Flexible Embedded Control Systems

Design and Implementation

Eker, Johan

1999

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

Flexible Embedded Control Systems

Design and Implementation

Johan Eker

Automatic Control

Flexible Embedded Control Systems
Design and Implementation

Flexible Embedded Control Systems

Design and Implementation

Johan Eker

Lund 1999

Till Lotta

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

ISSN 0280–5316
ISRN LUTFD2/TFRT–1055–SE

Printed in Sweden by Wallin & Dahlholm Boktryckeri AB
Lund 1999

Contents

Acknowledgments

Starting on my first day of undergraduate studies in Lund I heard people talking about Professor Åström. Nobody seemed to know what he did. They had just heard about him. A couple of years later when I did my master's thesis at the department I got to know the object of all the rumors. Karl Johan Åström has so much energy and such a drive, that it is hard not to the get carried away. And I was. Now the thesis is written, and I owe my gratitude to many people who have helped me along the way.

My main thanks go to my supervisors Karl Johan Åström and Karl-Erik Årzén, without whom this thesis would never have been completed. Karl-Erik has done a great job in reading and improving this thesis, and for this I am very grateful! Karl Johan's overwhelming enthusiasm has kept me going when things felt tough. Working with Karl Johan and Karl-Erik has been a great pleasure and lots of fun.

Many thanks to Anders Blomdell, the best programmer I have ever met, for always being anxious to deconstruct my vague ideas.

Thanks to my roommate and dear friend Erik Möllerstedt for putting up with my mood swings, and for solving those equations for me.

Lui Sha changed my view on embedded control and much of the work in this thesis is directly or indirectly inspired by his ideas. Thanks for being so generous!

I truly enjoyed working with Jörgen Malmborg on the hybrid controller project. I have spent many good times together with him and Johan Nilsson.

I would like to thank all the people who work, or have worked, at the department during these five years. Per Hagander made sure that I got a Riccati equation into the thesis. Bo Bernhardsson has read and commented on various manuscripts. He is always willing to switch focus and help out. Anton Cervin did a great job on the toolbox. He also read different versions of the papers, turning proof reading into an art form. LEIFA has provided excellent computer facilities. Anders Robertsson was very helpful in getting the robotics experiments to work. John Hauser vigorously played the devil's advocate and pointed out some blank spots in the thesis. Klas Nilsson came with some useful last minute comments. Paolo Tona did great work putting PALSIMART together. Thank you all!

This work has been supported by the Swedish National Board for Industrial and Technical Development (NUTEK). I would also like to thank the Swedish real-time research network ARTES for their support of my work.

Finally, thanks to all my friends and family for supporting and encouraging me, and for giving me so many good reasons not to work! Most of all, thanks to Lotta for being the apple of my eye.

Johan

Introduction

1. Motivation

The number of embedded systems is increasing rapidly. Processors shipped for embedded systems are already outnumbering processors for desktop systems. According to [Hennessy, 1999], the ratio between the number of 32 and 64 bit processors aimed for embedded systems and desktop systems is expected to increase to three to one in the next five years. If 8 and 16 bits processors are also considered, the difference becomes even more profound. The large number of embedded systems puts high demands on the software. These systems are expected to function more or less autonomously and must be robust and adaptive to changes. The advantage of using a system may easily be overshadowed by the cost of a failure. If, for example, the cruise controller of a car fails ever so rarely, the potential benefits from using it become very small.

Most embedded systems today are designed in an ad-hoc fashion. They are tuned for well-defined software and hardware environments, and validated through extensive testing. If the conditions change, the behavior of the system may be unpredictable. For example, the failure of the Ariane-5 rocket was due to the use of a code module from Ariane-4, which caused a fault when executed on a more modern processor [Le Lann, 1996]. The more we start to depend on embedded control systems for critical tasks, the higher are the demands on reliability and safety. Maneuvering systems in airplanes are more and more relying on computers, and it is unlikely that the pilot can save the situation by pressing Ctrl-Alt-Delete when an unrecoverable application error occurs. By making embedded systems more adaptive to changes in the environment, they would be less likely to fail. Having a more flexible set-up would also allow the support for on-line software upgrades. Being able to change embedded software without shutting down the plant is highly desirable, as it would save both time and money. An upgrade must, however, be done in a safe and well-defined way. The implementation of advanced control systems require good interactive environments, which are nontrivial to design because of the real-time aspects.

Many applications for embedded systems are control systems. In this thesis, some software issues for embedded control systems are investigated. An interactive development tool is presented together with two languages for controller implementation.

There is rarely any interaction between the control loops and the underlying real-time kernel or operating system. From a control engineering perspective, the system executes in open loop. In this thesis, the possibility of using feedback in the scheduling of real-time tasks is explored. This is called *feedback scheduling*. An algorithm for selecting sampling rates based on the quality of the control performance is presented.

By viewing the control task, the underlying real-time kernel, and the controller processes as an interconnected system, it is possible to study how the interaction between the real-time tasks influences the control performance. A toolbox which supports simulation of such systems is presented in the thesis. This more holistic view on embedded control systems enhances the possibilities for designing systems with predictable behavior.

2. Outline and Summary of Contributions

The thesis consists of this introduction and six papers. Below, the contents and major contributions of each paper are summarized. References to related publications are also given.

Paper 1 and Paper 2

The PÅLSJÖ environment for embedded control systems is presented in two papers:

Eker, J.: "A Tool for Interactive Development of Embedded Control Systems." In *Proceedings of the 14th IFAC World Congress*, 1999, Beijing, China.

Eker, J., and A. Blomdell: "A Flexible Interactive Environment for Embedded Controllers." To appear in Control Engineering Practice.

Contributions

Design and implementation of a software environment for dynamically configurable embedded systems and design of the controller description language PAL.

Related publications

Tona P., J. Eker and M. M'Saad: "PALSIMART: A New Framework for Computer Aided Rapid Prototyping of Advanced Control Systems."

In *Proceedings of the European Control Conference*, 1999, Karlsruhe, Germany.

Robertsson, A., J. Eker, K. Nilsson and R. Johansson: "Application Specific Control for Industrial Robots." Poster presentation at the 3rd Portuguese Conference on Automatic control—Controlo98, 1998, Lisboa, Portugal.

Eker, J.: "A Framework for Dynamically Configurable Embedded Controllers." Licentiate Thesis ISRN LUTFD2/TFRT–3218–SE. Department of Automatic Control, Lund Institute of Technology, November 1997, Lund.

Eker, J., and A. Blomdell: "Patterns in Embedded Control." Technical Report TFRT-7567. Department of Automatic Control, Lund Institute of Technology, December 1997, Lund.

Eker, J., and A. Blomdell: "A Structured Interactive Approach to Embedded Control." In *Proceedings of 4th International Symposium on Intelligent Robotic Systems*, July 1996, Lisboa, Portugal.

Eker, J., and K. J. Åström: "A C++ Class for Polynomial Operations.", Technical Report TFRT-7541. Department of Automatic Control, Lund Institute of Technology, December 1995, Lund.

Paper 3

Based on the experiences from PÅLSJÖ and PAL, a new language called FRIEND is proposed. The FRIEND language supports the implementation of flexible embedded control systems by providing mechanisms for adding and removing code on-line.

Eker, J., and A. Blomdell: "A Contract-Based Language For Embedded Control Systems." In submission to the 25th IFAC/IFIP Workshop on Real-Time Programming WRTP'2000, Palma de Mallorca, Spain.

Contributions

FRIEND supports the implementation of flexible control systems through the use of contracts. In FRIEND, a control algorithm may be associated with a contract, which defines the expected behavior of the algorithm. The contracts allow the run-time system to distinguish between blocks that are working correctly or incorrectly. The notion of contracts in the language supports the development of systems with a high level of adaptivity and fault tolerance.

Paper 4

A feedback scheduler algorithm is presented. Given that all the control tasks are associated with cost functions, it finds the optimal set of sam-

pling frequencies.

Eker, J., P. Hagander and K.-E. Årzén: "A Feedback Scheduler for Real-time Control Tasks." In submission to Control Engineering Practice.

Contributions

A feedback loop between the control tasks and the real-time kernel is designed based on LQG cost functions. How the cost depends on available computing resources is investigated and an algorithm for finding the optimal resource allocation scheme is proposed.

Related publications

Årzén, K.-E., B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha: "Integrated Control and Scheduling." Technical Report ISRN-7586. Department of Automatic Control, Lund Institute of Technology, August 1999, Lund.

Paper 5

A toolbox that supports simulation of closed loop embedded control systems at task level is presented.

Eker, J., and A. Cervin. "A MATLAB Toolbox for Real-Time and Control Systems Co-Design." To appear in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, December 1999, Hong-Kong, China.

Contributions

A toolbox for simulation of embedded systems is designed and implemented. The control algorithm, the dynamic process, and the underlying real-time kernel is viewed as one system, and its behavior is investigated through simultaneous simulation of both the process dynamics and the real-time kernel. The toolbox also supports simulation of distributed control systems connected by a network.

Paper 6

This paper presents the design of a hybrid controller for a heating/ventilation process.

Eker, J., and J. Malmborg: "Design and Implementation of a Hybrid Control Strategy." IEEE Control Systems Magazine, vol. 19, number 4, August 1999.

Contributions

A hybrid control strategy based on Lyapunov function arguments is first designed. A simplified version of the control law is then proposed and

implemented on a commercial embedded control system. The controller is tested on a heating/ventilation process with good results.

Related publications

Malmborg, J., and J. Eker: "Hybrid Control of a Double Tank System." In *Proceedings of the IEEE International Conference on Control Applications*, October 1997, Hartford, Connecticut.

Other Related Publications

The nonlinear observer used in the robot and pendulum example later in this introduction is presented in

Eker, J., and K.J. Åström: "A Nonlinear Observer for the Inverted Pendulum." In *Proceedings of the IEEE Conference on Control Applications*, 1996, Dearborn, Michigan.

The work on a feedback scheduler for a set of double-tank processes, presented in Section 8, will appear in

Cervin, A., J. Eker and L. Sha: "Improving Control Performance by Feedback Scheduling." Manuscript in preparation.

Chapter Outline

The PÅLSJÖ environment and the PAL language are introduced in Section 3. The continuation of this work, the proposed FRIEND language, is discussed in the same section. The use of PÅLSJÖ/PAL is demonstrated by an example in Section 4. A brief overview of the state-of-the-art in integrated control and scheduling is given in Section 5. The presentation is based on [Årzén *et al.*, 1999]. Section 6 discusses tools for simulation of embedded controllers. The PALSIMART environment and a MATLAB/Simulink toolbox are presented. Section 7 discusses the implementation of the controller from Paper 6. Finally, in Section 8, a unifying example is given where tools and ideas from the different papers are combined.

3. Implementation of Control Systems

Many real-time control systems are components embedded in a larger engineering system. They are often implemented on small microcontroller systems. Examples of embedded systems are found in mechatronic systems such as industrial robots, in aerospace applications such as airplanes and satellites, in vehicular systems such as cars, and in consumer electronic products. Embedded controllers are often complex systems, con-

sisting of several concurrent tasks such as low-level feedback control, supervision logic, data logging, and user communication. Special language support is needed in order to implement embedded control systems in an easy and straightforward way. Control laws are conveniently described using polynomials or matrices, and access to these data types is an advantage. Furthermore, many algorithms are described as a combination of periodic behavior and sequential logic. Any good language should support both of these descriptions. Finally, it should be easy to reuse algorithms from one application to another. One of the problems with reusing real-time algorithms is that statements describing the algorithm are often mixed with statements specifying the real-time behavior. Code reuse may be better supported by separating the algorithm code and the real-time specific information. Implementing embedded control systems is an error prone and complicated task. The nondeterministic nature of the programs makes them hard to test and verify. A program that works in one environment may fail in another. One reason for this is the dynamic allocation of resources. For example, a change in the input stream may lead to changes in the execution order of the processes, and a new run-time scenario is created. Implementation of concurrent tasks and task communication requires firm programming discipline. Incorrect implementation of real-time tasks may cause errors that are very difficult to trace.

Embedded systems must be efficient, because the execution is constrained by deadlines that should not be missed. Many large and complex systems need to be implemented using low-level techniques, in order to get the desired performance. Many control applications are implemented in assembly language or low-level languages such as Forth or C, due to the requirements for fast execution and small program sizes. Other common languages are Modula-2 or Ada, which have built-in support for concurrency. They provide a higher abstraction level, but will give larger and slower programs. No matter which approach is used, the implementation of real-time controllers becomes time-consuming and error prone. The popular language C++ has no built-in support for concurrency, and the weak typing, the pointer arithmetics and the memory management are constant sources to problems. The more appealing object-oriented language Java is currently emerging as an alternative, not only for desktop applications, but also for real-time systems. Usually, there is a trade-off between performance and high abstraction levels; the more support the language gives for structured programming, the larger and more inefficient the resulting programs usually become.

Distributed control systems are common in the process and manufacturing industry, and are typically programmed using sequential function charts, function block languages, or relay or ladder diagram languages. Distributed control systems and programmable logic controllers

share many of the characteristics of embedded systems. Many of these systems have the bizarre property that the order of execution depends on the order in which the systems are configured. This can lead to many strange effects, particularly when changes are made.

Several real-time languages have been proposed. One is the functional language Erlang [Armstrong *et al.*, 1993], developed by the Swedish telecom industry. It has 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. The language HI-Pearl is an extension of the Pearl programming language, augmented with real-time constructs [Stoyenko and Halang, 1993]. Pearl was designed in the late 1960's as a process control language for industrial automation problems. Real-Time Euclid offers, similarly to HI-Pearl, schedulability analysis of the compiled code [Kligerman and Stoyenko, 1986]. It is a Pascal-style language where all constructs are space- and time-bounded. Hence, recursion and dynamic memory allocation are not allowed. The language FLEX [Natarajan and Lin, 1988] is designed to support the implementation of dynamic and flexible real-time systems. By incorporating imprecision in the language, FLEX makes it possible to adjust execution times of tasks, so that deadlines may be met. The synchronous language approach has emerged as a paradigm well suited for design and implementation of safety-critical real-time systems [Halbwachs, 1993; Benveniste and Berry, 1991]. At the design level, the synchronous languages use a concurrent task model, but after compilation the program is completely sequential. The resulting programs are deterministic and possible to analyze. There are currently a number of synchronous languages available, among which Esterel [Boussinot and de Simone, 1991], Lustre [Halbwachs *et al.*, 1991], and Signal [Halbwachs, 1993] are the most well known.

There is an obvious need for languages dedicated towards special problem domains. General purpose languages may be used for implementing real-time embedded control systems, but it is unnecessarily awkward and time consuming. By using languages targeted for real-time systems, the programmer will be better supported in the task of writing safe and predictable software.

Pålsjö

Many control systems have a similar internal structures. There are for example functions for control logic, sequencing, user communication, task management, and data logging. If the software modules for these basic common activities are arranged in a framework, the user of such a framework only needs to add the code that is specific for a certain application. In the case of embedded control systems, the necessary code is typically the control algorithm.

The PÅLSJÖ environment is an attempt to provide such a framework. The project was outlined in 1994 [Gustafsson, 1994], as a part of the project "Autonomous Control". The goal was to create an environment for experiments in automatic control. Initially, a C++ class library was proposed, but the library soon became very large and cumbersome to use. 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, which stands for PÅLSJÖ Algorithmic Language [Blomdell, 1997], and was designed to support controller implementation. 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 is supported 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 using PCL, PÅLSJÖ Configuration Language. PCL is a simple language for managing blocks and assigning variables. The system may be reconfigured on-line while the system is running. Introductions to PAL and PCL are found in Appendix A and B.

Friend

The more interactive way of working with embedded systems, as proposed in the PÅLSJÖ/PAL project, creates new problems. PÅLSJÖ admits that blocks are changed on-line. Allowing blocks to be added or deleted dynamically will, for example, give rise to timing and scheduling problems. In PÅLSJÖ, it is entirely up to the user to make sure that nothing goes wrong. There are no language constructs in PAL to assist the programmer.

The design of the FRIEND language addresses the need for built-in language support, when implementing fault-tolerant and adaptive embedded control systems. Similarly to PÅLSJÖ/PAL, FRIEND is block-oriented, i.e., a control system is defined by a number of blocks and their interconnections.

To support implementation of control systems that allow blocks to be inserted and replaced on-line, the use of *contracts* is introduced. The contract defines the behavior of a block and may be used by the run-time system to determine:

- If a new block *can* replace an old block. Does it, for example, have the desired interface towards other blocks?

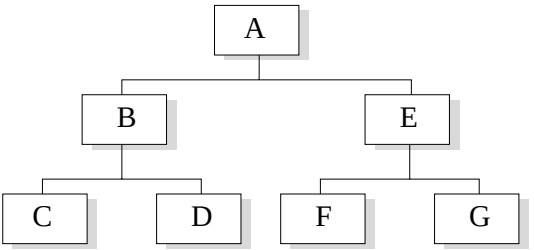


Figure 1. The hierarchical structure of a real-time control system designed in FRIEND. Resources are distributed from the top down. The parent blocks control the execution of its child blocks.

- If a new block executes *correctly*. The contract may contain ways to verify the functionality of a block.

Blocks are organized in a hierarchical fashion as shown in Figure 1. There are two block types in FRIEND: *algorithm* and *negotiator* blocks. An algorithm is an input-output block that calculates new output signals based on the values of the inputs signals, the internal states, and the parameters. A negotiator is a block that allocates resources and distributes them among its child blocks. The negotiator is a sort of resource broker. The *interface* specifies how a block interacts with other blocks. The interface is useful for on-line decisions about whether or not a new block can replace an old block. Figure 2 shows an example of a FRIEND block diagram. At at the top level it consists of three blocks. Block B has three child blocks inside: block D, E, and F. These blocks may either execute concurrently or exclusively, depending on the application. If yet another block should be added to B, it is necessary that it fulfills some requirements in order to work correctly with the rest of the blocks. This type of guarantees may be expressed with contracts. The resources of block B are distributed between the blocks D, E, and F, by the supervision logic of block B. The process of distributing resources among child blocks is in FRIEND called negotiation. The reasons for dividing blocks into subcomponents are:

- Better reuse of control algorithms by letting the control algorithm be described in an algorithm block and the application-specific information in a contract.
- Better support for on-line controller upgrades. By introducing interfaces it becomes easy to check whether or not the block reads and writes signals as specified. Furthermore, the interface may contain signal-data logs, needed to assign correct values to the controller states before switching in.

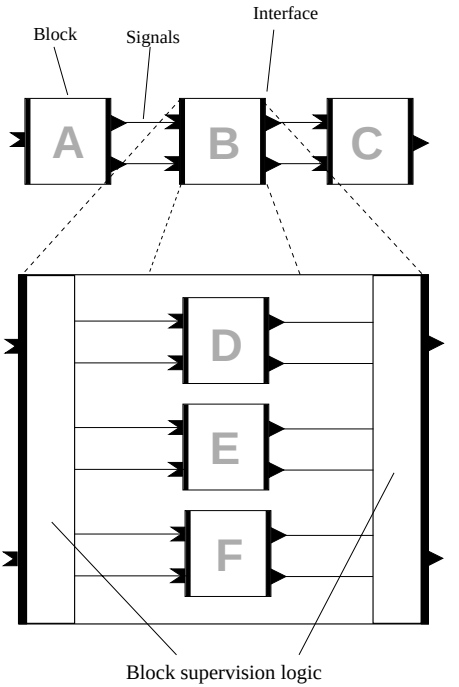


Figure 2. A controller in FRIEND may be organized in a hierarchical fashion. The interfaces between blocks at different levels in the hierarchy are called contracts.

- Better support for building systems with a high level of adaptivity. The negotiator decides which of its sub-blocks should execute when and how. New blocks may be added on-line. As long as there is a negotiator that is familiar with the contract of the new block, it is possible for the system to accept it and reconfigure itself.

The FRIEND framework does not give any solutions to the design of contracts or negotiation procedures. FRIEND merely provides the necessary mechanism for creating flexible and adaptive control software.

Related Work

There are several widely used prototyping systems today, for example Autocode [Integrated Systems, 1996a] which generates real-time code from SystemBuild-models [Integrated Systems, 1996b], or Real-time Workshop [MathWorks, 1997] which generates real-time code from Simulink-models [MathWorks, 1996]. The models are implemented graphically using block diagrams. In both SystemBuild and Simulink, the user has a

palette of pre-defined blocks for defining controllers. To create user-defined blocks, from which code may be generated, SystemBuild has a language called BlockScript, which is a simple block description language with support for basic data types such as floats, integers and booleans. Autocode generates C- or Ada-code from BlockScript blocks. A template file is used for customizing the generated code for different target environments. The template description is made in a template programming language and makes it possible to generate code that is tailor-made for a special system. User-defined blocks for Real-time Workshop must me written as C MEX S-functions, which are standard C-functions with a specified interface. Both Real-time Workshop and Autocode generate static systems, i.e., it is not possible to make any changes to the system without recompilation. Another tool that generates real-time code from a simulation description is Sim2DCC [Dahl, 1990]. Control algorithms are coded and simulated in the Simnon simulation environment [Elmqvist *et al.*, 1990], and translated to Modula-2 code using Sim2DCC. Sequential algorithms are implemented using the graphical GrafEdit interface.

Simulink is used both as the simulation environment and the real-time controller in [Blomdell, 1999]. Simulink is executed on a slightly modified Linux operating system. There are blocks for reading from analog inputs and writing to analog outputs. There are also timer blocks handling the sampling rate. This set-up offers a simple way to test algorithms in a near real-time environment.

ControlShell [RealTimeInnovations, 1995] is an advanced tool for designing complex real-time control systems. It consists of a C++ class library in combination with graphical editors, which are used for configuring control applications. ControlShell is block-based, and the blocks are implemented in C++. ControlShell gives the programmer great freedom, and it allows an application to be reconfigured without recompilation.

A commercial system that has some basic ideas in common with the PÅLSJÖ system is SattLine [Johannesson, 1994]. 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, and algebraic algorithms expressed with implicit equations. The SattLine language is based on LICS [Elmqvist, 1985], which has also inspired the work with 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 Pascal-type language with a structure similar to PAL. Function Block Diagram is a graphical language for specifying data flows through function blocks. Ladder Diagram is another graphical language that is used for

expressing ladder logic. The fourth language is Instruction List which is a low-level assembler language. Sequential Function Chart is a graphical language for implementing sequential algorithms, similar to Grafcet. The textual representation of Sequential Function Chart was used as a starting point when implementing support for Grafcet in PÅLSJÖ.

There are currently several initiatives trying to turn Java into a full-fledged alternative for implementation of embedded systems. The J-Consortium has a working document, "Core Real-Time Extensions for the Java Platform" [J-Consortium, 1999], trying to establish a real-time standard. The PERC virtual machine, which is a commercial offspring from the work of the J-Consortium, supports deterministic real-time tasking and real-time garbage collection [NewMonics, 1999]. SUN Microsystems together with the Real Time Expert Group quite recently announced the "Real-Time Specification for Java". The specification extends the Java platform to support deterministic real-time behavior.

The Simplex group at SEI/CMU has developed a framework, Simplex, that supports safe on-line upgrades of real-time systems, in particular safety-critical real-time control systems [Seto *et al.*, 1998a; Sha, 1998]. The basic building block of Simplex is the replacement unit, and a typical example of a replacement unit is a controller. Replacement units are organized into application units, that also contain communication and task management functions. A special safety unit is responsible for basic reliable operation and operation monitoring. A common set-up is that the system contains two units: a safety controller and a baseline controller. The baseline controller provides the nominal control performance, and it is assumed that it is executing when a new upgraded version of this controller is to be installed. If the new controller, which is added on-line, does not work correctly, Simplex switches in the safety controller. When the controlled process is eventually back in a safe state, Simplex switches the baseline controller back in.

Summary

PÅLSJÖ is a software environment for development of embedded real-time systems. Effort has been put on designing a system that allows the engineer to work interactively. PÅLSJÖ is highly modular, and components may be added while the system is running on-line. Control algorithms are coded in a dedicated controller description language called PAL. In PAL, control algorithms are described in terms of blocks that read input signals and calculate output signals. Compilation of PAL blocks results in C++ code which is subsequently compiled and linked to the PÅLSJÖ run-time system. The system is configured, i.e., blocks are instantiated and connected on-line using PCL. Data is exported over the network to stand-alone tools used for plotting and logging. The run-time system supports on-line edit-

ing of a running configuration; hence, blocks in a running controller can be replaced without stopping the system. Scheduling and synchronization problems encountered in many real-time systems are avoided by exploiting features of PAL control algorithms that make it easier to find a logical order of execution. PÅLSJÖ is primarily designed to run in a host-target configuration and currently runs on Sun/Solaris, VME-m68k, VME-ppc, and Windows NT. A complete description of the PÅLSJÖ environment, including the run-time system, the PAL compiler, and the PAL language, can be found in [Eker, 1997]. Paper 1 gives an introduction to PÅLSJÖ and PAL. Paper 2 focuses more on the concepts of the PÅLSJÖ framework, and details of the run-time system are discussed.

The proposed language FRIEND is designed to give high-level support for implementation of flexible control systems. The main concept is to use contracts to specify when and how control algorithms can be used. FRIEND is presented in Paper 3 with a discussion on flexible systems and some code examples.

4. The Robot 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 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. The ABB Irb-2000 industrial robot used in the experiments is shown in Figure 3. The robot holds the shaft of the pendulum using a gripper that is attached to the robot wrist, see Figure 4. Joint 1 of the robot is then used for swinging up and balancing the pendulum.

The Model

The design of the controller is based on the following linear model of the pendulum on the robot

$$\dot{x} = Ax + B\tau \qquad \text{where} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \\ \phi \\ \dot{\phi} \end{bmatrix}$$

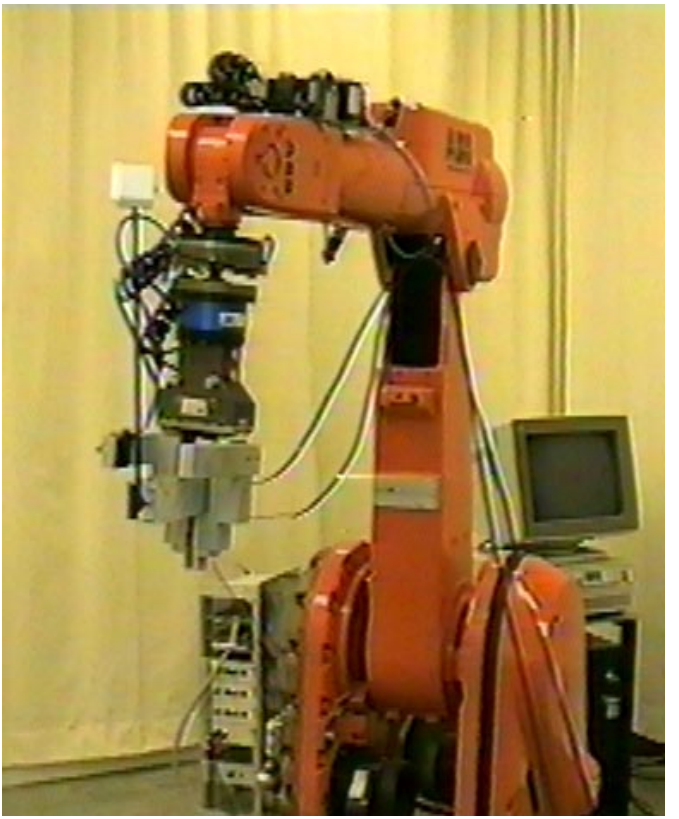


Figure 3. The ABB Irb2000 robot and the pendulum used in the experiment.

and τ is the torque applied to joint 1 of the robot. The states θ and $\dot{\theta}$ are the angle and the angular velocity of the pendulum, see Figure 5. The position and velocity of joint 1 are denoted ϕ and $\dot{\phi}$, respectively.

The model used in the control design is very much simplified. First it is assumed that the system may be viewed as two decoupled sub-systems, one describing the pendulum and one describing the robot-arm position. The Corioli forces due to the rotation and the interaction between the pendulum and the arm are neglected.

Let us first look at the pendulum, and model it as if attached to a cart running on a straight track, see Figure 5. The input signal, u, to the pendulum is the acceleration of the pivot, i.e., the acceleration of the cart.

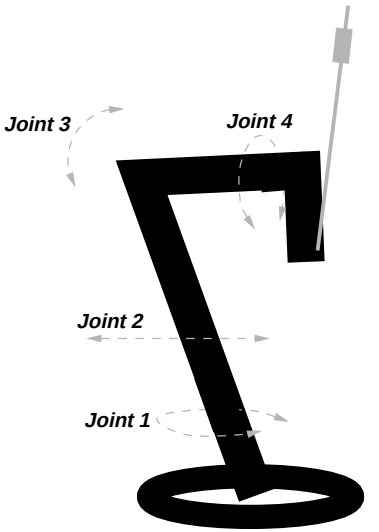


Figure 4. The pendulum is held by the gripper on the robot.

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 $\omega_0 = \sqrt{g/l}$ is the natural frequency for small oscillations around the stable equilibrium, and g is the gravity. The input signal u is the acceleration of the pivot. When the pendulum is attached to the robot, it can be approximated by $u = l\ddot{\phi}$, where l is the length of the robot arm. The equation of motion can be written as

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

where $x_1 = \theta$ and $x_2 = \dot{\theta}/\omega_0$. The input signal to the pendulum system is the acceleration of joint 1. The model for the joint is a second order system, where the input signal τ is the applied torque, and the output signal ϕ is the angle of the joint

$$\phi = \frac{k}{s(s+D)}\tau$$

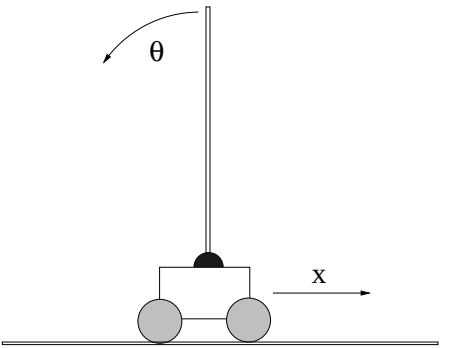


Figure 5. The pendulum on a cart.

The damping is denoted D. The input to the pendulum model is the acceleration of the pivot, i.e., $u = \ddot{\phi}$, which is given by

$$\ddot{\phi} = -D\dot{\phi} + k\tau$$

For the purpose of this experiment, the damping was neglected and hence

$$\ddot{\phi} \approx k\tau \quad \Rightarrow \quad u \approx lk\tau$$

The linearized model for the robot and the pendulum in the upright position may now be written as

$$\dot{x} = \begin{bmatrix} 0 & \omega_0 & 0 & 0 \\ \omega_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ kl\omega_0/g \\ 0 \\ k \end{bmatrix} \tau$$

The Controller

The controller is designed to swing up the pendulum and stabilize it in the upright position. This is done using a hybrid controller consisting of three sub-controllers: one for swinging up the pendulum, one for catching the pendulum, and one for stabilizing the pendulum. The switching logic for this hybrid controller is shown as a Grafcet in Figure 6. To swing up the pendulum an energy approach is used. The idea is to control the energy of the pendulum and move it to its upward position. The energy is calculated as

$$E = mg(\cos\theta - 1)l + \frac{ml^2\dot{\theta}^2}{2}$$

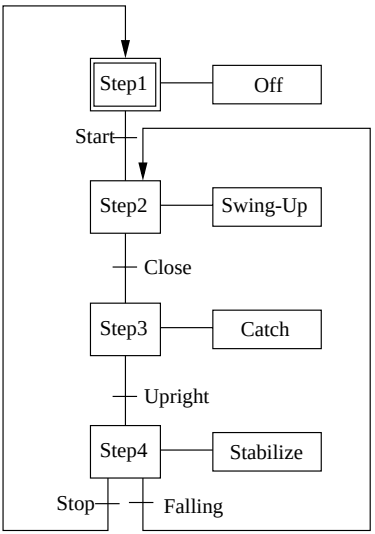


Figure 6. The Grafcet which describes the switching rules for the hybrid pendulum controller.

The pendulum standing in the upright position corresponds to zero energy. The algorithm used for controlling the energy was proposed in [Wiklund *et al.*, 1993; Åström and Furuta, 1996], and is given as

$$u = sat_{ng}(kE)sign\left(\dot{\theta}\cos\theta\right)$$

where k and n are design parameters. The function sat_{ng} saturates when its argument is larger than n times the gravity g. 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 stabilizing mode. The controller used in the experiments is the following state-feedback controller

$$u = -l_1 x_1 - l_2 x_2$$

This controller stabilizes the pendulum, but takes no concern to the position of the pivot. To balance the pendulum at the upright equilibrium, another state feedback controller is used. The control law is the following

$$u = -l_1 x_1 - l_2 x_2 - l_3(x_3 - x_r) - l_4 x_4$$

where x_r is the desired position of joint 1. The controller parameters are calculated using pole-placement design.

Designing an Observer

In order to implement the control laws discussed above, both the angle and the angular velocity values are needed. Since no angular velocity measurement is available, this must be estimated. A nonlinear observer is proposed. The need for a nonlinear observer becomes evident when considering the swing-up strategy, which requires both the angle and the angular velocity at all positions. A linear observer will only give accurate estimates values around some linearization point. The proposed nonlinear observer has the same structure as a linear observer but with the linear model replaced by a nonlinear model

$$\frac{d\hat{x}_1}{dt} = \omega_0 \hat{x}_2 + k_1(x_1 - \hat{x}_1)$$
$$\frac{d\hat{x}_2}{dt} = \omega_0 \sin \hat{x}_1 + u/g\,\omega_0 \cos \hat{x}_1 + k_2(x_1 - \hat{x}_1)$$

Note that the observer has a very simple structure. It combines a model for the nonlinear dynamics with a simple feedback from the measured angle. Conditions for stability and design based on linearized analysis are presented in Paper 5. Before applying the observer to the robot system it is

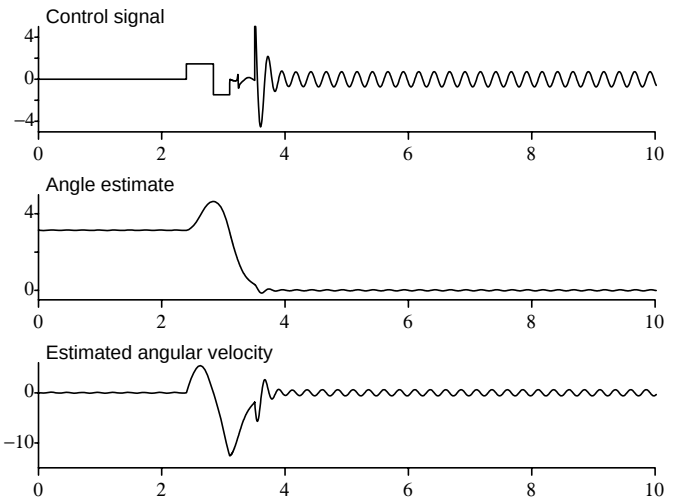


Figure 7. A simulation of swinging up and stabilizing the pendulum using the non-linear observer. The simulation starts with the pendulum in the downward position. The observer is initiated to start in this position.

tested in simulation, see Figure 7. The next step is to test the observer on a stand-alone pendulum. This experiment is shown in Figure 8. Both the simulation and the experiments show a swing-up followed by stabilization. The pendulum is started in the downward position. 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 experiment and simulation and the good performance of the nonlinear observer.

The Pendulum on the Robot

The controller used on the robot is the same as the one previously described. In that case, the input signal to the process was the acceleration of the pivot. In the robot set-up, 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 pivot of the pendulum on the cart moves along a straight line, the pivot of the pendulum on the robot moves along a circular path. This fact is not considered in 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.

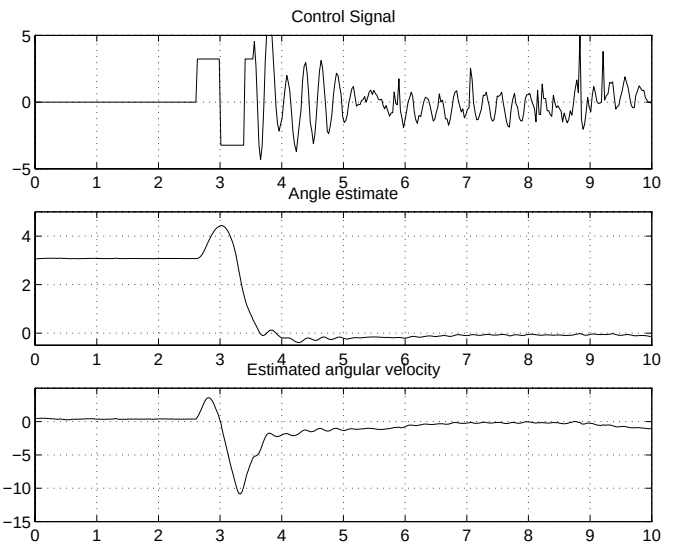


Figure 8. Plots from the swing-up and stabilization of a real pendulum process. The pendulum starts in the downward position.

The results from the experiments with the robot are presented in Figures 9 and 10. Initially the pendulum is in the downward position. The robot uses joint 1 to swing up and stabilize the pendulum.

