An Architecture for Autonomous Control

Gustafsson, Kjell

1994

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--7514--SE

An Architecture For Autonomous Control

Kjell Gustafsson

Department of Automatic Control
Lund Institute of Technology
January 1994

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

Document name
INTERNAL REPORT

Date of issue
January 1994

Document Number
ISRN LUTFD2/TFRT--7514--SE

Author(s)
Kjell Gustafsson

Supervisor

Sponsoring organisation

Title and subtitle
An Architecture for Autonomous Control

Abstract

This report outlines an architecture for experimentation with autonomous control. An important part of the architecture is a DDC (direct digital control) package. The DDC package takes care of real-time issues and provides a basic algorithmic building block in which the user easily can define his/hers control algorithms.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
12

Recipient's notes

Security classification

1. Introduction

At the Department of Automatic Control in Lund there is an ongoing research project on "Autonomous Control," see [2]. The project aims at developing the techniques needed to design autonomous controllers. Most controllers in use in industry today implements a control law of fixed type in which the parameters are tuned manually. Varying operating conditions or malfunctions require retuning or other actions by the operator in order to maintain proper operation. By extending the basic control law with methods for e.g. tuning, diagnosis, smart filtering, etc., one may construct a controller that is able to handle a variety of different situations. This type of controller would be easy to commission using its internal methods for parameter tuning. When operating conditions change, the situation could automatically be diagnosed, and depending on the result, a retune or a change of control strategy could be done or suggested. In case of malfunctions, the controller would contain routines for detecting what has gone wrong. Together these techniques create a controller that is autonomous, in the sense that it can maintain satisfactory closed loop performance in a large spectrum of operating conditions without direct intervention from the operator.

The development of autonomous controllers requires both new theory as well as practical experiments. One of the goals of our project is to develop a test bench where new algorithms and ideas can be evaluated conveniently. The intention is to make this implementation general enough that the same code can be used both during algorithm development (using a simulation of the plant) and actual testing on the real plant.

The implementation is work in progress, and in this report we will give a description of the planned architecture. We will focus on the real-time software that will be used to implement the control algorithms.

2. The Architecture

The general structure of the implementation, see Figure 1, separates the system into two main parts. One part is a real-time unit, which implements all algorithms that need to be run in *hard* real-time. This unit communicates with the plant through A/D and D/A or similar interfaces. It implements the control algorithms and all signal processing (filtering, parameter estimation, etc.) that needs to be done synchronously with the sampling rate of the controller. The hardware is either a (signal) processor (e.g. Motorola MC68040, Texas TMS32040) using a VME-bus or a standard PC.

The second part of the system is a workstation that communicates with the real-time unit. The workstation is used for tasks that requires more computation and/or need only be run in *soft* real-time. Different tools, e.g. the real-time expert system G2 [6], Matlab [7], etc, can be used at this level. In a typical scenario, the software in the workstation will be used to configure and supervise the controller that executes in the real-time unit. The software in the real-time unit implements the controller and preprocesses data for analysis and plotting in the workstation. In our experiments with autonomous control we plan to implement a library of different control algorithms. These algorithms act as building blocks in the real-time unit. From the workstation we will configure different types of controllers by combining different building blocks. The software in the workstation will analyze the behavior of the controller and if necessary reconfigure.

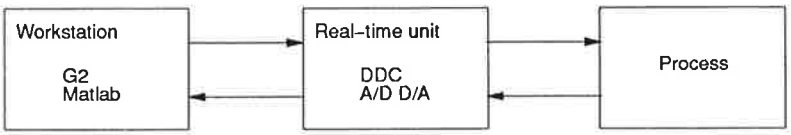

Figure 1. The general structure of the system.

Development Support

The architecture depicted in Figure 1 may be cumbersome to work with during algorithm development. It requires two different computers, a network, and interfaces to the plant. We have planned to provided two mechanisms that facilitates the use of the system. The first is to use a simulated plant. The software in the real-time unit will include primitives that make it possible to redirect A/D and D/A to an internal simulation of the plant. This makes it possible to debug a new controller without actually connecting it to the plant.

