ISSN 0280–5316
ISRN LUTFD2/TFRT--7567--SE

Patterns in Embedded Control Systems

Johan Eker
Anders Blomdell

Department of Automatic Control
Lund Institute of Technology
October 1997

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

Document name
INTERNAL REPORT

Date of issue
October 1997

Document Number
ISRN LUTFD2/TFRT--7567--SE

Supervisor

Sponsoring organisation

Author(s)
Johan Eker, Anders Blomdell

Title and subtitle
Patterns in Embedded Control Systems

Abstract

In this report we present a framework for implementation of embedded controllers. A discussion on design decisions are given. Further two pattern and one idiom used in the framework are presented. Finally a new language for implementation of embedded controllers are introduced.

Key words
Design pattern, framework, embedded control, real-time

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280–5316	*ISBN*

Language english	*Number of pages* 24	*Recipient's notes*
Security classification		

1. Introduction

The traditional way of implementing real-time systems using languages such as C or C++ gives deficient support for reuse of code. One problem is the difficulty to separate timing specifications from logical specifications.

Due to this, embedded control software often has to be constructed from scratch, even for minor changes of system requirements to existing implementations.

In the field of automatic control many implementations have a similar internal structure and one of the major driving forces when creating the PÅLSJÖ[8] framework has been to support this common structure.

To support implementation of features that are characteristic to control systems a dedicated language PAL (PÅLSJÖ Algorithm Language)[5] has been developed. Control algorithms can in most cases be described either as periodic tasks, or as finite state machines. PAL supports those types of algorithms. Furthermore, the language supports data-types such as polynomials and matrices, which are extensively used in control theory. PÅLSJÖ has been designed to support rapid prototyping of control systems. Off-line the engineer defines a set of blocks which at run-time can be instantiated on-line and connected to form a control system. Blocks in a running system can be replaced without having to stop the system.

The system consists of two main parts; a compiler and a run-time system. The run-time system provides a text interface for the user and a network interface for data transmission. PÅLSJÖ is primarily designed to run in a host-target configuration. Stand alone tools for on-line data display have been developed. Reuse of algorithms is possible through a block library facility.

In this report first the PÅLSJÖ framework will be presented. Then two patterns and one idiom used in the framework are discussed. The first pattern is called *CalculateOutput-UpdateState*, and deals with the execution of systems of periodic algorithms. The other pattern presented is *ParameterSwap*, which is pattern for assigning values to parameters of real-time processes. Finally the PAL language for controller design is presented.

2. A Framework for Real-time Control

2.1 Introduction

In this section the PÅLSJÖ framework for implementation of real-time control applications is presented. The presentation will loosely follow the outline for patterns given in [9]. First we will give a motivation why the pattern is needed, and then a description of the intent of the pattern is given. The driving forces are presented together with a solution. A number of well known patterns that are used in the framework will also be discussed.

2.2 Block diagrams

Block diagrams are used to schematically express the functional entities and their interconnections in a control system. Figure 1 shows a block diagram with a control block, a process block and a negative feedback loop.

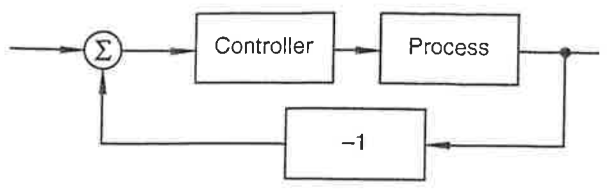


Figure 1 Controllers are usually described using block diagram. A block is defined by a set of input and output signals, parameters and states. A block diagram defines how data flows between a set of blocks.

A block diagram basically consists of connections and blocks that do some kind of processing to produce output values that somehow reflect their input values.

2.3 Motivation

Many control applications today are implemented using assembler or a low-level language such as Forth or C. The reason for this is the need for fast execution and small programs. Another choice is to use a language with built-in support for concurrency, e.g. Modula-2, ADA or or Grafcet. Those languages provides a higher abstraction level when it comes to real-time programming, but will give larger and slower programs. No matter which approach that is used, the implementation of real-time controllers becomes very time-consuming and error prone. 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. But even though the implementation structures resemble each other a lot, it is very hard to reuse code from one application to another. This is of course extremely frustrating having to rewrite previously implemented algorithms just in order to make them work in the new application. One reason why it is so difficult to reuse code is that logical and temporal statements are mixed together.

Figure 2 shows the structure of a typical control application. It consists of four main modules.

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

2.4 Intent

This section describes the goals that were set up when designing the framework. The main idea is to take advantage of the similar structure of many control applications. If those common features can be encapsulated in a framework, then a large percentage of the total code that have to be written could be avoided. Further it is possible to introduce a suitable abstraction level, that will support control algorithms in particular. The goal is to give such a high degree of support so that the programmer can focus

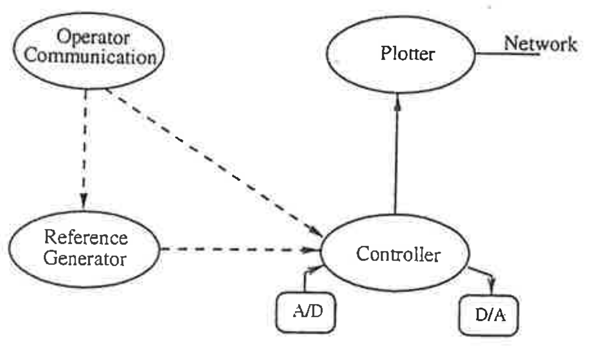


Figure 2 A typical control application consists of four main modules. *Operator Communication* which handles the interaction with the user, i.e. setting and reading parameters. *Reference Generator* which is used to calculate the desired position for the controller. *Controller* implements the control algorithms and *Plotter* handles displaying data for the user. The dashed lines mark asynchronous communication while the solid lines mark synchronous communication