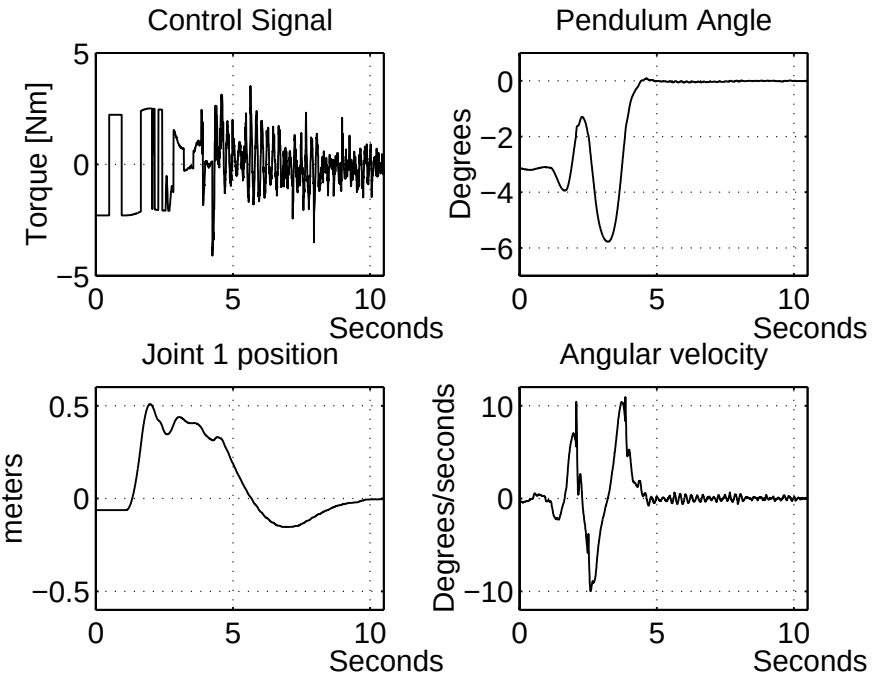


Figure 9. The control signal and the measurements logged during the swing-up shown in Figure 10.

The PAL Code

Example 1 shows the PAL code for the pendulum controller. The pendulum controller has one sequential part and one periodic part. The sequential part is described as a Grafcet with four steps, see Figure 6. 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. The periodic part, implemented using the **calculate** and **update** sections, contains the position and angle observers.

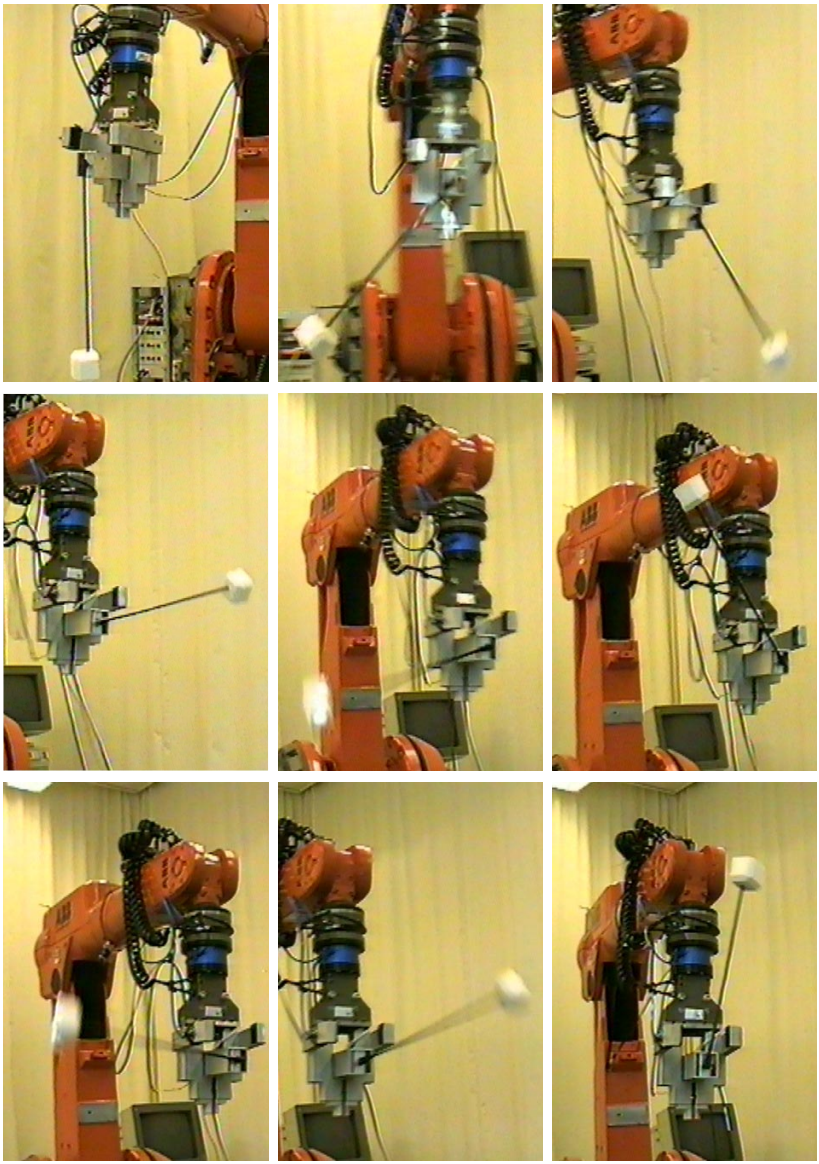


Figure 10. The robot swings up the pendulum by adding energy. The signals from this experiment are presented in Figure 9.

EXAMPLE 1

```
module Pendel;
  function cos(r : input real) : real; external "cos";
  function sin(r : input real) : real; external "sin";
  block PendulumController
    x, th : input real;
    u := 0.0 : output real;
    A22 : parameter matrix [1..2, 1..2] of real;
    B2, K1, K2 : parameter matrix [1..2, 1..1] of real;
    xhat : matrix [1..2, 1..1] of real;
    L : parameter matrix [1..4, 1..1] of real;
    x1hat := 0.0, x2hat := 0.0, dx1hat := 0.0, dx2hat := 0.0 : real;
    thetaHat := 0.0, dthetaHat := 0.0 : real;
    xr := 0.0, E := −2.0 : real;
    on := false : boolean;
    k := 280.0 : parameter real;
    w0 := 6.3 : parameter real;
    n := 0.35 : parameter real;
    minTh := 0.1, maxTh := 0.45 : parameter real;
    h : sampling interval;
    calculate
    begin
      dx1hat := w0 * x2hat + K2[1,1] * (th − thetaHat);
      dx2hat := w0 * sin(x1hat) − w0 * u/9.81 * cos(x1hat)
                +K2[2,1] * (th − thetaHat);
      x1hat := x1hat + dx1hat * h;
      x2hat := x2hat + dx2hat * h;
      thetaHat := x1hat;
      dthetaHat := w0 * x2hat;
    end calculate;
    update
    begin
      xhat := A22 * xhat + B2 * u + K1 * (x − xhat[1,1]);
    end update;
    action off;
    begin
      u := 0.0;
    end off;
    action swing;
    begin
      E := 0.5 * dthetaHat * dthetaHat/(w0 * w0) + cos(th) − 1.0;
      u := −sign(dthetaHat * cos(th)) * sat(n * 9.81, k * E);
    end swing;
```

```
action catch;
begin
  xr := x;
  u := −L[1, 1] ∗ th − L[2, 1] ∗ dthetaHat;
end catch;

action stabilize;
begin
  u := −L[1, 1] ∗ th − L[2, 1] ∗ dthetaHat
       −L[3, 1] ∗ (xhat[1, 1] − xr) − L[4, 1] ∗ xhat[2, 1];
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 step1 to step2 when on;

transition from step2 to step3 when
      th > minTh and th < maxTh or th < −minTh and th > −maxTh;

transition from step3 to step4 when
        th > −minTh and th < minTh;

transition from step4 to step1 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;
  else
    result := value;
  end if;
end sat;

function sign(r : input real) : real;
begin
  if r < 0.0 then
    result := −1.0;
```

```
      else
        result := 1.0;
      end if;
    end sat;

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

    procedure stop();
    begin
      on := false;
      u := 0.0;
    end stop;

  end PendulumController;

end Pendel.
```

□

The PCL script

The controller is configured by the PCL-script in Example 2. A number of PAL modules are made available through the `use` statement. Blocks are allocated using `new` statements. The top level block is of type `Periodic`, which is a system block, designated for executing other blocks. All blocks belonging to a `Periodic` block are periodically executed in sequence according to the data flow. The input and output signals of the blocks are connected using the `->` operator. Some signals are made available for export over the network using the `show` command. Figure 11 shows the resulting block diagram. The PCL language is presented in Appendix B.

EXAMPLE 2

```
use StandardBlocks
use Robot
use Gripper
use Pendel
{
s = new Periodic
with s
   gr = new EndEffector
   gref = new GripperRef
   gref.forceR -> gr.forceR
   gref.inhale -> gr.inhale
   gref.exhale -> gr.exhale
   gref.openclosed -> gr.openclosed
   j1 = new IRB2000JointWrite
```

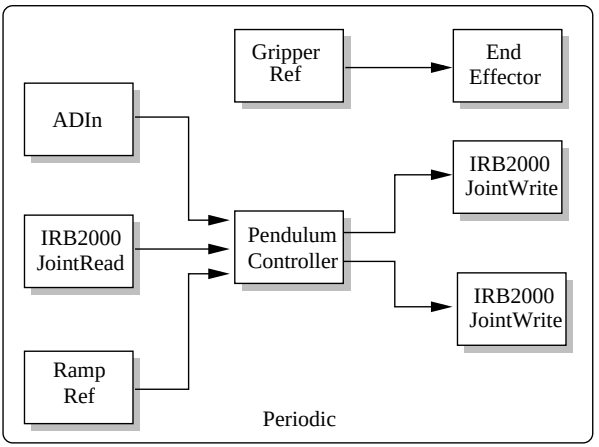


Figure 11. The PAL blocks used for controlling the robot. The PCL-script for creating this system is found in Example 2.

```
   j1.joint = 1
   j4 = new IRB2000JointWrite
   j4.joint = 4
   p2 = new PendulumController
   thetaIn = new ADIn
   j1r = new IRB2000JointRead
   j1r.joint = 1
   thetaIn.out -> p2.thdirect
   j1r.y       -> p2.xdirect
   ref4 = new RampRef
   ref4.out    -> p2.inref4
   p2.u1  -> j1.tauExtern
   p2.u4  -> j4.qrExtern
 endwith
 s.gref.OpenClosed = false
 s.gref.force=0.3
 s.tsamp = 5
}

s!start
show s.p2.u
show s.p2.thetaHat
show s.p2.dthetaHat
show s.p2.x
s!connect
```

□

Summary

The implementation and design of a robot controller for balancing and swinging up the inverted pendulum was briefly presented. The process model and the controller design were discussed. To get good estimates of the angular velocities a nonlinear observer was used. The implementation was presented with PAL and PCL code.

This application with PÅLSJÖ balancing the pendulum was also used in [Henriksson, 1998] to demonstrate scheduling principles for garbage collection in hard-real time systems. There, the underlying memory allocation and deallocation routines were replaced with functions supporting garbage collection.

5. Feedback Scheduling

Real-time systems is an inter-disciplinary research area that includes aspects of control engineering and computer science. It is an area of vital importance to all control engineers. Today almost all controllers are implemented on digital form with a computer. A successful implementation of a computer-controlled system requires a good understanding of both control theory and real-time systems.

Traditionally, when implementing computer control systems, the control community has assumed that the computer platform used can provide the deterministic, equidistant sampling that is the basis of sampled computer control theory. However, many of the computing platforms that are commonly used to implement control systems are not able to give any deterministic guarantees.

On the other hand, the real-time scheduling community generally assumes that all control algorithms should be modeled as tasks that:

- are periodic with a set of *fixed periods*,
- have *hard deadlines*, and
- have known *worst-case execution times* (WCETs).

This simple model has permitted the control community to focus on its own problem domain without worrying about how scheduling is being done, and it has relieved the scheduling community from the need to understand what impact scheduling delays have on the stability and performance of the plant under control. From a historical perspective, the separate development of control and scheduling theories for computer-based control systems has produced many useful results and served its purpose.

However, upon closer inspection it is quite clear that neither of the three above assumptions need necessarily to be true. Many control algorithms are not periodic, or they may switch between a number of different fixed sampling periods. Control algorithm deadlines are not always hard. On the contrary, many controllers are quite robust towards variations in sampling period and response time. It is also in many cases possible to compensate on-line for the variations by, e.g., recomputing the controller parameters. It is also possible to consider control systems that are able to do a trade-off between the available computation time, i.e., how long time the controller may spend calculating the new control signal, and the control loop performance.

By integrating control design and scheduling it is possible to go beyond the simple "fixed sample period and known WCET" model and to develop theory that also addresses the dynamic interaction between control and scheduling. The optimality of computer control is subject to the limitations of available computing resources, especially in advanced applications with fast plant dynamics and sophisticated state estimation and control algorithms. On the other hand, the true objective of real-time scheduling for control is to allocate limited computing resources in such a way that the state estimation and control algorithms can ensure stability and optimize the performance of the system. The computing resources could include CPU time and communication bandwidth.

The approach taken in Paper 4 is based on using dynamic feedback between the scheduler and the control loops. The idea of feedback has been used informally for a long time in scheduling algorithms for applications where the dynamics of the computation workload cannot be characterized accurately. For instance, the VMS operating system uses multi-level feedback queues to improve system throughput, and Internet protocols use feedback to help solve the congestion problems. Recently, under the title of quality of service (QoS), the idea of feedback has also been exploited in multi-media scheduling R&D.

Given this, one might expect that the use of feedback in the scheduling of feedback control systems would have been naturally an active area. On the contrary, the scheduling research of feedback control systems are dominated by open loop analytic scheduling methods such as rate or deadline based algorithms. This is not an accident, but rather the consequence of some serious theoretical challenges that require the close cooperation between control and scheduling communities.

The Vision

The work in this thesis is based upon a vision of a dynamic, flexible, and interactive integrated control and scheduling environment with the following features:

- The control design methodology should take the availability of computing resources into account.
- The requirement of known worst-case execution times should be relaxed. Instead the system should be able to guarantee stability and a certain level of control performance based only on knowledge of nominal execution times.
- The system should be able to adapt task parameters in overload situations in such a way that stability and an acceptable level of control performance are maintained.
- The system should be able to dynamically trade-off control performance and computing resource utilization.
- The system should support on-line information exchange between the on-line scheduler and the control tasks. The information could for example consist of mode change requests from the control tasks, execution time allotments from the on-line scheduler, etc.
- The system should be able to measure the actual execution time spent by the different tasks, and take appropriate actions in case of over-runs.
- The system should admit on-line experiments. Controller code should be allowed to be added, tested and replaced on a running system under safe condition as suggested in [Seto *et al.*, 1998a].

In order to make the above visions possible, a lot of information needs to be provided. The control tasks must be able to express the quality of the control performance in relation to needed resources, if, and how, the sampling rate and deadline may be adjusted, and what the consequences will be. Paper 4 investigates how the LQ-cost depends of the sampling rate and how this may be used in a feedback scheduling loop.

Designing a Feedback Scheduler

Viewing a computing system as a dynamical system or as a controller is an approach that has proved to be fruitful in many cases. For example, the step-length adjustment mechanism in numerical integration algorithms can be viewed as a PI-controller [Gustafsson, 1991], and the traveling salesman optimization problem can be solved by a nonlinear dynamical system formulated as a recurrent neural network. This approach can also be adopted for real-time scheduling, i.e., it is possible to view the on-line scheduler as a controller. Important issues that then must be decided are what the right control signal, measurement signals, and setpoints are, what the correct control structure should be, and which process model that

may be used. Using a controller approach of the above kind it is important to be able to measure the appropriate signals on-line, for example the deadline miss ratio, controller performance, the CPU utilization, or the task execution times.

Control and Scheduling Co-Design

A prerequisite for an on-line integration of control and scheduling theory is to be able to make an integrated design of control algorithms and scheduling algorithms off-line. Such a design process should allow an incorporation of the availability of computing resources into the control design by utilizing the results of scheduling theory. This is an area where, so far, relatively little work has been performed. One of the first references that addressed this problem was [Seto *et al.*, 1996]. An algorithm was proposed, which translates a control performance index into task sampling periods considering schedulability among tasks running with pre-emptive priority scheduling. The sampling periods were considered as variables and the algorithm determined their values so that the overall performance was optimized subject to the schedulability constraints. Both rate monotonic scheduling (RM) and earliest deadline first scheduling (EDF) were considered. The performance index was approximated by an exponential function only and the approach did not take input-output latency into account. The approach was further extended in [Seto *et al.*, 1998b].

An approach to optimization of sampling period and input-output latency subject to performance specifications and schedulability constraints is presented in [Ryu *et al.*, 1997; Ryu and Hong, 1998]. The control performance is specified in terms of steady state error, overshoot, rise time, and settling time. These performance parameters are expressed as functions of the sampling period and the input-output latency. A heuristic iterative algorithm is proposed for the optimization of these parameters subject to schedulability constraints. The algorithm is based on using the period calibration method [Gerber *et al.*, 1995] for determining the task attributes. The tasks are scheduled using EDF and a cyclic executive is used for run-time dispatching.

On-line Task Attribute Adjustments

A key issue in any system that allows dynamic feedback between the control algorithms and the on-line scheduler is the ability to dynamically adjust task parameters. Reasons for the adjustments could for example be to improve the performance in overload situations and to dynamically optimize control performance. Examples of task parameters that could be modified are periods and deadlines. One could also allow the maximum allowed worst-case execution time for a task to be adjusted. In order for this to be realistic, the controllers must support dynamically changing

execution times. Changes in the task period and in the execution time both have the effect of changing the utilization that the task requires.

Quality-of-service resource allocation

Much of the work on dynamic task adaptation during recent years is motivated by the requirements of multimedia applications. Activities such as voice sampling, image acquisition, sound generation, data compression, and video playing are performed periodically, but with less rigid timing requirements than those that can sometimes be found in closed-loop control systems. Missing a deadline may decrease the QoS-level, but without causing any critical system faults. Depending on the requested QoS, tasks may adjust their attributes to accommodate the requirements of other concurrent activities.

On-line admission control has been used to guarantee predictability of services where request patterns are not known in advance. This concept has also been applied to resource reservation for dynamically arriving real-time tasks, e.g. in the Spring kernel [Stankovic and Ramamritham, 1991]. A main concern of this approach is predictability. Run-time guarantees given to admitted tasks are never revoked, even if they result in rejecting subsequently arriving, more important requests competing for the same resources. In soft real-time systems, services are more concerned with maximizing overall utility, by serving the most important requests first, than guaranteeing reserved resources for individual requests. Priority-driven services can be categorized this way, and are supported in real-time kernels such as Mach [Tokuda *et al.*, 1990]. Under overload conditions, lower-priority tasks are denied service in favor of more important tasks. In the Rialto operating system [Jones and Leach, 1995], a resource planner attempts to dynamically maximize user-perceived utility of the entire system.

In [Abdelzaher *et al.*, 1997] a QoS renegotiation scheme is proposed as a way to allow graceful degradation in cases of overload, failures or violation of pre-run-time violations. The mechanism permits clients to express, in their service requests, a range of QoS levels they can accept from the provider and the perceived utility of receiving service at each of these levels. Using this, the application designer, e.g., the control engineer, is able to express acceptable tradeoffs in QoS and their relative cost/benefit. The approach is demonstrated on an automated flight-control system.

Sampling Rate Adjustments

In [Buttazzo *et al.*, 1998] an elastic task model for periodic tasks is presented. Each task is characterized by five parameters: computation time C_i, a nominal period T_{i_0}, a minimum period $T_{i_{min}}$, a maximum period $T_{i_{max}}$, and an elasticity coefficient $e_i \geq 0$. A task may change its period within

its bounds. When this happens the periods of the other tasks are adjusted so that the overall system is kept schedulable. An analogy with a linear spring is used, where the utilization of a task is viewed as the length of a spring that has a given rigidity coefficient $(1/e_i)$ and length constraints. The elasticity coefficient is used to denote how easy or difficult it is to adjust the period of a given task (compress or decompress the spring). A task with $e_i = 0$ can arbitrarily vary its period within its range. The approach can be used under fixed or dynamic priority scheduling.

Adjustment of task periods has also been suggested by others. For example, [Kuo and Mok, 1991] propose a load-scaling technique to gracefully degrade the workload of a system by adjusting the task periods. Tasks are assumed to be equally important and the objective is to minimize the number of fundamental frequencies to improve schedulability under static priority assignments. In [Nakajima and Tezuka, 1994] a system is presented that increases the period of a task whenever the deadline of the task is missed. In [Lee *et al.*, 1996] a number of policies to dynamically adjust task rates in overload conditions are presented. In [Nakajima, 1998] it is shown how a multimedia activity can adapt its requirements during transient overloads by scaling down its rate or its computational demands.

The MART scheduling algorithm [Kosugi *et al.*, 1994; Kosugi *et al.*, 1996; Kosugi and Moriai, 1997] also supports task-period adjustments. The system handles changes in both the number of periodic tasks and in the task timing attributes. Before accepting a change request the system analyzes the schedulability of all tasks. If needed it adjusts the period and/or execution time of the tasks to keep them schedulable with the rate monotonic algorithm. For the task execution time it is assumed that a minimum value exists in order for the task to guarantee a minimum level of service. For the task-period, neither minimum nor maximum are assumed to exist. The MART system is implemented on top of Real-Time Mach.

In [Shin and Meissner, 1999] the approach in [Seto *et al.*, 1996] is extended, making on-line use of the proposed off-line method for processor utilization allocation. The approach allows task-period changes in multiprocessor systems. A performance index for the control tasks is used to determine the value to the system of running a given task at a given period. The index is weighted for the importance of the task to the overall system. The paper discusses the issue of task reallocation from one processor to another, and the need to consider the transient effects of task reallocations. Two algorithms are given for task reallocation and period adjustments.

In [Stankovic *et al.*, 1999] a PID controller is proposed as an on-line scheduler under the notion of Feedback Control-EDF (FC-EDF). The mea-

surement signal (the controlled variable) is the deadline miss ratio for the tasks and the control signal is the requested CPU utilization. Changes in the requested CPU utilization are effectuated by two mechanisms. An admission controller is used to control the flow of workload into the system and a service level controller is used to adjust the workload inside the system. The latter is done by changing between different versions of the tasks, which have different execution times. A simple liquid tank model is used as an approximation of the scheduling system.

Imprecise calculations

The possibility to adjust the allowed maximum execution time for a task necessitates an approach for handling tasks with imprecise execution times, particularly the case when the tasks can be described as "anytime algorithms", i.e., algorithms that always generate a result but where the quality of the result, the QoS-level, increases with the execution time of the algorithm.

The group of Liu has worked on scheduling of imprecise calculations for a long time [Liu *et al.*, 1987; Chung *et al.*, 1990; Liu *et al.*, 1994]. In [Liu *et al.*, 1991] imprecise calculation methods are categorized into *milestone methods*, *sieve function methods*, and *multiple version methods*. Milestone methods use monotone algorithms that gradually refine the result and each intermediate result can be returned as an imprecise result. Sieve function methods can skip certain computation steps to tradeoff computation quality for time. Multiple version methods have different implementations with different cost and precision for a task.

Examples of all three types of imprecise algorithms can be found in control. Algorithms that are based on on-line numerical solution to an optimization problem every sample can be viewed as milestone or "anytime" methods. The result generated from the control algorithm is gradually refined for each iteration in the optimization up to a certain bound. Examples of this type of control algorithms are found in model-predictive control.

It is also straightforward to find examples of controllers that be cast on sieve function form. For example, in an indirect adaptive controller, the updating of the adapted parameters can be skipped when time constraints are hard, or, in a LQG controller the updating of the observer can be skipped when time constraints are hard. Similarly, in a PID controller the updating of the integral part may be skipped if needed. The extreme case is of course to skip the entire execution of the controller for a certain sample. Scheduling of systems that allow skips is treated in [Koren and Shasha, 1995] and [Ramanathan, 1997]. The latter paper considers scheduling that guarantees that at least k out of n instantiations will execute.

Finally, it is also possible to find examples of multiple version methods in control, e.g., situations with one nominal control algorithm and one backup algorithm. However, in most cases the motivation for having the backup algorithm is control performance rather then timing issues.

Mode changes

Mode changes for priority-based preemptive scheduling is an issue that has received some interest. In the basic model, the system consists of a number of tasks with task attributes. Depending on which modes the system and the tasks are in, the task attributes have different values. During a mode change, the system should switch the task attributes for a task and/or introduce or remove tasks in such a way that the overall system remains schedulable during and after the mode change.

A simple mode change protocol was suggested in [Sha *et al.*, 1989]. The protocol assumes that an on-line record of the total utilization is kept. A task may be deleted at any time, and its utilization may be reclaimed by a new task at the end of the period of the old task. The new task is accepted if the resulting task set is schedulable according to the rate-monotonic analysis. The locking of semaphores, according to the priority ceiling protocol, during the mode change is also dealt with.

In [Tindell *et al.*, 1992], it was pointed out that the analysis of Sha *et al.* was faulty. Tasks may miss their deadlines during a mode change, even if the task set is schedulable both before and after the switch. The transient effects of a mode change can be analyzed by extending the deadline-monotonic framework. Formulas for the worst-case response times of old and new tasks across the mode change were given. New tasks may be given release offsets, relative to the mode change request, to prevent deadlines to be missed. It is interesting to note, that under EDF scheduling, the reasoning about the utilization from [Sha *et al.*, 1989] actually seems to hold. In [Buttazzo *et al.*, 1998], EDF scheduling of a set of tasks with deadlines equal to their periods is considered. It is shown that a task can decrease its period at its next release, as long as the total utilization remains less than one.

Summary

A brief overview of state of the art in integrated control and scheduling was given. Three papers in this thesis relate to this type of work. Paper 4 presents a feedback scheduling algorithm which optimizes control performance with respect to available computing resources. The FRIEND language in Paper 3 is designed to support the implementation of control systems that are adaptive to changes in resources. Using the toolbox presented in Paper 5, it is possible to study how CPU-scheduling algorithms and network protocols affect the control performance.

One could argue that this type of research is uninteresting and soon to be superfluous. The computing power increases rapidly today and prices are going down. Why then bother with scheduling at all? Just buy another set of CPUs and give each task its own one, and the need for scheduling is gone! This is probably true for some applications. On the other hand the more powerful the CPUs get, the more software can and will be put onto them. Available CPU power will almost always be used, no matter speed or number of processors. Furthermore, adding more CPUs may solve the scheduling problem, but will instead worsen the network congestion problem.

6. Simulation of Embedded Control Systems

The complexity of embedded control systems often makes it difficult to investigate all their properties analytically. It is therefore necessary to have good simulation tools available. In this section two different approaches to simulation of embedded control systems are discussed. First, the PALSIMART simulation environment, in which PAL blocks may be simulated, is presented. Second, a MATLAB/Simulink toolbox for integrated simulation of process dynamics and discrete controllers running in a virtual computer is introduced. The toolbox is the subject of Paper 5.

PALSIMART

Rapid prototyping of advanced control systems can be properly supported provided that the design phase is closely coupled to the real-time implementation. In many systems, once the controller is designed and tested in simulation, the control algorithm must be rewritten, usually from scratch, in another environment for real-time embedded implementation. Apart from an obvious overhead, this procedure may as well cause significant deviations from the predicted behavior, due to the basically different nature of the underlying code.

One possible rapid prototyping approach, mentioned earlier, and indeed the most widespread so far, consists of using commercial tools like Simulink together with Real-Time Workshop, or SystemBuild together with AutoCode. One problem with this approach is that the languages used in simulation may not be suitable for designing real-time applications. With PALSIMART, another approach was chosen, based on the integration of the control description language PAL and the CACSD environment SIMART, see [M'Saad *et al.*, 1997].

SIMART offers a wide set of advanced control techniques as well as powerful analysis tools and a user-friendly graphical interface. The PAL

compiler produces C++ code that fits both SIMART and the PÅLSJÖ runtime system. For an algorithm whose code has been validated in simulation, no further compilation is needed for real-time implementation. This constitutes the main advantage of the proposed environment over the mentioned commercial packages.

SIMART

SIMART is a general purpose CACSD package designed to support all levels of controller development, from design to implementation. It contains a set of built-in toolboxes such as LQG and GPC for design support. Furthermore, it has tools for performance and robustness analysis, such as Bode diagrams and sensitivity functions. The user interface is graphical and menu-driven. SIMART has a range of predefined realistic nonlinear models for a number of processes. In addition, the user may enter linear model descriptions interactively. SIMART is currently available on PCs for MS-Windows, and in an X-11 version for UNIX workstations and Linux PCs.

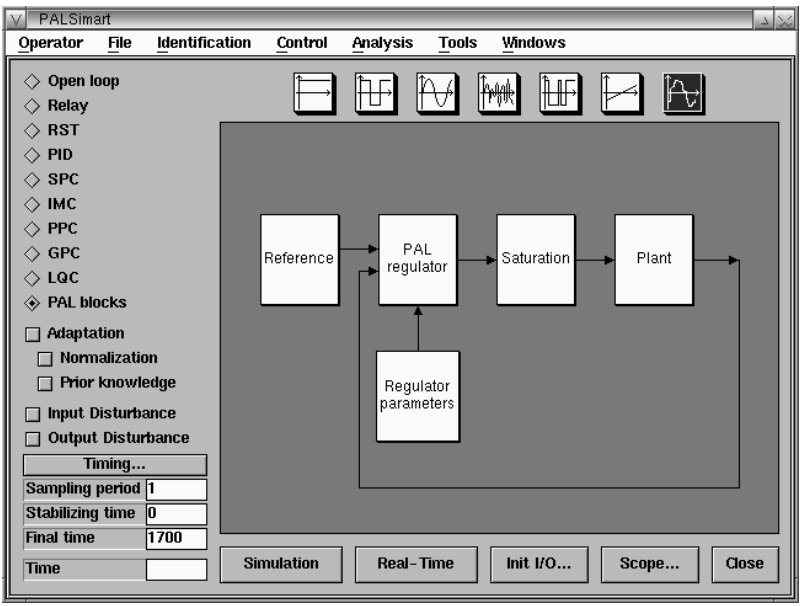


Figure 12. PAL inside SIMART

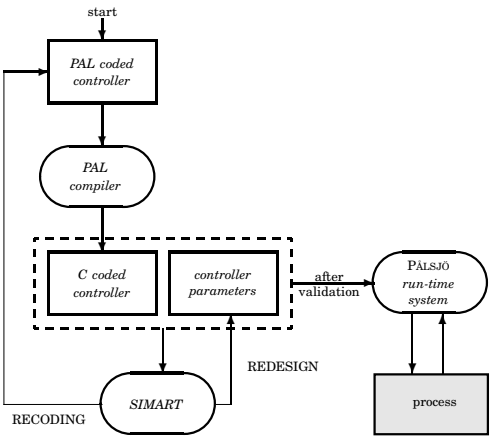


Figure 13. The integrated environment

The PALSIMART framework

The PALSIMART environment is an integrated environment created starting from three existing pieces of software: the CACSD package SIMART, the PÅLSJÖ run-time system and the PAL meta-compiler. Its main purpose is to quicken prototyping of advanced controllers, by letting real-time algorithms be analyzed and tested in simulation prior to on-line implementation.

Using PAL to build control systems for real-time applications enables the creation of a reusable library of control algorithms that are both easy to write and to implement. Still, the designer spends a considerable amount of time testing the control system under development by means of real-time experiments. PALSIMART supports the integration of PAL blocks into SIMART. Allowing PAL blocks to execute in the SIMART simulation environment has some immediate benefits. First, the same PAL code may be used both in the simulation and in the real-time controller. Second, it is possible to evaluate and test the controller code off-line. A screen-shot from PALSIMART is shown in Figure 12. By double-clicking on the PAL regulator block a list of available PAL modules and blocks will appear. From this list the user chooses which PAL algorithm to be simulated. Once a block is instantiated, its parameters can be set and its variables can be connected to SIMART variables. A plant model is then chosen in a similar way.

The resulting PALSIMART framework is presented in Figure 13. Any PAL coded algorithm can be easily appended to a control library that is available both to SIMART and to PÅLSJÖ run-time system. After the controller has been validated through simulation, the same PAL block, with the exact same C coding, can be tested in the PÅLSJÖ run-time environment. Testing the controller in the simulation environment ahead of the experiments on the real process substantially reduces prototyping time.

The main asset of PALSIMART, with respect to other available prototyping systems is that algorithm coding is identical regardless of use; the same blocks are used both in simulation and real-time, and the same behavior is expected. With no need for code rewriting, the creation of a versatile library of control algorithms is straightforward. PALSIMART currently runs on Sun/Solaris workstations. A Windows NT version is under development.

A Toolbox for Simulation of Embedded Systems

Paper 5 presents a MATLAB/Simulink-toolbox for simulation of embedded systems. There currently exist many tools for simulation of control systems, for example Simnon [Elmqvist, 1973], OMSIM/OMOLA [Andersson, 1994], Modelica [Elmqvist and Mattsson, 1997], Simulink [MathWorks, 1996], and SystemBuild [Integrated Systems, 1996b]. Similarly, there are many tools available for simulation of real-time systems, for example STRESS [Audsley *et al.*, 1994], and DRTSS [Storch and Liu, 1996]. Very little effort has however been put into mixed simulations, where both the continuous dynamics of the process, the discrete algorithm of the controller and the behavior of the control task running on a computer are simulated. One noticeable exception is [Liu, 1998], where a control task and a continuous process are simulated using Ptolemy II [Davis II *et al.*, 1999].

The Toolbox

To get the full picture of the performance of an embedded control system, it may be necessary to consider the real-time behavior of the controller task. The well known and much used simulation-environment Simulink was chosen as the platform for a toolbox designed to support simulations of embedded systems at task level. Using the toolbox, the control engineer may first simulate and design the controller without any real-time considerations, and then at a later stage, with only minor modifications, simulate it as if running on a virtual computer. The toolbox is implemented as a set of Simulink-blocks and MATLAB-functions. A kernel block is the core of the virtual computer block. It samples continuous signals and writes discrete signals. Figure 14 shows a view from a simulation. The model consists of two dynamic processes, which are controlled by two

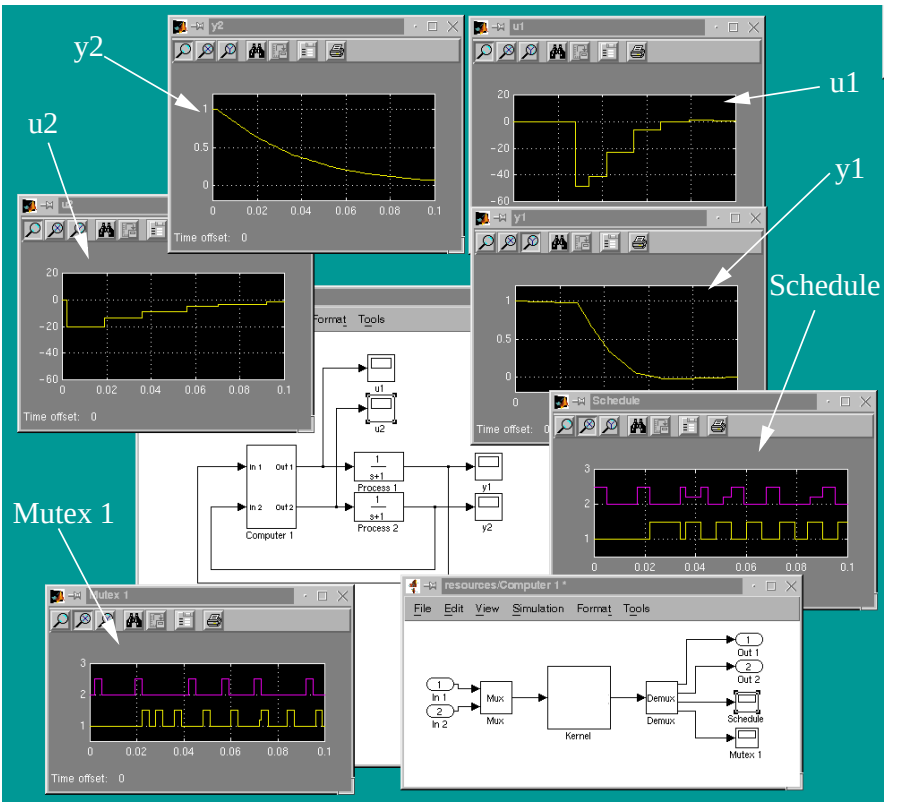


Figure 14. The simulation model is defined by a block diagram. There are two blocks for the dynamic processes and one block for the computer. The control tasks are implemented as MATLAB-files.

control tasks running in the virtual computer. Three types of output signals from the kernel block are plotted. The first type of output signals are the control signals `u1` and `u2` which are fed to the dynamic processes. The second type of output signals are the execution traces. The task activation graph shows the state of each task at every point in time. This signal is fed to the plot named `Schedule`. A task can be either running (high), blocked (medium), or sleeping (low). The `Mutex 1` signal indicates when the mutual exclusion monitor with the same name is occupied. Similar to the task activation graph, there is one line for each task, where high means that the task is inside the monitor, medium that the task is waiting to get in, and low that the task is outside the monitor.