The second mechanism is to simulate the whole real-time unit in the workstation. All the software in the real-time unit is based on the real-time kernel described in [1]. By providing a workstation implementation of this kernel, it is possible to move the software from the real-time unit to the workstation. The software from the real-time unit will run in a separate UNIX process, and the software in the workstation will communicate directly with this process instead of communicating with the real-time unit. We do not plan to equip the workstation with A/D and D/A. Moreover, the implementation of the kernel on the workstation will not be able to provide hard real-time, and therefore a simulated plant (as described above) will be used when running the whole system on the workstation. With this setup, a new control application can be developed and debugged on the workstation. Once this is done, the control algorithms can be moved to the real-time unit. This will be straightforward, because exactly the same implementation of the control algorithm is used both in the workstation and in the real-time unit.

The communication between the workstation and the software in the real-time unit is based on text strings. If required, the real-time unit can operate without being connected to the workstation. The only thing necessary is to implement a separate process in the real-time unit, and have this process feed the appropriate text strings to the rest of the software.

Implementation Language

The real-time kernel [1] is well tested and we will rely on it to provide the real-time functionality needed. The kernel is implemented in Modula-2. This causes a problem because good Modula-2 compilers are not available for all the platforms we intend to use. Currently, this problem has been overcome by using automatic translation from Modula-2 to C. Although originally written in Modula-2, the exact same version of the kernel is hence available in C. Finding good C compilers is relatively easy.

It is beneficial to use an object-oriented language when implementing the algorithms that are to run in the real-time unit. The algorithms have many common properties that can be described in a base class, and a specific algorithm is implemented by deriving from the base class. We have chosen C++ for the implementation. The combination of C++ and a real-time kernel in C (automatically translated from Modula-2) has been tested in a project in the real-time systems course given at the department. The setup worked reasonably well. We have also performed initial test with a version of the kernel

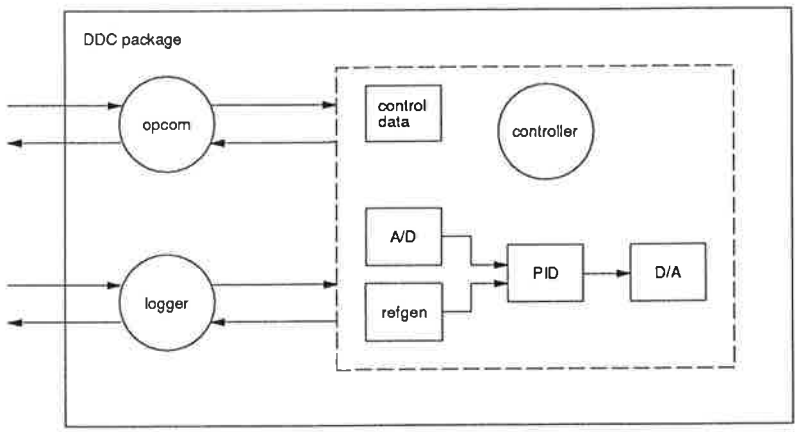

Figure 2. A PID controller implemented in the DDC package. The circles correspond to real-time processes and the rectangles correspond to algorithmic building blocks.

running under Solaris on a SUN Sparcstation. The test was promising and we count on being able to have an implementation on the workstations, which allows for a simulation of the whole system on a workstation.

3. The DDC Package

An important part of the system is the software that will run in the real-time unit. We intend to implement a simple DDC (direct digital control) package. The aim with this package is twofold:

- We want to provide an environment in which control algorithms can be tested easily. The environment takes care of real-time issues, e.g. synchronization, sampling, mutual exclusion when accessing common parameters, and provides for automatic data logging, operator communication, etc. The user only supplies the algorithm.
- All control algorithms share certain basic properties. We want to provide a basic building block that implements these properties. When experimenting with a new algorithm the user derives from the basic description and specifies only the new properties.

