s.rst.degAo = degAo

  s.ib.out -> s.rst.y
  s.rg.out -> s.rst.uc
  s.ib.out -> s.yFilter.u
  s.yFilter.y -> s.est.yf
  s.ctrl.out -> s.uFilter.u
  s.uFilter.y -> s.est.uf
  s.est.A -> s2.dsg.A
  s.est.B -> s2.dsg.B
  s.rst.u -> s.ob.in

  s2.dsg.R ->s.rst.R
  s2.dsg.S ->s.rst.S
  s2.dsg.T ->s.rst.T
}
```

When the system is created the next step is to assign values to all

parameters. This is done with the following PCL-script:

```
{
  s.rg.period = 8; s.rg.amplitud = 0.5
  s2.dsg.R2 = {1, -0.6741}
  s2.dsg.S2 = {9.317, -8.7807}
  s2.dsg.T2 = {2.9588, -2.4225}
  s.est.lambda = 0.995; s.est.P0 = 100
  s2.dsg.Am = {1, -1.8321, 0.8454};
  s2.dsg.Ao = {1, -0.8187}
  s.rst.R = {1, -0.6741}
  s.rst.S = {9.317, -8.7807}
  s.rst.T = {2.9588, -2.4225}
  s.est ! Reset
}
s ! start; s2 ! start
```

Now the system is configured and started. To view data signals must be exported, and this is done using the `show` command. The plot from a test run with the adaptive controller is shown in Figure 8.26.

```
show s.rg.out
show s.ib.out
show s.rst.u
show s.est.A
show s.est.B

s ! connect
```

The PAL code

In this section the PAL module `Adaptive` is discussed and each block will be presented. First the module head is shown. A number of C-functions are imported so that they can be used in the PAL.

module Adaptive;

  procedure RowAsRealArray(
    M : **input matrix** [0..m : **integer**, 0..n : **integer**] of **real**;
    row : **input integer**;
    res : **output array** [0..*n*] of **real**
  );

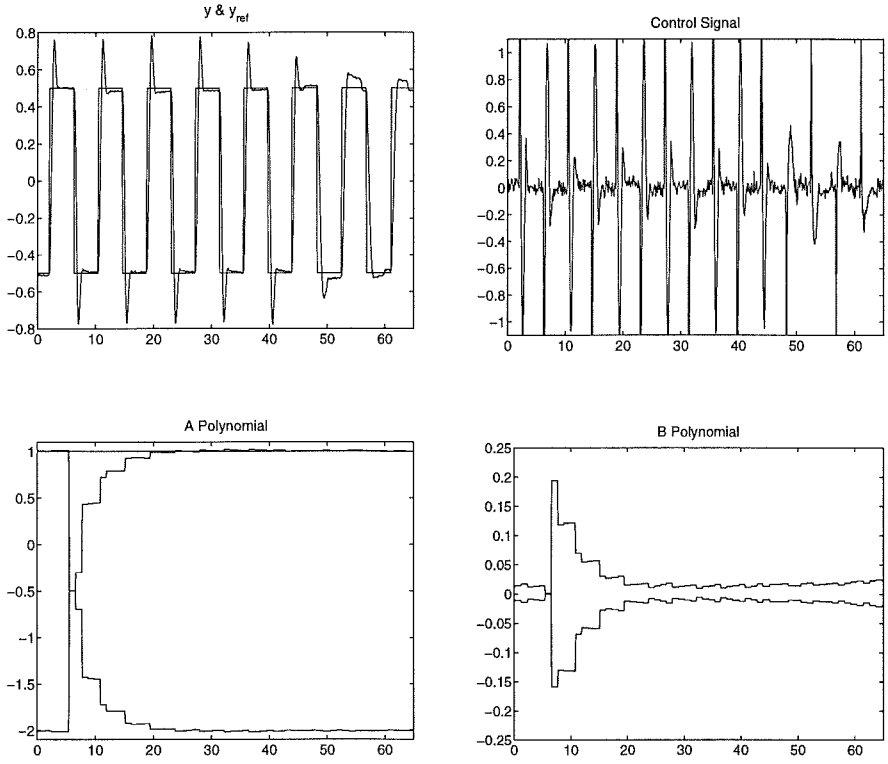


Figure 8.26 Running the adaptive controller on the servo. These are plots presented in real-time by a Matlab script.

```
external "RowAsRealArray";

procedure RealArrayAsRow(
  A : input array [0..n] of real;
  row : input integer;
  res : output matrix [0..m : integer, 0..n : integer] of real
);
external "RealArrayAsRow";

procedure ShiftBackward(
  p : output polynomial [degP : integer] of real;
  shift : real
);
external "ShiftBackward";

function MDPPNZCDesign(
  A : polynomial [degA : integer] of real;
  B : polynomial [degB : integer] of real;
  Am : polynomial [degAm : integer] of real;
```

```
  Ao : polynomial [degAo : integer] of real;
  R : output polynomial [degR : integer] of real;
  S : output polynomial [degS : integer] of real;
  T : output polynomial [degT : integer] of real
) : boolean;
external "MDPPNZCDesign";
```

RST-Controller

The RST controller has two input signals, the process value y and the set-point u_c. The degree of the polynomials in the controller is parameterized using dimension parameters.

```
block RST
  degAm, degAo, degA, degB : dimension;
  u : output real;
  uc, y : input real;
  R : parameter polynomial [degAm + degAo − degA] of real;
  S : parameter polynomial [degA − 1] of real;
  T : parameter polynomial [degAo] of real;
  U, Uc, Y : polynomial [degAm + degAo − degA] of real;
  preU := 0.0 : real;
  m = degAm + degAo − degA;
  n = degA − 1;
  o = degAo;
  delayT = m − n;
  delayS = m − o;

  forward
  begin
    ShiftBackward(Uc, uc);
    ShiftBackward(Y, y);
    u := −S[0] * Y[delayS] + T[0] * Uc[delayT] + preU;
    ShiftBackward(U, u);
  end forward;

  backward
    i : integer;
  begin
    preU := 0.0;
    for i := 1 to m do
      preU := preU − R[i] * U[i − 1];
    end for;
```

```
    for i := 1 to n do
      preU := preU − S[i] * Y[i + delayS − 1];
    end for;
    for i := 1 to o do
      preU := preU + T[i] * Uc[i + delayT − 1];
    end for;
  end backward;
end RST;
```

The Estimator

The Estimator has two input signals, the filtered process value y_f and the filtered control signal u_f. The output from the Estimator is the pulse transfer function for the process, i.e. the A and the B polynomial.

```
block Estimator
  degA, degB : dimension;
  uf, yf : input real;
  phi : array [0..degA + degB + 1] of real;
  theta : array [0..degA + degB + 1] of real;
  D : array [0..degA + degB + 1] of real;
  L : matrix [0..degA + degB + 1, 0..degA + degB + 1] of real;
  A : output polynomial [degA] of real;
  B : output polynomial [degB] of real;
  lambda, P0 : parameter real;

  calculate
    i : integer;
  begin
    for i := degA downto 2 do
      phi[i] := phi[i − 1];
    end for;
    phi[1] := −phi[0];
    phi[0] := yf;
    for i := degA + degB + 1 downto degA + 2 do
      phi[i] := phi[i − 1];
    end for;
    phi[degA + 1] := uf;
    LDFilter(theta, D, L, phi, lambda);
    for i := 0 to degA do
      A[i] := theta[i];
    end for;
```

```
  A[0] := 1.0;
  for i := 0 to degB do
    B[i] := theta[degA + 1 + i];
  end for;
end calculate;

procedure DyadicReduction(
  a : output array [0..n : integer] of real;
  b : output array [0..n] of real;
  alpha : output real;
  beta : output real;
  i0 : input integer;
  i1 : input integer;
  i2 : input integer
);
  i : integer;
  w1, w2, b1, gamma : real;
begin
  b1 := b[i0];
  w1 := alpha;
  w2 := beta * b1;
  alpha := alpha + w2 * b1;
  beta := beta * w1/alpha;
  gamma := w2/alpha;
  for i := i1 to i2 do
    b[i] := b[i] - b1 * a[i];
    a[i] := a[i] + gamma * b[i];
  end for;
end DyadicReduction;

procedure LDFilter(
  theta : output array [0..n : integer] of real;
  d : output array [0..n] of real;
  l : output matrix [0..n, 0..n] of real;
  phi : array [0..n] of real;
  lambda : input real
);
  i, j : integer;
  e, w : real;
  tmp1, tmp2 : array [0..degA + degB + 1] of real;
begin
  d[0] := lambda;
```