Example 3 below shows the initialization script used for the simula-

tion in Figure 14. At the top, some general settings are made. A priority function `prioRM`, which implements rate monotonic priority assignment, is chosen and the tick size of the kernel is set to 1 millisecond. Two periodic tasks are then defined using the `task`-function. The first parameter is the name of the task, followed by the *code object*, the period and the deadline. The code object contains the statements that are to be executed by the task.

EXAMPLE 3

```
function rtsys = resources_init

%% General settings
rtsys.prioFun = 'prioRM';
rtsys.tickSize = 0.001;

%% Create tasks
% TASK 1
rSegs = {'rseg1' 'rseg2'};
rParams.inCh = 1;
rParams.outCh = 1;
rCode = code(rSegs,[],rParams);
rPeriod = 0.012;
rDeadline = 0.012;
rTask = task('Task1', rCode, rPeriod, rDeadline);
% TASK 2
oSegs = {'oseg1' 'oseg2'};
oParams.inCh = 2;
oParams.outCh = 2;
oCode = code(oSegs,[],oParams);
oPeriod = 0.017;
oDeadline = 0.017;
oTask = task('Task2', oCode, oPeriod, oDeadline);

rtsys.tasks = {rTask oTask};

%% Create mutexes
m1 = mutex('M1');
m1.data = 0;
rtsys.mutexes = {m1};

%% Create events
e1 = event('E1','M1');
rtsys.events = {e1};
```

□

A code object is built up from *code segments*, which is the smallest entity in the kernel and implemented as a MATLAB-function. The hierarchy of

objects is shown in Figure 15. A code segment has two parts; the *enter part*, which is executed when the segment is started, and *exit part*, which is executed when the execution time of the segment has passed. In Example 3 each of the code objects, `rCode` and `oCode`, consists of two code segments. These four segments are shown in Example 4 below.

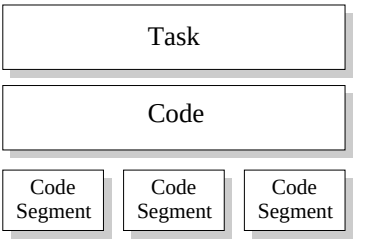


Figure 15. Each task is associated with a code object, which consists of one or more code segments.

EXAMPLE 4

```
function[exectime,states] = rseg1(flag,states,params)
switch flag,
 case 1, % enterCode
   if lock('M1')
     data=readData('M1');
     if data < 2
       await('E1');
       exectime=0;
     else
       exectime=0.003;
     end
   else
     exectime=0;
   end
 case 2,  % exitCode
   unlock('M1');
end

function [exectime,states] = rseg2(flag,states,params)
switch flag,
 case 1, % enterCode
   y=analogIn(params.inCh);
   states.u=-50*y;
   exectime=0.003;
 case 2,  % exitCode
```

```
   analogOut(params.outCh, states.u)
end

function [exectime,states] = oseg1(flag,states,params)
switch flag,
 case 1, % enterCode
   y=analogIn(params.inCh);
   states.u=-20*y;
   exectime=0.002;
 case 2, % exitCode
   analogOut(params.outCh, states.u);
end

function [exectime,states] = oseg2(flag,states,params)
switch flag,
 case 1, % enterCode
   if lock('M1')
     data=readData('M1');
     writeData('M1',data+1);
     exectime=0.003;
   else
     exectime=0;
   end
 case 2, % exitCode
   cause('E1');
   unlock('M1');
end
```

□

The functions `lock`, `unlock`, `await`, `cause`, `readData`, and `writeData` are toolbox functions implementing the mutual exclusion monitor with the associated event.

In the initialization script above, the scheduling type was set by giving a priority function. The priority function defines the criterion according to which the ready queue is sorted. A number of different strategies is available, and it is also straightforward for the user to implement a new strategy. The priority function for the rate monotonic case is the following

```
function prio = prioRM(task)
  prio = task.period;
```

In Example 3, `Task1` has the shortest period, and since rate monotonic priority assignment is used it gets the highest priority. A low number corresponds to a high priority.

Summary

The PALSIMART environment allows the use of exactly the same PAL code, and C++ code in both simulation and implementation. By bridging the gap between simulation and implementation the prototyping time is shortened. The MATLAB/Simulink toolbox supports task-level simulation of closed loop embedded controllers. This opens up the possibility to evaluate different scheduling strategies from a control performance point of view. The toolbox does also support simulation of computer networks, which makes it possible to investigate and simulate distributed control systems. For example, it is possible to look at time delays in networks and their influence on the control performance.

7. Implementation of a Hybrid Controller

Paper 6 describes the design and implementation of a hybrid control strategy for a heating/ventilation system in a school in Klippan, a small town not far from Lund. The project, which mainly consisted of control design, also gave some valuable insights through the use of commercial control hardware/software. While the design and the test implementations were made in the lab using PÅLSJÖ, the actual implementation was programmed in Forth on a Diana ADP 2000 system. The Diana ADP 2000, manufactured by Diana Control AB, is mainly used in heating-ventilation applications. The software is quite flexible and it is possible to install new controller code without having to rebuild the system. A unit can contain several control loops, and on large installations several units can be interconnected in a network. In the network configuration, one of the control units is a master and the others are slaves. The master unit can be reached over a modem connection and from the master it is possible to communicate with all the others. It is possible to log data, change parameters, and download new code for every controller over the network. The identification experiments presented in Paper 6 were made over a telephone connection from Lund. Before trying the controller on the heating/ventilation process it was tested against a process simulation, implemented in real-time SIMNON, see [Elmqvist *et al.*, 1990]. The lack of an integrated tool, like the PALSIMART environment previously described, forced us to use four different languages during the development. The simulations were coded in OMOLA [Andersson, 1994] for the OMSIM simulation tool. The first real-time implementation of the controller was made in PAL, and later translated into Forth. The Forth code was tested on the real-time simulation of the process. A flavor of the Forth code is given in Example 5 below.

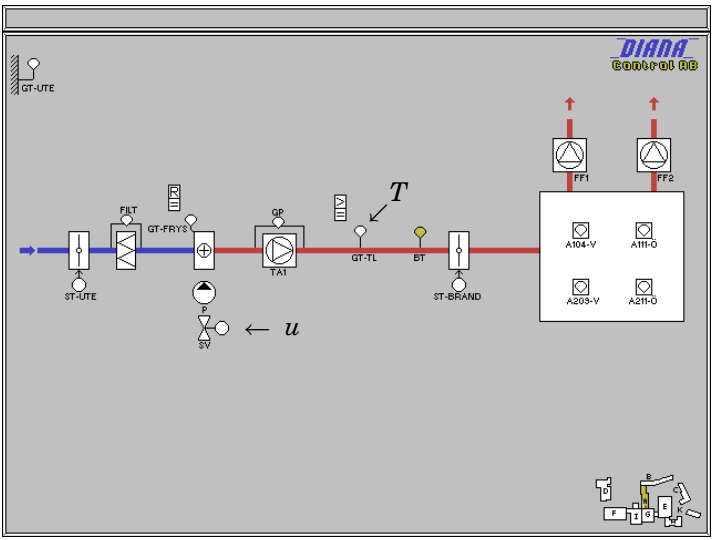


Figure 16. The air temperature control loop. The water flow to the heat-exchanger is controlled by a valve via the control signal u. The output is the air temperature T after the fan.

EXAMPLE 5
A fragment of the Forth code for the PID sub-controller. Even a simple algorithm like this becomes quite complex due to the lack of proper language support.

```
PAR REF 2@ PAR y1 2@ F- PAR EE P2!
PAR EE 2@ PAR K 2@ F* 2DUP PAR P P2!
PAR II 2@
PAR D1 2@ PAR D 2@ F*
PAR D2 2@ PAR y1old 2@ PAR y1 2@ F- F*
F+ 2DUP PAR D P2!
F+ F+  PAR UU P2!
```

□

The process diagram is shown in Figure 16. The temperature T of the inflow air is controlled by adjusting the flow of hot water in a heat exchanger. The control signal for the hot water valve is u. The previously installed controller worked well when the process was in steady state. However, the response to reference point changes was quite oscillative. One solution to this problem is to use a hybrid controller consisting of two sub-controllers, one PID controller and one time-optimal controller, together with a supervisory switching scheme. In process control it is common practice to use PI control for steady state regulation and to use manual control for large

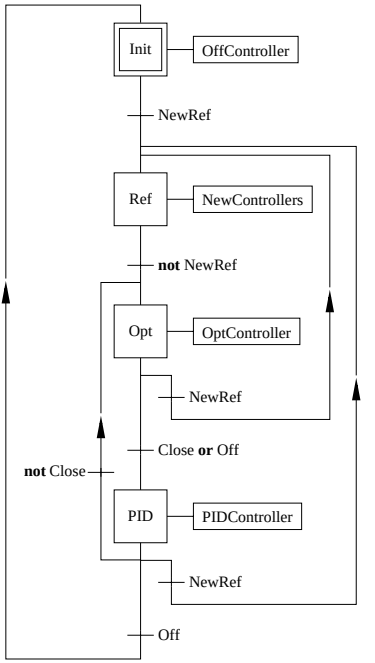


Figure 17. The Grafcet diagram for the hybrid controller consists of four steps. The execution starts in the **Init** step. **Opt** is the step where the time-optimal controller is active and **PID** is the step for the PID controller. The **Ref** step is an intermediate state used for calculating new controller parameters.

changes. In this case the PID steady state regulation is combined with a minimum time controller for the setpoint changes. 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 setpoint the controller is redesigned, keeping the same structure but using setpoint dependent parameters. Figure 17 describes the algorithm with a Grafcet [David and Alla, 1992]. This controller is used in Section 8, as an example of a controller with highly varying execution times.

8. A Unifying Example

A feedback scheduling application is designed and simulated. The feedback scheduling algorithm is loosely based on the work presented in Pa-

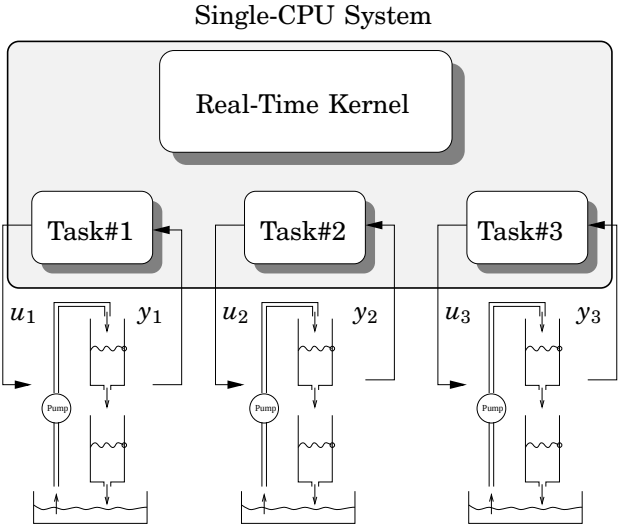


Figure 18. Several tank controllers execute concurrently on one hardware unit. The interaction between the control tasks affects the control performance.

per 4. The simulations are made using the toolbox described in Paper 5, and the control problem is taken from Paper 6.

Several double-tank processes should be controlled by a single-CPU system, see Figure 18. Since the control tasks execute on the same CPU, they will compete for the resources and interact with each other. The goal of this example is to show how higher resource utilization and better control performance may be achieved by a more flexible scheduling approach. The controller used in the simulation is the same as the one presented in Section 7. The computer executes each hybrid controller as a periodic real-time task. The execution times for the different modes vary a lot, and this makes the scheduling difficult.

The Problem

How should the tasks be scheduled in order to get the best performance from the system? If the computing resources were unlimited, the controllers could be designed to execute at very high rates, and with very good control performance. But with limited computing resources, the controllers must execute at much lower rates in order to avoid overload. When the utilization of the CPU is close to 100 percent, the interaction between the control tasks also becomes noticeable. Preemption and blocking causes jitter and delays, and the control performance deteriorates even further.

During a transient overload, some controllers might shut down completely, depending on the scheduling policy used.

The general problem is extremely complicated because of the different modes that the hybrid controllers might switch between. A hypothetical optimal solution would continuously redistribute the computing resources between the tasks depending on the current and possible future combinations of the controller states, the controller modes, and the execution history. Even if the optimal solution was obtainable, it would never be implementable in a system with limited computing power. Instead, one should aim for a simple heuristic solution that can be applied on-line, and without too much detailed knowledge about the system. An approximate version of the feedback scheduler from Paper 4, combined with standard rate-monotonic scheduling would form such an solution. In the following, some non-feedback and feedback scheduling solutions to the problem are described and evaluated by simulations. It is shown that a more flexible scheduling approach can give better control performance.

The Set-Up

The number of double tank systems to control is assumed to be constant and equal to three. The systems are identical, except for the tank cross sections which are different for the different double tanks. This causes the processes to have different time constants: $T_1 = 60$ s, $T_2 = 45$ s, and $T_3 = 30$ s. Setpoint changes for the levels of the lower tanks are assumed to occur with different frequencies for the different systems: $f_1 = 0.0067$ s^{-1}, $f_2 = 0.0094$ s^{-1}, and $f_3 = 0.0133$ s^{-1}. The differences between the processes make the scheduling problem more interesting.

The control algorithm executes in either one of the modes **Ref** (for setpoint changes), **Opt** (for optimal control), or **PID** (for PID control). The setpoint change frequencies described above will cause each controller to spend about half of its time in the **Opt** mode, and the other half in the **PID** mode. The **Ref** mode is only active for a single sample at a time, and only when a setpoint change has been detected by the controller. In this example, the behavior of the controller is typically the following: It executes in the **Ref** mode for one sample, then in the **Opt** mode for several samples, then in the **PID** mode for several samples, and then the cycle repeats.

The relative execution times of the different modes were obtained by measurements on an existing PÅLSJÖ implementation of the hybrid controller. It was found that the **Opt** mode took about twice as long to execute as the **PID** mode, and that the **Ref** mode had about six times the execution time of the **PID** mode. In this example, it is assumed that the execution times are constant in each mode, and equal to $C_{PID} = 300$ ms, $C_{Opt} = 600$ ms, and $C_{Ref} = 1800$ ms. More on the measurements of exe-

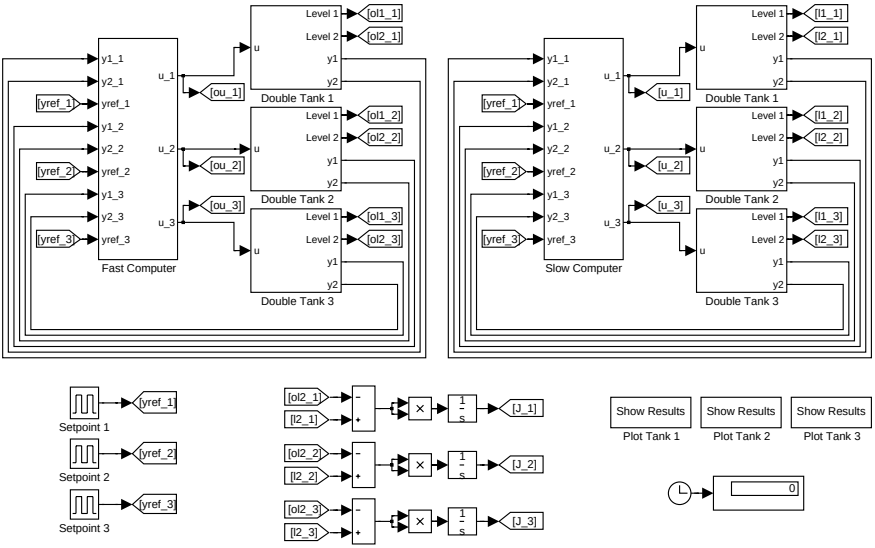


Figure 19. A screen-shot of the Simulink model of the control system. To the left, the tanks are controlled by a fast, ideal, computer. To the right, the tanks are controlled by a slow, real, computer. The behavior of the left system is used as a reference when evaluating the performance of the right system.

cution times for this hybrid controller is found in [Persson, 1998].

Simulation

The toolbox described in Paper 5 is used to evaluate different scheduling strategies. The hybrid control algorithm is very easily rewritten into a MATLAB function

```
[states,exectime] = hybridController(flag,states,params)
```

which is the control algorithm format prescribed by the toolbox. The function is executed periodically during simulation. It returns the new controller states (including the controller mode), and the simulated execution time of the algorithm.

To isolate the effects of scheduling on control performance, two complete control systems are simulated in parallel, see the screen-shot shown in Figure 19. To the left, the three double tanks are controlled by an ideal, very fast, computer. To the right, the three double tanks are controlled by a real, slower, computer. The behavior of the ideal system is used as a reference when evaluating the performance of the real system. Figure 20 shows the level of the lower tanks when controlled by the ideal computer.

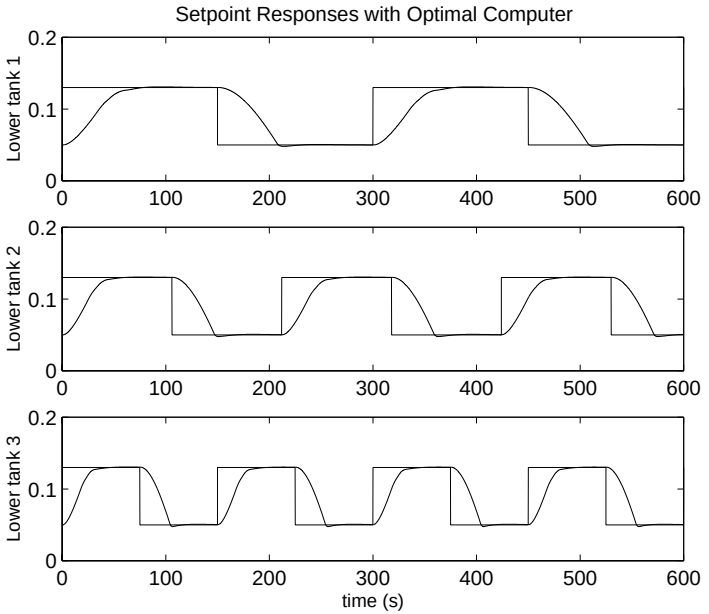


Figure 20. The level of the lower tanks when controlled by the ideal computer.

The simulations display very good setpoint responses and steady state behavior.

To compare different scheduling strategies, the control performance needs to be quantified somehow. This is a difficult task even for a single non-hybrid controller. We decided to measure the accumulated squared distance between the outputs of the ideal system, y_{ideal}, and the outputs of the real system, y_{real}, during a simulation of length T_{sim}.

$$V = \int_0^{T_{sim}} (y_{ideal}(t) - y_{real}(t))^2 dt$$

The function V is called the *performance loss due to scheduling*. It gives a rough estimate of the performance of the real system by judging all deviations from the ideal system as undesired. Typically, the outputs from the real system are delayed compared to the outputs from the ideal system.

Scheduling

Three different scheduling approaches are compared: worst-case scheduling, average-case scheduling, and feedback scheduling. All approaches are based on rate-monotonic scheduling.

Worst-Case Scheduling

Worst-case scheduling is probably the most common scheduling approach in real-time control systems today. Tasks are assigned periods off-line, and schedulability analysis is carried out assuming worst-case conditions during execution. In our case, the worst-case condition occurs when all controllers happen to execute in the **Ref** mode at the same time. It is, however, unnecessarily pessimistic to use the worst-case execution time of $C_{Ref} = 1800$ ms in the scheduling analysis. We know that, in our example, the controller only spends a single sample in the **Ref** mode and then switches to the **Opt** mode in the next sample.

Average-Case Scheduling

The drawback with worst-case scheduling is that the average resource utilization becomes low, with poor control performance as a result. An alternative would be to use average-case scheduling, where tasks are assigned periods to give a higher average utilization of the CPU. Of course, tasks may now miss their deadlines. If rate-monotonic scheduling is the underlying scheduling principle, the lower-priority tasks may miss a lot of deadlines, while the higher-priority tasks do not need to miss a single deadline. This unfair distribution of computing resources during runtime could cause the overall control performance to be quite poor.

Feedback Scheduling

Feedback scheduling in this example means that task periods are assigned on-line, depending on the modes of the different controllers. The underlying scheduling principle is rate-monotonic scheduling. The feedback scheduler is activated each time a control task switches mode. The feedback loop between the scheduler and the control tasks is illustrated in Figure 21.

In this example, we assume that each controller has an associated, known, cost function

$$J_i = \alpha_i + \beta_i h_i^2, \tag{1}$$

where h_i is the sampling frequency for task i. The goal for the feedback scheduler is to optimize the total performance, i.e.

$$\text{minimize } J = \sum_{i=1}^{n} J_i,$$

subject to the utilization constraint

$$\sum_{i=1}^{n} C_i/h_i = U,$$

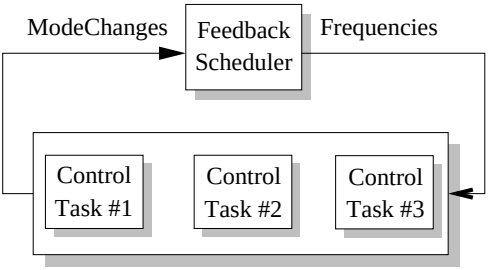


Figure 21. A feedback loop is formed between the control tasks and the scheduler. The control tasks notify the scheduler of their current mode. The scheduler then adjusts the frequencies of the control tasks so that the CPU load becomes the desired.

where U is the desired utilization and C_i is the execution time for task i. The execution time of the feedback scheduler itself is zero in the simulations. We do, however, reserve some slack for the feedback scheduler by setting the desired utilization to 80 %.

Results

The choice of sampling intervals for control loops is a research area in itself. In this case, the choice of sampling intervals can be motivated by the process dynamics and the setpoint change frequencies. The fastest process with the most frequent setpoint changes obviously requires the most frequent actions from its controller. The sampling intervals are chosen so that the ratio h_i/T_i is the same for all tree tanks. The whole system, i.e., the three control tasks, the kernel and the three double tank systems, was simulated for 1500 seconds using three different scheduling approaches. Rate monotonic priority assignments were used in all cases. Table 1 also presents some results from the simulations.

Worst-Case Scheduling

In the worst-case scheduling scenario, the sampling intervals are chosen so that no deadlines are missed. The sampling intervals are through simulation found to be $h_1 = 7$ s, $h_2 = 4.9$ s and $h_3 = 3.5$ s. These are the shortest sampling intervals that will not cause missed deadlines. The utilization will however be very poor, and is in the simulation found to be only 33.1 %. The slow sampling rates result in poor performance for all three controllers. The output of the system controlled by Controller 1 is shown in Figure 22 (top).

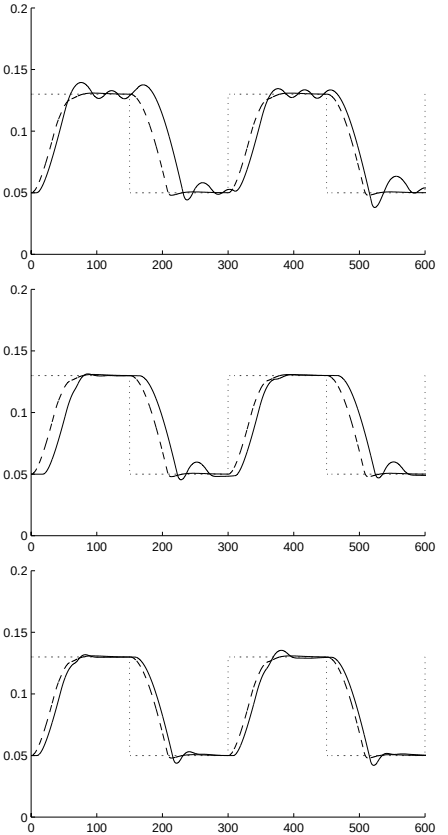


Figure 22. The output from the tank controlled by the controller with lowest priority for the three different scheduling strategies. The dashed line shows the ideal behavior and the dotted line shows the reference signal. The plot show the WCET scheduling (top), Average scheduling(middle), and Feedback scheduling (bottom).

Average-Case Scheduling

The average-case scheduling scenario allows deadlines to be missed and instead tries to keep a high utilization level. The sampling intervals $h_1 = 2.6$ s, $h_2 = 1.8$ s, $h_3 = 1.3$ s are calculated to give approximately 80 % utilization. Actual utilization in simulation is 78.5 %. This approach will favor the controllers with high priorities. Controller 1, however, misses a lot of deadlines with poor performance as a result. The output from the system controlled by Controller 1 is shown in Figure 22 (middle).

Strategy	V_1	V_2	V_3	U_{eff}
WCET	0.21	0.20	0.15	32.7 %
Average	0.21	0.05	0.037	78.5 %
Feedback	0.067	0.067	0.095	80 %

Table 1 The different scheduling strategies are compared in terms of the additional cost due to scheduling. The U_{eff} is the actual utilization from the simulation. The feedback scheduling strategy distributes the computing resources more evenly over the tasks which results in lower total cost.

Feedback Scheduling

The feedback scheduling approach tries to achieve both high utilization and few missed deadlines. Each task is associated with a cost function which reflects how the task performance relates to the sampling interval. The cost function is given by Equation 1, and the parameters are assigned heuristically. The feedback scheduling in this case is very much simplified since the same cost functions are used independently of which mode the controller executes in. A better approach would be to derive the cost function parameters analytically and have one cost function for each controller mode. However, in this example, this simple approach works very well.

The desired utilization is set to 80 %, and actual utilization in simulation is 79.3 %. Furthermore there are no missed deadlines. The output from the system controlled by Controller 1 is shown in Figure 22 (bottom). The feedback scheduler gives much better control performance for Controller 1, and slightly worse for Controllers 2 & 3.

The results for all three double-tank systems from all three simulations are presented in Table 1.

Summary

This simple example shows how the control performance may be improved by adjusting the sampling frequencies so that high CPU utilization is obtained and at the same time few deadlines are missed. This example also demonstrates the type of simulation that can be made with the real-time kernel toolbox. The double tank system is quite robust towards missed deadlines, and if the simulations were made using a more time-critical process the results would probably have been different. A possible extension to the feedback scheduler would be to introduce performance observers, that give the estimated cost based on measurements.

9. References

Abdelzaher, T., E. Atkins, and K. Shin (1997): "QoS negotiation in real-time systems, and its application to flight control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

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., B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha (1999): "Integrated control and scheduling." Report ISRN LUTFD2/TFRT--7586--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

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.

Audsley, N., A. Burns, M. Richardson, and A. Wellings (1994): "STRESS—A simulator for hard real-time systems." *Software—Practice and Experience*, **24:6**, pp. 543–564.

Benveniste, A. and G. Berry (1991): "The synchronous approach to reactive and real-time systems." In *Proceedings of the IEEE*, vol. 79.

Blomdell, A. (1997): "PAL – the PÅLSJÖ algorithm language.". Report ISRN LUTFD2/TFRT--7558--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Blomdell, A. (1999): "Linux in control." http://www.control.lth.se.

Boussinot, F. and R. de Simone (1991): "The ESTEREL language." In *Proceedings of the IEEE*, vol. 79.

Buttazzo, G., G. Lipari, and L. Abeni (1998): "Elastic task model for adaptive rate control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Chung, J.-Y., J. Liu, and K.-J. Lin (1990): "Scheduling periodic jobs that allow imprecise results." *IEEE Trans on Computers*, **39:9**.

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 II, J., M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong (1999): "Overview of the Ptolemy project." Technical Report ERL Technical Report UCB/ERL No. M99/37. University of California, Berkeley, EECS, CA, USA 94720.

Eker, J. (1997): *A Framework for Dynamically Configurable Embedded Controllers*. Lic Tech thesis ISRN LUTFD2/TFRT--3218--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. (1973): "SIMNON—User's guide." Technical Report TFRT-3106. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

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.

Elmqvist, H. and S. E. Mattsson (1997): "Modelica—The next generation modeling language, an international effort." In *Proceedings of the 1st World Congress on System Simulation, WCSS'97*. Singapore.

Gerber, R., S. Hong, and M. Saksena (1995): "Guaranteeing real-time requirements with resource-based calibration of periodic processes." *IEEE Trans on Software Engineering*, **21:7**.

Gustafsson, K. (1991): "Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods." *ACM Transactions on Mathematical Software*, **17:4**, pp. 533–554.

Gustafsson, K. (1994): "An architecture for autonomous control." Report ISRN LUTFD2/TFRT--7514--SE. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Halbwachs, N. (1993): *Synchronous programming of reactive systems*. Kluwer Academic Publishing.

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud (1991): "The synchronous data flow programming language LUSTRE." In *Proceedings of the IEEE*, vol. 79.

Hennessy, J. (1999): "The future of systems research." *IEEE Computer*, August.

Henriksson, R. (1998): *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technolgy, Lund, Sweden.

Integrated Systems (1996a): *AutoCode User's Guide*. Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054-3309, USA.

Integrated Systems (1996b): *SystemBuild User's Guide*. Integrated Systems Inc, 3260 Jay Street, Santa Clara, CA 95054-3309, USA.

J-Consortium (1999): "International J-Consortium specification." http://www.j-consortium.org/.

Johannesson, G. (1994): *Object-oriented Process Automation with SattLine*. Chartwell Bratt Ltd. ISBN 0-86238-259-5.

Jones, M. and P. Leach (1995): "Modular real-time resource management in the Rialto operating system." In *Proceedings of the of the Fifth Workshop on Hot Topics in Operating Systems*.

Kligerman, E. and A. D. Stoyenko (1986): "Real-Time Euclid: A language for reliable real-time systems." *IEEE Trans. on Software Eng.*, **SE-12:9**, pp. 941–949.

Koren, G. and D. Shasha (1995): "Skip-over: Algorithms and complexity for overloaded systems that allow skips." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Kosugi, N., A. Mitsuzawa, and M. Tokoro (1996): "Importance-based scheduling for predictable real-time systems using MART." In *Proceedings of the 4th Int. Workshop on Parallel and Distributed Systems*, pp. 95–100. IEEE Computer Society.

Kosugi, N. and S. Moriai (1997): "Dynamic scheduling for real-time threads by period adjustment." In *Proceedings of the World Congress on Systems Simulation*, pp. 402–406.

Kosugi, N., K. Takashio, and M. Tokoro (1994): "Modification and adjustment of real-time tasks with rate monotonic scheduling algorithm." In *Proceedings of the 2nd Workshop on Parallel and Distributed Systems*, pp. 98–103.

Kuo, T.-W. and A. Mok (1991): "Load adjustment in adaptive real-time systems." In *Proceedings of the 12th IEEE Real-Time Systems Symposium*.

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.

Lee, C., R. Rajkumar, and C. Mercer (1996): "Experiences with processor reservation and dynamic QoS in real-time Mach." In *Proceedings of Multimedia Japan 96*.

Lewis, R. (1995): *Programming industrial control systems using IEC 1131–3*. The Institution of Electrical Engineers, London, U.K.

Liu, J. (1998): "Continuous time and mixed-signal simulation in Ptolemy II." Technical Report UCB/ERL Memorandum M98/74. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Liu, J., K.-J. Lin, and S. Natarajan (1987): "Scheduling real-time, periodic jobs using imprecise results." In *Proceedings of the IEEE Real-Time System Symposium*, pp. 252–260.

Liu, J., K.-J. Lin, W.-K. Shih, A. Yu, J.-Y. Chung, and W. Zhao (1991): "Algorithms for scheduling imprecise computations." *IEEE Trans on Computers*.

Liu, J., W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung (1994): "Imprecise computations." *Proceedings of the IEEE*, Jan, pp. 83–93.

MathWorks (1996): *Simulink-Dynamic System Simulation for MATLAB*. The MathWorks inc., 24 Prime Park Way, Natick, MA 01760-1500.

MathWorks (1997): *Simulink–Real-Time Workshop*. The MathWorks inc., 24 Prime Park Way, Natick, MA 01760-1500.

M'Saad, M., J. Chebassier, and P. Tona (1997): "Advanced control using the CACSD package SIMART." In *IFAC Conference on Systems, Structure and Control*. Bucharest, Romania.

Nakajima, T. (1998): "Resource reservation for adaptive QoS mapping in real-time Mach." In *Proceedings of the Sixth International Workshop on Parallel and Distributed Real-Time Systems*.

Nakajima, T. and H. Tezuka (1994): "A continuous media application supporting dynamic QoS control on real-time Mach." In *Proceedings of ACM Multimedia'94*.

Natarajan, S. and K. Lin (1988): "FLEX: towards flexible real-time programs." In *Proceedings of the International Conference on Computer Languages*. IEEE.

NewMonics (1999): *Making Java Applications a Reality in Embedded Systems Today*. NewMonics Inc., 1755 Park Street, Suite 260, Naperville, IL 60563. http://www.newmonics.com.

Persson, P. (1998): "A tool measuring and analyzing real-time task execution times." Project Report, Real-Time Systems Course. Department of Automatic Control, Lund.

Ramanathan, P. (1997): "Graceful degradation in real-time control application using (m,k)-firm guarantee." In *Proceedings of the IEEE Real-Time Systems Symposium*.

RealTimeInnovations (1995): *ControlShell–Object–Oriented Framework fro Real–Time System Software*. Real-Time Innovations, Inc., 155A Moffet Park Drive, Suite 111, Sunnyvale, CA 94089, USA.

Ryu, M. and S. Hong (1998): "Toward automatic synthesis of schedulable real-time controllers." *Integrated Computer-Aided Engineering*, **5:3**, pp. 261–277.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Seto, D., B. Krogh, L. Sha, and A. Chutinan (1998a): "Dynamic control system upgrade using the Simplex architecture." *IEEE Control Systems*, August.

Seto, D., J. Lehoczky, and L. Sha (1998b): "Task period selection and schedulability in real-time systems." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Seto, D., J. Lehoczky, L. Sha, and K. Shin (1996): "On task schedulability in real-time control systems." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Sha, L. (1998): "Dependable system upgrade." In *Proceedings of IEEE Real Time Systems Symposium*.

Sha, L., R. Rajkumar, and J. Lehoczky (1989): "Mode change protocols for priority-driven pre-emptive scheduling." *Real-Time Systems*, **1:3**, pp. 244–264.

Shin, K. G. and C. L. Meissner (1999): "Adaptation of control system performance by task reallocation and period modification." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 29–36.

Stankovic, J., C. Lu, and S. Son (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Stankovic, J. and K. Ramamritham (1991): "The Spring kernel: A new paradigm for real-time systems." *IEEE Software*, **8**.

Storch, M. F. and J. W.-S. Liu (1996): "DRTSS: A simulation framework for complex real-time systems." In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 160–169.

Stoyenko, A. and W. Halang (1993): "Extending Pearl for industrial real-time applications." *IEEE Software*, **10:4**, pp. 65–74.

Tindell, K., A. Burns, and A. J. Wellings (1992): "Mode changes in priority preemptively scheduled systems." In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pp. 100–109.

Tokuda, H., T. Nakajima, and P. Rao (1990): "Real-time Mach: Towards a predictable real-time kernel." In *Proceedings of USENIX Mach Workshop*.

Wiklund, M., A. Kristenson, and K. J. Åström (1993): "A new strategy for swinging up an inverted pendulum." In *Preprints IFAC 12th World Congress*, vol. 9, pp. 151–154. Sydney, Australia.

Paper 1

A Tool for Interactive Development of Embedded Control Systems

Johan Eker

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 paper presents a software framework for implementation of embedded control systems. The PAL language 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.

1. Introduction

This paper presents the PÅLSJÖ system for development of embedded control systems. PÅLSJÖ is an experimental environment designed to support code reuse and rapid prototyping. To facilitate this and to relieve the control engineer of real-time programming obstacles the controller description language PAL was created. Algorithms are designed and packaged so that code reuse becomes straightforward.

It is difficult to design reusable real-time code using general purpose languages such as C/C++. Timing requirements and algorithm specifications are hard to separate, and it becomes cumbersome to use the same code module in two real-time scenarios, for a dramatic example of this see [Le Lann, 1996]. If instead real-time code is generated from a simulations environment code reuse may be better supported. Examples of simulation environments that support code generation are SystemBuild [Integrated Systems, 1996a; Integrated Systems, 1996b] and Simulink [MathWorks, 1997]. A problem with code generators is that they may give large and inefficient programs. The real-time specifications in simulation models are usually very sparse and this makes it difficult to generate optimized real-time code. Code generators usually do not take into account that the sampling delay can be reduced significantly by rearranging the code, see [Åström and Wittenmark, 1997].

Development tools of today have a very low degree of interactivity. The control application is designed, compiled and tested. If any errors are found the plant must be stopped and restarted before the user can test the corrected version. There is a great need for a higher level of interactivity when developing real-time control systems. It must be possible to insert and delete algorithms without having to stop the plant and recompile the whole application. More flexible control software is not only needed in the control labs, but also in the industry. The software in embedded controllers needs 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 costs reduction.

2. The Pålsjö System

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. There are functions for user interaction, task management, data logging, network interaction, etc. The main idea with the PÅLSJÖ project is to capture the common behavior of

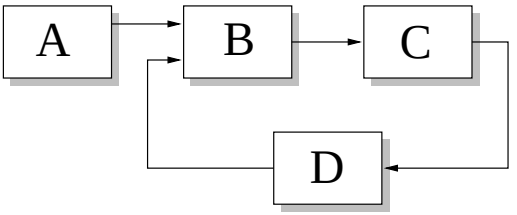


Figure 1. Control systems are conveniently described using block diagrams.

typical control systems. If those common features can be encapsulated in a framework, then the size of the code that needs to be written for each new application can be significantly reduced. Further it is possible to introduce a suitable abstraction level, that will support control algorithms in particular. The goal is to give a high level of support for those common features and let the control engineer focus on the control algorithm implementing. The user of such a framework only needs to add the code that is specific for a certain application, i.e. the control algorithm.