To get a general overview of the DDC package, consider the simple example in Figure 2. The Figure depicts the processes and building blocks present when implementing a PID controller. Two processes, `opcom` and `logger`, are always present in the DDC package. `opcom` handles the communication with the operator, which may be either a user or another program. The operator sends text commands asking for different actions to be performed, and `opcom` decodes these commands and executes them. A typical request would be to e.g. create a new PID controller, to connect the output from one block to the input of the next, or to change a parameter in a specific block. The second process, `logger`, handles data logging. A block that wants to make data available for external programs, e.g. Matlab in the workstation, declares this by calling routines in `logger`. `logger` then handles the data collection and the communication with the external program.

A controller (there may be many present simultaneously) consists of a control process (`controller` in Figure 2) and a set of connected algorithmic building blocks, e.g. A/D, refgen, PID, and D/A in Figure 2. The control

process, the building blocks, and the interconnections are all created by calls to opcom. Data that are common for the whole controller, e.g. sampling interval, is contained in the data block `control data`. The process `controller` will make sure that the different building blocks get executed in the correct order and with the correct sampling interval.

The opcom Process

The opcom process is the main link between the DDC package and an external operator. The interface is text based. opcom reads text strings from an input channel. These strings contain the commands that are to be executed, and opcom tells the result of the action by sending a string on an output channel. By using this type of interface the DDC package can be used in several different ways: the commands could come from a keyboard connected to the real-time unit, they could be sent from the workstation over the network, or they could be fetched from a file.

The commands that can be sent to opcom can be divided into a few different categories:

- creating and deleting building blocks, assigning blocks to a control process
- starting and stopping a controller, changing its priority or sampling interval
- connecting and disconnecting blocks
- obtaining information from a block
- changing parameters in a block

The opcom process has access to a library of different building blocks, e.g. A/D and D/A converters, signal generators, different types of filters and control algorithms, etc. These blocks can be instantiated and named by sending commands to opcom. Once several blocks have been created, they can be connected to form a complete controller, see Figure 2. When restructuring the controller a block may be disconnected and deleted.

When running a set of blocks as a controller, one uses opcom to create a control process. The different blocks are assigned to this process and the control process executes them according to the specified sampling interval. Associated with the control process there is a data block (`control data` in Figure 2) containing information that is relevant to the whole set of blocks, e.g. sampling interval, whether the controller is stopped or running, priority, etc. opcom accepts commands to access and modify this data.

Internally, opcom administrates a data structure that keeps track of how the different building blocks are connected. The control process uses this information to execute the algorithms in the different blocks in the correct order.

Through opcom one can get information about the names and internal data in the different building blocks. The basic description for a building block contains appropriate routines that opcom can use to access this information. There are also routines that can be used to modify the data.

The logger Process

Klas Nilsson has suggested a convenient way [8, 9] for data logging in a real-time control system. We intend to use his methods in the DDC package.

The data logging is implemented by the `logger` process. `logger` provides two interfaces: one to the building blocks in the controller and one to an external user. Each building block that wants to make a signal available for

logging, calls the routine `submit` in `logger` with the signal and its name. Every time a value is to be stored, the `update` routine in `logger` is called. This will make `logger` collect the new value and store it internally.

Through the second interface an external user can ask what signals are available and get a list of all signals that have been submitted. The user can, by sending commands to `logger`, decide how and in what form the different signals should be stored. `logger` will send the data in the requested form to the user. Nothing will be stored until the user actually asks to get access to a certain signal. This is convenient, because it allows the controller to make many signals available without significant overhead. The communication between `logger` and the user is similar to the text interface to `opcom`.