only on implementing the actual control algorithms, and let the framework take care of user interaction and network interaction. Figure 3 shows the general structure of such a framework. The left figure shows the processes of the framework. There exist processes for network management and operator communication. Further there are two processes that represent the actual control application. Those two processes are initiated by the user through the user interface. Each of those processes consists of a set of algorithmic blocks that describe the algorithm of each processes. The dashed lines marks asynchronous communication while the solid line marks synchronous communication. To the right in Figure 3, a user processes is shown. A number of algorithmic blocks are connected together and form the algorithm of the processes, i.e. each processes represents a block-diagram of its own. Another very important aspect when designing a framework is the support of code reuse. This is achieved through the possibility to reuse blocks and order them in libraries.Finally as little overhead as possible should be introduced when building an application using the framework compared to building it from scratch.

2.5 Forces

There are a number of forces that are taken into account when designing the framework.

- *Rapid prototyping*
  One of the main reasons for us to use a framework is to decrease development time. The framework is not intended to be used in the creation of end-user products, but instead as a flexible lab tool.
- *Code Reuse*
  In order to support rapid prototyping there must be good support for reuse of algorithms.
- *Expandable*
  The framework must be expandable so that new features easily can be introduced. For example it should be possible to use data-types that were not available in the original setup.

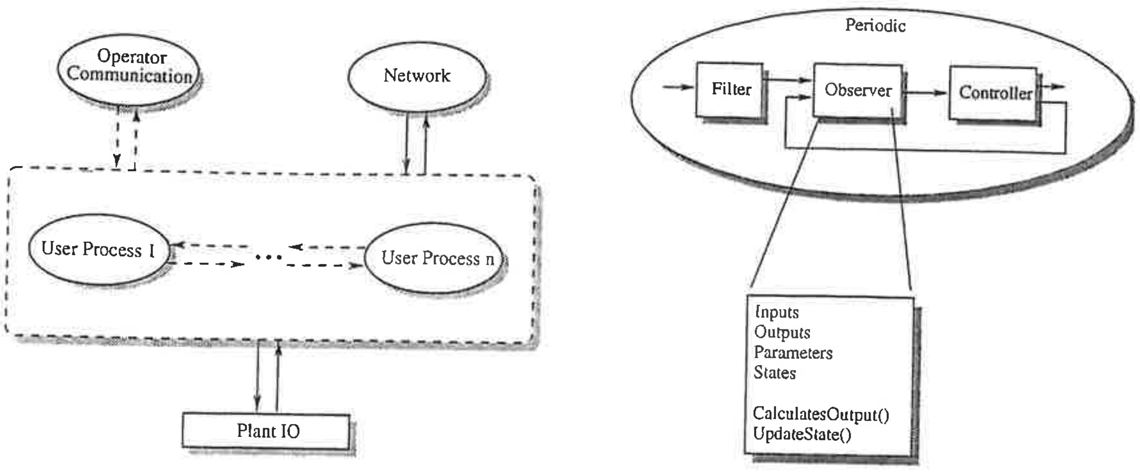


Figure 3 The left figure shows the processes of the framework. The processes for network management and operator communication are initiated automatically. The processes marked *User Process 1–n* are user defined process which contains the actual control algorithms. The dashed lines mark asynchronous communication while the solid lines mark synchronous communication. To the right a user process is shown. The algorithm is decsribed by a blockdiagram.

- *On-line configurable*
  The system should be configured on-line, and not at compile-time. Changes in running setups should be allowed without stops. One way to handle this could be by interpeting the algorithms, another way would be to use dynamic linking.
- *Efficient*
  For the framework to become really useful it must be efficient and allow fast sampling rates. The timing must also be accurate.
- *Distributed*
  Many control applications are distributed over several components and support for such systems must be taken into account.
- *Fault-Tolerant*
  Some control systems are mission-critical and are not allowed to crash. In case of a fault, the system should support recovery features so that a failure can be avoided.
- *Generic interfaces*
  We want the framework to provide interfaces to the controller. One interface towards the user for asynchronous information and another synchronous interface for network communication and data logging. The user interface should be implemented in such a way that it is platform independent.
- *Execution Analysis* Global versus local sorting

These are the forces that have beePÅLSJÖ framework.

2.6 Solution

In this section the basic ideas behind the implementation will be presented, i.e. the major design choices made to fulfill the specification discussed above.

The main abstraction used in control engineering is *block diagrams*. Block diagrams are very powerful ways of describing algorithms and dataflows. Further they support modular programming and are thus well suited as an implementation model. In the system a block is the smallest programming entity. A block can be described as a seven tuple $B = (I, O, P, S, E, L, P)$.

- A block can have a set of input signals I. An input signal must be connected to the output signal of another block or itself. It is not allowed to assign values to input signals.
- A block can have a set of output signals O. An output signal may be connected to an input signal of another block or itself.
- A block can have a set of parameters P. Parameters can only be set by the user or the system. The value of a parameter cannot be changed internally in the algorithm.
- A block can have a set of states S, which describe the internal state of the block. A state can only be set internally.
- A block can have a set of events E that it responds to. An event can be either synchronous or asynchronous. Synchronous here means that the event is taken care of at the next sampling instance. A asynchronous event on the other hand will be handled immediately when it arrives. A synchronous event could for example be a request for changing controller mode, while an emergency stop should be a asynchronous event.
- A block can contain sequential logic L, which is described by a state machine.
- A block can contain a periodic algorithm P that describes the periodic behavior of a block. If a block contains periodic algorithms it must be executed periodically.