Pålsjö System Model

Control engineers extensively describe control system using block diagrams, see Figure 1. A block diagram contains information of the algorithms and how they are interconnected. Real-time code could be generated from this this simple system description, but the resulting system would be equally simple. If instead some additional real-time information were added to the block diagram, its real-time behavior would become much more precise. Figure 2 shows a system with some extra real-time specifications. The kind of information is which blocks should be mapped onto which real-time tasks, which signal connections are synchronous and which are asynchronous, etc. In PÅLSJÖ blocks are implemented in the language PAL (PÅLSJÖ Algorithm Language). Control systems are then configured on-line using the script language PCL (PÅLSJÖ Configuration Language). The additional real-time information discussed above is hence in PÅLSJÖ added on-line. In PÅLSJÖ it is allowed to edit the system configuration without halting it first. Figure 3 shows the structure of a PÅLSJÖ application. Two user defined processes execute in the workspace, which is connected to modules managing operator communication, network communication, plant IO, etc. When an application is written in PAL for the PÅLSJÖ system, only the tasks specific to the application must be implemented by the control engineer. In Figure 3 there are two processes that execute the actual control application from Figure 1, and these processes are created on-line by the user. Each of these processes consists of a two PAL blocks. Each PAL block has a set of functions which are called by the

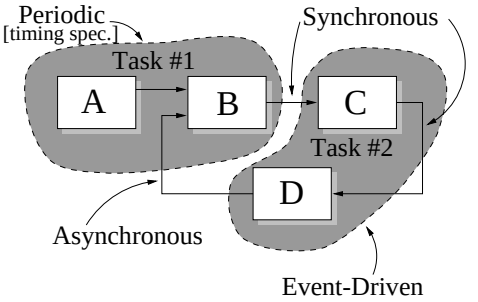


Figure 2. In order to execute the block diagram in a real-time application more information needs to be added. Such information could be which blocks that should be mapped to which processes, timing constraints, and what kind of connections, synchronous or asynchronous.

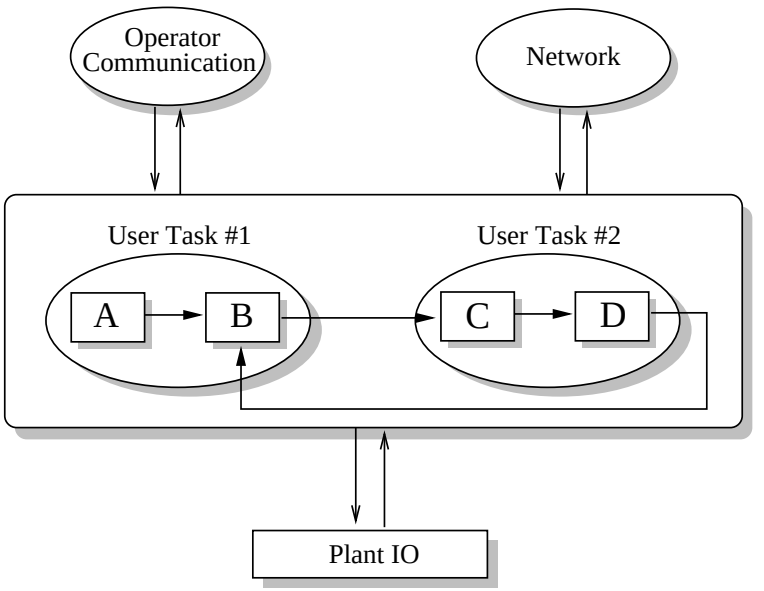


Figure 3. A schematic view of the PÅLSJÖ system. There are a set of predefined modules managing operator communication for example. Furthermore there is a set of user defined processes which is the actual controller.

process in order to execute the block. Connections between blocks residing in the same process is synchronous, while connections between different processes are asynchronous.

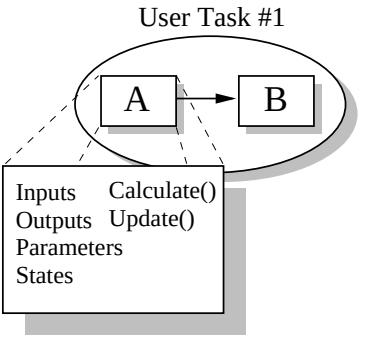


Figure 4. A PAL block has a set of methods that are executed by the owner process.

3. Configuring the System

PAL blocks are instantiated and connected on-line using PCL. PCL is a simple language for administrating blocks and assigning variables.

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 during run-time using PCL. Instances of the system blocks `Periodic` and `Sporadic` are used to manage the execution. 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 the `Periodic` block. Consider the block diagram in Figure 5. 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 and is executed at a higher sampling rate than the control blocks. The desired real-time behavior is the following:

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

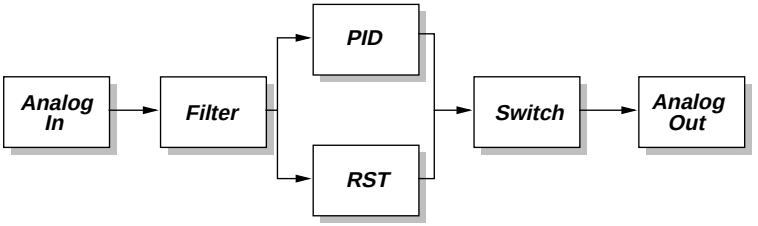


Figure 5. The block diagram created in Example 1.

EXAMPLE 1

```
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*> 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
```

□

The `Periodic` block executes its child blocks according to a schedule automatically created on-line. Since the child blocks are executed synchronously no data sharing problems will arise. Mutual exclusion is thus handled automatically by the system. The data-flow mechanism is implemented, so that unnecessary copying is avoided

4. PAL - Pålsjö Algorithm Language

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 sequen-

tial algorithms are available. Furthermore complex data types such as polynomials and matrices are fully supported. For a more in depth description see [Blomdell, 1997], which this section is based upon.

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 internal states and the parameters. A PAL block has the following structure:

EXAMPLE 2

```
module MyPI
  block PI
    r, y, u : input real;
    v, r2 : output real;
    I := 0.0, e := 0.0 : real;
    K := 0.5, Ti := 10.0 : parameter real;
    Tr := 10.0 : parameter real;
    h : sampling interval;
    bi = K * h/Ti; br = h/Tr;

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

    update begin
      r2 := Limiter(minRef, maxRef, r);
      I := I + bi * e + br * (u - v);
    end update;

    function Limiter(
      max : real;
      min : real;
      value : real
    ) : real;
    begin
      ...
    end Limiter;
  end PI;
end MyPI.
```

□

The top level concept of PAL is the module. Several blocks may be grouped together in a module. In the example above the block **PI** is declared within the module **MyPI**. The first section in the block defines the interface. There are a set of input signals, output signals and parameters. There are

also two state variables I and e, and two indirect parameters bi and br. After the interface section comes the algorithm section. The two sections **calculate** and **update** define the periodic algorithm of the block.

Data Types

PAL has a set of predefined data types. Among these are regular scalar ones, e.g. real, integer, boolean, as well as aggregate types such as arrays, matrices, and polynomials. The availability of arithmetic operations for polynomials and matrices makes it straightforward to express control algorithms. Below is an example of scalar data types:

r : **real**;

i : **integer**;

bool := true : **boolean**;

The type **dimension** is an integer variable used to specify the dimension for aggregate data types. 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. There is a special data type **sampling interval** for handling the period time for periodic processes. The sampling time is set by the user or the run-time system. Below is an example of more complex data types:

n : **dimension**;
in1 : **input array**[1..n] of **real**;

m : **dimension**;
in2 : **array**[0..m] of **input real**;

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

par : **parameter polynomial**[n] of **real**;

There are built in functionality to fully support the use of matrices and polynomials. Function libraries for Diophantine equation solving, recursive model estimation, etc are available.

Events

All procedures in PAL when executed in the PÅLSJÖ environment are registered as events. For example, let the procedure `Reset` below be defined in the block `Estim`. The `Reset` procedure may then be called from the PCL command line by the following command

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

Grafcet

Grafcet, see [David and Alla, 1992] is a convenient and powerful way of implementing sequential algorithms. 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]. A Grafcet designed to control a boiler process is shown in Figure 2.

Each state in a sequential algorithm is defined as a step in Grafcet. A step may be active or inactive. Several steps 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;
```

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 StartFilling
  to KeepFilling, StartHeating when L1;

transition from StartHeating
  to StopHeating when T >= Tref;
```

A block algorithm may be described using both the **calculate**/**update** procedures and one or or several Grafcets. This is useful when an algorithm is composed of one periodic part and one sequential part. Consider for example a hybrid controller. The control law is conveniently expressed using Grafcet, where each step corresponds to a sub-controller. Assume that several of the control laws used are of state feedback type, and that all plant states cannot be measured. In this case we need to estimate the states. We want an estimator inside the block, but we do not want to have an estimator in each step. By implementing the estimator in the **calculate**/**update** section the estimates are available for all the control laws.

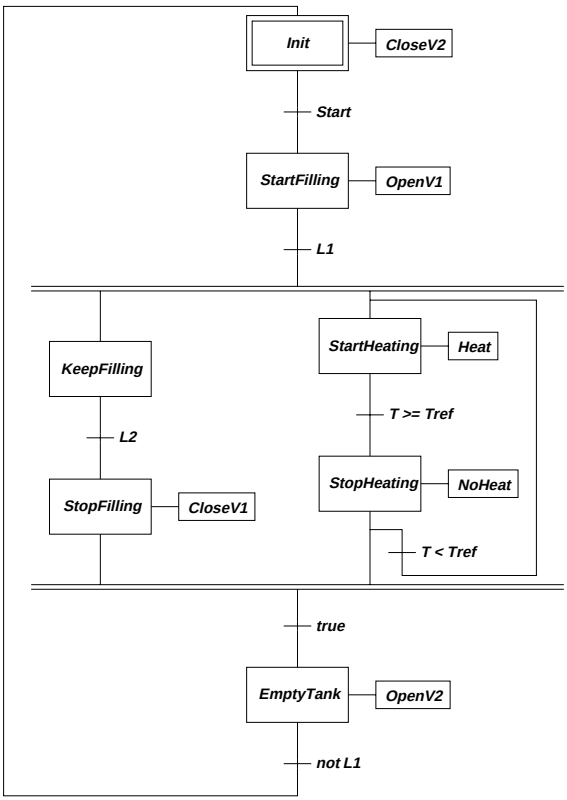


Figure 6. Grafcet for the controller of a boiler process. This figure is generated by the PAL compiler.

5. Calculate and Update

Periodic algorithms in PAL are divided into the two sections **calculate** and **update**. When a PAL 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 two main reasons to divide the code into two parts are to minimize the sampling delay and to handle signal limitations correctly.

Minimizing Sampling Delay

Consider the control system in Figure 7. First the value of the plant is sampled, and then the signal propagates from the first block to the second one, and so on. In a control system the stability of the closed loop

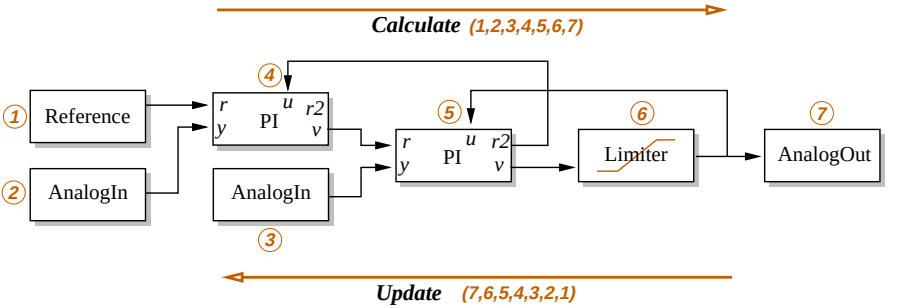


Figure 7. First the **calculate** method is executed for each block, starting with block number one. Then the **update** method is executed for each block in the reverse order.

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.

The algorithm for the PI-controller in Example 2 is divided into two parts, one for calculating the output signal and one for updating the integral state. It is important to finish the calculation of the new control signal as fast as possible in order to minimize the sampling delay. The update of the internal state is on the other hand allowed to be more time consuming. It can be shown that by treating a control block as two separate blocks, calculate and update, and scheduling them separately the control performance is improved, see [Cervin, 1999].

Handling Limiters

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 PI controller in Example 2 has 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. This execution order of the blocks is illustrated in Figure 7.

6. On-Line Configurations

PÅLSJÖ allows the block configuration to be edited without having to stop and restart the system. This is managed by using two identical sets of blocks called *running* and *shadow*. The running configuration is the set of blocks that are executing and the shadow configuration is used for editing. All commands that deal with creating blocks, moving blocks, connecting blocks, etc., are executed on the shadow configuration. When the edit session is completed the shadow configuration becomes the running configuration. The values of all internal states are copied to the shadow configuration before it is switched in. The new shadow configuration is now updated and turned into an identical copy of the current running configuration.

7. Summary

The PÅLSJÖ system consists of two main parts; a compiler and a framework. The compiler translates PAL code into C++ code that 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.

Related Work

Two previous research projects at the Department of Automatic Control in Lund that have influenced this work are LICS [Elmqvist, 1985] and Sim2DDC [Dahl, 1990]. The Simplex [Seto *et al.*, 1998] project at Carnegie Mellon University has inspired the support for variable structure controllers. Other interesting systems are the ControlShell framework [Schneider *et al.*, 1995], and the port based object approach built on top of the Chimera system[Stewart *et al.*, 1993].

8. References

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*, third edition. Prentice Hall, Upper Saddle River, New Jersey.

Blomdell, A. (1997): "PAL – the PÅLSJÖ algorithm language.". Report ISRN LUTFD2/TFRT--7558--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Cervin, A. (1999): "Improved scheduling of control tasks." In *Proceedings of the 11th Euromicro Conference on Real-time Systems*. Euromicro, Department of Automatic Control, Lund Institute of Technology, Sweden. In Submission.

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 International(UK).

Elmqvist, H. (1985): "LICS—Language for implementation of control systems." Report TFRT-3179. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Integrated Systems (1996a): *AutoCode User's Guide*. Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054-3309, USA.

Integrated Systems (1996b): *SystemBuild User's Guide*. Integrated Systems Inc, 3260 Jay Street, Santa Clara, CA 95054-3309, USA.

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.

MathWorks (1997): *Simulink–Real-Time Workshop*. The MathWorks inc., 24 Prime Park Way, Natick, MA 01760-1500.

Schneider, S. A., V. Chen, and G. Pardo-Castellote (1995): "The controlshell component-based real-time programming system." In *IEEE Conference on Robotics and Automation*. Real-Time Innovations, Inc., 954 Aster, Sunnyvale, California 94086.

Seto, D., B. H. Krogh, L. Sha, and A. Chutinan (1998): "Dynamic control system upgrade using the simplex architechture." *IEEE Control Systems*, **18:4**, pp. 72–80.

Stewart, D. B., R. A. Volpe, and P. K. Khosla (1993): "Design of dynamically reconfigurable real-time software using port-based objects.". Advanced Manipulators Laboratory, The Robotics Institute, and Department of Electrical and Computer Engineering, Carnegie Mellon University.

Paper 2

A Flexible Interactive Environment for Embedded Controllers

Johan Eker and Anders Blomdell

Abstract

Today embedded control systems are created using tools that give insufficient support for rapid prototyping and code reuse. New tools for the implementation of embedded controllers are necessary. This paper presents the experimental software framework PÅLSJÖ, and the language PAL. The PÅLSJÖ system is a rapid prototyping environment designed to support on-line system reconfigurations and code reuse. PAL is a new language, designed for implementation of control algorithms.

1. Introduction

The use of embedded control systems is growing rapidly. Nowadays advanced control systems can be found in a wide range of areas from industrial processes and aeroplanes to consumer products such as washing machines and home stereo equipment. In everyday life computers are becoming more and more indispensable for assistance. Some tasks are non-critical and may fail without dramatic consequences. If the word processor or email software crashes it is usually not a big problem, it is simple to reboot and carry on. On the other hand if the maneuvering systems in an airplane malfunctions it is unlikely that the pilot will save the situation by pressing Ctrl-Alt-Delete.

The development of embedded systems requires special tools. This paper introduces the PÅLSJÖ framework [Eker, 1997] and the PAL language [Blomdell, 1997]. PÅLSJÖ is an experimental rapid prototyping tool, developed with an emphasis on code reuse and interactivity in embedded systems. PAL, PÅLSJÖ Algorithm Language, is a dedicated controller description language used for implementing controllers that will run in PÅLSJÖ.

When developing real-time systems it is highly desirable to reuse components and modules from one program version to another or from one software project to another. The mixture of timing statements and logical statements in languages such as ADA and C/C++ makes it cumbersome to reuse code. Even if two applications share the implementation structure, code reuse is hindered by differences in timing requirements. Therefore to better support code reuse, timing specifications are kept separate from the control algorithm in PÅLSJÖ and PAL.

It is common to generate ADA or C/C++-code for controllers directly from a simulation environment such as Simulink [MathWorks, 1997] or SystemBuild [Integrated Systems, 1996a; Integrated Systems, 1996b]. The same control algorithm description may then be used both in simulation and in implementation. This is practical, if simulation is the main goal and implementation is just a feature. It may however be a serious drawback if the main goal is to implement a safe and efficient controller. One drawback with code generation is that it does not take into account the fact that sampling delay can be significantly reduced by rearranging the code, see [Åström and Wittenmark, 1997]. Another drawback is that most simulation languages are not designed to deal with execution in real-time, and therefore lack the necessary functionality. It would be better to use a tool that is designed especially for controller implementations, but that also supports simulation in some way. The requirements of a real-time language are typically quite different from those of a simulation language. For example, the weak typing in MATLAB, where basically everything is

matrices, is useful while designing and during simulations as the control engineer does not have to worry about data types and variable allocations. During implementation, on the other hand, strong typing and no global variables are needed to create safe real-time systems. If a simulation fails due to a programming error it is usually easy to correct it and restart the simulation. However, if the controller fails during an experiment instead it is much more time consuming and sometimes even dangerous.

The control engineer may add new types of blocks to Simulink and use them later for code generation, but must write them C. In PÅLSJÖ algorithms are written in PAL, which offers strong typing, high-level data-types and well defined real-time semantics. It is easy to translate PAL-code to MATLAB S-functions for use in simulation. To go in the other direction is much more complicated since PAL is more strongly typed than MATLAB. The code generated from Simulink and SystemBuild is static. It is not possible to replace blocks or components at run-time. The whole system must be recompiled with each change. The PÅLSJÖ environment has a much higher level of interactivity. It allows system reconfiguration and updates without having to restart the system. This offers the potential of replacing algorithms while the system is operating. The authors believe that an interactive development environment enhances the development process and speeds up controller implementation substantially.

Updating control software is a related issue. The software in embedded systems needs to be upgraded periodically. Since the cost for shutting down a manufacturing cell is high, upgrading the control algorithms might not be feasible, even if it would give better performance and cost reduction. There is a definite need of embedded control system software that allows on-line updates and reconfigurations. It should be possible to extend the system with new data types and classes while it is operating. Furthermore it should be possible to replace a running algorithm.

The PÅLSJÖ environment provides a framework designed to support on-line configurations and code reuse. 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.

Basic Idea

Many control systems have a similar internal structure. The same kind of building blocks are used and their internal communication follows certain patterns. There are usually functions for:

- **User interaction.** The operator must be able to set and inspect parameters and variables, make mode changes, start and stop operation, etc.
- **Task management.** Scheduling of real-time activities.

- **Data logging.** Logging of events, signals, etc.
- **Network interaction.** Managing the communication with other controllers and with the host system.

The code for these functions can only be reused if it is implemented in a flexible, modular fashion with well defined interfaces. Modules for a number of these basic common activities could then be arranged in a framework. Then the user of such a framework only needs to add code that is specific for a certain application, i.e., only the control algorithm.

Design Goals

A number of important issues were taken into account when the framework was designed. The most important ones were:

- **Rapid prototyping.** The framework should reduce the development time for prototype controllers.
- **Code reuse and algorithm libraries.** The framework must have a structure that allows algorithms to be stored and reused.
- **Expandable.** It should be possible to add new features, for example new data types to an existing system.
- **On-line configurable.** Changes in running setups should be allowed without stops.
- **Efficient execution.** Neither the code size nor the execution speed of an application built using the framework should differ from an application built from scratch.

2. PAL - Pålsjö Algorithm Language

PAL is a block based imperative language for implementation of embedded controllers. It is designed to support and simplify the coding 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.

In this section a brief introduction will be given to the PAL language. For a more in depth description see [Blomdell, 1997].

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.

The top level concept of PAL is the module. Several blocks may be grouped together in a module. In Example 1 the block SimplePI is declared within the module $MyBlocks$. The first section is the block interface, which here consists of two input signals y and $yref$, one output signal u, and two parameters K and Ti. The variable a is an *indirect* parameters, i.e. a function of other parameters. There are also two state variables e and I. The algorithm section comes after the interface section . The two methods **calculate** and **update**, define the periodic algorithm of the block.

EXAMPLE 1

```
module MyBlocks;
  block SimplePI

    y, ref : input real;
    u : output real;
    I := 0.0, e : real;
    K, Ti : parameter real;
    h : sampling interval;
    a = K * h/Ti;

    calculate begin
      e := ref − y;
      u := K * e + I;
    end calculate;

    update begin
      I := I + a * e;
    end update;
  end SimplePI;
end MyBlocks.
```

□

The **calculate** section is used to calculate output signals from input signals, and the **update** section is used to update the internal states. The algorithm is divided into two parts to minimize the sampling delay, i.e. the delay from the reading of the input signal to the writing of the output signal, and to enable the states to be correctly updated in case of signal limitations [Eker, 1997]. Typically the control signal is sent to the plant after **calculate** is executed, but before **update** is executed.

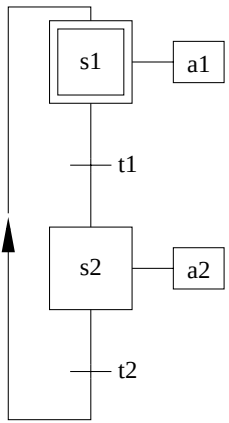
Grafcet

Grafcet [David and Alla, 1992] is a convenient and powerful tool for implementing sequential algorithms. Grafcet statements are expressed in PAL with a textual representation based on Sequential Function Charts in IEC-1131 [Lewis, 1995]. An example of a Grafcet in PAL is shown in Example 2.

As shown in Example 2, a block algorithm may be described using both the **calculate**/**update** procedures and one or more Grafcets. This is useful when an algorithm is composed of one periodic part and one sequential part. Consider for example a hybrid controller. The control law are conveniently expressed using Grafcet, where each step corresponds to a sub-controller. Assume that several of the control laws used are of state feedback type, and that all plant states cannot be measured. In this case the states need to be estimated. It is desireable to have an estimator inside the block, but not to have an estimator in each step. By implementing the estimator in the **calculate**/**update** section the estimates are available for all the control laws.

EXAMPLE 2

```
block Steps
  t1, t2 : input boolean;
  initial step s1;
    activate a1;
  end s1;
  step s2;
    pulse activate a2;
  end s2;
  action a1;
  begin
    ...
  end a1;
  action a2;
  begin
    ...
  end a2;
  transition from s1 to s2 when t1;
  transition from s2 to s1 when t2;
  calculate begin
    ...
  end calculate;
  update begin
    ...
  end update;
end Steps;
```



□

3. The Pålsjö framework

Fig. 1 shows the inheritance structure of the framework using using object diagram notation, see [Gamma *et al.*, 1995]. The functionality of the framework can be divided into three different parts: the *User Interface* handles commands from the operator; the *Network Communication* transmits and receives data over the network; and the *Workspace Manager* administers the blocks and the block type libraries. The fourth component is the block type library which contains the super class for all user defined block classes, and a set of system classes used for execution.

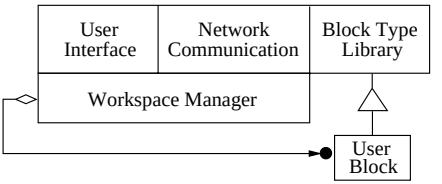


Figure 1. The framework handles the user interaction, the network communication and the block administration.

Definition of a Block

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 set of output signals O. An output signal may be connected to an input signal.
- A set of parameters P. Parameters can only be set from outside the block by the user or by the system. The value of a parameter cannot be changed internally in the block.
- A set of states S, which describe the internal states of the block. A state can only be assigned internally.
- A set of events E, which the block responds to. An event can be either synchronous or asynchronous. Synchronous events are executed at the next sampling instance. Asynchronous events are executed immediately when they arrive. A synchronous event could, for example be a request for a mode change in the controller. An emergency stop event should typically be asynchronous.
- Sequential logic L, which is described by one or several Grafcets.
- An algorithm A, that describes the periodic behavior of the block. If a block contains periodic algorithms it must be executed periodically.

To support the reuse of algorithms, the design decision was made to separate temporal and functional specifications in the design. 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

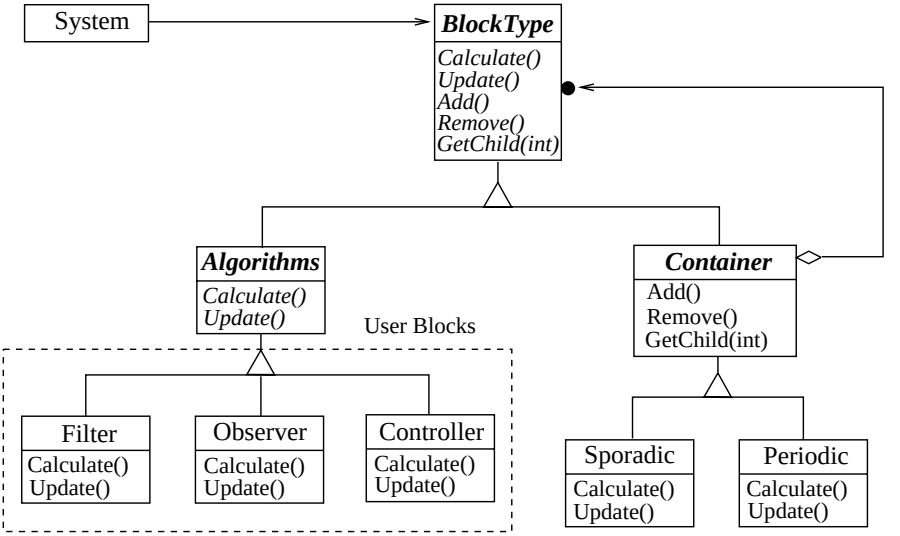


Figure 2. A user defined block is inherited from the pre-defined class `Algorithms`. `Container` is a super class designed to administer and encapsulate a set of blocks. Two subclasses `Periodic` and `Sporadic` are available. They are both used for managing the execution of other blocks.

programmer does not have to deal with any traditional real-time programming. Furthermore it is possible for the systems to arrange the blocks so that the execution is optimized.

The Framework and its Classes

The inheritance structure of the block type library is shown in Fig. 2. The super block to all block classes is `BlockType`, which contains the basic functionality that a class needs to be integrated into the PÅLSJÖ environment. The subclasses implement the abstract methods `Calculate` and `Update()`. Two subclasses from `BlockType` exist. One is `Container`, designed to encapsulate other blocks and the other is `Algorithms`, the super class for all user defined PAL blocks. Two subclasses from `Container` are available, `Periodic` and `Sporadic`. These are used to manage the execution of PAL blocks at run-time. The inheritance structure is known as the 'composite pattern' [Gamma *et al.*, 1995]. Using `Container` blocks the system supports a hierarchical structure. A block derived from Algorithm must have a `Container` block as a parent in order to be executed.

Creating Blocks and Signals

In PÅLSJÖ blocks and data-types are loosely coupled to the rest of the framework through the use of an 'abstract factory' pattern, see [Gamma

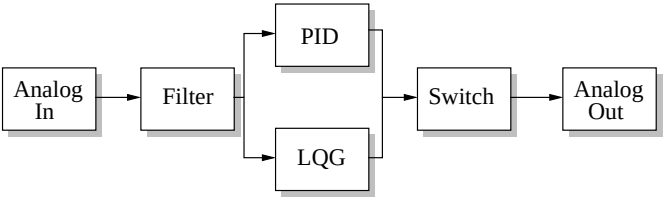


Figure 3. The block diagram created by the PCL commands in Example 3.

et al., 1995]. The run-time system calls the 'block factory' upon initialization and registers each available block type by giving a name tag and a constructor function. The 'block factory' keeps a table over all registered block types. When a client wants to create a new block it simply calls the 'block factory'. It is possible to register new block types and delete old ones during execution. Hence it is possible to extend the system without shutting it down. A similar factory is also used for dealing with data types.

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 during run-time using the PÅLSJÖ Configuration Language, PCL. Instances of the system blocks `Periodic` and `Sporadic` are created to manage the execution. Consider the block diagram in Fig. 3. 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 and is executed at a higher sampling rate than the control blocks. The desired real-time behaviors are the following:

- Execute **LQG**, **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-commands to achieve this are the following:

EXAMPLE 3

```
pcl>{
pcl*> p1 = new Periodic
pcl*> p2 = new Periodic
pcl*> p1.adin = new AnalogIn
```

```
pcl*> p1.filter = new Filter
pcl*> p2.lqg = new LQG
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.lqg.u
pcl*> p2.pid.y -> p2.switch.u1
pcl*> p2.lqg.y -> p2.switch.u2
pcl*> p2.switch.y -> p2.daout.u
pcl*> p1.tsamp = 0.010
pcl*> p2.tsamp = 0.050
```

□

The execution model used is shown in Fig. 4. The system blocks `Periodic` and `Sporadic` are mapped down to threads in the real-time kernel. The threads are scheduled with fixed priority scheduling.

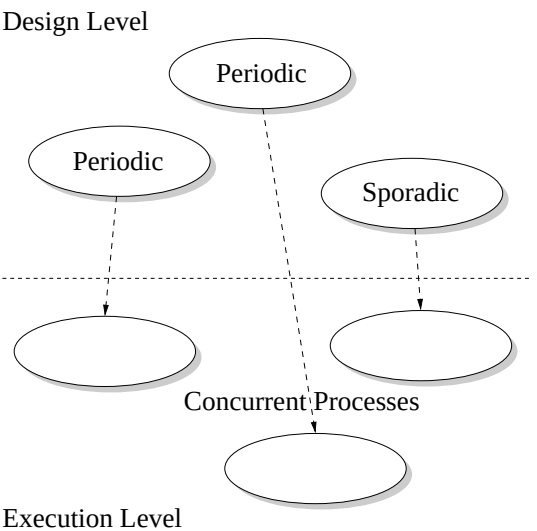


Figure 4. The `Periodic` and `Sporadic` blocks are mapped down to threads in the real-time kernel.

The `Periodic` block executes its child blocks according to the data flow between them using a schedule automatically created on-line. All timing and synchronization between the child blocks is taken care of by the scheduler in the `Periodic` block. Since the child blocks are executed

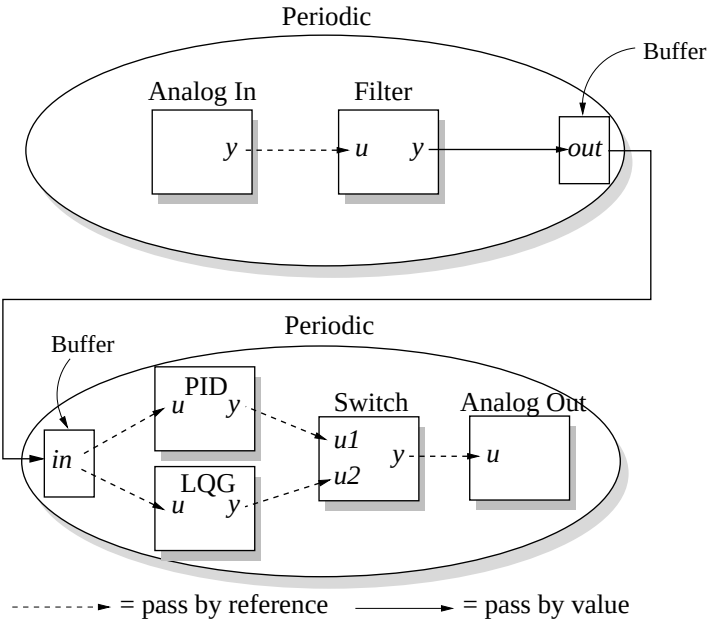


Figure 5. A system with one process for down sampling the measurement signal, and one process for the control algorithms.

sequentially no data sharing problems will arise. The data flow mechanism is implemented so that unnecessary copying is avoided. Input signals in blocks are implemented as pointers, which simply point at output signals, see Fig. 5. Thus no data needs to be copied when passing signals from one block to another within the same `Periodic` block. For connected blocks belonging to different `Periodic` blocks, mutual exclusion buffers are automatically created to preserve the consistency of data.

By grouping blocks together, the execution gets more efficient. Consider, for example, if each block was running as a separate process instead. 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 is be possible that data would be delayed by one sample for each block, from input to output. Another disadvantage would be the overhead introduced due to the context switching, when processes are stopped and restarted.

4. Signal Management

PAL has a set of predefined data types. Among those are regular scalar ones, e.g. real, integer, string. Furthermore there are aggregate types such as arrays, matrices, and polynomials. The availability of arithmetic operations for polynomials and matrices makes it straightforward to express control algorithms, see Example 4.

EXAMPLE 4

A : **matrix** [1..2, 1..2] of **real**;
B, x : **matrix** [1..2, 1..1] of **real**;
u : **input real**;
$x := A * x + B * u$;

□

The size of the aggregate data types cannot be changed from within the PAL code. They may however be parameterized using *dimension* variables. Dimension is a special parameter type used for defining the sizes of aggregate data types. The value of a dimension variable can only be set by the runtime system. The need of a special data type for handling sizes of aggegate data types is motivated by the need for synchronized size changes. In a situation where an output polynomial is connected to an input polynomial and the degrees are changed, it is necessary for this change to be done simultaneously throughout the system. Dimension variables are used in Fig. 6 where *in1* is an input array with n elements and *in2* is an array of m inputs.

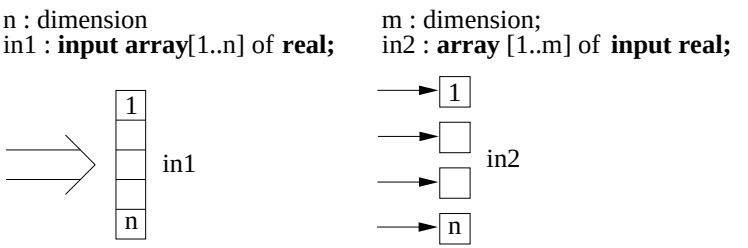


Figure 6. The input signal can be declared either as an input array or as an array of inputs. Dimension variables can be used to specify the size.

Connections

There are three types of connections in PÅLSJÖ:

1. Output signal to input signal;

2. Output signal to parameter;
3. Dimension to dimension.

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/update-procedures.

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

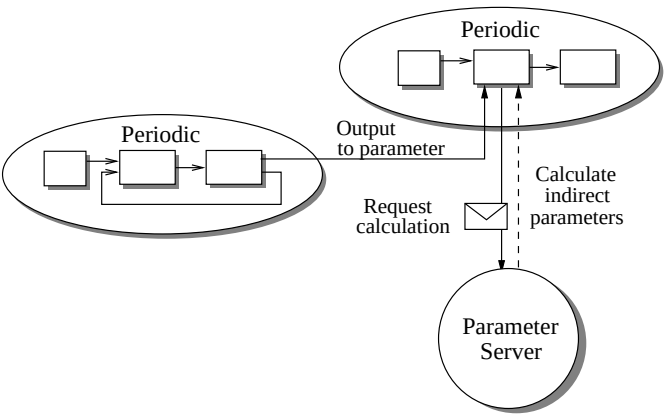


Figure 7. To handle connections of the type "output to parameter", special care must be taken. In PÅLSJÖ a special process, the Parameter Server process is used to handle this.

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 cause deadlines to be missed.

In PÅLSJÖ when a parameter receives a new value from an output signal, it sends a request to the 'parameter server', which then becomes responsible for calculating the indirect parameters, see Fig. 7. The 'parameter server' runs at a lower priority than the `Periodic/Sporadic` tasks and signals going via the 'parameter server' will be delayed. This is usually not a problem since this type of connection is designed for adaptive systems where the adaptation loop is allowed to be much slower than the feedback loop, see [Åström and Wittenmark, 1995].