```
    e := phi[0];
    for i := 1 to n do
      e := e - theta[i] * phi[i];
      w := phi[i];
      for j := i + 1 to n do
        w := w + phi[j] * l[i,j];
      end for;
      l[0,i] := 0.0;
      l[i,0] := w;
    end for;
    for i := n downto 1 do
      RowAsRealArray(l, 0, tmp1);
      RowAsRealArray(l, i, tmp2);
      DyadicReduction(tmp1, tmp2, d[0], d[i], 0, i, n);
      RealArrayAsRow(tmp1, 0, l);
      RealArrayAsRow(tmp2, i, l);
    end for;
    for i := 1 to n do
      theta[i] := theta[i] + l[0,i] * e;
      d[i] := d[i]/lambda;
    end for;
  end LDFilter;

  procedure Reset();
    i,j : integer;
  begin
    for i := 0 to degA + degB + 1 do
      D[i] := P0;
      for j := 0 to degA + degB + 1 do
        L[i,j] := 0.0;
      end for;
      L[i,i] := 1.0;
      theta[i] := 0.0;
    end for;
  end Reset;

end Estimator;
```

The Designer

The Designer receives the transfer function polynomials A and B from the Estimator. In order to calculate a new RST-controller it calls the

$MDPPNZC$-function in the polynomial library, see Appendix B. The outputs from the Designer block are connected to the R, S, and T polynomials of the RST block. It is important to notice that the output signals are only updated when new controller parameters are calculated, i.e. not at every cycle. As soon as the output signals are assigned new values the system will propagate their values to the parameters of the RST block.

block Designer

$degAm, degAo, degA, degB$: **dimension**;
R : **output polynomial** $[degAm + degAo - degA]$ of **real**;
S : **output polynomial** $[degA - 1]$ of **real**;
T : **output polynomial** $[degAo]$ of **real**;
$mode := 0$: **integer**;
A : **input polynomial** $[degA]$ of **real**;
B : **input polynomial** $[degB]$ of **real**;
$tmpR$: **polynomial** $[degAm + degAo - degA]$ of **real**;
$tmpS$: **polynomial** $[degA - 1]$ of **real**;
$tmpT$: **polynomial** $[degAo]$ of **real**;
Am : **parameter polynomial** $[degAm]$ of **real**;
Ao : **parameter polynomial** $[degAo]$ of **real**;
$R2$: **parameter polynomial** $[degAm + degAo - degA]$ of **real**;
$S2$: **parameter polynomial** $[degA - 1]$ of **real**;
$T2$: **parameter polynomial** $[degAo]$ of **real**;

calculate
  i : **integer**;
begin
  if $mode = 0$ **then**
    $R := R2$;
    $S := S2$;
    $T := T2$;
    $mode := 1$;
  elsif $mode = 2$ **then**
    if $MDPPNZCDesign(A, B, Am, Ao, tmpR, tmpS, tmpT)$ **then**
      $R := tmpR$;
      $S := tmpS$;
      $T := tmpT$;
    end if;
  end if;
end calculate;

```
    procedure on();
    begin
      mode := 2;
    end on;

    procedure off();
    begin
      mode := 0;
    end off;

  end Designer;
```

8.6 A Variable Structure Controller

In this section the implementation of a fault tolerant controller is described. The goal is to create a controller structure that allows new algorithms to replace old algorithms in a safe way. The idea is to have a set of controllers which vary in complexity and performance. The basic assumption is that the more complex the controller, the better its performance. At the same time the more complex the controller becomes, the more likely it is to contain design fault, which will cause it to fail. The set of controllers vary from simple, very reliable algorithms to more complex algorithms. All controllers execute in parallel and then a special decision block is used to decide which controller to use. This architecture is proposed in the Simplex project [Sha *et al.*, 1995], which aims at creating on-line evolvable control systems. In this section it is shown how the Simplex ideas could be implemented in the PÅLSJÖ environment. This example has not been implemented in PÅLSJÖ since the new PÅLSJÖ features presented below are not available in the current version.

The Simplex Architecture

The Simplex architecture has been developed to support safe and reliable online upgrades of hardware and software components in spite of errors in the new modules. Three types of faults are addressed by the Simplex architecture, see Figure 8.28. The problem with faults due to resource sharing, where the failure of one process may cause the failure of other processes, is dealt with using protected memory. Each process

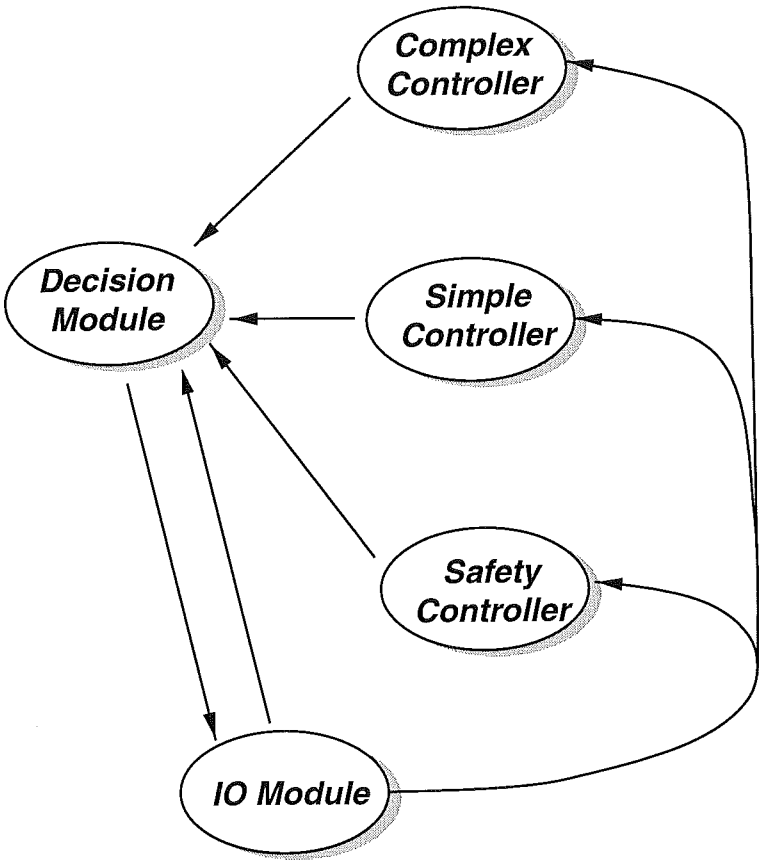


Figure 8.27 The structure of the fault tolerant controller.

has its own address space and is thus less likely to cause other processes to fail due to memory management problems. A semantic fault may be an incorrect algorithm or a software bug. Semantic fault are approached by having several similar software modules to perform a task. A special module is then used for selecting which calculations to use. This concept is called *analytic redundancy*. The timing problem is solved by using generalized rate monotonic scheduling. The Simplex architecture consists of three major parts, see Figure 8.27. The IO-module handles all communication with the environment. The Decision module decides which controller to use. The cyclic algorithm for Simplex is the following:

- Start of sampling interval.
- **IO-Module**
  - Write control signal to the process

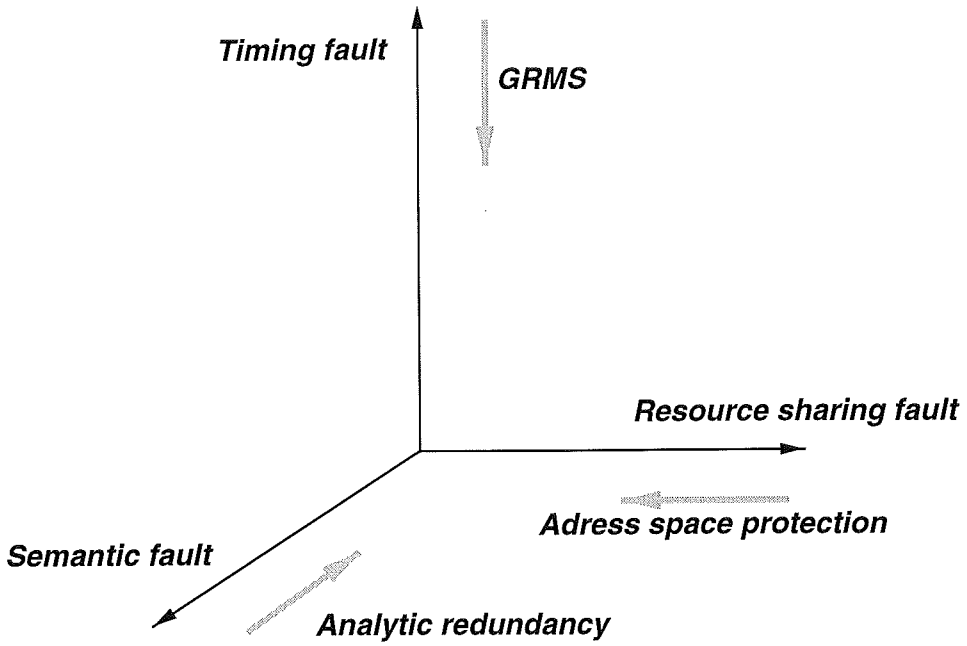


Figure 8.28 In Simplex three types of faults are addressed: Timing faults, resource sharing faults and semantic faults. Timing faults are solved using the Generalized Rate Monotonic Theory (GRMT), the resource sharing is dealt with using memory address protection, and semantic faults are approached with analytic redundancy.

  - Read the process output
  - Broadcast the process output to all controllers
- **Controller** Wait for process value, calculate control signal, and send it to the Decision-module.
- **Decision-module** Wait for process value to be broadcasted and then evaluate the state of the system, i.e. determine if the process is well controlled. Next, the Decision-module will wait for the controllers to finish their calculations. Controllers which have not delivered their control signal before the deadline are ignored. Finally the Decision-module picks the controller signal to use and sends it the IO-module.

Pålsjö

In this case study the concept with analytic redundancy is implemented using PÅLSJÖ. Other Simplex features such as protected memory are not regarded. In order to mimic the Simplex architecture in

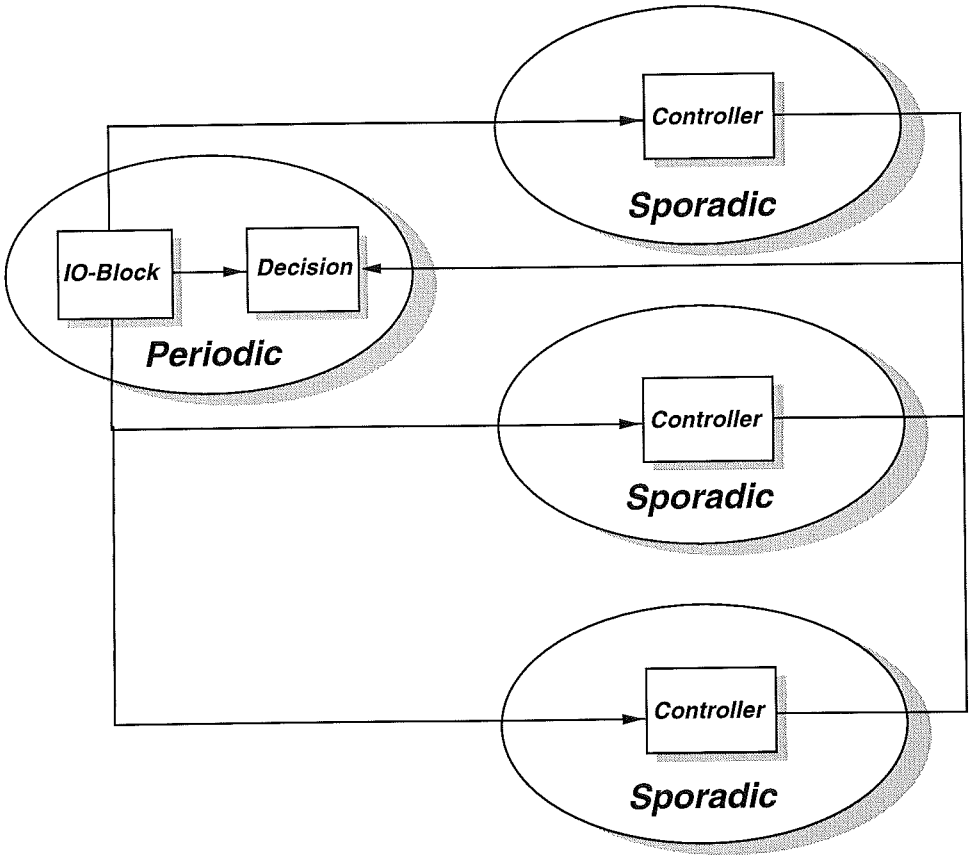


Figure 8.29 The structure of the PÅLSJÖ version of the Simplex architecture.

PÅLSJÖ some new features are added. The controller blocks must wait for the process value to be broadcasted from the IO-module, before they may start to calculate control signals. In order to synchronize processes like this, a new language construct is needed, and to solve this a new data type called **event** is introduced. The event data type works similar to a condition variable [Burns and Wellings, 1997]. Two operations are available for the event type: **wait** and **cause**. The IO-module executes a **cause** statement when the new process value is available, and the controllers which then execute **wait** get notified.

The PAL Code

The outline for PAL code which implements the basic ideas of the Simplex architecture is presented below. The structure for the PÅLSJÖ version is shown in Figure 8.29. The IO-Block and the Controller use the event variable to synchronize, so that when a new process value y

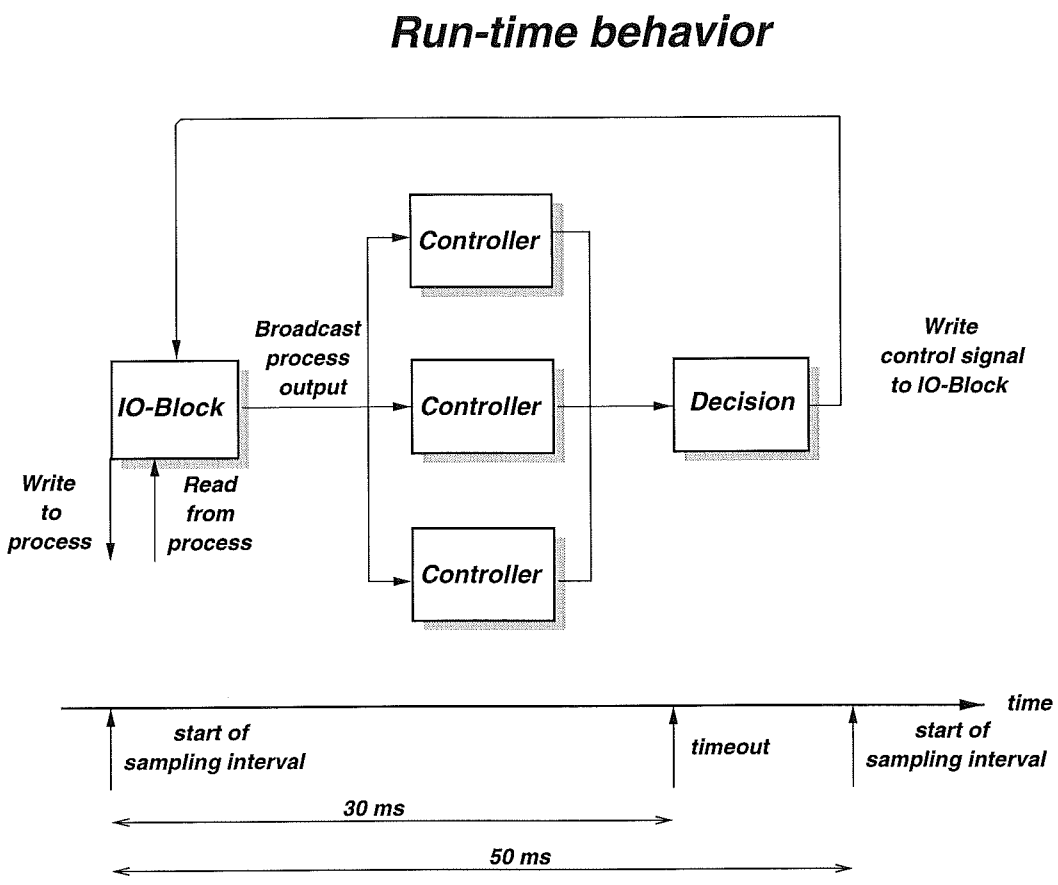


Figure 8.30 The cyclic behavior of the fault tolerant controller.

is available, the IO-Block notifies the controllers by executing a **cause** statement. The controllers are executed by `Sporadic` blocks, while the IO-Block and the Decision block are executed by a `Periodic` block. The controllers will however also execute periodically since they are synchronized with the IO-Block, via the event variable ev.

Below are the PAL code for implementing the controller structure. In the PCL script in the next section three different controllers are created. They all have the same structure as the `Controller` block. Note, that the PAL code is not complete, some functions are used but not defined.