The `controller` Process

A controller consists of a set of interconnected building blocks. `opcom` provides information about how the blocks are connected and the control process uses this information to execute the algorithms in the blocks in the correct order. At each sampling instance the blocks are visited twice. In the first sweep the outputs from each block are computed and propagated to the next block. In the second sweep the state in each block is updated. In the second sweep we also check if `opcom` has asked to change any parameter values in the block. If that is the case, the new value is moved to the internal data structure and will be used in the calculations in the next sample.

The reason for having two sweeps over the blocks is twofold:

- We want to minimize the delay from the sampling time until the control signal is put out through the D/A converter. Only a minimum of calculations should be done in the first sweep.
- When updating the state in a block one may need the value from the output of a different block. Consider for example a PID controller used in a cascade loop. Before being able to update the I-part in the outer PID we need access to the saturated control signal. This depends on the inner PID, and the only way to handle the situation correctly is to first calculate outputs for both PIDs and later revisit them and update their state.

For consistency, the parameters inside a block should not change between the two sweeps. That is the reason why we check for new parameters after having calculated the outputs as well as updated the state in a block.

Comparison with `sim2ddc`

The DDC package is very similar to the system described in [3, 4, 5]. Ola Dahl's program `sim2ddc` takes control algorithms expressed in the Simnon language and automatically translates them to building blocks in Modula-2. His program provides an environment where these building blocks can be operated and used as real-time controllers. There are a few differences between `sim2ddc` and the package we intend to develop:

- The algorithmic building blocks have many properties in common, e.g. there must be routines for setting parameters, reading inputs, setting outputs, asking for parameter names, etc. `sim2ddc` takes the control algorithm and automatically merges it with code for doing these general tasks. The actual control algorithm constitute a very small part of the generated Modula-2 code. This works fine for a controller with a structure that is easy to express using the Simnon language. For other algo-

rithms it is possible to generate the Modula-2 code corresponding to an "empty" Simnon system and later add the algorithm part, but the technique becomes cumbersome due to the complexity of the automatically generated Modula-2 code. We intend to use an object-oriented language and describe the common properties of the building blocks in a base class. A new building block is constructed by deriving from this base class and specifying what is new and unique. This construction allows both for automatic translation from e.g. Simnon and convenient direct implementation of a new building block.

- A typical building block will have inputs, outputs and internal parameters. In the case of a controller, the inputs may be measured values and reference values, the output the calculated control signal, and the parameters different coefficients used in the control law. The inputs and outputs will be connected to other blocks and the parameters will be set by the operator through the `opcom` process. In contrast to `sim2ddc` we plan to allow parameters to be set both through `opcom` as well as directly by output signals from a different block. This functionality is important when e.g. using a standard RST controller in an adaptive framework, where the controller parameters should be set by the block that implements the controller design.
- We plan to allow for vector signals and parameters. This is important when the number of signals in an interface may vary. An example is an estimator that fetches a data vector from the controller. The size of this vector depends on the order of the estimated model. This must be possible to change without having to stop the system and disconnect or connect signals. Similarly, sometimes several signals or parameters need to be handled in a block. One example is when changing parameters in an RST controller. All parameters need to be changed simultaneously in order not to get inconsistent behavior. A straightforward way to handle this is to always have vector signals and parameters, with the default case being a one element vector.
- The different blocks in a controller, as the one depicted in Figure 2, have to be synchronized so that signals flow from input to output without unnecessary delay. `sim2ddc` uses one separate process for each block, while we will use one process for a set of building blocks. This process will keep track of how the different blocks are connected and make sure that they get executed in the correct order.

4. The Algorithmic Building Blocks

The DDC package outlined above provides an environment in which different algorithmic building blocks can be combined to form a controller. The package takes care of the real-time aspects of the code and also handles a lot of the administration. This makes the task of implementing a controller much less tedious than writing software from scratch.