The third type of connection is dimension to dimension. The dimension parameters of one or more PAL blocks may be connected to a global dimension variable. When the global dimension variable changes, all local

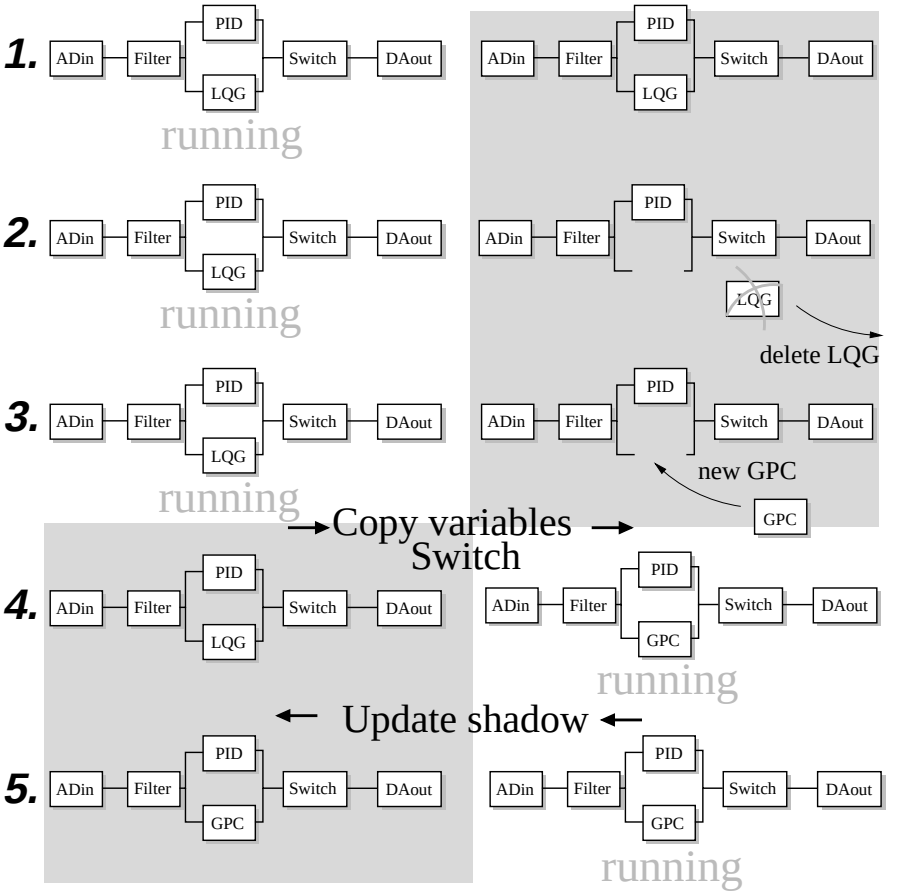


Figure 8. The internal structure during an edit session where one block is replaced by another.

dimension variables connected to it are updated. This update is made synchronously throughout the system. There is no restriction in what data types that may be used in connections of type one and two.

5. On-line Configurations

PÅLSJÖ allows the block configuration to be edited without having to stop and restart the system. It is handled by using two identical sets of blocks called *running* and *shadow*. The running configuration is the set of

blocks that are executing and the shadow configuration is used for editing. All commands that deal with creating blocks, moving blocks, connecting blocks, etc., are done on the shadow configuration. When the edit session is completed the shadow configuration becomes the running configuration. The values of all internal states are copied to the shadow configuration before it is switched in. The new shadow configuration is now updated and turned into an identical copy of the current running configuration. The copying of variables could become a problem if the data set is large. A better solution would be to pass a reference to the data set, thus making the switch independent of the size of the data set.

An edit session is shown in five steps in Fig. 8. Step one shows the initial state of the system. There are two identical copies of the block configuration. In step two the LQG block is deleted by the user. This operations only affects the shadow configuration at this stage. A new block called GPC is allocated and connected in step three. Now the shadow configuration is a correct and executable configuration. In step four a switch is made and the shadow configuration now becomes the running configuration. Before the switch is performed the values of all variables are copied from the running configuration to the shadow configuration. Finally the new shadow configuration is updated in step five.

6. Conclusion

The design of PÅLSJÖ has been inspired and influenced by a number tools. System descriptions in PÅLSJÖ are block oriented, similar to how systems are described in the simulation environments Simulink/Real-time Workshop and SystemBuild/Autocode. Complex controllers are formed by combining a number of basic building blocks. Many features in PÅLSJÖ were inspired by LICS [Elmqvist, 1985] and SIM2DDC [Dahl, 1990], both previous research projects at the Department. Ideas from the Simplex [Seto *et al.*, 1998] project on fault tolerant controllers have also influenced this work. Another interesting tool is ControlShell [RealTimeInnovations, 1995], a C++-class library combined with graphical editors.

The authors believe the goals stated in Section 1 have been reached with the PÅLSJÖ framework. The on-line configuration feature has however created a new set of problems. First the current switching mechanism is too crude. It is necessary to give the control engineer more power to specify how the switch shall be performed. There are also a number of other problems that have to be faced. For example, is it possible to predict the consequences of replacing one real-time task with another? Will all tasks still be schedulable in the new configuration, and if so, how will the system behave during the actual switching of the tasks? How can controllers be

designed so that they can be switched in and out without problems? The next step is to address some those issues. Better language is needed for writing controllers that can be switched in and out by a supervisor.

PÅLSJÖ is currently available for Motorola 68000 VME, Power PC VME and Windows NT. PÅLSJÖ is implemented on top of the STORK real-time kernel [Andersson and Blomdell, 1991]. STORK is a public domain real-time kernel available for Windows NT, Motorola 68000, Motorola Power PC and Sun Solaris 2.x. The real-time performance achieved on a Motorola 68040 are sampling intervals of around 5 milliseconds. The Power PC version is about 10-100 times faster.

7. Acknowledgment

This work was sponsored by National Swedish Board of Technical Development (NUTEK) under contract P9040-2. The authors wish to thank the anonymous reviewers for valuable comments.

8. References

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.

Åström, K. J. and B. Wittenmark (1995): *Adaptive Control*, second edition. Addison-Wesley, Reading, Massachusetts.

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*, third edition. Prentice Hall, Upper Saddle River, New Jersey.

Blomdell, A. (1997): "The PÅLSJÖ algorithm language.". Master's thesis, Department of Automatic Control, Lund Institute of Technology.

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 International(UK).

Eker, J. (1997): *A Framework for Dynamically Configurable Embedded Controllers*. Lic Tech thesis ISRN LUTFD2/TFRT--3218--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H. (1985): "LICS—Language for implementation of control systems." Report TFRT-3179. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Gamma, E., R. Helm, J. R., and J. Vlissides (1995): *Design Patterns-Elements of Reusable Object-Oriented Software*. Adison-Wesley, Reading, Massachusetts.

Integrated Systems (1996a): *AutoCode User's Guide*. Integrated Systems Inc., 3260 Jay Street, Santa Clara, CA 95054-3309, USA.

Integrated Systems (1996b): *SystemBuild User's Guide*. Integrated Systems Inc, 3260 Jay Street, Santa Clara, CA 95054-3309, USA.

Lewis, R. (1995): *Programming industrial control systems using IEC 1131–3*. The Institution of Electrical Engineers, London, U.K.

MathWorks (1997): *Simulink–Real-Time Workshop*. The MathWorks inc., 24 Prime Park Way, Natick, MA 01760-1500.

RealTimeInnovations (1995): *ControlShell–Object–Oriented Framework fro Real–Time System Software*. Real-Time Innovations, Inc., 155A Moffet Park Drive, Suite 111, Sunnyvale, CA 94089, USA.

Seto, D., B. H. Krogh, L. Sha, and A. Chutinan (1998): "Dynamic control system upgrade using the simplex architechture." *IEEE Control Systems*, **18:4**, pp. 72–80.

Paper 3

A Contract-Based Language for Embedded Control Systems

Johan Eker and Anders Blomdell

Abstract

The paper presents a new proposed block-oriented language called Friend, designed for the implementation of flexible and adaptive embedded control systems. Requirements on a controller block and its performance as a function of available resources are specified using contracts. A Friend control system is arranged in a hierarchical structure, where a parent-block is responsible for distributing resources to its child-blocks. The use of contracts simplifies the design and implementation of embedded systems that can adapt to altered operating conditions. The contracts allow the system to negotiate about resources, and redistribute them when necessary. Friend addresses the implementation of systems with adjustable quality of service-levels and the implementation of hybrid controllers. Friend is illustrated in the paper by two examples.

1. Introduction

Embedded control systems of today are generally not very flexible. They are often designed in an ad-hoc fashion, and the functionality is verified through testing. This makes them very sensitive to changes in the operating conditions, for example, decreased network bandwidth or less available CPU power. From a control-engineering point of view, the control tasks run in open loop, i.e. there is no feedback between the control tasks and the underlying real-time kernel. The purpose of feedback in automatic control is to adjust the control signal to accommodate for the control error. The same principle could be applied to real-time systems by viewing the resource-allocation level as the process output, and the priority of the tasks as the control signal. This paper proposes a controller description language, FRIEND, designed to support implementation of adaptive and flexible real-time control systems. The objectives of the FRIEND-project are to give good high-level support for flexible control systems, and to simplify the design of embedded systems with quality of service reasoning and feedback scheduling. A controller in FRIEND is associated with a *contract*, which specifies how the controller performance depends on the available resources. This information may be used by a supervisor to distribute resources to optimize the system in some respect. The FRIEND language also addresses the implementation of so called hybrid control systems, see [Branicky *et al.*, 1998]. A hybrid controller consists of a set of sub-controllers and a supervisor that selects which sub-controller to be active. The decision of the supervisor is based on so called switching rules, which in FRIEND are conveniently expressed in terms of contracts.

The paper is organized as follows. Section 2 gives some motivation and overview of related work. The concept of FRIEND is presented in Section 3 and the outline of the language is presented by two examples in Section 4. A summary is then given in Section 5.

2. Related Work

How can software be designed so that it can reconfigure itself, and load and unload software modules on-line in order to cope with changes in the environment? Such an adaptive behavior could prove useful in an embedded control system. The number of real-time tasks, the available resources and the usage thereof would then be allowed to change with time. A simple example of such a scenario, is an embedded control system where new control loops are added from time to time. Since it is a real-time system, care must be taken to ensure that enough computing and network resources are available for all tasks. A change in the task set re-

quires the task schedule to be recomputed. By introducing a feedback loop between the control loops and the kernel, an adaptive real-time system can be formed. Figure 1 illustrates this setup. The real-time scheduler

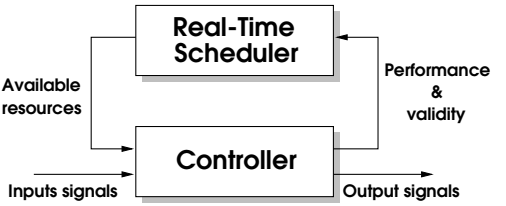


Figure 1. An adaptive real-time systems is formed by introducing a feedback loop between the real-time kernel and the tasks.

distributes system resources among the controllers, based on information given by the controllers. The inputs to the feedback scheduler from the control tasks are, for example, performance measurements, which indicate how well the tasks are working, and resource requirements, for example, CPU utilization. The output signals from the scheduler are the sampling frequencies of the control loops. By adjusting the sampling rates, the feedback scheduler can protect the system from overload. An example of a feedback scheduler is found in [Stankovic *et al.*, 1999].

A related language is FLEX, see [Natarajan and Lin, 1988], which supports the implementation of flexible real-time systems by introducing imprecise computations and constraint blocks. The body of a constraint block is executed only if the constraint is fulfilled, otherwise an exception is raised. A constraint may specify resource or timing requirements. FLEX allows the user to specify imprecise computations within a constraint block, and the run-time system then decides on a suitable precision level, that will not violate the constraint.

There are several existing software systems that are adaptive in one way or another. For example, RTPOOL [Abdelzaher *et al.*, 1997] allows the real-time processes to negotiate about system resources. Quality of service functions are associated with each process, and are used by the run-time system to optimize the global performance. Related work is found in [Andersson, 1999], where a framework for an adaptive multiprocessor system is suggested. The Rialto operating system [Jones *et al.*, 1995], from Microsoft Research, negotiates with its applications and distributes resources to maximize the performance of the entire system. The ALTAS testbed [Jehuda and Berry, 1995] implements a top-layer approach for handling dynamic job loads. The FARA framework [Rosu *et al.*, 1998] is designed for adaptive resource allocation. There are software components that represents the resources or the tasks, and they negotiate in order

to distribute the available resources. In [Krasner and Bernard, 1997] the design of the NASA deep-space mission software is described. Depending on the phase of the mission, the goals change and software modules are added or removed.

The ideas behind contracts have been adopted from [Helm *et al.*, 1990] and [Meyer, 1992]. A related architectural software pattern is the Broker pattern, see [Buschman *et al.*, 1996]. The Broker pattern is used for designing software with loosely coupled components which may be added or removed during run-time.

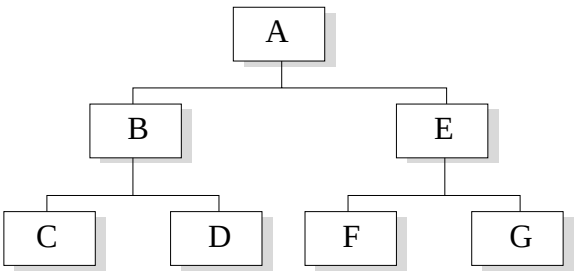


Figure 2. The hierarchical structure of a real-time control system designed in FRIEND. The blocks C, D, F, and G are control tasks, and blocks A, B, and E are supervisors.

3. Friendly Concepts

Flexible control systems may in FRIEND be designed in a hierarchical fashion, where the resources are distributed in a top-down manner. A block has one parent-block and possibly several child-blocks. It requests resources from its parent blocks, and distributes them among its child-blocks. Each block has one or more contracts with its parents-block that it must fulfill. This hierarchical structure is illustrated in Figure 2. The top level block A is assigned all available system resources and divides them between blocks B and E, by inspection of their contracts. Blocks B and C will then themselves distribute their alloted resources among their child-blocks. The contracts allow the parent-blocks to reason about how resources are best distributed. The blocks A, B, and E are *negotiator*-blocks. They allocate resources from their parent-block and divide them among their child-blocks. In Figure 2 only the blocks at the bottom level are actual control tasks, they are so called *algorithm*-blocks.

All blocks have an *interface*, which defines how the block communicates with the world, i.e. the format of its input and output signals. A block

may have none or several interfaces. The blocks C and D in Figure 3 may communicate with each other or with the world, but then only through the interface of block B. The interface of B may be viewed as the gateway to the rest of the system, for the blocks in B. In Figure 3, block B is assigned some Analog IO- and network-ports, and may divide them among its child blocks. The two child-blocks of B can be either algorithm or negotiator blocks, depending what type of controller block B is implementing.

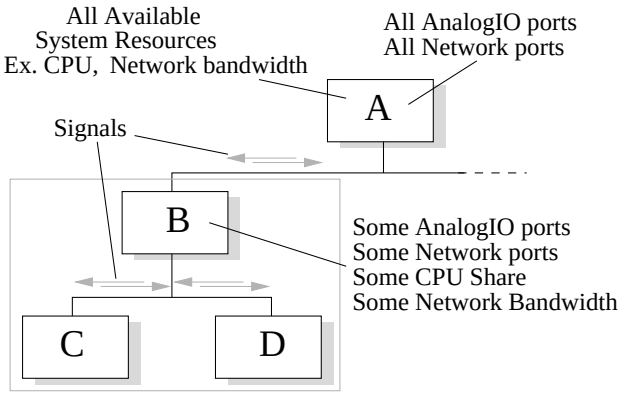


Figure 3. The resources of the system are distributed in a top-down fashion. The top block A contains all hardware specific information.

An Example

Consider a second order system, controlled by a hybrid controller with two sub-controllers. Which sub-controller to use, depends on the current position in the state space. Figure 4(a) shows how the state space is partitioned into two sectors, each corresponding to a controller mode, R_1 or R_2. Assume that the controller is doing well in the first and the third quadrants, but not so well in the second and fourth. The control engineer wants to improve the performance by adding two more controllers, one for the second quadrant and one for the fourth. How can this refinement process be made as simple and straightforward as possible? If the switching rules are implemented using automata all transitions must be altered. If, instead, each controller is associated with a contract, describing how well the controller operates at a certain point in the state space, the task of the supervisor would be to evaluate the contracts and select the best suited controller. If a new controller is added on-line, its contract is simply handed to the supervisor, which may now include it in the selection process. The contract approach to switching rules is well suited for in-

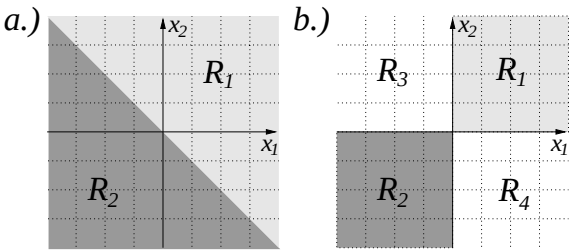


Figure 4. The switching rules for a hybrid controller are expressed as sectors in the states space. To the left the state space is divided between two controllers R_1 R_2, and to the right two more sub-controllers have been added.

cremental changes. Figure 4(b) shows how the state space is partitioned after two more sub-controllers are added.

System Resources

The only block in the hierarchy in Figure 2 that has any hardware specific information is block A. All the other blocks see the world through the eyes of block A. This means that it may be possible to write controller code that is parameterized in terms of the hardware. Instead of giving the execution times directly in seconds a more measure could be used. The task scheduling and resource allocation of the underlying real-time kernel are hidden from the user through the specifications of the top negotiator. A negotiator block may be viewed as a virtual computer, with a set of available resources.

4. The Friend Language

Does the world really need another language, and why cannot the proposed structure with contracts and negotiators be implemented using, for example, Java? The answers to those questions lie in what we are trying to achieve with FRIEND. The goal is to design a language that gives good support in writing adaptive control applications using contracts. In the same way that you could write object-oriented code in plain C, or write numerical software in Cobol, contract based control software can be implemented in the language of your choice. We believe, however, that the quality of the resulting application will be higher if the design paradigm and the implementation paradigm are the same. Hence the introduction of a new programming language. This section gives an introduction to FRIEND, and the syntax of the language is presented by examples.

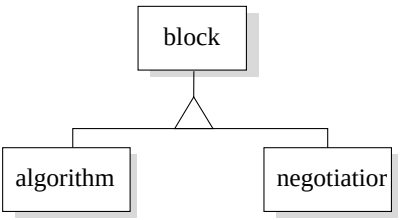


Figure 5. The block, the algorithm and the negotiator are the three types of block in FRIEND.

The Components

FRIEND is a small block based imperative language, and is a continuation of the PAL/PÅLSJÖ-project, see [Eker, 1997] [Blomdell, 1997]. In FRIEND there are four main components:

- **The Algorithm** An algorithm is a block that calculates new output signals given new input signals. The actual controller code is divided into two block-methods **calculate** and **update**. An algorithm block may consist of a set of sub-blocks of algorithm type.
- **The Negotiator** A Negotiator is a block that consumes and distributes resources. A negotiator may consist of a set of sub-blocks, which may be either algorithm blocks or negotiator blocks.
- **The Contract** A contract specifies how a block depends on available resources. A possible parent-block may require that a block fulfills a specified contract before it adopts it.
- **The Interface** An interface specifies how a block interacts with other blocks, i.e. the format of the input- and output-signals and of the parameters.

The two types of blocks, algorithm and negotiator, have the same abstract superblock, see Figure 5. The relation between the FRIEND components are shown in Figure 6. The labels on the arcs are the keywords used for connecting the components. For example, a negotiator may arbitrate contracts belonging to its child blocks, but also fulfill contracts with its parent block. A contract may require that the block that fulfills the contract has a certain interface. The FRIEND language is demonstrated below by two examples.

A Hybrid Controller

A hybrid controller consists of a combination of several different control strategies. Figure 7 shows an example of a hybrid control system. The

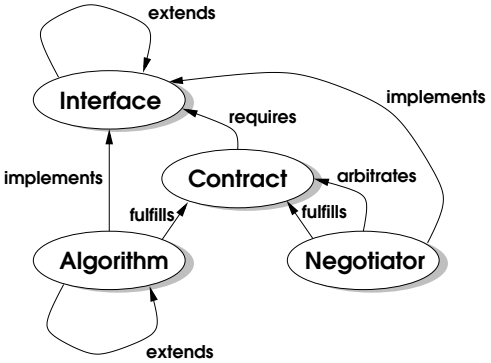


Figure 6. The relations of the Friend language components.

supervisor is used to select which sub-controller that should be active, and typically only one controller is affecting the plant at each point in time. The supervisor uses a set of switching rules to decide when to change the active controller. Consider an example with a hybrid controller, where

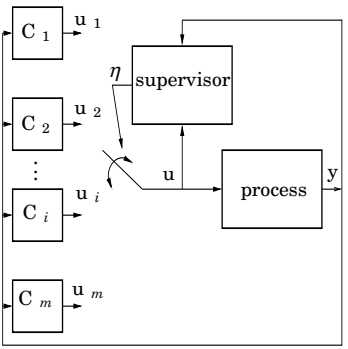


Figure 7. A hybrid controller consists of several sub-controllers C_i. A supervisor is used for selecting which control signal $u = u_\eta$, to be fed to the process.

each sub-controller is associated with a function V. The idea is to use the controller corresponding to the smallest value of the function V. For more on this switching criterion see [Malmborg, 1998]. This hybrid controller is implemented in block B, which consists of the two sub-controllers C and D, see Figure 2. Block B is a negotiator-block and block C and D are algorithm-blocks. Block B decides when and how the blocks C and D shall execute. Example 1 shows the outline of the FRIEND-code for such a hybrid controller.

EXAMPLE 1

```
interface SISO {
  y, y_ref : input real;
  u : output real;
}
```

The *SISO* interface defines a block interface with two input signals and one output signal. The signals may at run-time be connected to input and output signals of other blocks.

```
interface MISO extends SISO {
  y2 : input real;
}
```

The *SISO* interface is extended with an additional input signal and given the name *MISO*.

```
interface PI_pars {
  K, Ti : parameter real;
}
```

As shown in *PI_pars* an interface may also define block parameters.

```
contract C1 {
  negotiates sampling_interval : duration;
                 WCET : duration;
                 QoS : real;
       (sampling_interval) ↦ QoS;
  requires invocation < WCET ;
}
```

The contract *C1* requires that any block that fulfills it agrees with the following:

- The duration of an invocation must be less than *WCET* seconds. If the duration of an invocation exceeds *WCET* an exception will be raised and the parent block will be notified.
- A relation between the sampling interval and the quality of service (*QoS*) must be given in the block.

```
contract C2 extends C1 {
  requires V : real;
  requires interface io : MISO;
}
```

The contracts *C1* is extended to also require that:

- The *MISO* interface must be implemented.
- There must be an expression for calculating *V*.

```
algorithm PI {
  I := 0.0 : state real;
  implements io : SISO;
             pars : PI_pars;
  calculate {
    io.u := pars.K*(io.y_ref-io.y) + I;
  }
  update {
    I := I + pars.K/pars.Ti*(io.y_ref-io.y);
  }
}
```

The implementation of the PI-controller is divided into two parts: **calculate** and **update**. An algorithm block calculates new output signals based on the input signals, the states, and the parameters, and these calculations must always be implemented in **calculate** and **update**. The block *PI* implements two interfaces, one for communicating with the process and one for retrieving controller parameters.

```
negotiator B {
  active : block;
  implements io : MISO;
  fulfills myContract : C2 {
    QoS = active.QoS(sampling_interval);
    WCET = max(myBlocks.WCET);
    V = active.V;
  }
  arbitrates myBlocks[] : algorithm that
                          fulfills C2 {
    every myContract.sampling_interval {
      ∀i,   myBlocks[i].io = io;
      active := find_min(myBlocks[i].V);
      active.calculate;
      io = active.io;
      active.update;
    }
  }
}
```

Negotiator B accepts child blocks that fulfill the contract of type *C2*. At the beginning of every sampling interval the interfaces of all the child blocks are updated with new input values via the interface of B. The active sub-controller is then selected as the child-block with the smallest value of V (the function *find_min* is assumed to be available). The new output signal of B is given as the output signal from the active sub-controller.

```
algorithm C extends PI {
  α,β : parameter real;
  implements io2 : MISO {
    io2.u = io.u;
    io.y = io2.y;
    io.yref = io2.yref;
  }
  fulfills myContract : C2(io2 ↦ io) {
    QoS = α + β * sampling_interval^2;
    V = (io2.y1 − io2.y_ref)^2 + (io2.y2 − io.y_ref)^2;
    WCET = ...;
  }
}
```

Block C extends the *PI*-block by adding a contract. This is an example of how a control algorithm may be separated from application-specific information through use of contracts. All information that is related to the implementation of this specific hybrid controller is found in the contract. Note that *y2* is not used by the *PI* block.

□

A System with Feedback Scheduling

Consider a system where several control loops are executing on the same hardware. Furthermore, let the number of loops vary with time. Figure 8 illustrates this system. The CPU-consumption of a controller may be changed by adjusting its sampling frequency. The problem is twofold. First, the system must be kept schedulable, in spite of the changes in workload due to tasks arriving or tasks changing modes.

Secondly, the global control performance of the system should be optimized. This can be done by associating a cost function QoS_i, with each controller. The cost function indicates how the control performance depends on the level of alloted resources. The task of the parent-block is to

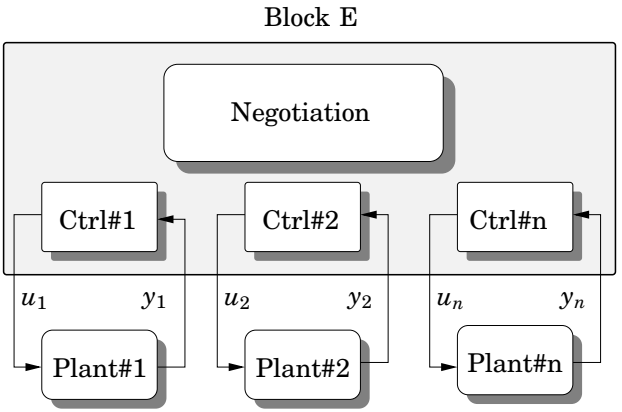


Figure 8. Several control loops execute at the same CPU, and are competing for the computing resources. If the number of control tasks change during run-time, the task schedule must be recalculated in order to avoid overload.

optimize the overall control performance, i.e.

$$\min QoS = \sum_{i=1}^{n} QoS_i,$$

and still keep the system schedulable. The outline of the FRIEND-code for this system is presented in Example 2 below.

EXAMPLE 2

```
contract C3 extends C1 {
  negotiates dQoS : real;
      (sampling_interval) ↦ (dQos);
  requires interface io : SISO;
}
```

The contract *C1* is extended with a new relation which specifies how the sampling interval affects *dQoS*, which is the derivative of the *QoS* with respect to the sampling interval.

```
negotiator E {
  implements io[] : SISO;
  fulfills myContract : C1 {
      QoS = ∑ myBlocks.QoS;
      WCET = ...;
```

```
  }
  arbitrates myBlocks[] : algorithm that
    fulfills C3 {
      on resourceChange {
        n := length(myBlocks);
        ...
        calculate H
        ...
        myBlocks.sampling_interval = H;
      }
      on new Blocks[] : algorithm {
        ...
        calculate H
        ...
        Blocks[i].sampling_interval = H;
        Blocks[i].io = io[i];
        schedule(Blocks);
      }
      on delete Blocks[] : algorithm {
        ...
      }
    }
  }
}
```

Negotiator *E* implements an array of interfaces. The actual size is determined at run-time and depends on available resources. It accepts blocks that fulfill contracts of type *C3*. The main difference between block *B* and *E* is that the child blocks of E execute concurrently. The task of block *E* is to distribute available computing resources. It determines the sampling intervals for its child-blocks through inspection of their contracts. By using the relations between the sampling interval and the quality of service, it may optimize the system with respect to available computing resources. The actual calculations of the new sampling intervals *H* are however omitted. Block *E* listens to three events, *resourceChange*, *new Block* and *delete Block*, which are caused by the run-time system. The event *resourceChange* is caused when the resources available for block *E* changes. The events *new Blocks* and *delete Blocks* are caused when blocks are added or removed. When the events occur, new sampling intervals are calculated for the child-blocks. The new blocks are then started by the schedule procedure, and will execute as periodic threads.

```
algorithm F extends PI {
  α, β : parameter real;
  fulfills myContract : C3 {
    QoS = α + β ∗ sampling_interval²;
    dQoS = 2 * β ∗ sampling_interval;
  }
}
```

Block *F* extends the *PI* algorithm by adding a contract.

□

5. Summary

This paper outlines the language FRIEND which is designed for the implementation of flexible, embedded control systems. The main goal is to support systems with an adaptive quality-of-service level and systems with feedback scheduling. The main idea is to separate the controller block into different parts that carry specific information, i.e. the actual algorithm, the contract, the interface, and the negotiator. The algorithm describes an input-output block. The contract describes how and when the controller should be used. The interface describes how the controller is connected to the environment. Contracts are suitable for expressing the demand and performance of a control algorithm. The negotiator allocates and distributes resources. The top negotiator contains all platform- and hardware-specific information. The run-time system is responsible for checking and evaluating contracts. It is assumed that there are underlying mechanisms that support dynamic linking of block code in real-time.

There are currently no FRIEND compiler available, and it is most likely that the language will evolve during the implementation. This paper only discusses the contract aspects of FRIEND. The rest of the language is very similar to PAL, and FRIEND may be viewed as 'PAL with contracts'. FRIEND is not intended as a general real-time language, but instead as a tool that will support the incorporation of adaptive behavior in embedded control systems.

6. References

Abdelzaher, T., E. Atkins, and K. Shin (1997): "QoS negotiation in real-time systems, and its application to flight control." In *Proceedings of*

the IEEE Real-Time Systems Symposium.

Andersson, B. (1999): "Adaption of time-sensitive tasks on shared memory multiprocessors: A framework suggestion.". Master's thesis 777, Department of Computer Engineering, Chalmers University of Technology.

Blomdell, A. (1997): "PAL – the PÅLSJÖ algorithm language.". Report ISRN LUTFD2/TFRT--7558--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Branicky, M. S., V. S. Borkar, and S. K. Mitter (1998): "A unified framework for hybrid control: Model and optimal control theory." *IEEE Trans. Automatic Control*, **43(1)**, p. 31.

Buschman, F., R. Meunier, P. Rohnert, H. Sommerlad, and M. Stal (1996): *A System of Patterns-Pattern Oriented Software Architechture*. Wiley, Chichester, West Sussex.

Eker, J. (1997): *A Framework for Dynamically Configurable Embedded Controllers*. Lic Tech thesis ISRN LUTFD2/TFRT--3218--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Helm, R., I. M. Holland, and D. Gangopadhyay (1990): "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pp. 169–180. Published as ACM SIGPLAN Notices, volume 25, number 10.

Jehuda, J. and D. Berry (1995): "A top layer design approach for adaptive real-time software." In *IFAC Real Time Programming*. IFAC, Fort Lauderdale.

Jones, M. B., P. J. Leach, R. P. Davis, and J. S. Barerra III (1995): "Modular real-time resouce management in the rialto operationg system." Technical Report MSR-TR-95-16. Microsoft Research, Advanced Technology Division, Microsoft Corpration, One Microsoft Way, Redmond, WA 98052.

Krasner, S. M. and D. E. Bernard (1997): "Integrating autonomy technologies into an embedded spacecraft system-flight software system engineering for new millennium." In *Aerospace Conference*, vol. 2, pp. 409–420. IEEE.

Malmborg, J. (1998): *Analysis and Design of Hybrid Control Systems*. PhD thesis ISRN LUTFD2/TFRT--1050--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Meyer, B. (1992): "Applying "design by contract"." *Computer*, **25:10**, pp. 40–51.

Natarajan, S. and K. Lin (1988): "FLEX: towards flexible real-time programs." In *Proceedings of the International Conference on Computer Languages*. IEEE.

Rosu, D., K. Schwan, and S. Yalamanchili (1998): "Fara - a framework for adaptive resource allocation in complex real-time systems." In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pp. 79 –84. IEEE.

Stankovic, J., C. Lu, and S. Son (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Paper 4

A Feedback Scheduler for Real-time Controller Tasks

Johan Eker, Per Hagander and Karl-Erik Årzén

Abstract

The problem studied in this paper is how to distribute computing resources over a set of real-time control loops in order to optimize the total control performance. Two subproblems are investigated: how the control performance depends on the sampling interval, and how a recursive resource allocation optimization routine can be designed. Linear quadratic cost functions are used as performance indicators. Expressions for calculating their dependence on the sampling interval are given. An optimization routine, called a feedback scheduler, that uses these expressions is designed.

1. Introduction

Control design and task scheduling are in most cases today treated as two separate issues. The control community generally assumes that the real-time platform used to implement the control system can provide deterministic, fixed sampling periods as needed. The real-time scheduling community, similarly, assumes that all control algorithms can be modeled as periodic tasks with constant periods, hard deadlines, and known worst case execution times. This simple model has made it possible for the control community to focus on its own problem domain without worrying how scheduling is being done, and it has released the scheduling community from the need to understand how scheduling delays impact the stability and performance of the plant under control. From a historical perspective, the separated development of control and scheduling theories for computer based control systems has produced many useful results and served its useful purpose.

Upon closer inspection it is, however, quite clear that neither of the above assumptions need necessarily be true. Many of the computing platforms that are commonly used to implement control systems, are not able to give any deterministic guarantees. This is especially the case when commercial off-the-shelf operating systems are used. These systems are, typically, designed to achieve good average performance rather than high worst-case performance. Many control algorithms are not periodic, or they may switch between a number of different fixed sampling periods. Control algorithm deadlines are not always hard. On the contrary, many controllers are quite robust towards variations in sampling period and response time. It is in many cases also possible to compensate for the variations on-line by, e.g., recomputing the controller parameters, see [Nilsson, 1998]. It is also possible to consider control systems that are able to do a tradeoff between the available computation time, i.e., how long time the controller may spend calculating the new control signal, and the control loop performance.

For more demanding applications, requiring higher degrees of flexibility, and for situations where computing resources are limited, it is therefore motivated to study more integrated approaches to scheduling of control algorithms. The approach taken in this paper is based on dynamic feedback from the scheduler to the controllers, and from the controllers to the scheduler. The idea of feedback has been used informally for a long time in scheduling algorithms for applications where the dynamics of the computation workload cannot be characterized accurately. The VMS operating system, for example, uses multi-level feedback queues to improve system throughput, and Internet protocols use feedback to help solve the congestion problems. The idea of feedback has also been ex-

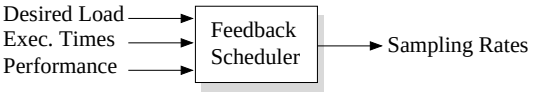


Figure 1. New sampling rates are calculated based on the desired CPU load, the execution times of the controllers, and the performance of the controllers.

ploited in multi-media scheduling R&D, recently, under the title of quality of service (QoS).

In this paper we want to schedule a set of real-time control loops to maximize their total performance. The number of control loops and their execution times may change over time and the task schedule must hence be adjusted to maintain optimality and schedulability. Since the optimizer works in closed loop with the control tasks, it is referred to as a feedback scheduler. The feedback scheduler adjusts the control loop frequencies to optimize the control performance while maintaining schedulability. The input signals are the performance levels of each loop, their current execution times, and the desired workload level, see Figure 1. To design such a feedback scheduler two problems are studied. The first is to find a suitable performance index and calculate how it depends on the sampling frequency. The second problem is to design an optimization routine. It is assumed that measurements or estimates of the execution times and system workload are available from the real-time kernel at run-time.

As a performance index, a linear quadratic (LQ) formulation is used. The controllers are state feedback algorithms designed to minimize this quadratic cost function. The performance index is calculated as a function of the sampling interval. Given this control performance indicator an optimization routine is designed. The optimization routine aims at finding the control performance optimum, given a desired CPU utilization level (workload) and the execution times for the tasks. The global cost function that should be minimized consists of the sum of the cost function of each control loop. The minimization is performed subject to a schedulability constraint. The minimization of the global cost function constitutes a nonlinear programming problem. The solution of this problem is facilitated by the knowledge of the first and second derivatives of the cost functions with respect to the sampling rate. Expressions for the derivatives are calculated and used in the optimization.

Using a feedback scheduling strategy it is now possible to design real-time control systems that are more robust against uncertainties in execution times and workload. When a control task performs a mode change and this leads to a change in its execution time the feedback scheduler adjusts the sampling frequencies so that the system remains schedulable.

Deadlines may be missed if a change in execution time causes the system to overload. In such a situation, the feedback scheduler adjusts the sampling rates to regain schedulability, but deadlines may, however, still be missed. If the feedback scheduler, instead, is aware of a forthcoming mode change, it could avoid an overload by changing the sampling rates in advance. This can be dealt with by establishing a communication channel between the feedback scheduler and the control tasks. The control tasks notify the feedback scheduler, by sending a request, prior to a mode change and may not proceed with the mode change until given permission to do so. The module that handles this is called the *admission controller*. It is also responsible for handling the arrival of new tasks and the termination of old tasks. Figure 2 shows a block diagram of the feedback scheduler with the admission controller.

Outline

An overview of related work is given in Section 2. Section 3 defines the problem, and the LQ-based cost function calculation are presented in Section 4. Section 5 describes the design of the feedback scheduler.