```
module Simplex;

  function ReadFromProcess() : real;
  external "ReadFromProcess";

  procedure WriteToProcess(
```

```
  value : real
);
external "WriteToProcess";

block IOBlock

  u : input real;
  y : output real;
  ev : output event;

  calculate
  begin
    WriteToProcess(u);
    y := ReadFromProcess();
    cause(ev);
  end calculate;

end IOBlock;

block Decision

  n : dimension;
  y : input real;
  v : array [1..n] of input real;
  u : output real;
  mode : integer;

  calculate
  begin
    u := ChooseController(v, mode);
  end calculate;

end Decision;

block Controller

  y : input real;
  u : output real;
  ev : input event;

  calculate
    wait(ev);
    u := ControlLaw(...);
  begin
  end calculate;

end Controller;
```

end Simplex.

The PCL Script

The PCL script that will create the system shown in Figure 8.29 is presented below. In the Simplex architecture the Decision-module waits for the controllers to deliver a control signal, and timeouts if not all the controllers are finished before the deadline. Timeouts are not possible to implement in PÅLSJÖ. Instead this is solved by scheduling the IO-Block and the Decision so that they execute at the same rate but with an offset that corresponds to the deadline. The execution order is shown in Figure 8.30. The IO-Block writes the control signal to the process, reads the process output, and sends it to the controller blocks. The controller blocks, which are waiting at the **wait** statement, starts to execute when a new process value is available. When 30 milliseconds have passed since the beginning of the sampling interval, the Decision block is started. It decides which control signal to use, based on the system states and the values of the control signals delivered by the controllers.