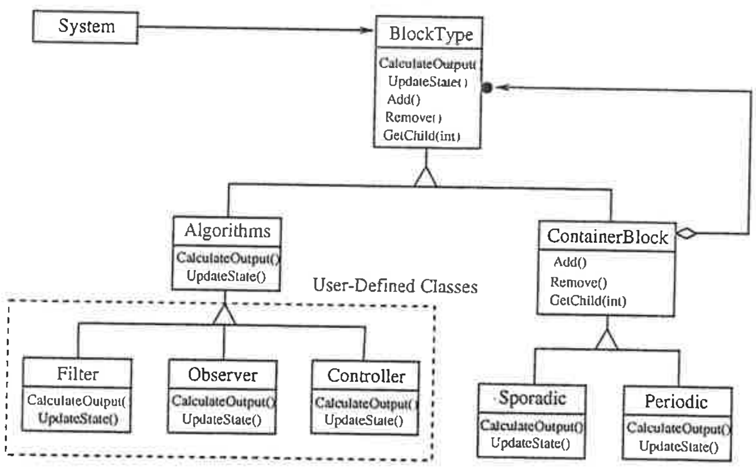


Figure 4 The inheritance structure of the framework. A `ContainerBlock` is a block type that is designed to encapsulate a set of blocks. Two subclasses are available, `Periodic` and `Sporadic`. The `Periodic` block executes its child blocks periodically according to the data flow between the child blocks. All timing and synchronization between the child blocks is taken care of by the scheduler in `Periodic`.

To support reuse of algorithms a design decision was made to separate temporal and functional specifications. A user-defined block cannot contain

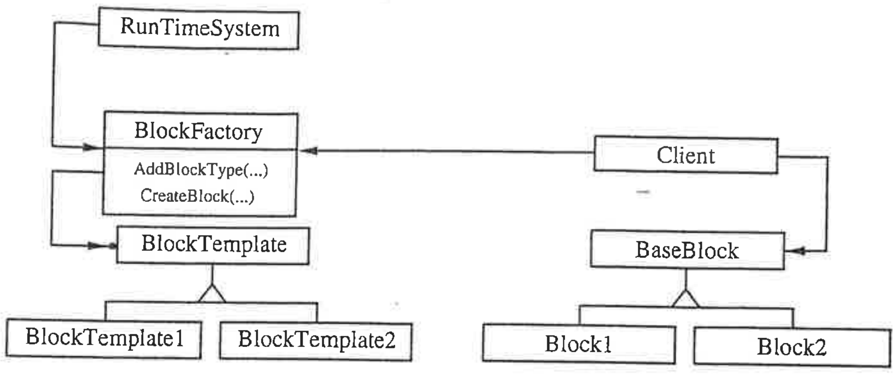


Figure 5 The run-time system adds new block-types to the BlockFactory. When the client, in this case the user by typing in a block creation command, wants to create a new block it calls the BlockFactory, which does the actual allocation.

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 user-defined blocks. Using this approach the programmer does not have to deal with any real-time programming, further it is possible for the systems to optimize the execution and prevent problems, for example jitter[1] [18]. The inheritance structure of the different block types is shown in Figure 4.

A `ContainerBlock` is a block type that is designed to encapsulate a set of blocks. Two subclasses are available, `Periodic` and `Sporadic`. The `Periodic` block executes its child blocks periodically according to the data flow between the child blocks. All timing and synchronization between the child blocks is taken care of by the scheduler in `Periodic`. The inheritance structure is know as the Composite pattern[9]. A `Sporadic` block only executes when it receives an event, i.e. similar to interrupt handling. A `Periodic` block can be viewed as a special case of the `Sporadic` driven by time events. Using `ContainerBlocks` the system supports the hierarchical structure that was specified above.

A block is coded, compiled, and linked off-line. When the system is started, blocks are instantiated and connected on-line by the operator, using a special configuration language. Instances of the system blocks `Period` and `Sporadic` are created to manage the execution. A user block must have a `ContainerBlock` as a parent in order to be executed at all.

Dynamic Creation of Data types and Block types In order to make the system as useful as possible it must be possible to extend the framework in a simple way. In the Pålsjö framework algorithmic blocks and all data-types used by the system are only loosely coupled through the use of an Abstract Factory pattern [9].

Figure 5 show the class diagram for the block factory. The run-time system calls the `BlockFactory` upon initialization and registers each available block type by giving a name tag and a constructor function. The `BlockFactory` keeps a table over all registered block types.

When the client wants to create a new block it calls the `BlockFactory` with the name of the block. The `BlockFactory` tries to find and run the constructor for the wanted block type, and upon success it returns a handle

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

to the new block instance.

It is possible to register new block types and delete old during execution. This means that it is possible to extend the system with new functionality without having to shut it down.

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

The available block and data types can thus be changed during run-time without having to stop and restart the system.

Portability The system is implemented on top of our real-time kernel STORK [1]. The kernel is available for Windows NT, Motorola 68000, Motorola Power PC and Sun Solaris 2.x, and so is Pålsjö.

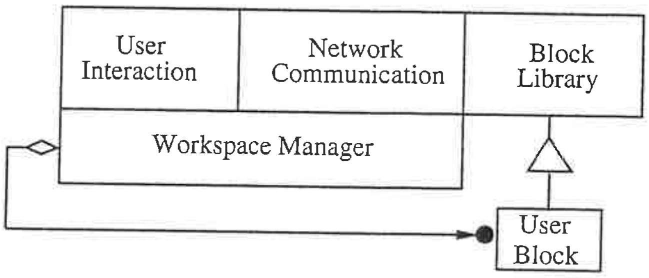


Figure 6 The framework takes care of the interaction with the user, the network communication and manages the block instantiation and execution. A user defined block is inherited from a pre-defined class. The user will simply modify a small number of functions to implement the control algorithm for the new block type.