2. Integrated Control and Scheduling

In order to achieve on-line interaction between control algorithms and the scheduler a number of issues must be considered. Control design methods must take schedulability constraints into account. It must be possible to dynamically adjust task parameters, e.g., task periods, in order to compensate for changes in workload. It can also be advantageous to view the task parameters adjustment strategy in the scheduler as a controller. In this section an overview is given of the work that has been performed in these areas. A more detailed survey on control and scheduling can be found in [Årzén *et al.*, 1999].

Control and scheduling co-design

A prerequisite for an on-line integration of control and scheduling theory is that we are able to make an integrated off-line design of control algorithms and scheduling algorithms. Such a design process should ideally allow an incorporation of the availability of computing resources into the control design by utilizing the results of scheduling theory. This is an area where so far relatively little work has been performed. In [Seto *et al.*, 1996] an algorithm was proposed that translates a system performance index into task sampling periods, considering schedulability among tasks running with pre-emptive priority scheduling. The sampling periods were

considered as variables, and the algorithm determined their values so that the overall performance was optimized subject to the schedulability constraints. Both fixed priority rate-monotonic and dynamic priority, Earliest Deadline First (EDF) scheduling were considered. The loop cost function was heuristically approximated by an exponential function. The approach was further extended in [Seto *et al.*, 1998].

A heuristic approach to optimization of sampling period and input-output latency subject to performance specifications and schedulability constraints was also presented in [Ryu *et al.*, 1997; Ryu and Hong, 1998]. The control performance was specified in terms of steady state error, overshoot, rise time, and settling time. These performance parameters were expressed as functions of the sampling period and the input-output latency. An iterative algorithm was proposed for the optimization of these parameters subject to schedulability constraints.

Task attribute adjustments

A key issue in any system allowing dynamic feedback between the control algorithms and the on-line scheduler is the ability to dynamically adjust task parameters. Examples of task parameters that could be modified are period and deadline.

In [Shin and Meissner, 1999] the approach in [Seto *et al.*, 1996] is extended, making on-line use of the proposed off-line method for processor utilization allocation. The approach allows task period changes in multiprocessor systems. A performance index for the control tasks is used, weighting the importance of the task to the overall system, to determine the value to the system of running a given task at a given period.

In [Buttazzo *et al.*, 1998] an elastic task model for periodic tasks is presented. A task may change its period within certain bounds. When this happens, the periods of the other tasks are adjusted so that the overall system is kept schedulable. An analogy with a linear spring is used, where the utilization of a task is viewed as the length of a spring that has a given spring coefficient and length constraints. The MART scheduling algorithm [Kosugi *et al.*, 1994; Kosugi *et al.*, 1996; Kosugi and Moriai, 1997] also supports task period adjustments. MART has been extended to also handle task execution time adjustments. The system handles changes both in the number of periodic tasks and in the task timing attributes. Before accepting a change request the system analyzes the schedulability of all tasks. If needed, it adjusts the period and/or execution time of the tasks to keep them schedulable with the rate monotonic algorithm.

Feedback scheduling

An on-line scheduler that dynamically adjusts task attributes can be viewed as a controller. Important issues that must be decided are what

the right control signals, measurement signals, and set-points are, what the correct control structure should be, and which process model that may be used.

So far, very little has been done in the area of real-time feedback scheduling. A notable exception is [Stankovic *et al.*, 1999] where it is proposed to use a PID controller as an on-line scheduler under the notion of Feedback Control-EDF. The measurement signal (the controlled variable) is the deadline miss ratio for the tasks, and the control signal is the requested CPU utilization. Changes in the requested CPU are effectuated by two mechanisms (actuators). An admission controller is used to control the flow of workload into the system, and a service level controller is used to adjust the workload inside the system. The latter is done by changing between different versions of the tasks with different execution time demands. A simple liquid tank model is used as an approximation of the scheduling system.

Using a controller approach of the above kind, it is important to be able to measure the appropriate signals on-line, e.g., to be able to measure the deadline miss ratio, the CPU utilization, or the task execution times.

An event feedback scheduler is proposed in [Zhao and Zheng, 1999]. Several control loops share a CPU, and only one controller may be active at each time instant. The strategy used is to activate the controller connected to the plant with the largest error. Similar ideas are found in [Årzén, 1999], which suggests an event based PID controller that only executes if the control error is larger than a specified threshold value.

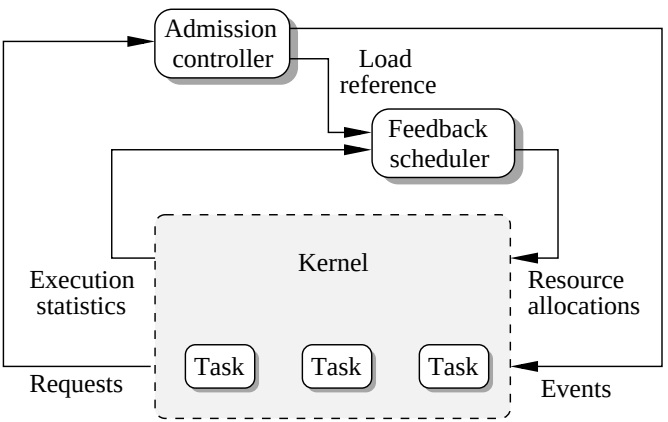


Figure 2. The real-time kernel is connected in a feedback loop with the feedback scheduler and the admission controller.

3. Problem Statement

Consider a control system where several control loops share the same CPU. Let the execution times of all tasks and the system workload be available from the real-time kernel at all times. Furthermore, associate each controller with a function, which indicates its performance. The execution times, the number of tasks, and the desired workload may vary over time. The aim of the feedback scheduler is to optimize the total control performance, while keeping the workload at the desired level.

Two separate modules are used, see Figure 2. The feedback scheduler adjusts the sampling intervals to control the workload. The workload reference is calculated by the admission controller which contains high-level logic for coordinating tasks.

The tasks and the feedback scheduler communicate using requests and events. A task can send a request for more computing resources and the scheduler may grant this by replying with the appropriate event. The kernel manages the tasks at the low level, i.e. task dispatching, etc. The feedback scheduler gets execution statistics, such as the actual execution times of the tasks, and the system workload from the kernel. This information is then used to calculate how CPU resources should be allocated in an optimal fashion.

Let the number of tasks on the system be n, and let each task execute with a sampling interval h_i (task period) and have the execution time C_i. Each task is associated with a cost function $J_i(h_i)$, which gives the control cost as a function of sampling interval h_i. The following optimization problem may now be formulated

$$\text{minimize} \quad J = \sum_{i=1}^{n} J_i(h_i), \tag{1}$$

subject to the constraint $\sum_{i=1}^{n} C_i/h_i \leq U_{ref}$, where U_{ref} is the desired utilization level.

4. Cost Functions

In this section the LQ cost function and its derivatives with respect to the sampling interval are calculated. These functions will be used in the feedback scheduler algorithm, as described in Section 5.

Let the system be given by the linear stochastic differential equation

$$dx = Axdt + Budt + dv_c. \tag{2}$$

where A and B are matrices and v_c is a Wiener process with the incremental covariance $R_{1c}dt$. The sampled system is then given by

$$x(kh+h) = \Phi x(kh) + \Gamma u(kh) + v(kh),$$

where $\Phi = e^{Ah}$, $\Gamma = \int_0^h e^{As}dsB$, and $v(kh)$ is noise with the following property:

$$\mathrm{E}v(kh)v^T(kh) = R_1(h) = \int_0^h e^{A\tau}R_{1c}e^{A^T\tau}d\tau.$$

The cost function for the controller is chosen as

$$J(h) = \frac{1}{h}\mathrm{E}\int_0^h \begin{bmatrix} x^T(t) & u^T(t) \end{bmatrix} Q_c \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt,$$

where

$$Q_c = \begin{bmatrix} Q_{1c} & Q_{2c} \\ Q_{2c}^T & Q_{3c} \end{bmatrix}.$$

Note that only the cost in stationarity is regarded, and that the cost is scaled by the time horizon, i.e. the sampling interval h. The controller $u = -Lx(t)$ that minimizes the cost is given by solving the stationary Riccati equation with respect to the matrices S and L.

$$\begin{bmatrix} S + L^TGL & L^TG \\ GL & G \end{bmatrix} = \begin{bmatrix} \Phi^T \\ \Gamma^T \end{bmatrix} S \begin{bmatrix} \Phi & \Gamma \end{bmatrix} + Q_d \tag{3}$$

where $G = \Gamma^T S\Gamma + Q_3 d$, and

$$Q_d = \int_0^h e^{\Sigma^T t}Q_c e^{\Sigma t}dt, \qquad \Sigma = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \tag{4}$$

The minimal value of J is given by [Åström and Wittenmark, 1997]; [Gustafsson and Hagander, 1991] as:

$$minJ(h) = \frac{1}{h}(\mathrm{tr}SR_1 + \bar{J}) \qquad \text{where}$$
$$\bar{J} = \mathrm{tr}(Q_{1c}\int_0^h R_1(\tau)d\tau)$$

In order to use the cost function in the optimization it is useful to know the derivatives with respect to the sampling period.

THEOREM 1
The first derivative of J is given as

$$\frac{dJ}{dh} = \frac{1}{h}(\mathrm{tr}\frac{dS}{dh}R_1 + \mathrm{tr}S\frac{dR_1}{dh} + \frac{d\bar{J}}{dh}) - \frac{1}{h^2}(trSR_1 + \bar{J})$$

where

$$\begin{aligned}\frac{dS}{dh} =& (\Phi - \Gamma L)^T \frac{dS}{dh}(\Phi - \Gamma L) + W,\\ W =& \begin{bmatrix}\Phi - \Gamma L\\ -L\end{bmatrix}^T Q_c \begin{bmatrix}\Phi - \Gamma L\\ -L\end{bmatrix} +\\ & \{(\Phi - \Gamma L)^T A^T - L^T B^T\} S(\Phi - \Gamma L) +\\ & (\Phi - \Gamma L)^T S\{A(\Phi - \Gamma L) - BL\},\end{aligned}$$

The matrix R_1 is calculated using the expressions in Equation 1.

$$\begin{aligned}R_1 =& \Psi_{22}^T \Psi_{12}, \quad \text{where}\\ & \begin{bmatrix}\Psi_{11} & \Psi_{12}\\ 0 & \Psi_{22}\end{bmatrix} = \exp\Big(\begin{bmatrix}-A & R_{1c}\\ 0 & A^T\end{bmatrix} h\Big),\\ \frac{dR_1}{dh} =& e^{Ah} R_{1c} e^{A^T h},\\ \frac{d\bar{J}}{dh} =& \mathrm{tr}Q_{1c}R_1\end{aligned}$$

□

See Appendix A for the proof. The expression for the second derivative of the cost functions is also given there.

EXAMPLE 1
Consider the linearized equations for a pendulum:

$$\begin{aligned}dx &= \begin{bmatrix}0 & 1\\ \alpha\omega_0^2 & -d\end{bmatrix} x dt + \begin{bmatrix}0\\ \alpha b\end{bmatrix} u dt + dv_c\\ y &= [\,1 \quad 0\,]\, x, \qquad R_{1c} = \begin{bmatrix}0 & 0\\ 0 & \omega_0^4\end{bmatrix}\end{aligned}$$

The natural frequency is ω_0, the damping $d = 2\zeta\omega_0$, with $\zeta = 0.2$, and $b = \omega_0/9.81$. If $\alpha = 1$, the equations describe the pendulum in the upright

position, and if $\alpha = -1$ they describe the pendulum in the downward position. The incremental covariance of v_c is $R_{1c}dt$, which corresponds to a disturbance on the control signal.

The cost functions J for the closed loop control of the inverted pendulum as a function of the sampling interval is shown in Figure 3. The corresponding function for the stable pendulum is shown in Figure 4.

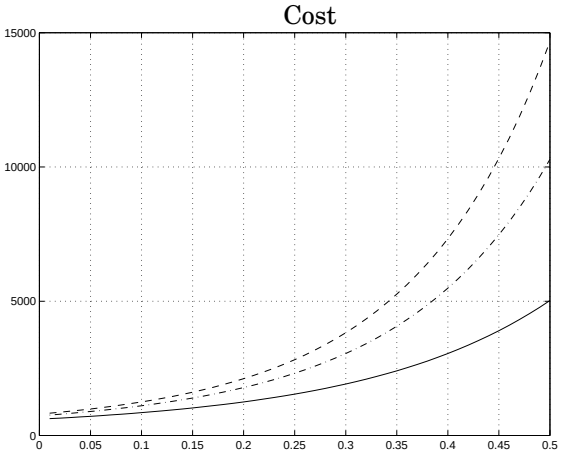


Figure 3. The cost $J_i(h)$ as a function of the sampling interval for the inverted pendulum. The plot shows the graph for $\omega_0 = 3.14$(full), 3.77(dot-dashed), and 4.08(dashed).

Figure 4 clearly demonstrates that faster sampling not necessarily gives better control performance. Sampling the pendulum as slowly as this is however unrealistic. The rule of thumb from [Åström and Wittenmark, 1997] is to chose the sampling rate as $\omega_0 h \approx 0.2 - 0.6$. □

5. A Feedback Scheduler

In this section a feedback scheduler, see Figure 2, for a class of control systems with convex cost functions is proposed. Standard nonlinear programming results are used as a starting point for the feedback scheduler design. First, an algorithm, using the cost functions presented in the previous section is designed, and then an approximate, less computation-intense algorithm is presented. Simulation results for a system with three control loops are also given.

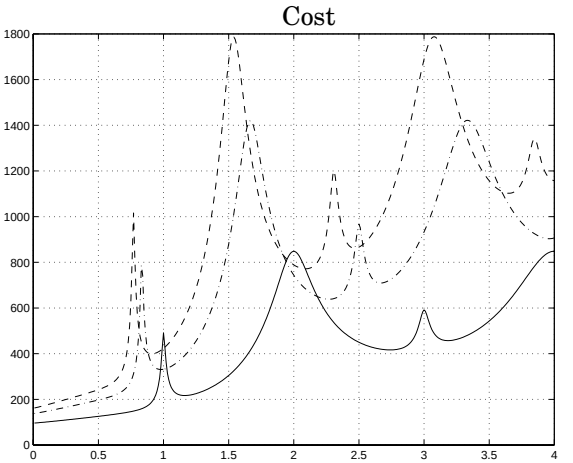


Figure 4. The cost $J_i(h)$ as a function of the sampling interval for the pendulum. The plot shows the graph for $\omega_0 = 3.14$(full), 3.77(dot-dashed), and 4.08(dashed). The peaks are due to the resonance frequencies of the pendulum.

Static Optimization

For the class of systems for which the cost functions are convex, ordinary optimization theory may be applied. The optimization criterion in Equation (1) has nonlinear constraints and is first rewritten. By optimizing over the frequencies instead of the sampling intervals, the following optimization problem is given:

$$\begin{aligned} \min_f V(f) &= \sum_{i=1}^{n} J_i(1/f_i), \\ \text{subject to} \quad & C^T f \leq U_{ref} \\ & f = [f_1, \ldots, f_n]^T \end{aligned}$$

The Kuhn-Tucker conditions, see for example [Fletcher, 1987], give that if $\bar{f} = [\bar{f}_1, \ldots, \bar{f}_n]^T$ is an optimal solution then:

$$\begin{aligned} V_f(\bar{f}) + \lambda C &= 0 \\ \lambda[U_{ref} - C^T \bar{f}] &= 0, \\ \lambda &\geq 0 \end{aligned} \tag{5}$$

where V_f is the gradient, and $C = [C_1, \ldots, C_n]^T$.

Recursive Optimization

Since changes in the computer load U_{ref} and the execution times C change the optimization problem, they need to be resolved at each step in time. In principle, one can repeat the solution to the static optimization problem at each step in time. However, instead of looking for a direct solution we will consider a recursive algorithm. Assume that we have a solution $f(k)$, at step k, which is optimal or close to optimal. We will now construct a recursive algorithm to compute new values for f and λ. Let the execution

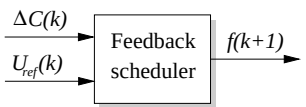


Figure 5. The feedback scheduler calculates new sampling frequencies to accomodate for changes in the execution times C or in the desired workload level.

time vector $C(k)$ be time-varying. We want to design a control loop that adjusts the sampling frequencies so that the control performance cost is kept at the optimum, see Figure 5. Let

$$\begin{aligned} f(k+1) &= f(k) + \Delta f(k) \\ \lambda(k+1) &= \lambda(k) + \Delta\lambda(k) \\ C(k+1) &= C(k) + \Delta C(k) \end{aligned}$$

We assume that $\lambda > 0$, i.e. that the CPU constraint is active. Linearization of Equation (5) around the optimum gives

$$\begin{aligned} V_f + V_{ff}\Delta f + (\lambda + \Delta\lambda)(C + \Delta C) &= 0 \\ (C^T + \Delta C^T)(f + \Delta f) &= U_{ref} \end{aligned}$$

If the quadratic delta terms are disregarded we get

$$\begin{bmatrix} V_{ff} & C \\ C^T & 0 \end{bmatrix} \begin{bmatrix} \Delta f \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -(V_f + \lambda C) - \lambda\Delta C \\ U_{ref} - C^T f - \Delta C^T f \end{bmatrix}$$

Now the increments in f and λ are given as

$$\begin{bmatrix} \Delta f \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} V_{ff} & C \\ C^T & 0 \end{bmatrix}^{-1} \begin{bmatrix} -(V_f + \lambda C) \\ U_{ref} - C^T f - \Delta C^T f \end{bmatrix} \tag{6}$$

A solution exists if V_{ff} is positive (or negative) definite and $C \neq 0$. Note that the matrix V_{ff} is diagonal.

$$V_{ff} = diag(\begin{bmatrix} \frac{\partial^2 V}{\partial f_1^2} & \cdots & \frac{\partial^2 V}{\partial f_n^2}, \end{bmatrix})$$

Thus, there is a solution to Equation (6) if $\frac{\partial^2 V}{\partial f_1^2} > 0, \forall i$.

REMARK 1 It can be shown that this optimization routine corresponds to constrained Newton optimization, see for example [Gill *et al.*, 1981].

EXAMPLE 2
A single CPU-system is used to control three inverted pendulums of different lengths. Every pendulum is controlled by one controller. Each control loop has an execution time C_i and a sampling frequency f_i. Let the initial sampling frequencies be f_i^0. One of the tasks operates in two modes with very different execution times. The feedback scheduler adjusts the sampling frequencies for the tasks, so that the total control performance is optimized, given the current execution times and the workload reference. The workload reference is given by the admission controller.

Figure 6 shows some plots from a simulation, where execution times, sampling frequencies and load are plotted as functions of the iteration steps. From the initial frequencies $f^0 = [4, 4.5, 5]$, an optimal solution is found after 7-8 iterations. At step 15, Task #2 sends a request to the admission controller for more execution time, due to a forthcoming mode change. The admission controller must first lower the workload reference, before any task is allowed to increase its execution time, in order to avoid overload. At step 15 the request for more CPU time from Task #2, results in the workload reference changing from 1 to 0.7. At step 20, the actual workload is sufficiently near the reference and the admission controller grants the request from Task #2, which then immediately performs a mode change. At the same time the admission controller changes the workload reference back to 1. The final frequencies are $f = [\,4.822 \quad 4.378 \quad 5.276\,]$. □

An approximate version

The feedback scheduling algorithm proposed above, includes solving both Riccati and Lyapunov equations on-line. This is expensive, and a computationally cheaper algorithm is desirable. From Figure 3 it seems that the cost functions could be approximated as quadratic functions of the sampling interval h. The relation between performance and sampling rates for LQG controllers was also discussed in [Åström, 1963], where it was shown that the cost is indeed quadratic for small sampling intervals.

Let the approximative cost $\tilde{J}(h)$ be defined as

$$\tilde{J}(h) = \alpha + \beta h^2.$$

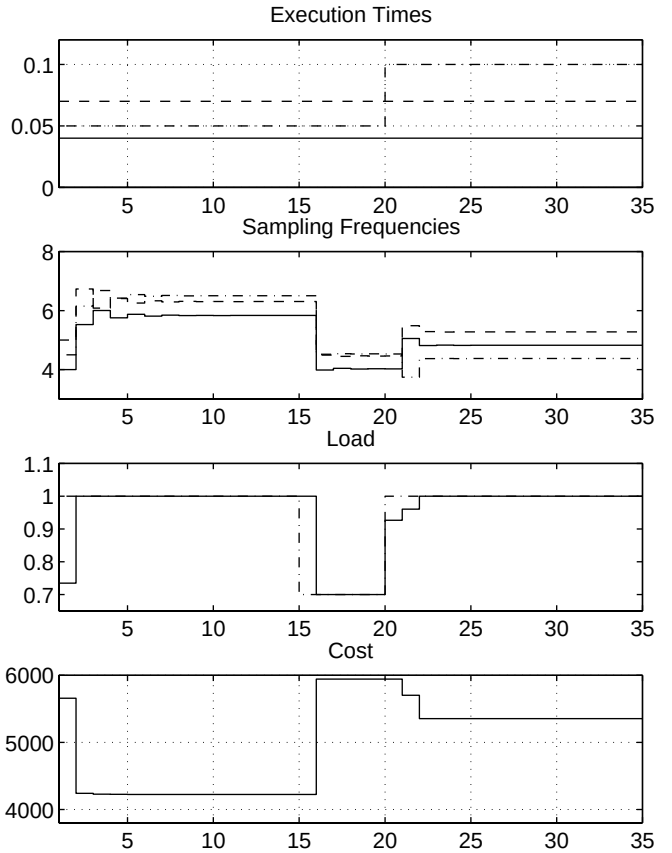


Figure 6. Plots from Example 2. The top plot shows how the execution time for task #2 doubles at step 20. The second plot from the top shows how the sampling rates are adjusted by the feedback scheduler (Task #1–full, Task #2–dash-dotted, Task#3–dashed). The load plot shows how the workload reference (dash-dotted) goes from 100% to 70% at step 15, and back to 100% at step 20. The full line is the actual workload. The bottom plot shows the sum of the cost functions, i.e. the optimization criterion.

By choosing a suitable nominal sampling interval h_0, and using $\tilde{J}(h_0) = J(h_0)$, $\tilde{J}_h(h_0) = J_h(h_0)$ the coefficients are given as:

$$\beta = J_h(h_0)/(2h_0)$$
$$\alpha = J(h_0) - \beta h_0^2$$

The following approximate cost function and derivate are then obtained:

$$\begin{aligned}\tilde{V}(f) &= \sum_{i}^{n} \alpha_i + \beta_i / f^2 \\ \tilde{V}_f(f) &= -2\,[\beta_1/f^3, \quad \ldots \quad, \beta_n/f^3\,]^T \\ V_{ff} &= 6\ \mathrm{diag}\,[\beta_1/f^4, \quad \ldots \quad, \beta_n/f^4\,]\end{aligned}$$

Using these approximate functions instead of the exact ones will give a much less computation intense problem.

The optimal solution is obtained from Equation (5), and by inserting the approximative expression for V_f we get

$$\begin{aligned}&2\beta_i/f_i^3 = \lambda C_i \;\Rightarrow\; f_i = (2\beta_i/(\lambda C_i))^{1/3} \\ &\Rightarrow U = \sum_{1}^{n} C_i f_i = \sum_{1}^{n} C_i (2\beta_i/(\lambda C_i))^{1/3}\end{aligned}$$

Now, an explicit expression for the optimal solution is given by

$$\begin{cases} \lambda = (1/U \sum_1^n C_i^{2/3} (2\beta_i)^{1/3})^3 \\ f_i = (2\beta_i/(\lambda C_i))^{1/3} \end{cases} \tag{7}$$

EXAMPLE 2 *continued*
Again, consider the system with three pendulum controllers. From the explicit analytical expressions in Equation (7) the following frequencies are calculated:

$$f = [\,4.866 \quad 4.347 \quad 5.29\,]$$

Note that these frequencies are very close to those from Example 2. This example shows that the approximative cost functions are sufficient to achieve good optimization results for some processes. □

6. Conclusion

In this paper a novel feedback scheduler has been proposed. For a class of control systems with convex cost functions, the feedback scheduler calculates the optimal resource allocation pattern. The calculation of cost functions and their dependence on sampling intervals has been investigated. Formulas for calculating the cost functions and their derivatives

have been presented. The feedback scheduler is demonstrated on a three control loops system, and the result is promising. The algorithm is, however, complex and quite computation-intense. The execution time needed for solving the optimization problem would probably exceed the execution times of most control tasks. By instead using approximative cost functions good results may be achieved at much less computational cost.

Many questions have been left out in our discussion on feedback scheduling. One thing that has been neglected is what happens during the transient phase, when a sampling rate is changed. This problem must be treated differently depending on the used scheduling algorithm, e.g EDF or RMS.

Using a feedback scheduler approach it would be possible to design real-time, plug-and-play control systems. Tasks and resources are allowed to change and the system adapts on-line.

7. References

Årzén, K.-E. (1999): "A simple event based PID controller." In *Proceedings of the 14th World Congress of IFAC*, vol. Q, pp. 423–428. IFAC, Beijing, P.R. China.

Årzén, K.-E., B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha (1999): "Integrated control and scheduling." Report ISRN LUTFD2/TFRT--7586--SE.

Åström, K. J. (1963): "On the choice of sampling rates in optimal linear systems." Technical Report RJ-243. San José Research Laboratory, IBM, San José, California.

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*, third edition. Prentice Hall.

Buttazzo, G., G. Lipari, and L. Abeni (1998): "Elastic task model for adaptive rate control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Fletcher, R. (1987): *Practical Methods of Optimization*, second edition. Wiley.

Gill, P. E., W. Murray, and M. H. Wright (1981): *Practical Optimization*. Academic Press.

Gustafsson, K. and P. Hagander (1991): "Discrete-time LQG with cross-terms in the loss function and the noise description." Report TFRT-7475. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Kosugi, N., A. Mitsuzawa, and M. Tokoro (1996): "Importance-based scheduling for predictable real-time systems using MART." In *Proceedings of the 4th Int. Workshop on Parallel and Distributed Systems*, pp. 95–100. IEEE Computer Society.

Kosugi, N. and S. Moriai (1997): "Dynamic scheduling for real-time threads by period adjustment." In *Proceedings of the World Congress on Systems Simulation*, pp. 402–406.

Kosugi, N., K. Takashio, and M. Tokoro (1994): "Modification and adjustment of real-time tasks with rate monotonic scheduling algorithm." In *Proceedings of the 2nd Workshop on Parallel and Distributed Systems*, pp. 98–103.

Nilsson, J. (1998): *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Ryu, M. and S. Hong (1998): "Toward automatic synthesis of schedulable real-time controllers." *Integrated Computer-Aided Engineering*, **5:3**, pp. 261–277.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Seto, D., J. P. Lehoczky, and L. Sha (1998): "Task period selection and schedulability in real-time systems." In *Proceedings of the 19th IEEE Real-Time Systems Symposium*.

Seto, D., J. P. Lehoczky, L. Sha, and K. G. Shin (1996): "On task schedulability in real-time control systems." In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 13–21.

Shin, K. G. and C. L. Meissner (1999): "Adaptation of control system performance by task reallocation and period modification." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 29–36.

Stankovic, J., C. Lu, and S. Son (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

van Loan, C. (1978): "Computing integral involving the matrix exponential." *IEEE Transactions on Automatic Control*, **No AC–23**, pp. 395–404.

Zhao, Q. C. and D. Z. Zheng (1999): "Stable real-time scheduling of a class of pertubed hds." In *Proceedings of the 14th World Congress of IFAC*, vol. J, pp. 91–96. IFAC, Beijing, P.R. China.

Appendix A – Proof and Continuation of Theorem 1

Proof: To find out how J depends on the sampling interval it remains to investigate S. Equation (3) is differentiated with respect to h:

$$\begin{bmatrix} 0 & 0 \\ \frac{dL}{dh} & 0 \end{bmatrix}^T \begin{bmatrix} S & 0 \\ 0 & G \end{bmatrix} \begin{bmatrix} I & 0 \\ L & I \end{bmatrix} +$$
$$\begin{bmatrix} I & 0 \\ L & I \end{bmatrix}^T \begin{bmatrix} \frac{dS}{dh} & 0 \\ 0 & \frac{dG}{dh} \end{bmatrix} \begin{bmatrix} I & 0 \\ L & I \end{bmatrix} +$$
$$\begin{bmatrix} I & 0 \\ L & I \end{bmatrix}^T \begin{bmatrix} S & 0 \\ 0 & G \end{bmatrix} \begin{bmatrix} 0 & 0 \\ \frac{dL}{dh} & 0 \end{bmatrix} =$$
$$\frac{dQ_d}{dh} + \begin{bmatrix} \frac{d\Phi^T}{dt} \\ \frac{d\Gamma^T}{dt} \end{bmatrix} S \begin{bmatrix} \Phi & \Gamma \end{bmatrix} + \begin{bmatrix} \Phi^T \\ \Gamma^T \end{bmatrix} \frac{dS}{dh} \begin{bmatrix} \Phi & \Gamma \end{bmatrix} + \begin{bmatrix} \Phi^T \\ \Gamma^T \end{bmatrix} S \begin{bmatrix} \frac{d\Phi}{dt} & \frac{d\Gamma}{dh} \end{bmatrix}$$

Rearranging the terms yields

$$\begin{bmatrix} 0 & \frac{dL^T}{dh} G \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \frac{dS}{dh} & 0 \\ 0 & \frac{dG}{dh} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ G\frac{dL}{dh} & 0 \end{bmatrix} =$$
$$\begin{bmatrix} \Phi^T - L^T\Gamma^T \\ \Gamma^T \end{bmatrix} \frac{dS}{dh} \begin{bmatrix} \Phi - \Gamma L & \Gamma \end{bmatrix} + \begin{bmatrix} I & 0 \\ -L & I \end{bmatrix}^T \bar{W} \begin{bmatrix} I & 0 \\ -L & I \end{bmatrix} \quad (8)$$

where the block matrix $\bar{W}$ is defined as

$$\bar{W} = \frac{dQ_d}{dh} + \begin{bmatrix} \frac{d\Phi^T}{dh} \\ \frac{d\Gamma^T}{dh} \end{bmatrix} S \begin{bmatrix} \Phi & \Gamma \end{bmatrix} + \begin{bmatrix} \Phi^T \\ \Gamma^T \end{bmatrix} S \begin{bmatrix} \frac{d\Phi}{dh} & \frac{d\Gamma}{dh} \end{bmatrix}$$

The following Lyapunov equations for $\frac{dS}{dh}$ and $\frac{dL}{dh}$ are obtained by extracting elements from Equation (8), and introducing $\Psi = (\Phi - \Gamma L)$.

$$\frac{dS}{dh} = \Psi^T \frac{dS}{dh} \Psi + \begin{bmatrix} I & -L^T \end{bmatrix} \bar{W} \begin{bmatrix} I \\ -L \end{bmatrix}$$
$$G\frac{dL}{dh} = \Gamma^T \frac{dS}{dh} \Psi + \begin{bmatrix} 0 & I \end{bmatrix} \bar{W} \begin{bmatrix} I \\ -L \end{bmatrix}$$

The derivative $\frac{dL}{dh}$ is needed later for the calculation of the second derivative of J. To calculate $\bar{W}$ formulas for $\frac{dQ_d}{dh}$, $\frac{d\Phi}{dt}$, and $\frac{d\Gamma}{dh}$ are needed. Since

$$\begin{bmatrix} \Phi & \Gamma \\ 0 & I \end{bmatrix} = \exp\left(\begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} h \right) = e^{\Sigma h}$$

and $\frac{Q_d}{dh}$ is given from Equation (4) it is straightforward to calculate

$$\frac{d\Phi}{dh} = A\Phi, \qquad \frac{d\Gamma}{dh} = A\Gamma + B, \qquad \frac{dQ_d}{dh} = e^{\Sigma^T h} Q_c e^{\Sigma h}$$

$\bar{W}$ can now be written as

$$\bar{W} = \begin{bmatrix} \Phi & \Gamma \\ 0 & I \end{bmatrix}^T Q_c \begin{bmatrix} \Phi & \Gamma \\ 0 & I \end{bmatrix} + \\ [\, A\Phi \quad A\Gamma + B \,]^T S\, [\, \Phi \quad \Gamma \,] + [\, \Phi \quad \Gamma \,]^T S\, [\, A\Phi \quad A\Gamma + B \,]$$

and now let W be

$$W = [\, I \quad -L^T \,] \bar{W} \begin{bmatrix} I \\ -L \end{bmatrix} = \begin{bmatrix} \Psi \\ -L \end{bmatrix}^T Q_c \begin{bmatrix} \Psi \\ -L \end{bmatrix} + \\ \{\Psi^T A^T - L^T B^T\} S\Psi + \Psi^T S\{A\Psi - BL\}.$$

which now leads to the following expression, and $\frac{dJ}{dh}$ may now be calculated.

$$\frac{dS}{dh} = \Psi^T \frac{dS}{dh}\Psi + W$$
$$G\frac{dL}{dh} = \Gamma^T \frac{dS}{dh}\Psi + [\, \Gamma^T \quad I \,] Q_c \begin{bmatrix} \Psi \\ -L \end{bmatrix} + \Gamma^T S(A\Psi - BL) + (A\Gamma + B)^T S\Psi$$

THEOREM 2
The second derivative is given by

$$\frac{d^2J}{dh^2} = \frac{1}{h}\Big\{tr(\frac{d^2S}{dh^2}R_1) + 2tr(\frac{dS}{dh}\frac{R_1}{dh}) + tr(S\frac{d^2R_1}{dh^2}) + \frac{d^2\bar{J}}{dh^2}\Big\} \\ - \frac{2}{h^2}\Big\{tr(\frac{dS}{dh}R_1) + tr(S\frac{R_1}{dh}) + \frac{d\bar{J}}{dh}\Big\} + \frac{2}{h^3}\Big\{tr(SR_1) + \bar{J}\Big\}$$

where

$$\frac{d^2S}{dh^2} = \Psi^T \frac{d^2S}{dh^2}\Psi + W_2, \qquad W_2 = \frac{d\Psi^T}{dh}\frac{dS}{dh}\Psi + \Psi^T \frac{dS}{dh}\frac{d\Psi}{dh} + \frac{dW}{dh}$$
$$\frac{d\Psi}{dh} = A\Phi - (A\Gamma + B)L - \Gamma\frac{dL}{dh}$$

$$\frac{dW}{dh} = \begin{bmatrix} \frac{d\Psi}{dh} \\ -\frac{dL}{dh} \end{bmatrix}^T Q_c \begin{bmatrix} \Psi \\ -L \end{bmatrix} + \begin{bmatrix} \Psi \\ -L \end{bmatrix}^T Q_c \begin{bmatrix} \frac{d\Psi}{dh} \\ -\frac{dL}{dh} \end{bmatrix} +$$
$$\{\Psi^T A^T - L^T B^T\}(\frac{dS}{dh}\Psi + S\frac{d\Psi}{dh}) +$$
$$(\Psi^T \frac{dS}{dh} + \frac{d\Psi^T}{dh} S)\{A\Psi - BL\} +$$
$$\{\frac{d\Psi^T}{dh}A^T - \frac{dL^T}{dh}B^T\}S\Psi + \Psi^T S\{A\frac{d\Psi}{dh} - B\frac{dL}{dh}\}$$

$$\frac{d^2\bar{J}}{dh^2} = tr(Q_{1c}\frac{dR_1}{dh}), \qquad \frac{d^2R_1}{dh^2} = A\Phi R_{1c}\Phi^T + \Phi R_{1c}A^T\Phi^T$$

□

Proof: Proven with similar techniques as Theorem 1.

LEMMA 1
The integral

$$\int_0^t e^{A\tau} Q e^{A^T\tau} d\tau$$

is calculated using the following equations taken from [van Loan, 1978]. Consider the linear system

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -A^T & Q \\ 0 & A \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \tag{9}$$

which has the following solution

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \exp(\begin{bmatrix} -A & Q \\ 0 & A^T \end{bmatrix} t) \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix}$$

If

$$\begin{bmatrix} \phi_{11} & \phi_{12} \\ \phi_{21} & \phi_{22} \end{bmatrix} = \exp(\begin{bmatrix} -A & Q \\ 0 & A^T \end{bmatrix} t)$$

then

$$x_1(t) = \Psi_{11}x_1(0) + \Psi_{12}x_2(0)$$
$$x_2(t) = \Psi_{21}x_1(0) + \Psi_{22}x_2(0)$$

An alternative way of solving Equation (9) is

$$\begin{aligned} & x_2(t) = e^{A^T t} x_2(0) \quad \Rightarrow \dot{x}_1(t) = -A x_1(t) + Q e^{A^T t} x_2(0) \\ \Rightarrow \quad & x_1(t) = e^{-At} x_1(0) + e^{-At} \int_0^t e^{A\tau} Q A^T \tau \, d\tau \, x_2(0) \end{aligned}$$

Identification of the terms from the different solutions now gives

$$\int_0^t e^{A\tau} Q e^{A^T \tau} d\tau = \Psi_{22}^T \Psi_{12},$$

which concludes the lemma. □

Paper 5

A Matlab Toolbox for Real-Time and Control Systems Co-Design

Johan Eker and Anton Cervin

Abstract

The paper presents a Matlab toolbox for simulation of real-time control systems. The basic idea is to simulate a real-time kernel in parallel with continuous plant dynamics. The toolbox allows the user to explore the timely behavior of control algorithms, and to study the interaction between the control tasks and the scheduler. From a research perspective, it also becomes possible to experiment with more flexible approaches to real-time control systems, such as feedback scheduling. The importance of a more unified approach for the design of real-time control systems is discussed. The implementation is described in some detail and a number of examples are given.