```
use Simplex
{
  dim n
  n = 3
  main = Periodic
  main.io = new IOBlock
  main.decision = new Decision
  ctrl1 = new Sporadic
  ctrl1.controller = new Controller1
  ctrl2 = new Sporadic
  ctrl2.controller = new Controller2
  ctrl3 = new Sporadic
  ctrl3.controller = new Controller3

  main.decision.n = n
  main.io.y -> main.decision.y
  main.io.y -> ctrl1.controller.y
  main.io.t -> ctrl1.controller.t
  ctrl1.controller.u -> main.decision.u[1]
  main.io.y -> ctrl2.controller.y
  main.io.t -> ctrl2.controller.t
```

```
  ctrl2.controller.u -> main.decision.u[2]
  main.io.y -> ctrl3.controller.y
  main.io.t -> ctrl3.controller.t
  ctrl3.controller.u -> main.decision.u[3]

  main.io.slave = false
  main.io.tsamp = 0.05
  main.io.offset = 0
  main.decision.slave = false
  main.decision.tsamp = 0.05
  main.decision.offset = 0.03

  main ! start
  ctrl1 ! start
  ctrl2 ! start
  ctrl3 ! start
}
```

Using the PCL script above the system is now configured so that three controllers are running in parallel. Assume that a new controller should be added to the system. This is done with the following PCL commands:

```
{
  n = 4
  ctr4 = new Sporadic
  ctrl4.controller = new Controller4
  main.io.y -> ctrl4.controller.y
  main.io.t -> ctrl4.controller.t
  ctrl4.controller.u -> main.decision.u[4
  ctrl4 ! start
}
```

Summary

In this section it has been shown how the Simplex architecture for fault tolerant controllers can be implemented in the PÅLSJÖ environment. Some extra features that are not available in the current version of PÅLSJÖ have been introduced. This example has thus not been implemented.

9

Conclusions and Future Work

The PÅLSJÖ environment for development of embedded control systems has been presented in this thesis. The environment consists of the PÅLSJÖ run-time system and the PAL compiler. Control algorithms are coded in PAL language, which was described in Chapter 4. A special language, PCL, is used for configuring the run-time system. This was described in Chapter 5. Implementation issues were discussed in Chapter 6 and 7. Finally, some case studies were presented in Chapter 8.

9.1 Conclusions

A number of design goals were presented in Chapter 3. In the implementation of PÅLSJÖ they were approached as follows:

- *Rapid prototyping*, is achieved through the controller description language PAL, which gives modular code that can be *reused*.
- The system is *expandable* in the sense that new data types and new block types may easily be added. This is achieved through the use of register functions and factory patterns as discussed in Chapter 7.
- The system configuration is done on-line with using the PCL configuration language. Changes to an executing configuration is al-

lowed without stopping the system.

- The use of the Calculate-Update pattern and Parameter Swap pattern contributes to make the execution *efficient*.

Related Tools

A mentioned in Chapter 1 the design of PÅLSJÖ has been inspired and influenced by a number of other similar tools. An attempt to express the relations to these tools is shown in Figure 9.1.

System descriptions in PÅLSJÖ are block oriented, similar to how systems are described in the simulation environments Simulink and SystemBuild. Complex controllers are formed by combining a number of basic building. In Simulink and SystemBuild real-time code may be generated from these system descriptions, using Real-time Workshop or Autocode, respectively. SystemBuild allows for the user to define new blocks for simulation and code generation, while this cannot be done in Simulink. Both SystemBuild and Simulink support state machine models.

Two previous research projects at the Department that have influenced this work are LICS [Elmqvist, 1985] and Sim2DDC [Dahl, 1990]. LICS, Language for Implementation of Control Systems, is a controller description language, and was the starting point for the commercial SattLine programming language. Many of the features found in LICS and SattLine, such as double sweep execution, are also found in PAL. Sim2DDC uses the Simnon simulation language [Elmqvist *et al.*, 1990] for code generation, and thus allows the same code to be used for simulation and implementation. GrafEdit is graphical editor that generates code that is compatible with Sim2DDC.

The ControlShell [Rea, 1995] is a C++-class library with tools for adding classes and executing algorithms. ControllShell is on-line configurable and real-time processes are organized similar to PÅLSJÖ, with scheduler blocks and algorithmic blocks.

The syntax and the semantics for Grafcet in PAL and PÅLSJÖ are strongly influenced by Sequential Function Charts in the IEC-1131-3 standard [Lewis, 1995]. In IEC-1131-3 it is also possible combine Grafcet with periodic control algorithms expressed in a function block language.

Finally, ideas from the Simplex [Sha *et al.*, 1995]cite project on

fault tolerant controllers have been useful in the design of support for variable structure controllers.

Current Status

The PÅLSJÖ system has been used and tested at the Department during 1996 and 1997 in research and teaching, and has proven to be useful for rapid prototyping and experimentation with control systems. The system has also been used for testing real-time garbage collection algorithms at the Department of Computer Science at Lund Institute of Technology.

PÅLSJÖ is currently available for Motorola 68000 VME and Windows NT. PÅLSJÖ is implemented on top of the STORK real-time kernel [Andersson and Blomdell, 1991]. This is a public domain real-time kernel which is available for Windows NT, Motorola 68000, Motorola Power PC and Sun Solaris 2.x.

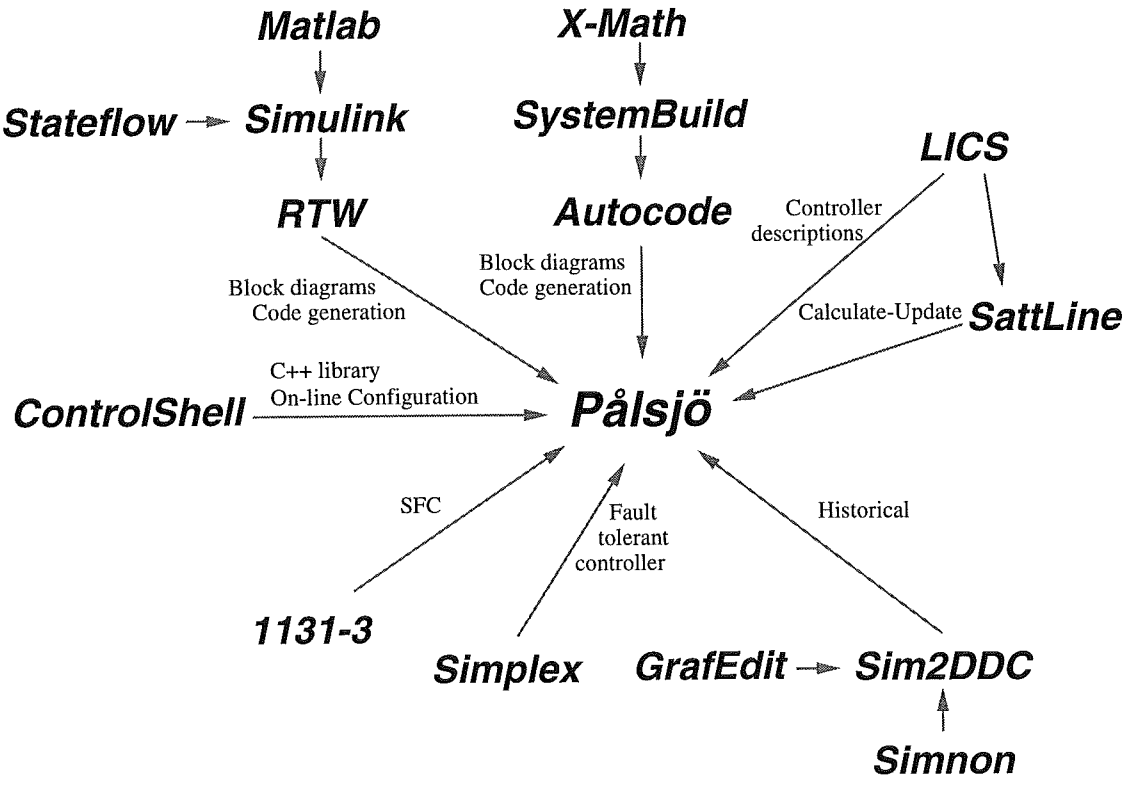


Figure 9.1 The relation between PÅLSJÖ and some other tools for creating embedded control systems.

9.2 Future Work

In this section some possible future directions for the PÅLSJÖ project are discussed.Extensions to both the PÅLSJÖ environment and to PAL care mentioned.

User Interface

A graphical user interface is an obvious possible extension. Today the user has a text interface for typing PCL commands, MATLAB for plotting data. An integrated interface for data display and system configuration is clearly needed.

Simulation

An interesting ongoing project is the integration of the PÅLSJÖ runtime system with the SIMART simulation environment [M'Saad *et al.*, 1997]. SIMART is a simulation environment for discrete systems. Currently, new algorithms that are added to the system must be implemented in C. An interface has been implemented so that PAL blocks may be executed in SIMART.

This has been done by stripping all real-time code from PÅLSJÖ, and only keeping the block administration functionality. The system is the interfaced to SIMART. PAL blocks may thus be used both for simulation and implementation. Figure 9.2 shows the interface of the SIMART simulation environment.

MATLAB and Modelica [Elmquist and Mattsson, 1997] are other interesting simulation environments. Clearly, the possibility to generate PAL code from Modelica or Matlab-files would be valuable. Generating Modelica or Matlab code from PAL files is another possibility.

9.3 Extending PAL

Several new data types and operators would be useful to further support the programmer when implementing control algorithms. A more compact notation can be obtained by using more powerful data types.

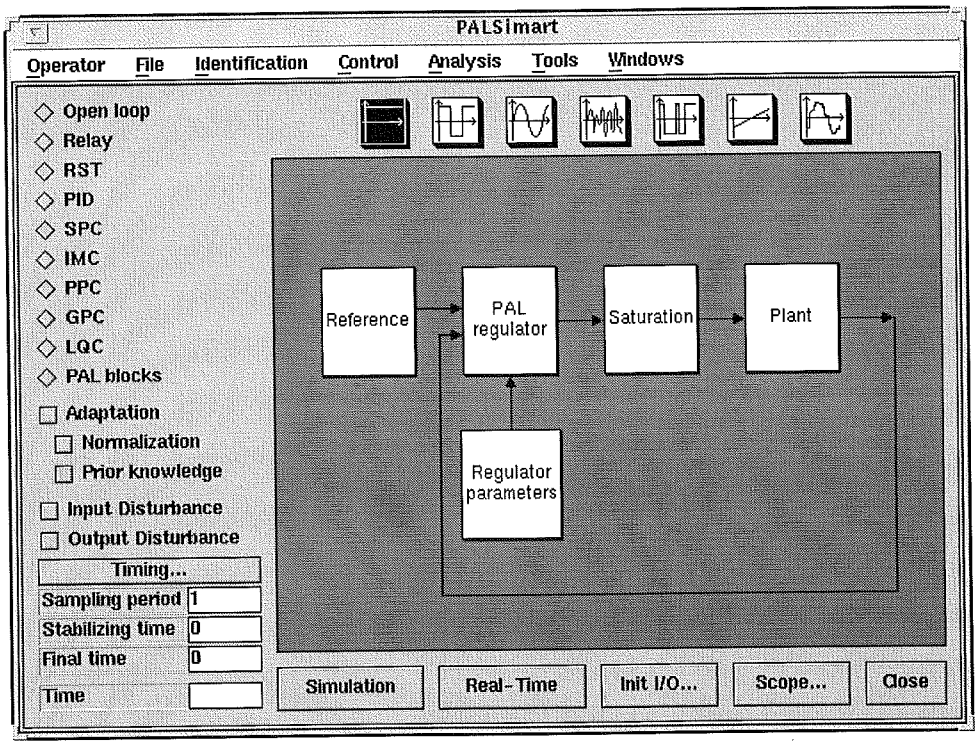


Figure 9.2 The graphical user interface of the SIMART simulation tool. It is possible to execute PAL blocks in SIMART.

Data types and operators

Event

A new data type called *event* was suggested in Section 8.6. Event is a condition variable, see [Burns and Wellings, 1997], intended for synchronizing user defined PÅLSJÖ processes. Two operations **wait** and **cause** are required to deal with events.

Delay Operator

When implementing digital controllers it is necessary to have access to previous values of certain variables. This is usually solved by introducing state variables. A more convenient way of solving this could be to introduce a backward shift operator. Let x be a state variable of a PAL block, and assume that x is updated every sampling interval. To get information of the previous value of x, n steps back in time, the

following expression is suggested

$$x(n) = x(t - n * tsamp) \tag{9.1}$$

The time horizon for a variable should be defined when the variable is declared. A ring buffer would be suitable for implementation.

Backward Polynomials

In PAL a polynomial is represented as $P = a_0q^n + a_1q^{n-1} + \cdots + a_n$, i.e. a forward shift polynomial. Some algorithms are more conveniently expressed using backward shift polynomials, $P = a_0 + a_1q^{-1} + \cdots + a_nq^{-n}$. The introduction of backward shift polynomials in PAL would give the programmer freedom to chose the representation.

Filters

A filter can be described as a pair of polynomials: a denominator and a numerator. A filter operates on an input signal and generates an output signal. The suggested filter data type consists of two polynomials as shown below

$$F(q) = \frac{B}{A} = \frac{b_0q^n + b_1q^{n-1} + \cdots + b_m}{a_0q^n + a_1q^{n-1} + \cdots + a_n}, \qquad m < n$$

Possible operations on a filter data type are

$$y = F_1 * u$$
$$F_3 = F_1 * F_2$$

where F_1, F_2 and F_3 are filters and u is a real.

Block Handles

To create autonomous systems there must be a way for one block to manipulate other blocks. This could be done in PAL by introducing block handles. A block handle is simply a reference to a block. A set of operations for block handles must also be defined. These operation would include allocation, deallocation, etc.

Packaging PAL code

In PAL, several block types are grouped together in modules. It is, however, not possible to package functions and procedures together in the same manner. It would be important to introduce functionality for handling function and procedure libraries. This includes PAL constructs for importing and exporting functions and procedures.

Inheritance

Inheritance could be a key ingredient to support code reuse and rapid prototyping. The basic idea of simple algorithms being extended through inheritance is very appealing, but introducing inheritance in PAL is not straightforward. In ordinary object-oriented languages such as Smalltalk and Simula, sub-classes are allowed to be extended and, thus, refine the super class by introducing new attributes and adding new methods or overloading old ones.

In PAL, classes correspond to block types, attributes to block variables, and methods to block algorithm descriptions.

Introducing the normal inheritance type into PAL could easily create problems, since the block algorithm is divided into at least two functions, and a Grafcet. The question is, should it be possible to change the **calculate** function without changing the **update** function? If the answer is yes, it would easily lead to inconsistent block algorithms, and if the answer is no, there would not be much of an inheritance. To deal with Grafcets is another problem. Assume that the base class contains a Grafcet, and a new Grafcet is defined in the sub-class. Should the Grafcet in the sub-class hide the Grafcet in the super class, or should the sub-class contain two Grafcets?

An inheritance model that is better suited for PAL is one where the super class simply defines a set of states and and an interface. The interface consists of input signal, output signal, and parameters. All sub-classes will share the same interface defined by the super class. The sub-classes will not inherit any functionality expressed in **calculate**, **update**, or by Grafcets.

Variable Structure Controllers

A variable structure controller consists of several sub-controller blocks, which are created and used when needed. When a running controller is

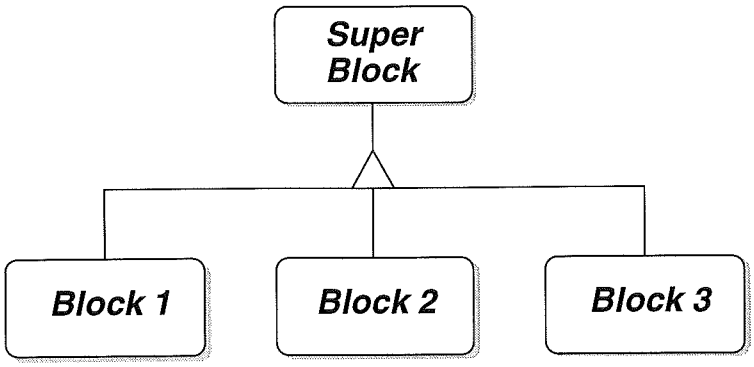


Figure 9.3 The classes Block 1, 2, and 3 are inherited from the super class Super Block.

replaced by a new algorithm, there is the problem of initiating the new controller. The information about the process states that are available in the running controller could be used to initialize the new controller. One way of handling this is to have an inheritance model where the base class constitutes an interface and a set of states. Assume that the inheritance structure is the one shown in Figure 9.3, and that Block 1 is being replaced by Block 2. Furthermore let the Super Block define an interface and a set of states. These states are used for transferring information between instances of the sub classes. Each sub class has a function F_i which calculates the internal states based on the common states defined in the super class. In a similar fashion each sub class has a function G_i which maps the internal states on the super class states. The following calculation would then initiate the new controller:

$$superblock.states = G_2(block1.states)$$
$$block2.states = F_2(superblock.states)$$

Conditional Execution

Hybrid controllers are receiving a lot of attention at the moment. A hybrid controller consists of a set of sub controllers, of which only one is active. A set of switching rules is used to decide which controller that should be active. The switching rules could be expressed using automata, if the hybrid controller was coded as one PAL block. If each sub controller is coded as PAL blocks, there must be mechanisms in PAL for starting and stopping blocks. The switching rules could be

implemented as a separate block or be taken care of by the run-time system.

Assume that the inheritance model used allows the base class to define the block interface. Classes inherited from this super class will then have exactly the same interface. If this is the case, a set of blocks that have the same super class could be treated as one logical block by other blocks. The execution of a setup where four interfaces are defined is shown in Figure 9.4.

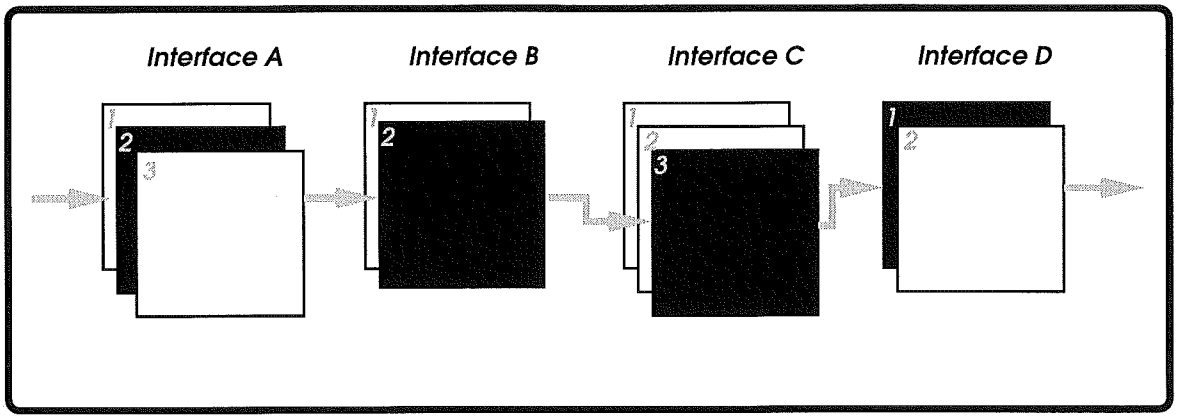


Figure 9.4 This figure demonstrates how the concept of conditional execution could work in PÅLSJÖ. Four interfaces are connected together. Each interface is implemented by a set of blocks. Which block is used depends on the working conditions. The arrows mark the data flow.

Contracts

It may be useful to add information about what a block can accomplish. To specify under which circumstances the algorithm will perform well. Each block could contain a set of rules, or preconditions, that must be fulfilled in order for the block to execute. Such a set of rules could be viewed as a contract between the run-time system and the block. Given a set of sub controllers the run-time system then selects the active controller based on the contracts. The idea with contracts is discussed in [Helm *et al.*, 1990]. In Eiffel [Meyer, 1992], there is a similar mechanism with invariants that must me fulfilled for a method to execute. For a control algorithm, contracts could look something like this:

Algorithm A

signals:
u : output real;
r, y : input real;
k : parameter real;
contract:
Periodic with h : period;
$h < 0.5$;
$0 < r < 10$ and $0 < y < 10$;
algorithm:
$u := k * (r - y)$;

Algorithm B

signals:
u : output real;
r, y : input real;
$yold$: state real;
contract:
Aperiodic with d : delay;
$d < 0.5$;
algorithm:
$u := k * (r - y)$;
exception:
$u := k * (r - yold)$;

The first sections of the algorithms above define the algorithm interfaces. The next section defines the conditions that must be fulfilled for the algorithm to execute correctly. In the aperiodic algorithm to the right there is also an exception definition which handles the case where the invariant is not true. The contract specifies under which conditions the algorithms will produce a valid result.

Consider the example with the inverted pendulum in Section 8.2. The controller consists of three sub-controllers. The sub-controller that is used is determined by the working condition, here the angle θ of the pendulum. The switching rules for the inverted pendulum controller could easily be expressed as contracts

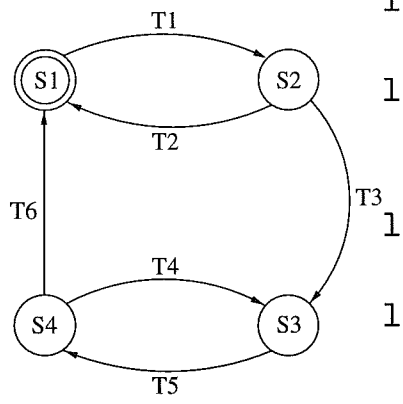
Evaluation Functions

Another approach for support of conditional execution is to attach an evaluation function to each algorithm. This evaluation function is defined by the user and gives information about how well suited the algorithm is for execution at the current working condition.

Extended Automata Primitives

PAL has primitives for expressing automata in order to support sequential algorithms. The current version of PAL supports automata expressed as Grafcets. There are several disadvantages with this notation, and a more expressive automata description is needed. One possible way is to use the type of automata that is used in the hybrid systems community, see [Alur *et al.*, 1993]. An example of such

an automata is shown in Figure 9.5. Another possibility would be to implement Statecharts [Harel, 1987] or Grafchart [Årzén, 1994], which is an object oriented extension of Grafcet.