The actual control algorithm is described in one or several of the building blocks. These blocks have a well defined structure and act like an interface between the DDC package and the algorithm itself. We will use an object oriented language (C++) for the implementation and all the basic properties of the block will be described in a base class. When implementing the algorithm we derive from this base class and "fill in the empty slots."

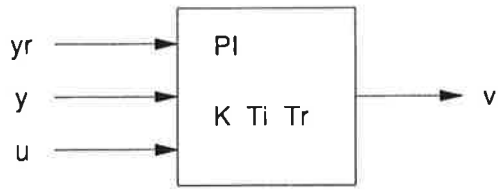

Figure 3. A PI controller.

To demonstrate the structure and some of the properties of these building blocks we will use a PI controller as an example. A PI controller with anti-windup can be expressed on discrete-time form as

$$e(k) = y_r(k) - y(k) \tag{1}$$

$$v(k) = Ke(k) + I(k) \tag{2}$$

$$I(k+1) = I(k) + b_i e(k) + b_r\,(u(k) - v(k)) \tag{3}$$

where y_r is the reference value, y the measured process output, v the calculated control signal, and u the control signal after saturation in the actuator. The coefficients b_i and b_r are defined as

$$b_i = \frac{Kh}{T_i}, \qquad b_r = \frac{h}{T_r}, \tag{4}$$

where h is the sampling interval, K the controller gain, T_i the integration time, and T_r the anti-windup time constant.

The PI controller may be viewed as a block (see Figure 3) taking y_r, y, and u as input and producing v as output. The block contains the parameters K, T_i, and T_r. The sampling interval h is propagated from the control process executing the PI block.

The Algorithm

The algorithm is divided into two parts. The first part is implemented in `CalculateOutput` and contains the minimum amount of calculations needed to produce the output. This routine is called by the control process during the first sweep over the building blocks. The second part is implemented in `UpdateState` and contains the rest of the calculations. `UpdateState` also performs all calculations that can be done to prepare for the next sample.

In our PI example, Equations 1 and 2 would be implemented in `CalculateOutput` and Equation 3 in `UpdateState`.

Normally, `CalculateOutput` and `UpdateState` are called once every sample with a sample rate set by the control process. The sampling interval h is propagated from the control process to the building blocks, and h can be used directly in the calculations. To allow for synchronized multirate, e.g. decimation/interpolation filters, each block contains a parameter N, and the controller process will only call `CalculateOutput` and `UpdateState` every N:th sample. The default value for N is 1, and the value of N is taken into account when propagating h down to the block. Using this mechanism we will be able to implement synchronous multirate systems. Asynchronous systems are implemented by assigning the building blocks to different control processes.

The Parameters

Most algorithms contain parameters that the user should be able to change. We intend to provide a special class `Parameter` implementing parameters. This class keeps track of e.g. the name, the default value, etc, of the parameter.

When implementing the PI block we will declare variables named `K`, `Ti` and `Tr` of type `Parameter`. The base class for the algorithmic building blocks (the one from which the PI block is derived) knows about the parameter class. It contains routines that `opcom` can use to ask how many parameters the building block contains, what their name is, what value they have, as well as routines for setting new parameter values. The base class is made aware of a parameter when running the constructor for the parameter. The constructor takes a reference to the block it belongs to and can call the appropriate routines in the base class. All this is automatic. The only thing the user needs to do is to declare the parameter. Once this is done, the parameter will be available to `opcom` and can also be used in `CalculateOutput` and `UpdateState`.

A parameter will be accessed both from `opcom` as well as from the control algorithm. The methods in the base class that access the parameters will automatically provide mutual exclusion.

A flag will be set when `opcom` changes the value of a parameter. At the end of the sampling interval, after having called `UpdateState`, the control process will check this flag. If there has been a change, the new value is moved to the internal data set, and will be used in the next sample. If several parameters need to be changes simultaneously one may call an access routine that records the new parameter value but does not set the change flag. After all parameters have been changed a call is made to a special routine that sets the flag.