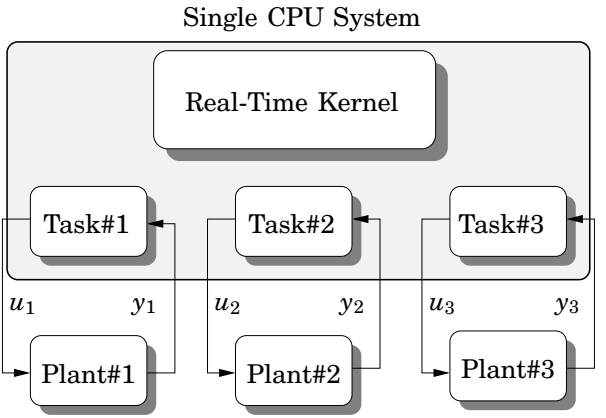


Figure 1. Several control loops execute concurrently on one CPU. The interaction between the control tasks will affect the control performance.

1. Introduction

Real-time control systems are traditionally designed jointly by two different types of engineers. The control engineer develops a model for the plant to be controlled, designs a control law and tests it in simulation. The real-time systems engineer is given a control algorithm to implement, and configures the real-time system by assigning priorities, deadlines, etc.

The real-time systems engineer usually regards control systems as hard real-time systems, i.e. deadlines should never be missed. The control engineer on the other hand expects the computing platform to be predictive and support equidistant sampling. In reality none of the assumptions are necessarily true. This is even more obvious in the case where several control loops are running on the same hardware unit. The controllers will interact with each other since they are sharing resources such as CPU, network, analog/digital converters, etc. see Figure 1.

A new interdisciplinary approach is currently emerging where control and real-time issues are discussed at all design levels. One of the first papers that dealt with co-design of control and real-time systems was [Seto *et al.*, 1996], where the sampling rates for a set of controllers sharing the same CPU are calculated using standard control performance metrics. Control and scheduling co-design is also found in [Ryu *et al.*, 1997], where the control performance is specified in terms of steady state error, overshoot, rise time, and settling time. These performance parameters are expressed as functions of the sampling period and the input-output

latency. A heuristic iterative algorithm is proposed for the optimization of these parameters subject to schedulability constraints.

Good interaction between control theory and real-time systems theory opens up for a unified approach and more integrated algorithms. Scheduling parameters could for example be adjusted automatically on-line by a kernel supervisor. Such a setup would allow much more flexible real-time control systems than those available today. Ideas on adaption of scheduling parameters are for example found in [Abdelzaher *et al.*, 1997] and [Stankovic *et al.*, 1999].

The development of algorithms for co-design of control and real-time systems requires new theory and new tools. This paper presents a novel simulation environment for co-design of control systems and real-time systems within the Matlab/Simulink environment. The advantages of using Matlab for this purpose are many. Matlab/Simulink is commonly used by control engineers to model physical plants, to design control systems, and to evaluate their performance by simulations. A missing piece in the simulations, however, has been the actual execution of the controllers when implemented as tasks in a real-time system. On the other hand, most of the existing tools for task simulations, for instance STRESS [Audsley *et al.*, 1994], DRTSS [Storch and Liu, 1996], and the simulator in [Ancilotti *et al.*, 1998], give no support for the simulation of continuous dynamics. Not much work has previously been done on mixed simulations of both process dynamics, control tasks, and the underlying real-time kernel. An exception is [Liu, 1998], where a single control task and a continuous plant was simulated within the Ptolemy II framework.

The simulator proposed in this paper is designed for simultaneous simulation of continuous plant dynamics, real-time tasks, and network traffic, in order to study the effects of the task interaction on the control performance.

2. The Basic Idea

The interaction between control tasks executing on the same CPU is usually neglected by the control engineer. It is however the case that having a set of control tasks competing for the computing resources will lead to various amounts of delay and jitter for different tasks. Figure 2 shows an example where three control tasks with the same execution times but different periods are scheduled using rate-monotonic priorities. In this case the schedule does not tell the whole story. In the example, the actual control delay (the delay from reading the input signal until writing a new output signal) for the low priority task varies from one to three times the execution time. Intuitively, this delay will affect the control performance,

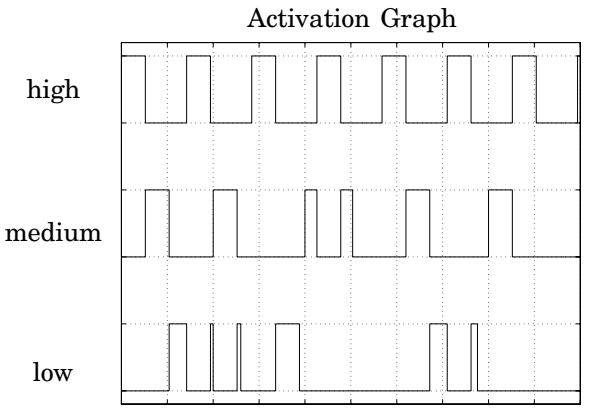


Figure 2. The activation graph for three control tasks, with fixed priorities (high, medium, low), running in a pre-emptive kernel. The execution times are the same for all three processes.

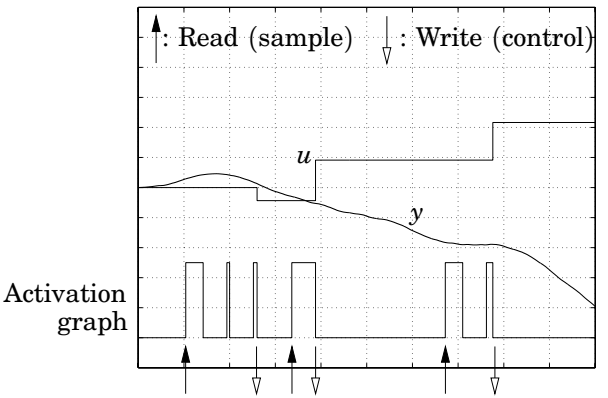


Figure 3. This is how the low priority task from Figure 2 interacts with its plant. (u is the control signal, y is the measurement signal.)

but how much, and how can we investigate this?

To study how the execution of tasks affects the control performance we must simulate the whole system, i.e. both the continuous dynamics of the controlled plant and the execution of the controllers in the CPU.

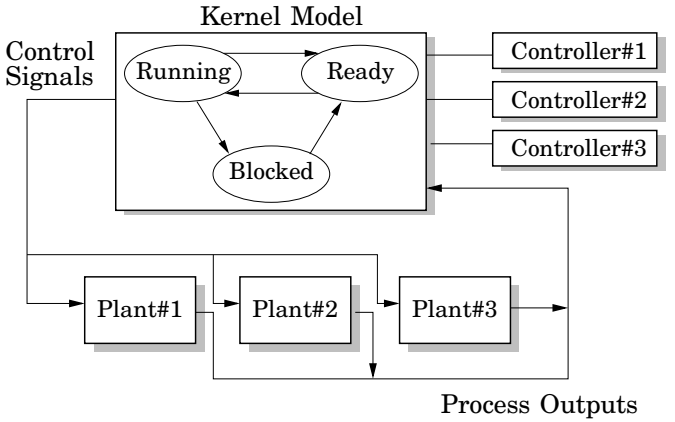


Figure 4. Schematic view of the simulation setup. The controllers are tasks executing in a simulated pre-emptive kernel. The controllers and the control signals are discrete while the plant dynamics and the plant output are continuous. The continuous signals from the plants are sampled by the control tasks.

We need not simulate the execution of the controller code on instruction level. In fact, it is enough to model the timely aspects of the code that are of relevance to other tasks and to the controlled plant. This includes computational phases, input and output actions, and blocking of common resources (other than the CPU).

Figure 3 shows the activation graph for the low priority task from Figure 2 and how it interacts with the continuous plant. The controller samples the continuous measurement signal from the plant (y) and writes new control outputs (u).

Figure 4 provides a schematic view of how we simulate the system. A model of a real-time kernel handles the scheduling of the control tasks and is also responsible for properly interfacing the tasks with the physical environment. The outputs from the kernel model, i.e. the control signals, are piecewise constant. The plant dynamics and the plant outputs, i.e. the measurement signals, are continuous.

3. The Simulation Model

The heart of the toolbox is a Simulink block (an S-function) that simulates a tick-driven preemptive real-time kernel. The kernel maintains a number of data structures that are commonly found in a real-time kernel: a set of task records, a ready queue, a time queue, etc. At each clock

tick, the kernel is responsible for letting the highest-priority ready task, i.e. the running task, execute in a virtual CPU. The scheduling policy used is determined by a priority function, which is a function of the attributes of a task. For instance, a priority function that returns the period of a task implements rate-monotonic scheduling, while a function that returns the absolute deadline of a task implements earliest-deadline-first scheduling. There currently exist predefined priority functions for rate-monotonic (RM), deadline-monotonic (DM), arbitrary fixed-priority (FP), and earliest-deadline-first (EDF) scheduling. The user may also write his own priority function that implements an arbitrary scheduling policy.

The execution model used is similar to the *live task model* described in [Storch and Liu, 1996]. During a simulation, the kernel executes user-defined code, i.e. Matlab functions, that have been associated with the different tasks. A code function returns an execution time estimate, and the task is not allowed to resume execution until the same amount of time has been consumed by the task in the virtual CPU.

The Task

Each task in the kernel has a set of basic attributes: A name, a list of code segments to execute, a period, a release time, a relative deadline, and the remaining execution time to be consumed in the virtual CPU. Some of the attributes, such as the release time and the remaining execution time, are constantly updated by the kernel during a simulation. The other attributes, such as the period and the relative deadline, remain constant unless they are explicitly changed by kernel function calls from the user code.

The Code

The local memory of a task is represented by two local, user-defined data structures `states` and `parameters`. The states may be changed by the user code, while the parameters remain constant throughout the execution.

To capture the timely behavior of a task, the associated code is divided into one or several code segments, see Figure 5. The execution time of a segment is determined dynamically at its invocation. Normally, the segments are executed in order, but this may be changed by kernel function calls from the user code.

On a finer level, actual execution of statements in a code segment can only occur at two points: at the very beginning of the code segment (in the `enterCode` part) or at the very end of the code segment (in the `exitCode` part), see Figure 6. Typically, reading of input signals, locking of resources, and calculations are performed in the `enterCode` part. Writing of output signals, unlocking of resources, and other kernel function calls

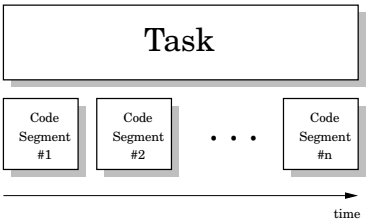


Figure 5. The execution structure of a task. The flexible structure supports data dependent execution times and advanced scheduling techniques.

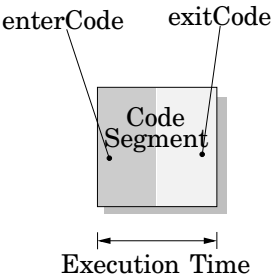
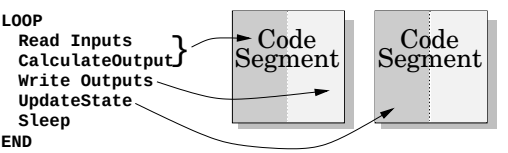


Figure 6. A code segment is divided in two parts: the `enterCode` part and the `exitCode` part.

are typically performed in the `exitCode` part. The following examples illustrate how code segments can model real-time tasks.

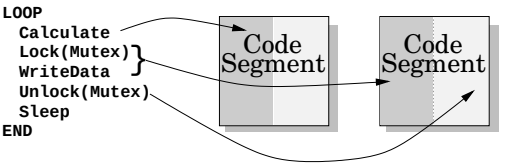
EXAMPLE 1
A task implementing a control loop can often be divided into two parts: one that calculates a new control signal and one that updates the controller states. The first part, called Calculate Output, has a hard timing constraint and should finish as fast as possible. The timing requirement for the second part, Update State, is that it must finish before the next invocation of the task. Two code segments are appropriate to model the task:



The `enterCode` of the first segment contains the reading of the measurement signals, and the calculation of a new control signal. In the same seg-

ment, in `exitCode`, the control signal is sent to the actuator. The control delay of the controller is thus equal to the execution time of the first segment. The `enterCode` of the second code segment contains Update State. When the last segment has completed, the task is suspended until the next period by the kernel. □

EXAMPLE 2
The structure of a periodic task that first calculates some data and then writes to a common resource could look like this:



Again, two code segments can capture the timely behavior. The first code segment contains the `Calculate` statement, located in the `enterCode` part. The `enterCode` part of the second code segment contains the `Lock(Mutex)` and `WriteData` statements, while `exitCode` contains the `Unlock(Mutex)` statement. When the last segment has completed, the task is suspended until the next period by the kernel. □

4. Using the Simulator

From the user's perspective, the toolbox offers a Simulink block that models a computer with a real-time kernel. Connecting the Computer block's inputs and outputs (representing for instance A-D and D-A converters) to the plant, a complete computer-controlled system is formed, see Figure 7.

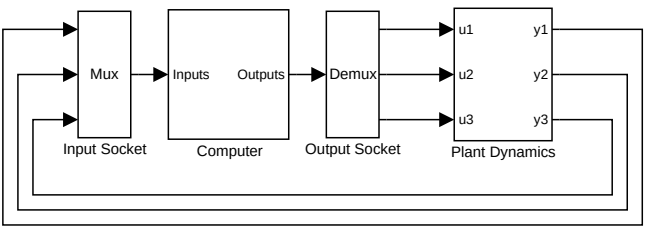


Figure 7. The simulation environment offers a Simulink Computer block that can be connected to the model of the plant dynamics.

The plant dynamics may have to be controlled by several digital controllers, each implemented as a periodic control task in the computer. Besides the controllers, other tasks could be executing in the computer, for instance planning tasks, supervision tasks, and user communication tasks.

Opening up the Computer block, the user may study detailed information about the execution of the different tasks, see Figure 8. It is for in-

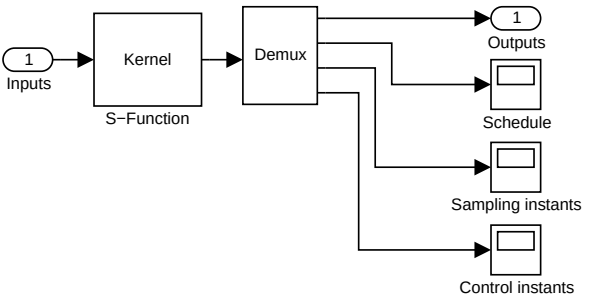


Figure 8. Inside the Computer block, it is possible to study details about the execution of different tasks.

stance possible to study the schedule, i.e. a plot that shows when different tasks are executing, at run-time. Further statistics about the execution is stored in the workspace and may be analyzed when the simulation has stopped.

Controller Implementation

It is highly desirable that the design of the kernel is flexible and allows components to be reused and replaced. Much effort has been put into writing control algorithms in Matlab, and these algorithms should be straightforward to reuse. In the toolbox, a control algorithm can be implemented as a code segment with the following format:

```
function [exectime,states] = ...
      myController(flag,states,params)
 switch flag,
  case 1, % enterCode
    y = analogIn(params.inChan);
    states.u = <place control law here>
    exectime = 0.002;
   case 2, % exitCode
    analogOut(params.outChan,states.u)
 end
```

The input variables to `myController` are the state variables `states`, and the controller parameters `params`. The `flag` is used to indicate whether the `enterCode` or the `exitCode` part should be executed. If the `enterCode` part is executed, the function returns the execution time estimate `exectime` and the new state variables. The control signal is sent to the plant in the `exitCode` part.

REMARK 1 The output signal u is not normally regarded as a state variable in a controller. In this example, however, we need to store the value of u between two invocations of the `myController` function.

Configuration

Before a simulation can start, the user must define what tasks that should exist in the system, what scheduling policy should be used, whether any common resources exist, etc. The initialization is performed in a Matlab script.

EXAMPLE 3
Three dummy tasks are initialized in the script below. The tick-size of the kernel is set to 0.001 s and the scheduling type is set to rate monotonic. The dummy code segment `empty` models a task that computes nothing for a certain amount a time. Each task is assigned a period, and a deadline which is equal to the period.

```
function rtsys = rtsys_init

% 1 = RM, 2 = DM, 3 = Arbitrary FP, 4 = EDF
rtsys.st = 1;
rtsys.tick_size = 0.001;

T = [0.10 0.08 0.06];  % Task Periods
D = [0.10 0.08 0.06];  % Deadlines
C = [0.02 0.02 0.02];  % Computation times

rtsys.Tasks = {}
code1 = code('empty',[],C(1))
code2 = code('empty',[],C(2))
code3 = code('empty',[],C(3))

rtsys.Tasks{1}=task('Task1',code1,T(1),D(1));
rtsys.Tasks{2}=task('Task2',code2,T(2),D(2));
rtsys.Tasks{3}=task('Task3',code3,T(3),D(3));
```

The initialization script is given as a parameter to a Computer block in

a Simulink model. Simulating the model for one second produces, among other things, the schedule plot shown in Figure 9. □

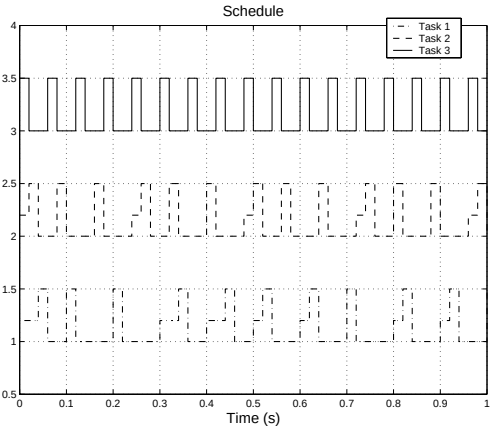


Figure 9. The schedule resulting from the simulation in Example 3. The bottom graph shows when Task 1 is running (high), ready (medium) or blocked (low). The other two graphs represent Task 2 and Task 3.

Connecting a Continuous Plant

Figure 10 shows a Simulink diagram where a Computer block is connected to three pendulum models. The continuous plant models are described by other Simulink blocks.

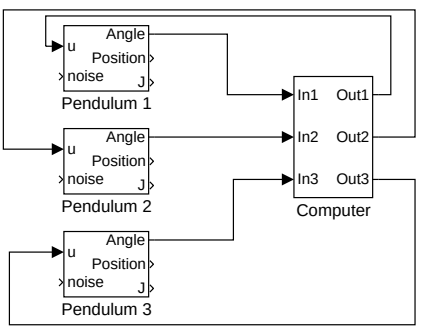


Figure 10. A Simulink diagram where three continuous pendulum models are connected with the real-time kernel. The simulation result from this system is both the activation graph and the output from the continuous plants.

A real-time system with three control loops are created in Example 4. One code segment named `myController` is associated with each task.

EXAMPLE 4

```
function rtsys = rtsys_init
% Scheduling type, 1=RM, 2=DM, 3=FP, 4=EDF
rtsys.st=1;
rtsys.tick_size=0.001;

% Desired bandwidths
omega=[3 5 7];
% Sampling periods
T=[0.167 0.100 0.071];
for i=1:3
  % Design controller
  params=ctrl_design(omega(i),T(i));
  % Initialize control code
  states.xhat=[0 0]';
  % The controller reads from input i
  params.inChan=i;
  % The controller writes to output i
  params.outChan=i;
  sfbcode=code('myController',states,params);
  % Create task
  tasks{i}=task(['Task 'num2str(i)],...
                sfbcode, T(i), T(i));
end
rtsys.tasks=tasks;
```

□

The outputs from a simulation of this system are a set of continuous signals from the plants together with an activation graph. It is hence possible to evaluate the performance of the real-time systems both from a control design point of view and from a scheduling point of view.

5. A Co-Design Example

Using the simulator, it is possible to evaluate different scheduling policies and their effect on the control performance. Again consider the problem of controlling three inverted pendulums using only one CPU, see Figure 11. The inverted pendulum may be approximated by the following linear differential equation

$$\ddot{\theta} = \omega_0^2 \theta + \omega_0^2 u/g,$$

where $\omega_0 = \sqrt{g/l}$ is the natural frequency for a pendulum with length l. The goal is to minimize the angles, so for each pendulum we want to minimize the accumulated quadratic loss function

$$J_i(t) = \int_0^t \theta_i^2(s)ds. \tag{1}$$

Three discrete-time controllers with state feedback and observers are designed. Sampling periods for the controllers are chosen according to the desired bandwidths (3, 5 and 7 rad/s respectively) and the CPU resources available. The execution times of the control tasks, τ_i, are all 28 ms, and the periods are $T_1 = 167$ ms, $T_2 = 100$ ms, and $T_3 = 71$ ms. Task objects

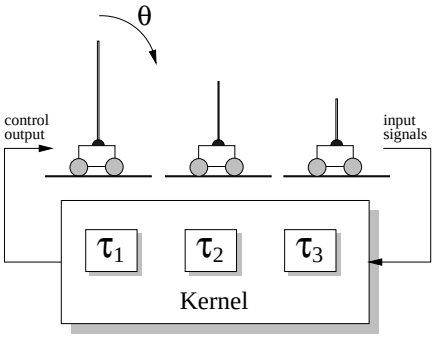


Figure 11. The setup from described in Section 5. Three inverted pendulums with different lengths are controlled by three control tasks running on the same CPU.

are created according to Example 4. Also, similar to Example 1, the control algorithm is divided into two code segments, Calculate Output and Update State, with execution times of 10 and 18 ms respectively.

In a first simulation, the control tasks are assigned constant priorities according to the rate-monotonic schema, and the two code segments execute at the same priority. In a second simulation, the Calculate Output code segments are assigned higher priorities than the Update State segments, according to iterative priority/deadline assignment algorithm suggested in [Cervin, 1999]. The accumulated loss function for the slow pendulum ($T_1 = 167$ ms) is easily recorded in the Simulink model, and the results from both simulations are shown in Figure 12.

A close-up inspection of the schedule produced in the second simulation is shown in Figure 13. It can be seen that the faster tasks sometimes allow the slower tasks to execute, and in this way the control delays in the slower controllers are minimized. The result is a smaller accumulated loss, and thus, better control performance.

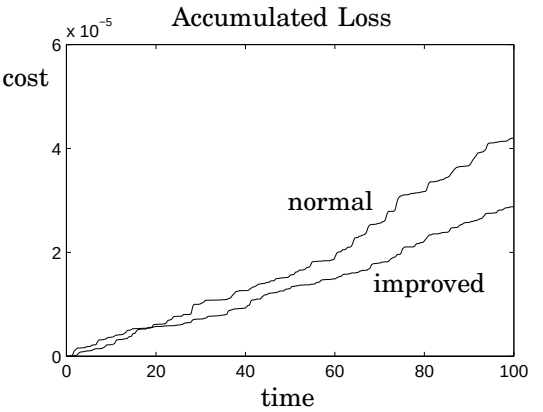


Figure 12. The accumulated loss, see Equation (1), for the low priority controller using normal and improved scheduling. The cost is substantially reduced under the improved scheduling.

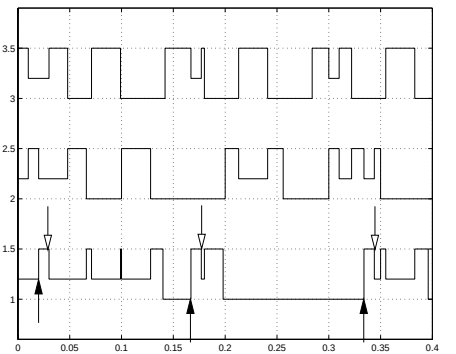


Figure 13. The activation graph when the improved scheduling strategy is used. Note that the control delay for the low priority task is approximately the same as for the other tasks.

6. Simulation Features

Further features of the toolbox are the support for common real-time primitives like mutual exclusion locks, events (also known as condition variables), and network communication blocks.

Locks and Events

The control tasks do not only interact with each through the use of same CPU, but also due to sharing other user-defined resources. The kernel allows the user to define monitors and events, for implementing complex dependencies between the task. The syntax and semantics of the `mutex` and `event` primitives are demonstrated by a small example. Two tasks `Regul` and `OpCom` are sharing a variable called `data`. To ensure mutual exclusion the variable is protected by the mutex variable `M1`. Associated with `M1` is a monitor event called `E1`. The `Regul`-task consists of two code segments called `rseg1` and `rseg2`, that are shown in Example 5. Each time the `Regul`-task is released it tries to lock the monitor variable `M1`. Once the monitor is locked it may access the shared data. If the value of the `data`-variable is less than two, it waits for the event `E1` to occur.

EXAMPLE 5

```
function [exectime, states] = ...
            rseg1(flag,states,params)
 switch flag,
  case 1,   % enterCode
    if lock('M1')
      data = readData('M1');
      if data < 2
        await('E1');
        exectime = 0;
      else
        exectime = 0.003;
      end
    else
      exectime = 0;
    end
  case 2,   % exitCode
    unlock('M1');
 end

function [exectime,states] = ...
            rseg2(flag,states,params)
 switch flag,
  case 1,   % enterCode
    y = analogIn(params.inChan);
    states.u = -50*y;
    exectime = 0.003;
  case 2,   % exitCode
    analogOut(params.outChan,states.u)
 end
```

□

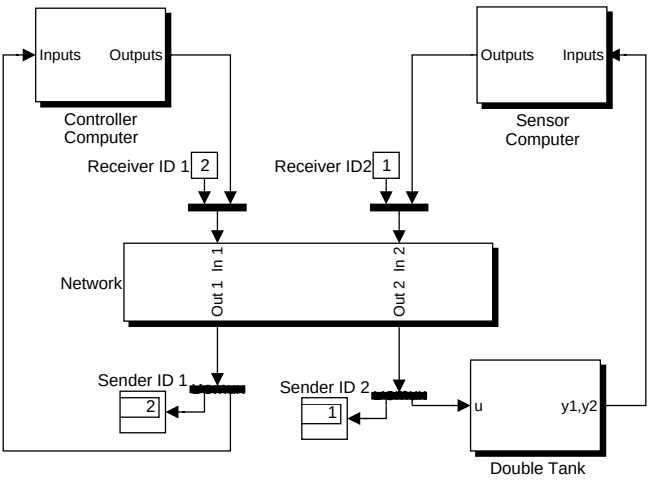


Figure 14. A distributed control system where the sensor and the CPU are dislocated. The controller and the sensor are implemented as periodic tasks running on separate CPUs.

The locks and the events are designed similarly to how monitors and events are implemented in a standard real-time kernel, i.e. using queues associated with the monitor for storing tasks blocking on locks or events. The execution time used for trying, but failing to lock, is in the example above zero.

Network Blocks

It is possible to include more than one Computer block in a Simulink model, and this opens up the possibility to simulate much more complex systems than the ones previously discussed. Distributed control systems may be investigated. Furthermore, fault-tolerant systems, where, for redundancy, several computers are used for control, could also be simulated. In order to simulate different communication protocols in such systems, communication blocks for sending data between the different Computer blocks are needed. Figure 14 shows a simulation setup for a simple distributed system where the controller, and the actuator and sensor, are located at different places. Besides the kernel blocks there is a network block for communication. The network block is event driven, and each time any of the input signals change, the network is notified. The user needs to implement the network protocol, since the blocks simply provides the mechanisms for sending data between kernels.

High Level Task Communication

One of the main reasons for designing the kernel and the network blocks was to facilitate the simulation of flexible embedded control system, i.e. systems where the task set is allowed to change dynamically and the underlying real-time system must compensate for this. From a control theory perspective we might say that we want to design a feedback connection between the control tasks and the scheduler, see Figure 15. To support the

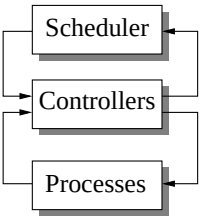


Figure 15. The control tasks and the task scheduler are connected in a feedback loop.

simulation of feedback scheduling, there must be ways for the tasks and the task scheduler to communicate. Therefore the kernel also supports system-level message passing between tasks.

7. Conclusions

This paper presented a novel simulator for the co-design of real-time systems and control systems. The main objective is to investigate the consequences on control performance of task interaction on kernel level. This way, scheduling algorithms may be evaluated from a control design perspective. We believe that this is an issue of increasing importance. There are many more things to be implemented and improved before this block set will become a truly useful tool. Currently the kernel is tick-based, and has little support for external interrupts. The next version of the kernel block will probably be event-based in order to better support interrupts and event-based sampling. To make the simulations more realistic, the scheduler itself could also be modeled as a task that consumes CPU time. This would also enhance the possibilities for the user to implement new scheduling strategies for control tasks.

8. Acknowledgments

This work was sponsored by the Swedish national board of technical development (NUTEK) and by the Swedish network for real-time research and education (ARTES).

9. References

Abdelzaher, T., E. Atkins, and K. Shin (1997): "QoS negotiation in real-time systems, and its application to flight control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Ancilotti, P., G. Buttazzo, M. D. Natale, and M. Spuri (1998): "Design and programming tools for time critical applications." *Real-Time Systems*, **14:3**, pp. 251–269.

Audsley, N., A. Burns, M. Richardson, and A. Wellings (1994): "STRESS—A simulator for hard real-time systems." *Software—Practice and Experience*, **24:6**, pp. 543–564.

Cervin, A. (1999): "Improved scheduling of control tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 4–10.

Liu, J. (1998): "Continuous time and mixed-signal simulation in Ptolemy II." Technical Report UCB/ERL Memorandum M98/74. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Seto, D., J. Lehoczky, L. Sha, and K. Shin (1996): "On task schedulability in real-time control systems." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Stankovic, J., C. Lu, and S. Son (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Storch, M. F. and J. W.-S. Liu (1996): "DRTSS: A simulation framework for complex real-time systems." In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 160–169.

Paper 6

Design and Implementation of a Hybrid Control Strategy

Johan Eker and Jörgen Malmborg

Abstract

A hybrid controller for set-point changes is presented. The strategy combines a time-optimal controller with a PID-controller to achieve both good set-point response and steady-state behavior. The control strategy is designed and implemented both for a double tank lab process and a real-life heating/ventilation system. Both simulations and experiments are presented.

1. Introduction

Traditionally there is a trade-off in design objectives when choosing controller parameters. It is usually hard to achieve the desired step change response and at the same time get the wanted steady-state behavior. An example of contradictory design criteria is tuning a PID controller to achieve both fast response to set-point changes, fast disturbance rejection, and no or little overshoot. In process control it is common practice to use PI control for steady state regulation and to use manual control for large set-point changes. Figure 1 shows a system that is controlled (left) by a PID controller and (right) with manual control combined with a PID controller. The simulations show a step response at time zero and a load disturbance rejection at time 80. This paper describes a hybrid control structure which combines a steady state PID controller with a minimum time controller for the set-point changes. Both good response to set-point changes and good disturbance rejection are achieved.

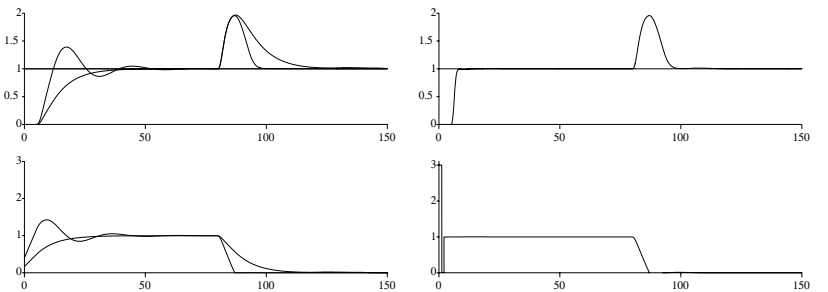


Figure 1. In the design of a PID control system there is often a choice between a fast controller giving a large overshoot or a slow controller without overshoot. In the left figure this is illustrated by two examples of PID control. The right figure shows a PID controller combined with manual control. The top figures shows the set-point y_{sp} and the measured variable y. The bottom figures shows the control signal u.

Design and implementation of such a hybrid control strategy for two different plants are presented. The ideas were first tested in the laboratory on a double tank system, and later applied to a heating/ventilation system in a public school. The experiments were performed in two different software environments but with the same basic control algorithm.

The main focus of this work has been to apply optimal control theory and hybrid control theory to realistic processes using as simple methods as possible. This means taking hardware and software limitations into account when designing the control algorithm. In practice we have been forced to simplify and approximate the control strategy. A positive aspect

of this has been that we have found a fairly easy way of synthesizing our controller. This in turn makes the control strategy much more accessible for industrial use. A Lyapunov based strategy is first investigated theoretically and simulated. A simplified version of this controller is then implemented for two different processes.

This paper is based on work found in [Malmborg, 1998], [Eker, 1997], and [Malmborg and Eker, 1997].

2. The Controller Structure

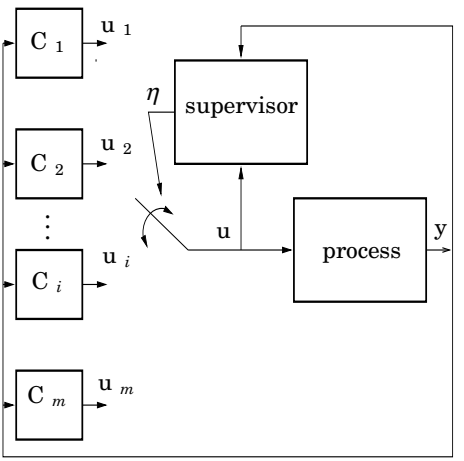


Figure 2. A multi controller architecture, used in [Morse, 1995]. Several controllers C_i execute in parallel and a supervisor is used to select which control signal $u = u_\eta$ to be fed to the process.

Using a hybrid control scheme it is possible to combine several control algorithms and thus get a controller that consists of several subcontrollers, each designed for a special purpose. An example of a multi control architecture is shown in Figure 2. The basic idea here is that the process outputs y are fed back to a set of controllers. Then each controller calculates a candidate control signal. Which control signal is finally used is decided by the supervisor. For an overview of hybrid systems see [Branicky, 1995] [Antsaklis and Nerode, 1998] [Antsaklis *et al.*, 1998] [Branicky *et al.*, 1998].

A Two Mode Hybrid Controller

A controller structure with two sub-controllers and a supervisory switching scheme 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 3 describes the algorithm with a Grafcet diagram, see [David and Alla, 1992]. Grafcet, also known as Sequential Function Charts (SFC), is a graphical language for implementation and specification of sequential algorithms. A Grafcet consists of *steps* and *transitions*. A step corresponds to a state and can be *active* or *inactive*. Each state may be associated with an action which is executed when the step is active. A transition may fire if its condition is true and the the preceding step is active. The Grafcet diagram for our hybrid controller consists of four states. Initially no controller is in use. 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 signal **Close** tells if the controller should use the PID controller or

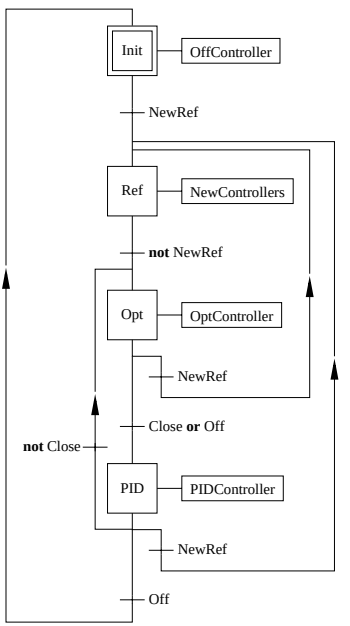


Figure 3. A Grafcet diagram describing the control algorithm.

the time optimal controller. Two different ways to calculate **Close** is given in the section on Lyapunov function modifications.

3. 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

$$\begin{aligned} \dot{x} &= f(x,t,u_i) \\ u_i &= c_i(x,t), \\ i(t) &\in \{1,\ldots,n\} \end{aligned} \tag{1}$$

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, see for example [Johansson and Rantzer, 1998] [Ferron, 1996]. For the applications in this paper the min-switch strategy described below is used.

The min-switching strategy is defined as follows. Let there be a Lyapunov function V_i associated with each controller c_i in Equation 1. Furthermore let the controller c_i be admissible only in the region Ω_i. The idea is then to use the controller corresponding to the smallest value of the Lyapunov function. More formally:

DEFINITION 1—THE MIN-SWITCHING STRATEGY
Let $f_i(x,t)$ be the right-hand side of Equation 1 when control law c_i is used. Use a control signal u^* such that,

$$\dot{x} = \sum_{i=1}^{n} \alpha_i f_i(x,t). \tag{2}$$

where $\alpha_i \geq 0$ satisfies $\sum \alpha_i = 1$, and $\alpha_i = 0$ if either $x \notin \Omega_i$ or if $V_i(x,t) > \min_j[V_j(x,t)]$. If only one controller achieves the minimum then $\alpha_i = 1$ for that controller and all the other α_i are zero. □

THEOREM 1—STABILITY OF HYBRID SYSTEMS

Let the system be given by Equation (1). Introduce W as

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

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

For a proof of the theorem and a more detailed description of the strategy, see [Malmborg, 1998]. The case where several Lyapunov functions achieve the minimum is discussed in [Malmborg, 1998].