```
loc: s1 do <Action1a>;
     when <T1> do <Action1b> goto S2;
loc: s2 do <Action2a>;
     when <T2> do <Action2b> goto S1;
     when <T3> do <Action2c> goto S3;
loc: s3 do <Action3a>;
     when <T5> do <Action3b> goto S4;
loc: s4 do <Action4a>;
     when <T4> do <Action4b> goto S3;
     when <T6> do <Action4c> goto S1;
```

Figure 9.5 The automata consists of a set locations with guarded transition. There may be actions assisted with both locations and transitions.

A

The Standard Block Library

The StandardBlocks module contains blocks for interfacing the environment as well as some standard controllers.

A.1 StandardBlocks.pal

```
module StandardBlocks;

  function AnalogIn(channel : integer) : real;
  external "Double_ADIn";

  procedure AnalogOut(channel : integer; value : real);
  external "Double_DAOut";

  block ADIn

    out := 0.0 : output real;
    channel := 0 : parameter integer;

    calculate
    begin
      out := AnalogIn(channel);
      out := AnalogIn(channel);
    end calculate;

  end ADIn;

  block DAOut

    in : input real;
```

```
    channel := 0 : parameter integer;

    calculate
    begin
      AnalogOut(channel, in);
    end calculate;

  end DAOut;

  block RefGen

    out := 0.0 : output real;
    sign := 1.0 : real;
    time := 0.0 : real;
    type := 0 : parameter integer;
    mean := 0.0, amplitud := 0.5, period := 5.0 : parameter real;
    k := 1.0 : real;
    h : sampling interval;

    calculate
    begin
      time := time + h;
      if type = 0 then
        if time >= period/2.0 then
          time := time - period/2.0;
          sign := -sign;
        end if;
        out := amplitud * sign + mean;
      else
        if time >= period/2.0 then
          time := time - period/2.0;
          sign := -sign;
          out := amplitud * sign + mean;
          k := -sign * 4.0 * amplitud/period;
        else
          out := out + k * h;
        end if;
      end if;
    end calculate;

  end RefGen;
```

```
block PI
  r, y, u : input real;
  v := 0.0 : output real;
  I := 0.0, e := 0.0 : real;
  K := 0.5, Ti := 10000.0, Tr := 10000.0 : parameter real;
  h : sampling interval;
  bi = K * h/Ti;
  br = h/Tr;

  calculate
  begin
    e := r - y;
    v := K * e + I;
  end calculate;

  update
  begin
    I := I + bi * e + br * (u - v);
  end update;

end PI;

block SimplePI
  vel, pos : input real;
  u := 0.0 : output real;
  I := 0.0, e := 0.0 : real;
  K := 0.5, Ti := 10000.0 : parameter real;
  r := 0.3 : parameter real;
  h : sampling interval;
  x0, laps, time : real;
  mode : integer;
  bi = K * h/Ti;

  calculate
  begin
    if mode = 1 then
      e := r - vel;
      u := K * e + I;
      laps := (pos - x0)/0.23;
      time := time + h;
```

```
    end if;
  end calculate;

  update
  begin
    I := I + bi * e;
  end update;

  procedure zero();
  begin
    I := 0.0;
    e := 0.0;
    laps := 0.0;
    mode := 0;
    u := 0.0;
  end zero;

  procedure on();
  begin
    mode := 1;
    x0 := pos;
    time := 0.0;
  end on;

end SimplePI;

block PID

  r, y : input real;
  v := 0.0 : output real;
  I := 0.0, e := 0.0, D := 0.0 : real;
  Dold := 0.0, yOld := 0.0 : real;
  K := 0.5, Ti := 10000.0 : parameter real;
  Td := 0.0, N := 0.0 : parameter real;
  h : sampling interval;
  bi = K * h/Ti;
  tsamp : sampling interval;

  calculate
  begin
    D := Td/(Td + N * tsamp) * Dold - K * Td * N/(Td + N *
tsamp) * (y - yOld);
```

```
      e := r − y;
      v := K ∗ e + I + D;
    end calculate;

    update
    begin
      I := I + bi ∗ e;
      yOld := y;
      Dold := D;
    end update;

    procedure zero();
    begin
      I := 0.0;
      e := 0.0;
      D := 0.0;
      Dold := 0.0;
      yOld := 0.0;
    end zero;

  end PID;

  block Filter

    u : input real;
    y : output real;
    h : sampling interval;
    N := 10.0 : parameter real;

    calculate
    begin
      y := (1.0 − N ∗ h) ∗ y + N ∗ h ∗ u;
    end calculate;

  end Filter;

end StandardBlocks.
```

B

The Polynomial Library

B.1 Introduction

In this appendix the numerical polynomial functions included in the package are described. The reader is assumed to have knowledge of polynomial design methods and to be familiar to the notations used in [Åström and Wittenmark, 1990] and [Åström and Wittenmark, 1995]. All algorithms in the functions below originate from those books unless otherwise stated.

```
void ShiftForward(Polynomial P, real r);
```

This function shifts the coefficients of the polynomial to the left and sets the last coefficient to `r`. Let $A = 2.3q^2 + 3.5q + 1.1$, then the following operation

```
A.ShiftForward(999);
```

changes A to $A = 3.5q^2 + 1.1q + 999$.

```
void ShiftBackward(Polynomial P, real r);
```

This function shifts the coefficients of the polynomial to the right and sets the highest power coefficient to `r`.

```
void GCD(Polynomial A, Polynomial B, real reps,
         Polynomial& G, Polynomial& X, Polynomial& Y,
         Polynomial& U, Polynomial& V);
```

This function calculates the greatest common divisor of polynomials `A` and `B` using Euclid's algorithm. It also solves the two equations

$$AX + BY = G \tag{B.1}$$

$$AU + BV = 0 \tag{B.2}$$

G is the greatest common divisor of A and B. A more detailed description of the algorithm is found in [Åström and Wittenmark, 1995].

```
void DiophantineMDS(Polynomial A, Polynomial B, Polynomial Ac,
                    Polynomial& R, Polynomial& S);
```

`DiophantineMDS` finds the minimal degree solution to the equation $AR + BS = A_c$. The equation is solved in two steps. First the two equations B.1 and B.2 are solved using the GCD function. Then in the second step the general solution is found through

$$R = R^0 + QU$$

$$S = S^0 + QV$$

with

$$R^0 = XA_c \operatorname{div} G$$

$$S^0 = YA_c \operatorname{div} G$$

where G is the greatest common divisor of A and B. The minimal degree solution is now given by simply choosing
$Q = S^0 \operatorname{div} V$. The greatest common G must divide A_c otherwise the equation has no solution.

```
void Sfactorize(Polynomial B, Polynomial& A);
```

Let

$$A(z) = z^{na} + a_1 z^{na-1} + \cdots + a_{na}. \tag{B.3}$$

The *reciprocal polynomial* of A, denoted A^* is obtained by reversing the order of the coefficients of A.

$$A^*(z) = 1 + a_1 z + \cdots + a_{na} z^{na} = z^{na} A(z^{-1}). \tag{B.4}$$

`Sfactorize` takes a polynomial $B = ee^*$ and returns a stable polynomial A so that $AA^* = ee^*$. This algorithm is described in [Kučera, 1979].