2.7 Structure

The inheritance structure of the framework from an end user's point of view is shown in Figure 6. The framework is a so called black box framework[13].

The framework takes care of the interaction with user, the network communication and manages the block instantiation and the execution. A user defined block is inherited from the pre-defined class *Algorithms*, see Figure 4. The user will simply modify a small number of functions to implement the control algorithm for the new block type.

2.8 Consequences

- *Rapid prototyping,* is resolved through the use of a new dedicated language PAL, which is a highly specialized language for description of control algorithms. The use of it will speed up the implementation process. PAL supports code modularization and will thus resolve the *code reuse* force.
- The system is *Expandable* in the sense that new data types and new block types easily may be introduced. This is achieved through the use of register functions and factory patterns as discussed above.
- All configuration of the system is done on-line with with a special configuration language, the systems is thus *On-line configurable*. Further new types can be added to the factories on-line.
- The use of the Forward Backward pattern and Parameter swap pattern is aimed at making the execution efficient.

- Distributed applications are simply treated as several stand-alone applications communicating using system network blocks.
- The problems regarding *fault-tolerant* control application is not handled sufficiently in the current version of the framework.
- The system provides two interfaces. One text-based interface for configuration and one network interface for data logging.

3. The CalculateOutput-UpdateState Pattern

Intent This pattern addresses the problem where the computational result from a function can be divided into two parts, one with a harder time constraint than the other. A typical example is a control algorithm, where it's important to finish the calculation of the new control signal as fast as possible, but where the calculation of the state updates is allow to be more time consuming.

Another important use of this pattern is when several blocks are used to calculate an output signal, which depends both on external inputs and on internal states in each block. For example, consider the case when three functions are needed to calculate a result, and the output from the first block is used as input to the second, and so on. It is common in control algorithms, that to update the states in the first block, the result of the second block must be know. This problem is also addressed by this pattern.

Variations – Also known as Forward-Backward

Motivation Consider the control system in Figure 7. The controller internally consists of a set of blocks. First the value of the process is sampled and the signal propagates from the first block to the second one. The output of the second block is then passed on to the third block and so on. The input signal is sampled and a new output signal is calculated periodically at a specified sampling rate. In a control system the stability of the closed loop system is dependent on the time it takes for a process measurement to show up in the actuation signals to the process. If there are multiple blocks between the input and output, special care has to be taken to ensure that no unnecessary delays are introduced due to improper ordering of calculations. Consider the following two different execution orders for the blocks in the controller in Figure 7.

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

If the calculations are done in backward order, the input values reach the output after 6 sampling intervals, while in forward order they reach

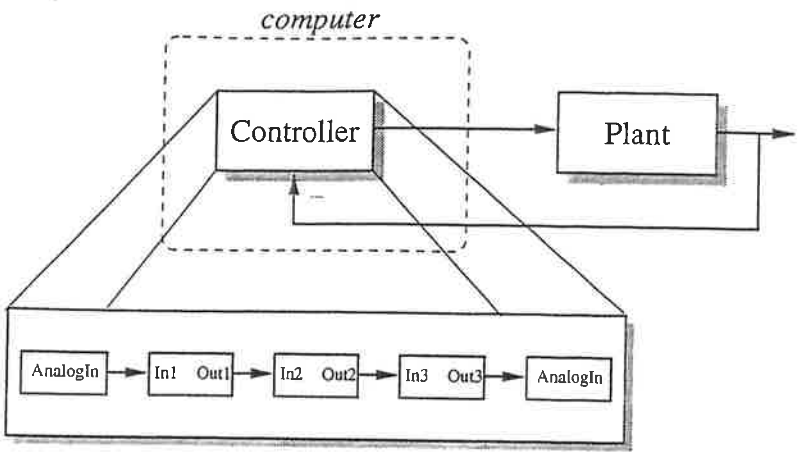


Figure 7 A schematic view of a controller which consists of a number of sub blocks. The output from the plant is sampled by *AnalogIn*, and then the signal is propagated from left to right until it reaches *AnalogOut*, which sends the control signal to the actuator. The CalculateOutput-UpdateState pattern addresses the problem of minimizing the time delay between process measurement and actuation.

the output in a fraction of a sampling interval, and this seems to be the obvious choice since we want to minimize the time delay.

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

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

CalculateOutput (Forward)	Read inputs Do calculations needed for outputs Write outputs
UpdateState (Backward)	Do all other calculations

Again consider the controller in Figure 7, and let each of the sub-blocks have two functions *CalculateOutput()* and *UpdateState()*. The execution of the blocks would now be the following.

```
AnalogOut.CalculateOutput()
Block2.CalculateOutput()
Block3.CalculateOutput()
AnalogOut.CalculateOutput()
AnalogOut.UpdateState()
Block3.UpdateState()
Block2.UpdateState()
AnalogOut.UpdateState()
```

3.1 Participants

In the Pålsjö Framework the execution of a block-diagram is taken care of by an object inherited from the class `ContainerBlock`, see Figure 4.

3.2 Sample Code

Consider the implementation of a PI-controller. The control law is described by the following equation, where $e(t) = r(t) - y(t)$ is the control error and $u(t)$ is the control signal.