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, $\tilde{V}_i = g_i(V_i)$, where $g_i(\cdot)$ are monotonously increasing functions.

In some cases there can be very fast switching, known as chattering, between two or more controllers having the same value of their respective Lyapunov function. The hybrid controller is still stabilizing but it may not lead to the desired behavior in a practical implementation. One way to avoid chattering 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. For two controllers with Lyapunov functions V_1 and V_2 the equations are $V_1 = V_1 + \Delta$ and $V_2 = V_2$ if controller two is in use and $V_1 = V_1$ and $V_2 = V_2 + \Delta$ if controller one is controlling the process. This guarantees that a controller is used for a time period $t > 0$ before it is switched out. It is easily shown that the hybrid controller is globally stabilizing with this addition, if it is stabilizing without it. More information on chattering in hybrid systems can be found in [Hay and Griffin, 1979] [Park and Barton, 1996].

4. The Processes

The first process is a double tank system, a standard laboratory process. The second process is a real-life process, a heating/ventilation system at a public school.

A Double Tank System

The first process is the double tank system from our laboratory. It consists of two water tanks in series, see Figure 4. The goal is to control the level, x_2, of the lower tank and indirectly the level, x_1, of the upper tank. The

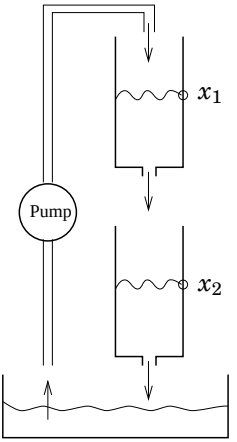


Figure 4. The double-tank process. The input signal u is the voltage to the pump and the output signal $y = x_2$ is the level of the lower tank.

two tank levels are both measurable. The following state space description is derived

$$\dot{x} = f(x,u) = \begin{bmatrix} -\alpha_1\sqrt{x_1} + \beta u \\ \alpha_1\sqrt{x_1} - \alpha_2\sqrt{x_2} \end{bmatrix}, \tag{3}$$

where the flow u into tank 1 is the control variable. The inflow can never be negative and the maximum flow is $\bar{u} = 27 \cdot 10^{-6}\ m^3/s$. Furthermore, in this experimental setting the tank areas and the outflow areas are the same for both tanks, giving $\alpha_1 = \alpha_2$. The square root expression is derived from Bernoulli's energy equations.

A heating/ventilation process

The second process is a heating/ventilation system in a public school in Klippan, Sweden. The process, see Figure 5, is a combined heating and ventilation system where pre-heated air is blown into the classrooms. The control problem here is to design a controller that has both the desired steady-state behavior and transient behavior. The participating company was interested in new ways of improving start-up and reference change performances.

Specifically, the control problem is that they want to have two different settings for the ventilation air temperature, one during the day and one during the night. Another related problem is that when the system has been shut down for some reason it is necessary to start from considerably lower temperatures. The air is heated in a heat-exchanger and the incoming air has outdoor temperature, which can be very low in Sweden during

Figure 5. The control valve for the heating/ventilation system.

the winter. The variation in outdoor temperature will cause considerable variation in the process parameters.

The existing PID controller was tuned very conservatively to be able to handle these cold starts without too large overshoots. This was exactly the problem type that the fast set-point hybrid controller previously discussed was designed for.

The control signal, u, is the set-point for the opening of the valve. Hot water flows through the valve to a heat-exchanger and the air is heated. The output of the system is the air temperature, T, after the fan, which blows the air into the classrooms. The final temperatures in the classrooms are controlled with radiators on a separate control system.

While the double tank system was well known, there was no model available for the heating/ventilation system. A system model was identified. The identification data was logged running the system in manual control mode and then uploaded over a modem connection. System identification data was collected by exciting the system with the input profile sketched in Figure 6. The identification software used is described in [Wallén, 1999]. The only measurable state of the system is the outlet air temperature. During normal use the reference temperature varies between 17 and 22 degrees centigrade. The goal of the experiment was to also be able to handle the start-up phase from considerably lower temperatures. Therefore the transfer function was estimated at a much lower

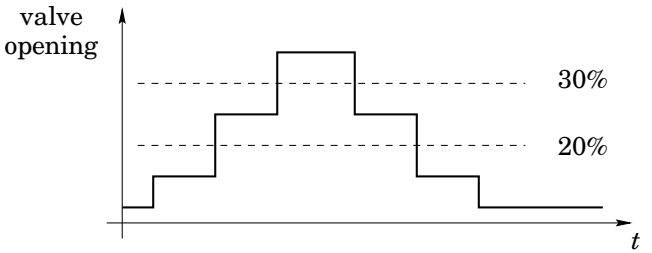


Figure 6. Input profile for the system identification. The figure shows valve opening in percent of full opening as a function of time.

temperature as well.

The actual modeling was done by fitting step responses of low order to match the input-output data.The identification data was separated into several sets, each covering different working-points. In fact there were actually two sets for each working-point, one for raising the temperature and one for lowering the temperature. There was no significant difference in the models estimated from raising or lowering the temperature or at different set-points. Basically the only difference was in the static gain. The identification experiments gave that

$$G(s) = \frac{b}{(s+a_1)(s+a_2)} = \frac{0.03}{(s+0.01)(s+0.05)} \tag{4}$$

was a reasonable model of the system. During the experiments the parameter b had a variation of $\pm 20\%$. At least part of the variation in this parameter is due to the outdoor temperature changes. To raise the temperature, more energy needs to be added to the outdoor air when it is colder.

The chosen model is complicated enough to capture the dynamics of the system and still simple enough to allow an analytical solution to the time-optimal control problem.

There is some freedom in choosing a state space representation for the transfer function in Equation 4. In the real process only one state, the temperature, is measurable and to simplify the filtering needed in the real-time implementation, the second state was chosen as the derivative of the temperature. A suitable state-space representation for simulation and implementation is the controllable form:

$$\begin{aligned} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ -a_1 a_2 & -(a_1+a_2) \end{bmatrix} x + \begin{bmatrix} 0 \\ b \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x. \end{aligned} \tag{5}$$

The parameters a_1, a_2 are almost constant and only the gain b contributes to the process variations as a function of the working conditions.

5. Controller Design

This section describes the design of the subcontrollers for each of the processes. First the design of the PID-controller for both the double tank system and the heating/ventilation process is presented. Next, the time optimal controller is discussed for both of the plants.

PID controller design

A standard PID controller of the form

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

with an additional lowpass filter on the derivative is used. The design of the PID controller parameters K, T_d and T_I is based on the linear second order transfer function,

$$G(s) = \frac{k}{(s + a_1)(s + a_2)}$$

Both the double tank system and the heating/ventilation system may be written in this form. The PID controller parameters are then derived with a pole-placement design method, where the desired closed loop characteristic equation is

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

For the double tank system the sub-controller designs are based on a linearized version of Equation 3:

$$\dot{x} = \begin{bmatrix} -a & 0 \\ a & -a \end{bmatrix} x + \begin{bmatrix} b \\ 0 \end{bmatrix} u. \tag{6}$$

For the tank system $a_1 = a_2 = a$, and $k = ba$, compare Equation 6. The parameter b has been scaled so that the new control variable u is in $[0, 1]$. The parameters a and b are functions of α, β and the reference level around which the linearization is done. It is later shown how the neglected nonlinearities affect the performance. To be able to switch in the PID controller, a fairly accurate knowledge of the parameters is needed. The

parameters $(\omega, \zeta, \alpha) = (0.06, 0.7, 1.0)$ are chosen to get a good behavior under load disturbances.

For the heating/ventilation process the parameters in the closed loop characteristic equation are $(\omega, \zeta, \alpha) = (0.025, 1.0, 1.0)$. The controller parameters are chosen to give a well damped closed loop system but with approximately the same speed as the open system. The controller could be more aggressively tuned than the currently used PID controller, as the error and control signal never are large.

Time-optimal controller design

The problem of set-point response can be viewed as a purely deterministic problem: to change the process from one state to another in shortest possible time, possibly without any overshoot, subject to constraints on the control signal. The constraints are typically bounds on the control signal or its rate of change. This is an optimal control problem, and the theory of optimal control, see [Lewis, 1986] and [Leitman, 1981], is applied to derive minimum time strategies to bring the system as fast as possible from one set-point to another. The time-optimal control is the solution to the following optimization problem

$$\max \int_0^T -1 \cdot dt \tag{7}$$

under the constraints

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

This together with the system dynamics give the control law.

For a wide class of systems the solution to Equation 7 is of bang-bang character where the optimal control signal switches between its extreme values. For problems with a two-dimensional state space the strategy can be expressed in terms of a switching curve that separates the state space into two subsets, where the control signal assumes either its high or low value. This changes the control principle from feed-forward to feedback.

Equilibrium points and switching curves around these may be derived from the state space representations. These switching curves may then be used as reference trajectories. For many systems analytic expressions for the switching curves are hard to derive. The solution is then to use approximate switching curves. If the system is known it is easy to get numerical values for the switching curves through simulation.

The Hamiltonian, $H(x,u,\lambda)$, for the double tank system described by Equation 3 is

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

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

$$\begin{bmatrix} \dot{\lambda}_1 \\ \dot{\lambda}_2 \end{bmatrix} = \begin{bmatrix} -\frac{a}{2\sqrt{x_1}} & \frac{a}{2\sqrt{x_1}} \\ 0 & -\frac{a}{2\sqrt{x_2}} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}. \tag{8}$$

The complete solution to these equations is not needed to derive the optimal control signal. It is sufficient to note that the solutions to the adjoint equations are monotonic. The switching function from solving Equation 8 is $\sigma = \lambda_1 bu$. It gives the optimal control signal sequence that minimizes $H(u)$, and the possible control sequences are then

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

For linear systems of order two, there can be at most one switch between the maximum and minimum control signal value (it is assumed that the tanks never are empty, i.e. $x_i > 0$).

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 pre-calculated switching times. Hence an analytical solution for the switching curves is needed. For the linearized equation for the double tank systems it is possible to derive the switching curve

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

where $\bar{u}$ takes values in $\{0,1\}$. The time-optimal control signal is $u = 0$ above the switching curve and $u = 1$ below, see Figure 7.

The fact that the nonlinear system has the same optimal control sequence as the linearized system makes it possible to simulate the nonlinear switching curves and to compare them with the linear switching curves. Simulation is done in the following way: initialize the state to the value of a desired set-point and simulate backwards in time.

Note that the linear and the nonlinear switching curves are quite close for the double-tank model, see Figure 7. The diagonal line is the set of equilibrium points, $x_1^R = x_2^R$. Figure 7 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.

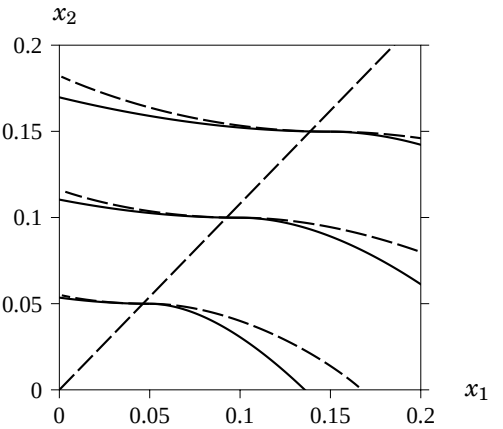


Figure 7. Switching curves, for different set-points, for the double tank system. The set-points lie on the diagonal $x_1 = x_2$. The full lines show the switching curves for the nonlinear system, while the dashed lines show the linearized, approximate switching curves. Above the switching line the minimum control signal is applied and below it the maximum control signal is used.

It is not necessary to use 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 takes over.

The heating/ventilation system is a linear, second order system with real poles, and a time optimal controller for such a system is known to be of bang-bang type. The switching curves for the heating/ventilation system were derived through simulation, and a simple linear function was fitted to the simulated curves. The approximate linear function is simply a function of the steady state coordinate. Both the simulated switching curves and the linear approximations are shown in Figure 8.

6. 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 made in the Omola/Omsim environment [Andersson, 1994], which supports hybrid systems.

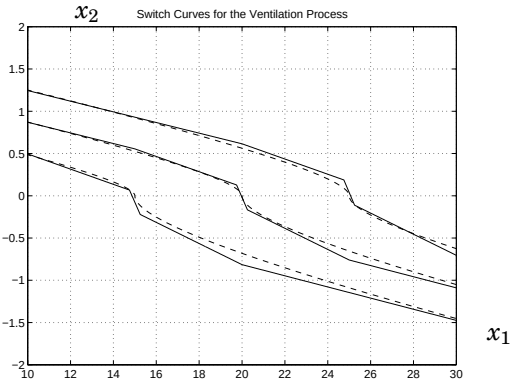


Figure 8. The switching curves for the heating/ventilation system. The figure shows both the actual simulated switching curves (dashed) and the approximate, linear ones (full).

The Double Tank System

A natural simple 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,

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

where

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

The switching strategy here is to start with the time-optimal controller and then switch to the PID controller when $V_{close} < \rho$. The size and shape of the catching region may be changed with the γ and θ parameters. The P matrix above gives ellipsoidal catching curves. In this simulation switching back to the time-optimal controller is not allowed until there is a new reference value. See Figure 3 for a graphical description of the algorithm. The simulation results in Figure 9, show how the best parts from the sub-controllers are combined to give very good performance.

In this second simulation set, the *min switching strategy* that guarantees stability for the linearized system is used. The two Lyapunov func-

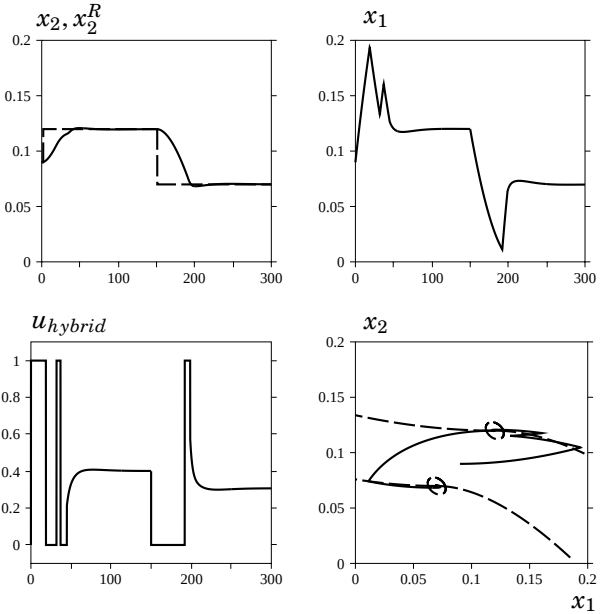


Figure 9. Simulation of the simple switching strategy for the double tank system. Lower left figure shows the control signal. The time-optimal controller makes one extra min-max switch because of the nonlinearity. Catching regions are shown in lower right sub-figure.

tions are defined as

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

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

The second Lyapunov function is associated with the time optimal control law and is the time it takes to reach the new setpoint starting from the current position in the state space if time optimal control is used. The state x_3 is the integrator state in the PID controller and x_3^R is its steady state value. As in the previous simulation set, the parameters γ and 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 not updated until after the first switch to PID control. Using this method a similar two-dimensional catching region as in the case with the *simple switching strategy*, is created. The simulation results are presented in Figure 10.

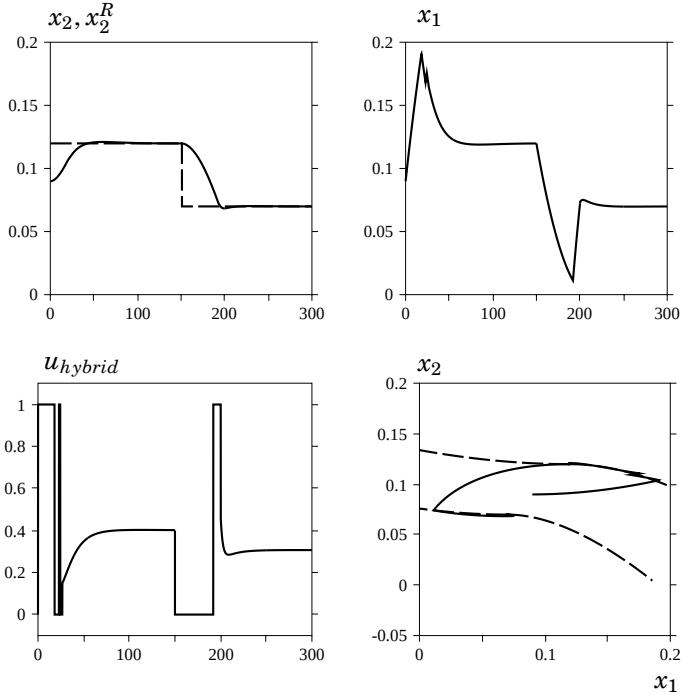


Figure 10. Lyapunov based switching for the double tank system. Compare Figure 9.

This supervisory scheme may lead to two types of chattering behavior. One is only related to the time-optimal controller and is due to the nonlinearities. The nonlinear switching curve lies above the linear, see Figure 7. That causes the trajectory of the nonlinear system to cross the linear switching curve and the control signal goes from 0 to 1 somewhat too late or too early. One way to remove this problem is to introduce a hysteresis when 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 if their corresponding 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 Heating/Ventilation Process

In practical applications it can be an advantage to approximate the switching curves with simpler functions. To investigate the behavior for such approximations, the switching curves for the system in Equation 5 were approximated with five straight lines. As can be seen in Figure 11 the

overall performance did not deteriorate very much. The main difference from using the theoretically correct switching strategy is that a few extra $\langle min, max \rangle$ switches are needed. Simulations to test the method's robustness to process variations were also made. There was no significant decrease in performance for process parameter variations of $\pm 50\%$.

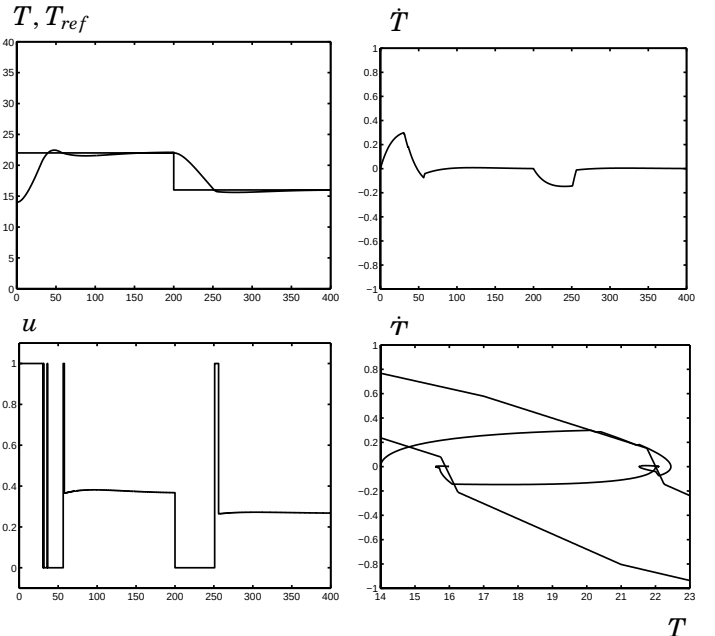


Figure 11. Simulation of the ventilation model for the school in Klippan. The affine approximation of the ideal switching curves is seen together with two set-point change trajectories in the last sub-plot.

7. Implementation

The traditional way of implementing real-time systems using languages such as C or ADA gives very poor support for algorithms expressed in a state machine fashion. The need for a convenient way to implement hybrid systems is evident. The programming language must allow the designer to code the controller as a state-machine, as a periodic process, or as a combination of both. The latter alternative is particularly useful in the case where the controller is divided into several controllers sharing some common code. A typical example of this is a state feedback controller for a plant, where it is not possible to measure all states. To get information about the non-measurable states an observer or a filter can be used.

Typically, this information is needed by the whole set of controllers, and thus the controller itself can be implemented as a hybrid system, consisting of one global periodic task that handles the filtering of process data, and a set of controllers that can be either active or inactive. Many hybrid controllers have the same sort of demands on the programming language.

Implementing complex control algorithms puts high demands on the software environment. Automatic code generation and verification are needed together with advanced debugging and testing facilities.

The Double Tank System

The hybrid controller for the double tank system was implemented in PAL for use in the PÅLSJÖ real-time environment.

PAL [Blomdell, 1997] is a dedicated control language with support for hybrid algorithms. Furthermore, the language supports data-types such as polynomials and matrices, which are extensively used in control theory. PAL has a run-time environment, PÅLSJÖ [Eker and Blomdell, 1996], that is well suited for experiments with hybrid control. PÅLSJÖ was developed to meet the needs for a software environment for dynamically configurable embedded control systems. PÅLSJÖ features include rapid prototyping, code re-usability, expandability, on-line configurability, portability, and efficiency For a more exhaustive description of PAL and PÅLSJÖ, see [Eker, 1997].

The Heating/Ventilation System

The hybrid controller was implemented in FORTH on a Diana ADP 2000 system and tested against a process simulation using real-time SIMNON, see [Elmqvist *et al.*, 1990].

The control system, Diana ADP 2000, manufactured by Diana Control AB, is mainly used in heating-ventilation applications. The system is flexible, modular and well suited for testing of new software. It is possible to install new controller code without having to rebuild the whole system for input/output handling, logging etc.

The controller hardware Diana ADP 2000 is modular. One unit can itself have several control loops, and on large installations several Diana ADP 2000 can be interconnected in a network. In the network configuration, one of the control units is a master and the others are slaves. This master unit could be reached over a modem connection and from the master it is possible to communicate with all the others. Over the telephone line and via a user interface it is possible to log data, change parameters, and down-load new code for every controller.

The programming of both the real-time operating system and the application programs is done in FORTH. Amongst its good properties are that

it is both interactive and compiled. Structure can be built with the notion of "words". Words are executable functions or procedures that can be arranged in a hierarchical structure. Diana ADP 2000 comes with some predefined objects and a limited number of mathematical functions.

8. Experiments

The theory and the simulations have been verified by experiments. For simplicity only the simple switching strategy previously presented was implemented. Figure 12 shows the results of that experiment with the double-tanks.

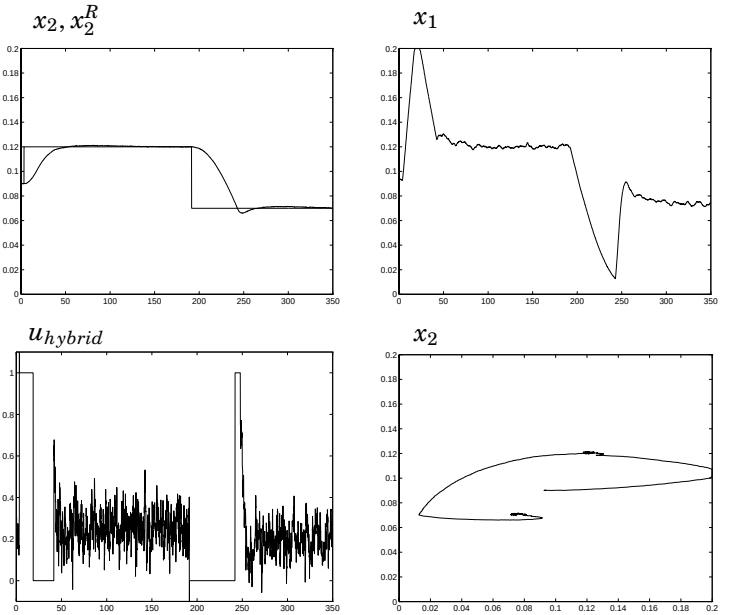


Figure 12. The double tank system. The simple switching strategy is used. The noise level is rather high, which is visible during PID control.

The measurements from the lab process have a high noise level as can be seen in Figure 12. A first order low-pass filter

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

is used 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 the lab process. It is thus possible to compare the experimental results directly with the simulations. A comparison of Figures 9 and 12 shows a close correspondence between simulation and experimental results.

During experiments it was found that the model mismatch, i.e. the difference between the linear and the nonlinear switching curves, did not affect the performance in the sense of fast tracking. It resulted in a few more $\langle min, max \rangle$ switches before reaching the target area where the PID controller took over. However, a good model of the static gain in the system is needed. If there is a large deviation it cannot be guaranteed that the equilibrium points are within the catching regions. The catching region is defined as an ellipsoid around the theoretical equilibrium points.

Field test

Finally the hybrid control strategy was tested at the school in Klippan. The results are shown in Figure 13. As expected there were a few extra $\langle min, max \rangle$ switches due to approximation of the switching curve and probably also due to unmodeled dynamics and nonlinearities. The parameter b was estimated from steady state values of the air temperature and the control valve position.

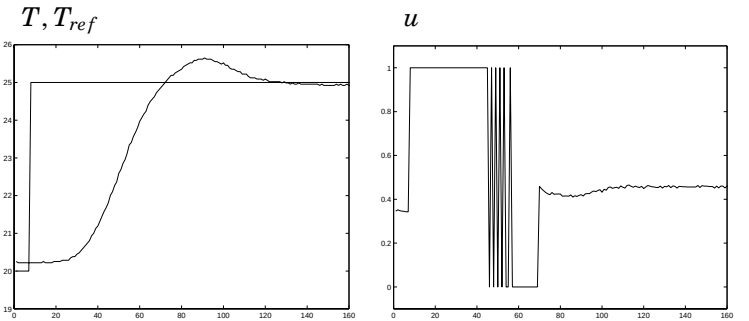


Figure 13. A 5 degree set-point change using the hybrid controller. Air temperature (left) and control signal (right).

A similar set-point change for the controller currently in use at the school showed that it took approximately twice as long time to reach the new set point. However, the main advantage with the proposed method is that it will not give a large overshoot even if started from a very low temperature.

It is not difficult to incorporate some process parameter estimation functions in the code. This would give a good estimate of the static gain. Points on the switching curves could be stored in a table instead of using

the analytical expressions. If the switch is too early or too late because of the nonlinearities, this information could be used to update the switching curve table.

9. Summary

Hybrid controllers for a double-tank system and a heating ventilation process have been designed and implemented. Both simulations and real experiments were presented. It was shown that a hybrid controller, consisting of a time-optimal controller combined with a PID controller gives very good performance. The controller is easy to implement. It gives, in one of its forms, a guaranteed closed loop stability.

The proposed controller solves a practical and frequent problem. Many operators switch to manual control when handling start-up and set-point changes. It is fairly easy to combine this method with a tuning experiment that gives a second order model of the system. From the tuning experiments it is easy to automatically generate a PID controller and an approximate time-optimal controller. The model mismatch was not serious. The nonlinearity led to some additional $\langle min,\ max \rangle$ switches.

The simple hybrid controller was tested on a fast set-point problem on a school. Simulations indicated that the theoretically derived switching curves could be approximated with simple functions without too much performance deterioration. Measurements at the school showed that the process variations were not significant and the variation in steady state gain could be estimated from input output data before or during a set-point change.

A fairly crude model of the system together with approximative switching curves gave significant improvements for set-point changes. The speed was doubled without large overshoots. At this first test, no attempts were made to smooth the control signal. The extra $\langle min, max \rangle$ switches could have been avoided with a hysteresis function.

10. References

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.

Antsaklis, P. J., W. Kohn, A. Nerode, and S. Sastry, Eds. (1998): *Hybrid Systems V*. Springer-Verlag.

Antsaklis, P. J. and A. Nerode (1998): "Hybrid control systems: An introductory discussion to the special issue." *IEEE Trans. Automatic Control*, **43(4)**, p. 497.

Blomdell, A. (1997): "PAL – the PÅLSJÖ algorithm language.". Report ISRN LUTFD2/TFRT--7558--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Branicky, M. (1995): *Studies in Hybrid Systems: Modeling, Analysis, and Control*. PhD thesis, LIDS, Massachusetts Institute of Technology.

Branicky, M. S., V. S. Borkar, and S. K. Mitter (1998): "A unified framework for hybrid control: Model and optimal control theory." *IEEE Trans. Automatic Control*, **43(1)**, p. 31.

David, R. and H. Alla (1992): *Petri Nets and Grafcet: Tools for modelling discrete events systems*. Prentice-Hall.

Eker, J. (1997): *A Framework for Dynamically Configurable Embedded Controllers*. Lic Tech thesis ISRN LUTFD2/TFRT--3218--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Eker, J. and A. Blomdell (1996): "A structured interactive approach to embedded control." In *The 4th International Symposium on Intelligent Robotic Systems*. Lisbon, Portugal.

Elmqvist, H., K. J. Åström, T. Schönthal, and B. Wittenmark (1990): *Simnon User's Guide*. SSPA, Göteborg, Sweden.

Ferron, E. (1996): "Quadratic stabilizability of switched systems via state and output feedback." Technical Report CICS-P-468. Center for intelligent control systems, MIT, Cambridge, Massachusetts, 02139, U.S.A.

Hay, J. L. and A. W. J. Griffin (1979): "Simulation of discontinuous dynamical systems." In *Proc. of the 9th IMACS Conference on Simulation of Systems*. Sorrento, Italy.

Johansson, M. and A. Rantzer (1998): "Computation of piecewise quadratic lyapunov functions for hybrid systems." *IEEE Transactions on Automatic Control*, **43:4**, pp. 555–559. Special issue on Hybrid Systems.

Leitman, G. (1981): *The Calculus of Variations and Optimal Control*. Plenum Press, New York.

Lewis, F. L. (1986): *Optimal Control*. Wiley.

Malmborg, J. (1998): *Analysis and Design of Hybrid Control Systems*. PhD thesis ISRN LUTFD2/TFRT--1050--SE, Department of Automatic Control, Lund Institute of Technology.

Malmborg, J. and J. Eker (1997): "Hybrid control of a double tank system." In *IEEE Conference on Control Applications*. Hartford, Connecticut.

Morse, A. S. (1995): "Control using logic-based switching." In Isidori, Ed., *Trends in Control. A European Perspective*, pp. 69–113. Springer.

Park, T. and P. Barton (1996): "State event location in differential algebraic models." *ACM Transactions on Modeling and Computer Simulation*, **6(2)**, p. 137.

Wallén, A. (1999): "A tool for rapid system identification." In *Proceedings of the 1999 IEEE International Conference on Control Applications*.

Appendix A

PAL – Pålsjö Algorithm Language

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 available, and complex data types such as polynomials and matrixes are fully supported.

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

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 internal states and the parameters. A PAL block has the following structure:

EXAMPLE 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 of the block defines the interface. In this example it consists of one input signal *in*, one output signal *out*, one parameter *p*, and one state variable *i*. After the interface section comes the algorithm section. The two procedures **calculate** and **update**, define the periodic algorithm of the block.

3. Block 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. The syntax for defining variables is:

name [:=value]: [**interface modifier**] **data type**

The possible data types and interface modifiers are described below.

4. 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 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 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.

5. Scalar Data Types

The predefined scalar data type are the following:

- **real** A real valued variable.
- **integer** An integer valued variable.
- **string** A text string.
  - str = "initial value" : **string**;
  - String concatenation is 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.

6. Aggregate Data Types

The predefined aggregate data types are:

- **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 a matrix.
  - The matrix data type maps down onto the Newmat matrix class [Davis, 1997]. 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.
  - 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 onto an external polynomial package [Eker and Åström, 1995].

7. 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 are the expressions and statements available in PAL.

- **Unary Expressions** A unary expression is an expression which only involves one operand. There are 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 are 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. $x[i]$), which returns a specific part of an array, matrix, or polynomial.
  - Function calls. A function reads input parameters and returns a value.

- **Assignment** A variable is assigned using the assignment operator :=.

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

- **The if Statement** Conditional execution is constructed using the **if** statement.

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

- **The for statement** One or more statements are repeated using the **for** statement.

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

8. Procedures, Functions and Events

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 parameters 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 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 a matrix row to an array, and back. The definition of the third function is omitted for simplicity.

EXAMPLE 3

```
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;
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;
```

The variables $tmp1$ and $tmp2$ are global variables declared in the block scope. The value of the integer variable n is dynamically bound to the size of the input vector *theta*. □

A function is a subprogram which consists of a sequence of statements. A value is returned after finishing execution. 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. The return value is stored in the predefined variable **result**.

```
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;
```

A PAL block may have a set of events that it responds to. An event itself is a text string. When a block receives an event it executes the procedure with the same, if there is any. All procedures in PAL are registered as events when executed in the PÅLSJÖ run-time environment. Let, for example, the procedure *Reset* 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
```

There are two predefined block procedures **calculate**, and **update**. They are used to implement periodic algorithms. The **calculate** procedure calculates an output signal, while **update** updates the state variables. The reason for dividing the algorithm like this is discussed in detail in [Eker, 1997].

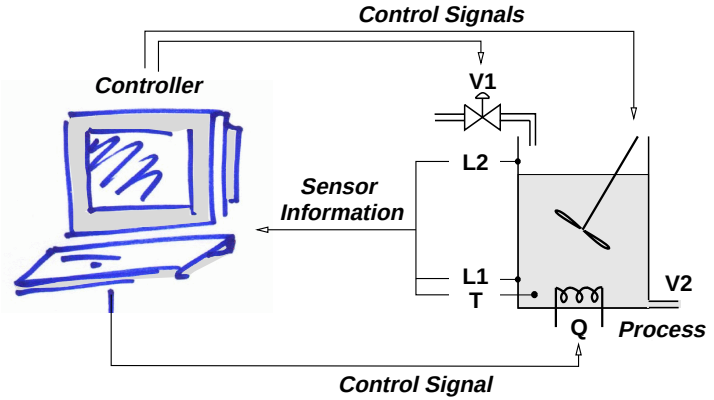


Figure 1. The reactor is used for mixing and heating fluids. When batch is started the valve is opened and the tank starts filling. When the tank level reaches $L1$, the heater and the mixer start. The valve stays open until the upper level $L2$ is reached. When the tank is filled and the content has reached the desired temperature, the tank is emptied and a new batch can be started.

Grafcet

Grafcet [David and Alla, 1992] is a convenient and powerful way of implementing sequential algorithms. 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].

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 PAL syntax for parts of the Grafcet in Figure 2 is presented below.

A step is defined as follows.

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

step StartHeating;
  activate Heat;
end StartHeating;

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

To each step an action can be attached. The action contains the statements

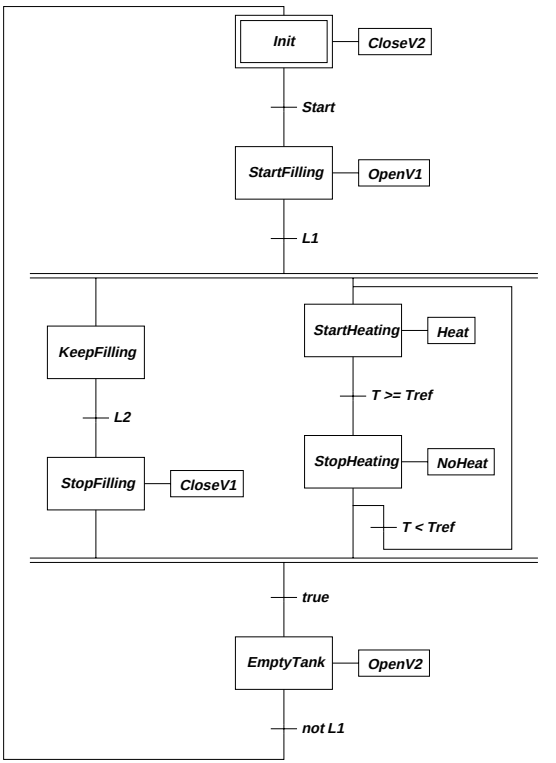


Figure 2. Grafcet for the controller of the boiler process, see Figure 1. This figure is generated by the PAL compiler using the following command: `pal -fig grafcet.pal`.

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, for example:

- **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 becomes active.

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** $T >= Tref$;

An Example

The PAL code for the Grafcet in Figure 2 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 L1;
    transition from StartHeating to StopHeating when T >= Tref;
    transition from StopHeating to StartHeating when T < Tref;
    transition from KeepFilling to StopFilling when L2;
    transition from StopFilling,StopHeating to EmptyTank
            when true;
    transition from EmptyTank to Init when not L1;
    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.
```

9. References

Åström, K. J. and B. Wittenmark (1995): *Adaptive Control*, second edition. Addison-Wesley, Reading, Massachusetts.

Blomdell, A. (1997): "PAL – the PÅLSJÖ algorithm language.". Report ISRN LUTFD2/TFRT--7558--SE, 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 International(UK).

Davis, R. B. (1997): "Newmat – A matrix library in C++." http://nz.com/webnz/robert/.

Eker, J. (1997): *A Framework for Dynamically Configurable Embedded Controllers*. Lic Tech thesis ISRN LUTFD2/TFRT--3218--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

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.

Lewis, R. (1995): *Programming industrial control systems using IEC 1131–3*. The Institution of Electrical Engineers, London, U.K.

Appendix B

PCL – Pålsjö Configuration Language

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.

2. Keywords

The predefined keywords in PCL are the following;

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

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 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 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 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 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 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` block, that owns the block must be connected over the network to a plot or data-log utility. This is done by sending the event `connect` to the `Periodic` block. 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 signals. `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>
```

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 trigger 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 make 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 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*>
```

□

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 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 4 and Example 5 are equivalent.

EXAMPLE 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
```

□

Lund Institute of Technology

Lund University

Department of Automatic Control

ISSN 0280–5316
ISRN LUTFD2/TFRT--1055--SE