```
void DyadicReduction(Polynomial& A, Polynomial& B, real& Alpha,
                     real& Beta, int i0, int i1, int i2);
```

Given vectors

$$\begin{aligned} a &= [1\ a_2 \cdots a_n]^T \\ b &= [1\ b_2 \cdots b_n]^T \end{aligned} \tag{B.5}$$

and scalars α and β, find vectors

$$\begin{aligned} \tilde{a} &= [1\ \tilde{a_2} \cdots \tilde{a_n}]^T \\ \tilde{b} &= [1\ \tilde{b_2} \cdots \tilde{b_n}]^T \end{aligned} \tag{B.6}$$

such that

$$\alpha a a^T + \beta b b^T = \tilde{\alpha} \tilde{a} \tilde{a}^T + \tilde{\beta} \tilde{b} \tilde{b}^T \tag{B.7}$$

The vectors $\tilde{a}$ and $\tilde{b}$ can be found using *dyadic decomposition*. `DyadicReduction` is very useful when doing square root recursive least square estimations. For a closer look at the algorithm and its applications see [Åström and Wittenmark, 1995].

```
void RobustIntegralDesign(Polynomial   A, Polynomial   B,
                          Polynomial  Ac, Polynomial&  R,
                          Polynomial&  S, real x0, real x1);
```

This function calculates an integral controller with zero gain at the
Nyquist frequency. Two additional closed loop poles are specified through `x0` and `x1` which are coefficients in the X-polynomial, see below.

First the minimal degree solutions R^0 and S^0 are calculated. If R^0 and S^0 satisfy

$$AR^0 + BS^0 = A_c$$

then

$$R = XR^0 + YB$$

$$S = XS^0 - YA$$

are solutions to the equation

$$AR + BS = XA_c$$

This gives a controller with the characteristic polynomial A_cX, where $X = q^2 + x_1q + x_0$. To get the desired controller first let $Y = y_oq - y_1$. Then solve

$$R(1) = 0 \Leftrightarrow 0 = -X(1)R^0(1) + Y(1)B(1)$$

$$S(-1) = 0 \Leftrightarrow 0 = X(-1)S^0(-1) - Y(-1)A(-1)$$

By using those equations the coefficients of the Y polynomial can be calculated.

$$y_0 = \frac{Y(1) - Y(-1)}{2}$$

$$y_1 = \frac{Y(1) + Y(-1)}{2}$$

The two following functions use the same method to calculate a robust controller and an integral controller.

```
void IntegralDesign(Polynomial  A, Polynomial  B, Polynomial C,
                    Polynomial& R, Polynomial& S, real x0);
```

This function first solves the diophantine equation and then forces the R polynomial to contain an integrator. This is done by designing the R polynomial so that $R(1) = 0$. The input parameter `x0` specifies the additional closed loop pole. The characteristic polynomial now becomes $A_c X$, where $X = q - x_0$.

```
void RobustDesign(Polynomial  A, Polynomial  B, Polynomial C,
                  Polynomial& R, Polynomial& S, real x0);
```

`RobustDesign` works similar to `IntegralDesign` but the constraint on the controller design is instead $S(-1) = 0$. This condition gives a controller with zero gain at the Nyquist frequency. The input parameter `x0` specifies the additional closed loop pole. The characteristic polynomial now becomes $A_c X$, where $X = q - x_0$.

```
void LQGDesign(Polynomial A, Polynomial B, Polynomial C,
               Polynomial& R, Polynomial& S, real rho);
```

Calculates a LQG-controller for the system A, B, and C with the loss function coefficient ρ. The computational procedure is described in detail in section 12.5 in [Åström and Wittenmark, 1990]. This implementation only handles the case where $A(0) \neq 0$.

```
void MDPPNZCDesign(Polynomial A, Polynomial B, Polynomial Am,
                   Polynomial Ao, Polynomial& R, Polynomial& S,
                   Polynomial& T);
```

The abbreviation stands for Minimal Degree Pole Placement with No Zero Cancelation. The function chooses $B^+ = 1$ and $B^- = B$.

Furthermore B_m is chosen so that the stationary gain will be unity.

$$B_m(q) = \frac{A_m(1)B(q)}{B(1)}$$

The closed-loop characteristic equation to be solved now becomes

$$AR + BS = A_o A_m$$

The T polynomial is given by

$$T(q) = \frac{A_m(1)A_o(q)}{B(1)}$$

```
void Roots(Polynomial A, Polynomial& rootRe,
           Polynomial& rootIm);
```

The roots of polynomial `A` are calculated and are returned in the two polynomials `rootRe` and `rootIm`. The real parts of the roots are stored in `rootRe` and the imaginary parts are stored in `rootIm`.