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

To implement this algorithm is must first be discretized. The proportional term P in Equation(1) is then replaced by

$$P(t_k) = Ke(t_k) \tag{2}$$

and the integral part is replaced by the following recursive expression which is extended with a tracking term for handling actuator saturations [3].

$$I(t_{k+1}) = I(t_k) + \frac{Kh}{T_i}e(t_k) + \frac{h}{T_r}(u - v) \tag{3}$$

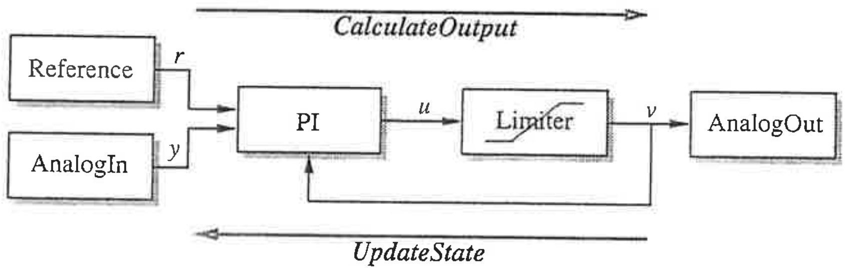
The pseudo code for the implementation of a PI-control algorithm would then look like this

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

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

    **update**
    **begin**
      $I := I + bi * e + br * (u - v)$;
    **end backward**;

  **end** PI;
**end** Controller.
```

Now consider the case when the controller consists of several sub-blocks, where each block is divided into two functions similar to the PI-controller above. The execution of the whole block-diagram would then be

```
block Periodic
  blocks : array [1..5] of block;
  n := 5 : integer;
  calculate
    i : integer;
  begin
    for i := 1 to n do
      blocks[i].calculate();
    end for;
  end forward;
  update
    i : integer;
  begin
    for i := n to 1 step −1 do
      blocks[i].update;
    end for;
  end backward;
end Periodic;
```

This algorithm is illustrated below.



3.3 Known Uses

This is a well known pattern in the control community. Use of this is suggested in many control textbooks. One known industrial application is SattLine from Alfa Lava Automation[12].

4. The Parameter-Swap Pattern

4.1 Intent

The Parameter-Swap pattern is a real-time pattern which deals with the problem of assigning values to process parameters in a fast and consistent way.

4.2 Motivation

Consider the following piece of code. It is an algorithm expressed in PAL. The block has one input signal *in*, one output signal *out*, one state *s*, and

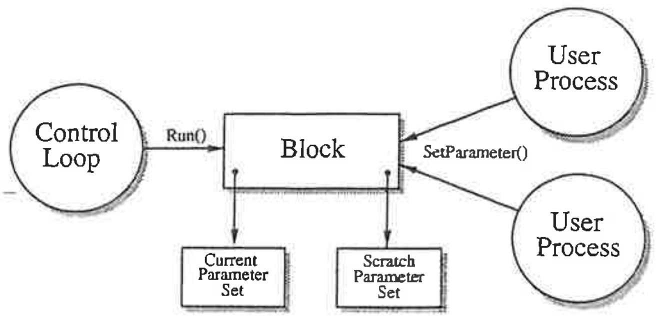


Figure 8 The block algorithm is executed by the Control Loop process. Here the algorithm uses the Current Parameter Set. For assigning values to parameters the Scratch Parameter Set is used. When an assignment operation is finished and all necessary calculations are made the two sets are swapped.

three parameters a, b and c.

```
block par
    in : input real;
    out : output real;
    s : state real;
    a : parameter real;
    b : parameter real;
    c = a * b;
    forward
    begin
      out := ...
    end forward;
    backward
    begin
      s := ...
    end backward;
end par;
```

Parameters are defined as special variables that can only be assigned from outside the block. When the parameters a and b are assigned new values, a new value for c must also be calculated. a and b are called *direct* parameters and c is an *indirect* parameter.

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

4.3 Structure

The solution that this patterns suggest is to have two parameters sets, see Figure 8. One parameter set that is used by the block algorithm, and another that is used for assigning direct parameters and calculating indirect parameters. When a assignment operation is finished the block gets access

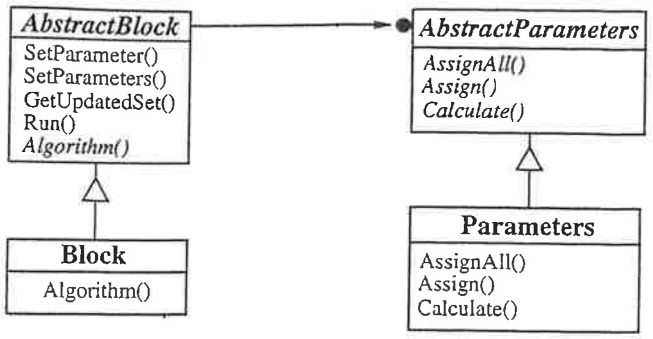


Figure 9 The two classes `AbstractBlock` `AbstractParameters` are super classes that the user will use when implementing new blocks. All interaction between the user, the framework and and the parameters are specified in the relation between `AbstractBlock` and `AbstractParameters`.

to the new parameter set through a pointer swap.

4.4 Sample Code

The class `AbstractBlock` constitutes a logical block that can be accessed from the surrounding framework. It has basically two interfaces, `Run` for executing the block algorithm, and `SetParameter` and `SetParameters` for assigning block parameters. Usually those two interfaces are access from different processes which are not synchronized. Further it is necessary that the `Run` method may be executed without any interference from other methods.