Often the parameters are not used directly in the calculations in the control algorithm, and the controller process will automatically call a conversion routine that the user may define. By default this routine is empty. In the PI example we may use this conversion routine to calculate b_i and b_r from the new values of K, T_i, and T_r.

The Inputs and Outputs

The inputs and outputs to a block are declared in a similar way as parameters. There are special classes implementing inputs and outputs, and when a variable of this type has been declared in a block, the base class of the building blocks provides methods for manipulating them. There are methods for asking how many inputs and/or outputs are present, what their names are, and what value they have.

An input has a default value that will be used if the input is not connected to another block. This is convenient for e.g. y_r in the PI example. Either we can connect y_r to a block computing reference trajectories, or we can leave it unconnected and change its value through `opcom`.

An output knows about what input signals (or parameters) it has been connected to. After the routine `CalculateOutput` has been executed the new values of the outputs from the block will be propagated to the these inputs. There will be two ways of propagating the value. If the other block is administrated with the same control process there is no need for mutual exclusion, but if the block is run by a different process (maybe with a different sampling interval, e.g. a controller and an estimator where the estimator runs with a slower sampling rate than the controller) the propagation has to be done with mutual exclusion. An input variable provides methods for both these cases, and it is only a matter of calling the correct one. This choice is done automatically.

Logging Signals

Some of the signals in a block may be of interest for an external user, e.g. for debugging or for data processing in the workstation. Data logging is taken care

of by the `logger` process. The base class for the blocks contains a routine that can be called to tell that a variable is to be available for data logging. This is typically done in the constructor of the block. In the PI example we may call this routine with e.g. I and u. Once the user have called this routine, the base class will inform `logger` about the variable. Every sample, after `UpdateState` has been executed, the new value will propagated to `logger`. Again, the user need only make one call and the rest is handled automatically.

5. The Next Step

We have outlined an architecture for experimentation with autonomous control. An important part of this architecture is the DDC package and it's predefined structure for algorithmic building blocks. A logical next step is to implement this package and start using it to gain experience. It is, however, of vital importance that the structure and interface of the DDC package provides the correct functionality and is easy and straightforward to use. We recommend that a detailed specification of both the external and internal interface to the DDC package as well as the base structure of the building blocks is made before any coding is done. When specifying these interfaces one may consider a few different scenarios (e.g. PID, cascade PID with different sampling intervals, Kalman filter and state feedback, adaptive controller with separate (asynchronous) blocks for estimation and design, relay autotuning, etc) and make sure that they map onto the specified structure in a natural way.

6. References

[1] L. Andersson. "Implementation of a real-time kernel." Report ISRN LUTFD2/TFRT--7511--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, November 1993.

[2] K. J. Åström. "Autonoma regulatorer – en projektansökan till NUTEK." 1993.

[3] O. Dahl. "Generation of structured Modula-2 code from a Simnon system description." Report TFRT-7416, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, February 1989.

[4] O. Dahl. "SIM2DDC—User's manual." Report TFRT-7443, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, January 1990.

[5] O. Dahl. "An interactive environment for real time implementation of control systems." In Barker, Ed., *Computer Aided Design in Control Systems*, Preprints of the IFAC Symposium, Swansea, UK, Oxford, UK, 1991. Pergamon Press.

[6] Gensym. *G2 Reference Manual, Version 3.0.* Gensym Corporation, 125 CambridgePark Drive, Cambridge, MA 02140, USA, 1992.

[7] The MathWorks Inc., Cochituate Place, 24 Prime Park Way, Natick, MA 01760, USA. *MATLAB – High-Performance Numeric Computation and Visualization Software*, 1992.

[8] K. Nilsson. *Application Oriented Programming and Control of Industrial Robots.* Lic Tech thesis ISRN LUTFD2/TFRT--3212--SE, Department of

Automatic Control, Lund Institute of Technology, Lund, Sweden, June 1992.

[9] K. Nilsson 1993. Private communication.