```
void LDFilter(double *l, Polynomial& d, Polynomial& phi,
              Polynomial& theta, double& lambda);
```

This function is an implementation of an estimator and would together with any of the design functions above form a complete adaptive controller. The algorithms behind `LDFilter` is taken from [Åström and Wittenmark, 1995]. Let the system to be estimated be on the form

$$y(t) = \varphi^T(t)\theta$$

where θ is a parameter vector and φ is a vector of signals. The following recursive least-square estimation algorithm is used

$$\begin{aligned}
\hat{\theta}(t) &= \hat{\theta}(t-1) + K(t)(y(t) - \varphi^T(t)\hat{\theta}(t-1)) \\
K(t) &= P(t)\varphi(t) \\
P(t) &= (I - K(t)\varphi^T(t))P(t-1)
\end{aligned}$$

The covariance matrix P has a decomposition $P = LDL^T$, where L is a lower triangular matrix and D is a diagonal matrix. Initially set $L = I$, which gives that $P(0) = D$. The function takes the following arguments:

- `double *l`
  An array of doubles with the size $deg(\theta) \times deg(\theta)$.
- `double *d`
  An array of doubles with the size $deg(\theta)$.
- `Polynomial& Phi`
  This a polynomial that contains old process values and old control signals. The polynomial is arranged on the following format:

$$\texttt{Phi} = [y(t), -y(t-1), \ldots, -y(t-n), u(t-d), \ldots, u(t-d-m)]$$

  where $n = deg(A), m = deg(B)$, and $d = n - m$.
- `Polynomial& theta`
  This is a polynomial with the degree set to $(deg(A) + deg(B) + 1)$ and with the coefficients to be estimated on the following format:

$$\texttt{theta} = [a_0, a_1, \ldots, a_n, b_0, \ldots, b_m]$$

- `double& l`
  This parameter is the forgetting factor λ.

C

Patterns and Framework

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
> Christopher Alexander

C.1 Introduction

This appendix will give a brief introduction to pattern and the notation used when describing them. Further there will a short description of the concept of frameworks. This text is based on [Gamma *et al.*, 1995], [Buschman *et al.*, 1996] and [Hedin, 1997].

C.2 Patterns

The main idea with patterns is to capture the expertise of an experience programmer and document it in a standard form, a pattern. Originally patters were introduced by the architecture Christopher Alexander, as a way of accumulating wisdom. He demonstrated the use of patterns in the context of architecture of buildings. Using pattern it is possible to capture qualities which are otherwise difficult to describe. The idea of recording experience using pattern has successfully

been adopted by the software community. The pattern approach has proved a very suitable way for documenting solutions to complex software problems. An experienced programmer has standard solutions to standard problems. Pattern is a good way of recording this experience and make it available to novices. Further it simplifies communication by establishing a common terminology.

Design patterns are usually described according to the template given in [Gamma *et al.*, 1995]. This template has the following items:

- **Pattern Name and Description** The name of the pattern and a short description.
- **Also Known As** Other names that the pattern may be known under.
- **Intent** The problem that the pattern is supposed to solve.
- **Motivation** A scenario that illustrates a situation
- **Applicability** In what situations the patterns are suitable.
- **Structure** Usually a graphical description of the class hierarchy and the object relations. The class diagram notation is illustrated in Figure C.1.
- **Participants** The different objects that interact in the pattern.
- **Consequences** A discussion on the advantages and disadvantages of using the pattern.
- **Implementation** Guide lines for implementing the pattern. What pit-falls, hints, or techniques the programmer should be aware of.
- **Sample Code** Code that illustrates the how the pattern could be implemented.
- **Known Uses** Applications where the pattern are used.
- **Related Patterns** Other patterns that are related either in the structure or in the intent.

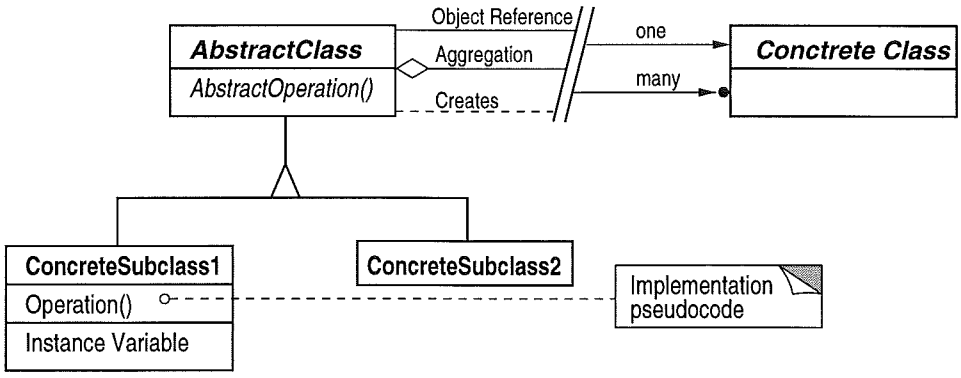


Figure C.1 Class Diagram Notation

C.3 Frameworks

While the basic idea with patterns is to reuse design, the basic idea with frameworks is to reuse both design and code. A framework is an integrated set of cooperating classes aimed for a family of similar applications. Frameworks usually consists of a shell application which handles the tasks that are regarded as general to the whole family of applications. The application programmer then simply creates subclasses from abstract framework classes, in order to create the final application. The "main event loop" of the application is often a part of the framework. The use of frameworks promotes faster application development. In order to create good flexible frameworks the notation of design patterns is often used. Further the use of patterns is a good way of documenting frameworks.

D

Bibliography

ALUR, R., C. COURCOUBETIS, T. A. HENZINGER, and P.-H. HO (1993): "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems." In GROSSMAN *et al.*, Eds., *Hybrid Systems, Lecture Notes in Computer Science*, pp. 209–229. Spronger-Verlag.

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

ANDERSSON, M. (1994): *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis ISRN LUTFD2/TFRT--1043--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ARMSTRONG, J., R. WIRDING, and M. WILLIAMS (1993): *Concurrent Programming in Erlang*. Prentice Hall.

ÅRZÉN, K.-E. (1994): "Grafcet for intelligent supervisory control applications." *Automatica*, **30:10**, pp. 1513–1526.

ÅRZÉN, K.-E. (1996): "Lecture notes on real-time control systems."

ÅSTRÖM, K. J. (1993): "Autonoma regulatorer-en projektansökan till nutek."

ÅSTRÖM, K. J. and K. FURUTA (1996): "Swinging up a pendulum by energy control." In *IFAC'96, Preprints 13th World Congress of IFAC*, vol. E, pp. 37–42. San Francisco, California.

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

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

ÅSTRÖM, K. J. and B. WITTENMARK (1995): *Adaptive Control*, second edition. Addison-Wesley, Reading, Massachusetts.

BENVINISTE, A. and G. BERRY (1991): "The synchronous approach to reactive and real-time systems." In *Proceedings of the IEEE*, vol. 79. IEEE.

BLOMDELL, A. (1997): "The pålsjö algorithm language.". Master's thesis, Department of Automatic Control, Lund Institute of Technology.

BOUNSSINOT, F. and R. DE SIMONE (1991): "The ESTEREL language." In *Proceedings of the IEEE*, vol. 79. IEEE.

BURNS, A. and A. WELLINGS (1997): *Real-time Systems and their programming languages*, 2nd edition. Addison-Wesley.

BUSCHMAN, F., R. MEUNIER, P. ROHNERT, H. SOMMERLAD, and M. STAL (1996): *A System of Patterns-Pattern Oriented Software Architechture*. Wiley.

DAHL, O. (1990): "SIM2DDC—User's manual." Report TFRT-7443. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

DAVID, R. and H. ALLA (1992): *Petri Nets and Grafcet: Tools for modelling discrete events systems*. Prentice-Hall.

DAVIS, R. B. (1997): "Newmat, a matrix library in c++." http://nz.com/webnz/robert/.

DESOER, C. A. and M. VIDYASAGAR (1975): *Feedback Systems: Input-Output Properties*. Academic Press, New York.

EKER, J. and K. J. ÅSTRÖM (1995): "A C++ class for polynomial operation." Report ISRN LUTFD2/TFRT--7541--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQUIST, H. and S.-E. MATTSSON (1997): "Modelica-the next generation modeling language an international effort." In *Proceedings of 1st World Congress on System Simulation*. Singapore. http://www.dynasim.se/Modelica/.

ELMQVIST, H. (1985): "LICS—Language for implementation of control systems." Report TFRT-3179. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H., K. J. ÅSTRÖM, T. SCHÖNTHAL, and B. WITTENMARK (1990): *Simnon User's Guide*. SSPA, Göteborg, Sweden.

FURUTA, K., S. KOBAYASHI, and M. NISHIMURA (1991): "A new inverted pendulum apparatus for eduction." *Preprints IFAC Conference on Advances in Control Educations*, pp. 191–196.

GAMMA, E., R. HELM, J. R., and J. VLISSIDES (1995): *Design Patterns-Elements of Reusable Object-Oriented Software*. Adison-Wesley.

GUSTAVSSON, K. (1994): "An architecture for autonomous control." Technical Report. Department of Automatic Control.

HALBWACHS, N. (1993): *Synchronous programming of reactive systems*. Kluwer Academic Pub.

HALBWACS, N., P. CASPI, P. RAYMOND, and D. PILAUD (1991): "The synchronous data flow programming language LUSTRE." In *Proceedings of the IEEE*, vol. 79. IEEE.

HAREL, D. (1987): "Statecharts:a visual formalism for complex systems." *Science of Computer Programming*, **8**, pp. 231–274.

HEDIN, G. (1997): "Lecture notes on patterns and frameworks in object-oriented programming."

HELM, R., I. M. HOLLAND, and GANGOPADHYAY (1990): "Contracts:specifying behavioral compositions in object-oriented systems.". ECOOP/OOPSLA.

INTEGRATED SYSTEMS INC., 3260 Jay Street, Santa Clara, CA 95054-3309, USA (1996a): *AutoCode User's Guide.*

INTEGRATED SYSTEMS INC, 3260 Jay Street, Santa Clara, CA 95054-3309, USA (1996b): *SystemBuild User's Guide.*

JOHANNESSON, G. (1994): *Object-oriented Process Automation with SattLine.* Chartwell Bratt Ltd. ISBN 0-86238-259-5.

KHALIL, H. K. (1992): *Nonlinear Systems.* MacMillan, New York.

KNUTH, D. E. (1969): *The Art Of Computer Programming, Volume2 / Semi-numerical Algorithms.* Addison-Wesley Publishing Company.

KOENIG, A. R. (1989): "Effective use of C++." Course material used in a course given by the author 1989 at the Department of Automatic Control, Lund, Sweden.

KUČERA, V. (1979): *Discrete Linear Control—The Polynomial Equation Approach.* Wiley, New York.

LE LANN, G. (1996): "The ariane flight 501 failure - a case study in system engineering for computing systems." In *Hand-Out European Educational Forum School on Embedded System.* European Educational Forum.

LEWIS, R. (1995): *Programming industrial control systems using IEC 1131–3.* The Institution of Electrical Engineers, London, U.K.

LIPMAN, S. B. (1991): *C++ Primer*, 2nd edition. Addison-Wesley Publishing Company.

MAGNUSSON, B., M. LÖFGREN, H. ELMQUIST, C. FERNSTRÖM, I. KRUZELA, and T. SCHÖNTAL (1981): "Regula - an interactive user oriented language for implementation of control systems." Technical Report LUTFD2/(TFRT-7213)/1-027/(1981). Department of Automatic Control, Lund Institute of Technology.

MALMBORG, J., B. BERHARDSSON, and K. J. ÅSTRÖM (1996): "A stabilizing switching scheme for multi-controller systems." In *Proceedings of the 1996 Triennal IFAC World Congress, IFAC'96*, vol. F, pp. 229–234. Elsevier Science, San Francisco, California, USA.

THE MATHWORKS INC., 24 Prime Park Way, Natick, MA 01760-1500 (1997): *Simulink–Real-Time Workshop.*

MEYER, B. (1992): "Applying design by contract." *Computer.*

M'SAAD, M., J. CHEBASSIER, and P. TONA (1997): "Simart: A cacsd package for advanced control." In *Proceedings of the Conference on System Structure and Control.* IFAC, Bucharest Romania. To appear.

NILSSON, K. (1996): "Private communication."

REAL-TIME INNOVATIONS, INC., 155A Moffet Park Drive, Suite 111, Sunnyvale, CA 94089, USA. (1995): *ControlShell–Object–Oriented Framework fro Real–Time System Software*, version 5.1 edition.

SCHAEFER, J. F. (1965): *On the bounded control of some unstable mechanical systems.* Thesis, Stanford University.

SCHAEFER, J. F. and R. H. CANNON (1967): "On the control of unstable mechanical systems." In *Automat. Remote Contr. III, Proc. 3rd Int. Fed. Automat. Contr. (IFAC)*, vol. 1, 6C.1-6C.13.

SHA, L., R. RAJKUMAR, and M. GAGLIARDI (1995): "A software architecture for dependable and evolvable industrial computing systems." http://www.sei.cmu.edu/.

SHAW, M. (1994): "Beyond objects: A software design paradigm based on process control." Technical Report CMU/SEI-94-TR-15. Software Engineering Institute, Carnegie-Mellon University.

SHIMKIM, N. and A. FEUER (1988): "On the necessity of 'block invariance' for convergence of adaptive pole-placement algorithm with persistently exciting input." *IEEE Transactions on Automatic Control*, **33:8**.

SHOPIRO, J. E. (1991): "Advanced C++." Course material used in a course given by the author 1991 at the Department of Automatic Control, Lund, Sweden.

TÖRNGREN, M. (1995): *Modelling and Design of Distributed Real-time Control Applications.* PhD thesis, DAMEK Researh Group, Department of Machine Design, Royal Institute of Technology, Stockholm.

WIKLUND, M., A. KRISTENSON, and K. J. ÅSTRÖM (1993): "A new strategy for swinging up an inverted pendulum." In *Preprints IFAC 12th World Congress*. Sydney, Australia.