```
class AbstractBlock {
 public:
  int SetParameter(int parID, parVal value);
  int SetParameters(Parameters *newpars);
  void GetUpdatedSet();
  void Run();
  virtual void Algorithm() {}

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

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

```
class AbstractParameters {
  Monitor mon;
 public:
  virtual int Assign(parID, valueType) { return 0; };
  virtual void AssignAll(newpar) {};
  virtual void Calculate() {};
};
```

A block is a way of encapsulating an algorithm in a straightforward and convenient way. The algorithm is executed by the surrounding framework through the `Run` method. `Run` simply does two things, first it calls `GetUpdatedSet` to get the latest parameter set and then it calls `algorithm`.

The parameters are shared by several processes and must thus be protected so that no inconsistencies arise. This is usually done using primitives such as semaphores and monitors. The problem with this approach is that it is possible for one process to block another. In our setup we do not want to allow the real-time process to be blocked. The solution suggested here is to simply disable the interupt when the real-time proces is accessing the data. A flag `parChanged` is used to indicate if new parameters are set.

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

  mask = Coroutines.Disable();
  if ( parChanged ) {
    tmp = scratch;
    scratch = current;
    current = tmp;
    parChanged = FALSE;
    event.Cause();
  }
  Coroutines.Reenable(mask);
}
```

`Run` is called by the framework to execute the block algorithm.

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

The `AbstractBlock` provides two methods for the assigning parameters, `SetParameter` for assign one parameter at a time, and `SetParameters` for assigning all parameters at once. The monitor `mon` is needed since there may be several user process trying to assign parameters.

```
int AbstractBlock::SetParameter(int parID, valueType value)
{
  int result;

  mon.Enter();
  mask = Coroutines.Disable();
  if ( parChanged ) {
    event.Await();
  }
  Coroutines.Reenable(mask);

  result = scratch->Assign(parID, value);
  if ( result ) {
    scratch->Calculate();
    mask = Coroutines.Disable();
```

```
      parChanged = TRUE;
      Coroutines.Reenable(mask);
    }
    mon.Leave();
    return result;
  }

  void AbstractBlock::SetParameters(Parameters *newpar)
  {
    mon.Enter();
    mask = Coroutines.Disable();
    if ( parChanged ) {
      event.Await();
    }
    mask = Coroutines.Disable();

    scratch->AssignAll(newpar);
    scratch->Calculate();

    mask = Coroutines.Disable();
    parChanged = TRUE;
    Coroutines.Reenable(mask);
    mon.Leave();
  }
```

The two classes `AbstractBlock` and `AbtractParameters` handle all the interaction with the environment. An algorithm is added to `AbstractBlock` in `Block`. The `algorithm` works on data in the `Parameters` class.

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


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

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

In the `Parameters` class all the parameter variables are added. Further three functions for managing them are implemented. The two methods `Assign` and `AssignAll` implement the assignment of one or several parameters, respectively. The third method `Calculate` implements the calculation of indirect parameters.

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

```
void Parameters::AssignAll(AbstractParameters *newpar)
{
  par1 := ((Parameters *) newpar)->par1;
  par2 := ((Parameters *) newpar)->par2;
}

int Parameters::AssignOne(parID id, valueType value)
{
  int result = TRUE;
  switch(id) {
   case 1:
    par1 := ((Parameters *) newpar)->par1 = value);
    break;
   case 2:
    par2 := ((Parameters *) newpar)->par1 = value);
    break;
   default:
    result = FALSE;
    break;
   }
  return result;
}

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

4.5 Known Uses

The basic ideas behind this is pattern are well known [2]. The sample code here is based on [16].

5. The Register Idiom

5.1 Intent

The register idiom provides a method to add new code to an existing framework without having to recompile the main program.

5.2 Motivation

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

When designing the Pålsjö we want to avoid this. Instead the user should only need to add the names of the new files to the makefile. To solve this we used the Register idiom.

5.3 Structure

The way the register idiom works is similar to dynamic linking of code. The basic idea is that that when a new class is added to the framework

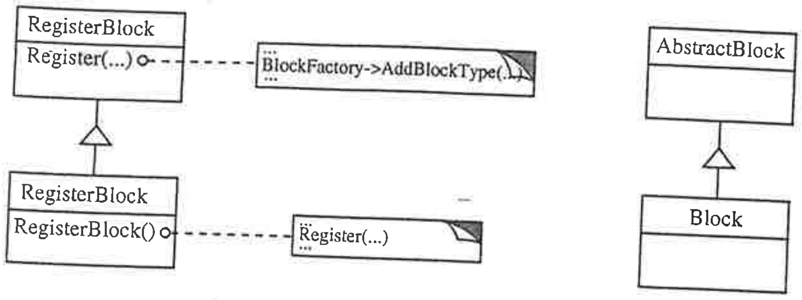


Figure 10

it notifies the main program that a new class is available and it also tells the main program how to create an instance of this new class. The class is said to register, when it notifies the framework. The registration can be done either on the start up by using static objects, see below, or during run-time. When a class has been registered with the framework the client may create instances of it through a so called factory [9].

5.4 Sample Code

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

```
#include "AbstractBlock.h"

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

In this example there exist one global instance of BlockFactory.

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

extern BlockFactory *blockfactory;

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

Now the user defined classes will be presented. First is the `Block` class that should be integrated with the framework, and the second class is `RegisterBlock`, of which a static instance is defined.

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

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

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

When registerblock_Block is allocated its constructor is executed. The constructor calls the BlockFactory object and passes on a function pointer to Create_Block, which is then used by the BlockFactory to allocate objects of the Block class. The static variable called is used as flag to prevent multiple registrations.

```
#include "Block.h"

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

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

int RegisterBlocks::called = 0;

RegisterBlocks::RegisterBlocks() {
  if (!called) {
    called = 1;
    Register("Block", Create_Block);
  }
}
```

5.5 Participants

The Factory[9] pattern is used for creating objects of the new classes.

5.6 Related Patterns

The idea behind Singleton[9] is similar to this.

6. PAL – Pålsjö Algorithm Language

Initially the Pålsjö project was aimed at designing a set of classes that would support implementation of embedded control systems. Typically there would be classes for real-time management, user-interaction and so on. The class library soon become very complicated and hard to use. Further the typical user of the framework would be a control engineer and not a experienced programmer. In that situation there were two possibilities. Either the class library could be made simpler to use, but less flexible and powerful, or the code could be generated automatically from a dedicated language. We choose to create a new language called PAL, which stands for PÅLSJÖ Algorithm Language. The language is designed to support implementations of control algorithms. The control engineer will specify an algorithm in PAL, and the compiler will generate the glue needed to incorporate the code with the framework.

In the control engineering discipline, a number of abstractions are used. These range from block diagrams to various mathematical formalisms. Here we will focus on

- Block diagrams, for defining the data flow between algorithms.
- Grafcet[7], as a way of describing sequential algorithms.
- Writing control laws, using
  - matrices for state-space design, and
  - polynomials for a transfer function approach.

6.1 Feedback

Feedback is a fundamental aspect of almost all control systems. The principle behind it is very simple:

> Observe the system to be controlled and take appropriate actions to keep the system within [some predefined] limits.

When implementing blocks in a digital computer, there are a few forces that have to be considered:

- All measurements have to be filtered so that frequencies above the Nyquist frequency are removed, otherwise aliasing will occur.
- The time delay between measurement and actuation should be kept as small as possible.

The first force can be dealt with by using an analog prefilter, but in practice it is often handled by a combination of analog and digital filters giving rise to the need of multi-rate sampling.

The second force can be dealt with by applying the **CalculateOutput-UpdateState** pattern.

6.2 Why a new language

PAL was designed as an attempt to insulate users of the PÅLSJÖ framework from the idiosyncrasies of both the framework itself, as well as the C++ language.

Since the PÅLSJÖ framework uses introspection to display the contents of blocks, the first thing a C++ programmer is burdened with is to keep the introspection code in sync with the actual implementation. Parts of this can of course be achieved by (ab)using the preprocessor, but in case of improper use, only the most experienced C++ programmers will be able to decipher the resulting error messages.

The next thing to keep in mind, is that the framework (currently) stipulates that all block inputs and outputs are accessed via an extra level of indirection, further burdening the programmer with a task unrelated to the control problem to be solved.

By introducing PAL, those tasks are handled by the compiler, which means that the programmer can be more productive. Thereby changes in the framework only triggers a re-compilation of code, instead of a [partial] rewrite. An added benefit of using a special language and associated compiler, is that multiple back-ends can be implemented, and designs can be used for more purposes, such as simulation and documentation.

Another driving force behind abandoning the C++ language, was its lack of support for data-types often used in control theory, such as matrices and polynomials. Even though this can be overcome with new classes, their use often turns out to be very confusing for an unsuspecting programmer who doesn't write C++ programs frequently.

There is also no simple way in C++ to represent Grafcets, which are commonly used in control engineering to express sequences in control systems. By introducing a language that includes the Grafcet notation, the programmer's productivity is further enhanced. A specialized language can also capture the lifetime patterns of variables in control algorithms, thereby decreasing the computational load on the control computer.

After initial experiments with the first version of the compiler, we found that in our environment the documentation aspect was more important than initially envisioned. The latest version of the compiler therefore includes back-ends to generate both HTML and $\TeX$ formatted documents as well as a prototype of a Grafcet figure generator. Due to this we have also extended the lexical analyzer to allow identifiers to include $\TeX$ formatting, making PAL programs a good candidate for documenting algorithms. By using this, the same code can be used both for documentation, simulation and control purposes, thereby reducing the risk of transcribation errors when moving an algorithm between different representations.

In its current form, PAL is not a very mature language, but it is still a much faster way to get new algorithms up and running than to code them in C++.

6.3 Decomposition and structure

During the design of a control system, the control engineer uses many different tools and abstractions. The design often starts with a decomposition into a block diagram, where blocks typically correspond to either physical parts of the process to be controlled or to functionally separate parts of the control strategy. This strategy is commonly referred to as a regulator. In this report we will only focus on the control strategy, and will not consider how the process should be designed.

Each block communicates with other blocks through **inputs** and **outputs**. The blocks also contains **state** variables, whose purpose is to let the block keep track of historical data. In order to make blocks more useful, they often contains **parameters** that do not affect the structure of the block, although they may well affect the computations done by the block.

Another way to make blocks more versatile is to design them in such a way that the sizes of internal and external variables can be decided at instantiation time by using **dimension** specifiers.

6.4 Inputs and outputs

Inputs are special variables whose values come from other blocks, often as a result of external stimuli. In the current version of PAL/PÅLSJÖ, inputs are read-only, but when back-propagation of variable limitations are implemented, they can be assigned to during the backward sweep.

Outputs are variables that propagate the results of calculations to other blocks and to the external environment. Outputs are currently write-only, but if back-propagation is used, they can be read during the backward sweep.

6.5 Parameters

A special kind of inputs are **parameters**, whose values are not directly affected by the process. Their most significant property is that they change at a much slower rate than other inputs. Often their values stay constant for the entire lifetime of a regulator.

```
$K$ := 1.0 : **parameter real**;
$T_i$ := 100.0 : **parameter real**;
```

Parameters can be changed as a result of operator intervention, change of operating mode or an auto-tuning experiment. Since parameters are seldomly changed, it is computationally efficient to use them to precalculate parts of expressions in control algorithms.

```
block PI
  $r, y, u$ : input real;
  $v := 0.0$ : output real;
  $I := 0.0, e := 0.0$ : real;
  $K := 0.5, T_i := 10000.0, T_r := 10000.0$ : parameter real;
  $h$ : sampling interval;
  $b_i = K * h/T_i$;
  $b_r = h/T_r$;

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

  backward begin
    $I := I + b_i * e + b_r * (u - v)$;
  end backward;
end PI;
```

6.6 Data types

PAL includes common data-types such as **boolean**, **integer** and **real**. In order to support the abstractions used in control engineering, it also supports **matrices** and **polynomials**. This unique feature in combination with the TEX-formatting of identifiers makes it possible to express algorithms in a way that closely match the theoretical design, thereby making the resulting programs less distant from its theoretical underpinnings than normally expected.

```
block estimator
  $y$ : input real;
  $G$ : output matrix [1..4, 1..1] of real;
  $\varepsilon$ : output real;
  $\lambda$ : parameter real;
```

```
    $\phi, \omega, \tilde{\theta}$ : matrix [1..4, 1..1] of real;
    $\phi^T$ : matrix [1..1, 1..4] of real;
    $P$ : matrix [1..4, 1..4] of real;
    $den, \lambda$ : real;

    forward begin
      $\varepsilon := y - \phi^T * \tilde{\theta}$;
      $\omega := P * \phi$;
      $den := \lambda + \phi^T * \omega$;
      $G := \omega * (1.0/den)$;
    end forward;

    backward begin
      $\tilde{\theta} := \tilde{\theta} + G * \varepsilon$;
    end backward;

  end estimator;
```

6.7 Sequencing

Often control includes sequencing, either of the controlled process or of the internal algorithms. Such sequencing is conveniently expressed by **Grafcet**, which is a simplified form of Petri-nets. In Grafcet one or more **step** may be active at the same time. Activation and deactivation of steps are done by firing **transitions** between them.

6.8 Sampling

Most control algorithms make use of the time elapsed between invocations of the algorithm, which is commonly referred to as the **sampling interval**. In PAL this time is expressed in seconds and then translated to the proper unit for the chosen target system. In PAL the name of the sampling can be chosen arbitrarily. What value the sampling interval will have during execution, is determined at system configuration time either by the user or the run-time system itself.

6.9 Dimensions

Many control algorithms can be expressed in a generic way, where only the size of constituent data structures need to be specified to create a concrete instance of the block. To make such blocks possible to write, **dimension** identifiers are used.

```
  block Sum

    $n$ : dimension;
    $in$ : array [1..$n$] of input real;
    $out$ : output real;

    forward
      $i$ : integer;
      $result$ : real;
    begin
      $result$ := 0.0;
      for $i$ := 1 to $n$ do
        $result := result + in[i]$;
      end for;
      $out := result$;
    end forward;

  end Sum;
```

7. Summary

In this report we have presented a framework for implementation of embedded controller. The working name for the framework is PÅLSJÖ and currently it is running on three different platforms, VME, Windows-NT and Solaris. It has been used both for control experiment and as a platform for real-time research. Further two patterns and one idiom used in the framework are presented. Finally a new dedicated language PAL is introduced.

8. Acknowledgment

We would like to thank Görel Hedin, Roger Henriksson, Niclas Landin and Finn Håkansson for giving valuable comments on the draft version of this report.

9. References

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

[2] K.-E. ÅRZÉN. "Lecture notes on real-time control systems." 1996.

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

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

[5] A. BLOMDELL. "The pålsjö algorithm language.". Master's thesis, Department of Automatic Control, Lund Institute of Technology, 1997.

[6] F. BUSCHMAN, R. MEUNIER, P. ROHNERT, H. SOMMERLAD, and M. STAL. *A System of Patterns-Pattern Oriented Software Architechture*. Wiley, 1996.

[7] R. DAVID. "Grafcet: A powerful tool for specification of logic controllers." *IEEE Transactions on Control Systems Technology*, **3:3**, pp. 253–268, 1995.

[8] J. EKER and A. BLOMDELL. "A structured interactive approach to embedded control." In Submitted to *The 4th International Symposium on Intelligent Robotic Systems, Lisbon, Portugal*, 1996.

[9] E. GAMMA, R. HELM, J. R., and J. VLISSIDES. *Design Patterns-Elements of Reusable Object-Oriented Software*. Adison-Wesley, 1995.

[10] J. GROSCH. "Toolbox introduction." Technical Report 25, Gesellschaft für Mathematik und Datenverarbeitung mbH, Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, 1991.

[11] INTERNATIONAL ELECTROTECHNICAL COMMISSION. *IEC 1131-3, Programmable controllers, Part 3: Programming languages*, 1992.

[12] G. JOHANNESSON. *Object-oriented Process Automation with SattLine.* Chartwell Bratt Ltd, 1994. ISBN 0-86238-259-5.

[13] E. R. JOHNSON and B. FOOTE. "Designing reusable classes." *Journal of Object-Oriented Programming*, June, June 1988.

[14] S. B. LIPPMAN. *C++ Primer.* Addison-Wesley, 1989.

[15] B. MEYER. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[16] K. NILSSON. "Private communication." 1996.

[17] H. RUTISHAUSER. *Description of Algol 60.* Handbook of Automatic Computation, Vol. 1a. Springer, Berlin, 1967.

[18] M. TÖRNGREN. *Modelling and Design of Distributed Real-time Control Applications.* PhD thesis, DAMEK Researh Group, Department of Machine Design, Royal Institute of Technology, Stockholm, 1995.

[19] N. WIRTH. *Programming in Modula-2.* Springer, New York, 3d edition, 1985.