Designing Event-Controlled Continuous Processing Systems

Class 325

Hans Peter Jepsen, Danfoss Drives
Finn Overgaard Hansen, Danish Technological Institute

ABSTRACT

Many embedded systems, such as systems in the process control and measurement domains, are designed to perform continuous processing of input values and to produce and maintain corresponding output values. At the same time these systems have to react on events that reconfigures the processing algorithms.

This paper will present an architectural model that separates the event handling and the continuous data processing. The event handling part selects the current active algorithm for the continuous processing part. The paper will also present an outline of an architecture for this kind of system and describe how the combination of several Design Patterns (Gamma et. al.) have been useful for building an object-oriented architectural model described with UML notation. The results presented will be relevant for most event-controlled continuous processing systems.

AUTHOR BIOGRAPHY

Hans Peter Jepsen is software technology leader at Danfoss Drives in Graasten, Denmark, responsible for methods and tools used in the development of frequency converters. As part of this job he participated in the Danish COT research project where he was a member of the industrial team looking at the architecture of embedded systems. He can be contacted at hans_peter_jepsen@Danfoss.com.

Finn Overgaard Hansen is consultant in object-oriented methods at the Danish Technological Institute. As part of this job he was project manager for the industrial team working on the architecture of embedded systems in the Danish COT research project. He has more than 20 years of experience in software development and software engineering methods. He can be contacted at finn.overgaard.hansen@teknologisk.dk.

INTRODUCTION

This paper presents some architectural concepts that we believe are relevant to most event-controlled continuous-processing systems. They are the results of a pilot project inside the Danish research project Centre for Object Technology (COT)[1], where an object-oriented model and a corresponding prototype implementation in C++ for the central parts of a VLT®[2] frequency converter were developed.

[1] The ***Centre for Object Technology (COT)*** was a Danish three-year project collaboration project with participants from industry, universities and technological institutes. The overall objective of Centre for Object Technology (COT) was to conduct research, development and technology transfer in the area of object-oriented software construction, for more details see [COT]. The project was financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry. The activities in COT are based on the actual requirements of the participating industrial partners. COT was organised in six industrial cases, and the pilot project, described in this paper was carried out inside case 2 entitled "Architecture of embedded systems". The purpose of COT case 2 was to experiment with the introduction of object technology in embedded systems. The participants in this pilot project came from University of Aarhus, Danish Technological Institute and Danfoss Drives

[2] VLT® is the trademark for frequency converters produced by Danfoss Drives.

A major breakthrough in the modelling of the frequency converter was the formulation of a conceptual model, which we call "the two-part architectural model". A major inspiration for this model was the architectural style known as "Process Control" from [Shaw&Garlan96] along with the oscilloscope and cruise control examples from the same book.

Although the two-part architectural model may seem - and in fact is - very simple, it has been of great help since we formulated it. Further we think, that this model will be relevant for many other types of embedded systems, such as systems in the process control and measurement domains, which are designed to perform continuous processing of input values and to produce and maintain corresponding output values. At the same time these systems have to react on events that control and reconfigure the processing algorithms.

In this paper we will present the two-part architectural model. Further we will present, how this architectural model has been realised in an object-oriented architecture, and finally describe the benefits we have experienced with this model.

FREQUENCY CONVERTERS

A **frequency converter** is an electronic power conversion device used to control the shaft speed or torque of a three-phase induction motor to match the need of a given application. This is done by controlling the frequency and corresponding voltage on the three motor cables. A frequency converter is often called a 'drive'.

Figure 1 shows a typical frequency converter application used in connection with a ventilator to maintain a desired room temperature.

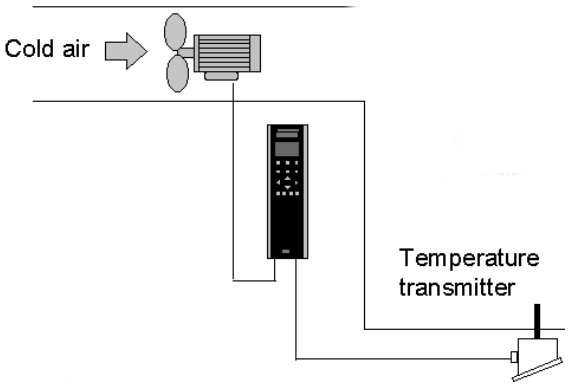

Figure 1. A drive used for controlling the speed of a ventilator.

It is possible to save a lot of energy by reducing the ventilator speed as much as possible while maintaining the desired temperature. This is a normal closed-loop feedback control problem. In this application, the ventilator 'windmills', or runs free, when the motor is not powered. Therefore, before applying power, the ventilator must be stopped or caught.

The following scenario will give an idea of the working of the software that this application requires. Initially the motor is coasted (i.e. no output is applied to the motor) and the fan is wind milling. When the frequency converter receives a *"start"* command, it starts performing an algorithm, that we call *"catch spinning motor"*. Whenever this algorithm has detected the motor speed and adjusted its output to the motor, we have reached a situation where the frequency converter is controlling the motor. The algorithm sends an event with this information. The algorithm is then replaced with a *"process closed loop"* algorithm, which maintains the desired temperature.

In the example above we see three *modes of steady-state operation*, namely *"coast"*, *"catch spinning motor"* and *"process closed loop"*. We also see, that the shift from one type of control to another happens as a result of an event, where the event can come from outside (the *"start"* command) or from the control algorithm *("spinning motor caught"*).

We want to stress that this is only a simple example. Advanced frequency converters can be used in a large number of applications and have a high degree of complexity.[3] For products made by Danfoss Drives the number of "modes of steady-state operation" is between ten and twenty, the number of the external events is at the same scale, and the number of internal events a bit higher. To adapt the frequency converter to the motor or the application, the user configures it, by applying values or choices to about 150 configuration items (called "parameters").

THE TWO-PART ARCHITECTURAL MODEL

As part of our first steps in developing the software architecture, we have formulated a model, that we have called the "the two-part architectural model".

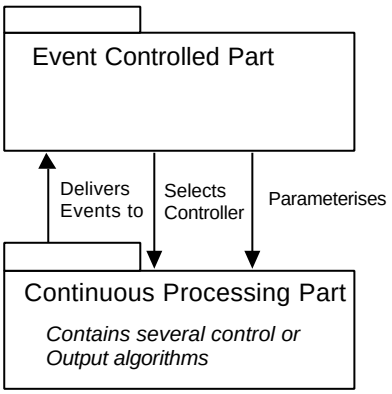

Figure 2 – Overview of the two-part architectural model

Here is a very short description of this model – shown on Figure 2:

- *The Event-Controlled Part* of the system is responsible for handling external and internal events - occurring asynchronously with the continuous data proc-

[3] As an example, the software for the VLT® 5000 series of frequency converters made by Danfoss Drives consists of approximately 150 000 lines of C code. The hardware design is based a Motorola MC68331 microcontroller for software processing and an ASIC that, measured by the number of gates, has a complexity equivalent to that of an 80386 processor.

essing. (The configuration data for the system is also placed here since configuration changes are event controlled.)

- *The Continuous Processing Part* of the system is responsible for the continuous data processing, e.g. process control and measurement.
- The event-controlled part configures (parameterises) the individual algorithms in the continuous data processing part, but also determines which one of the algorithms has to be the active control algorithm. On the other hand the continuous data processing part can produce internal events that have to be handled in the event-controlled part.

The continuous processing part contains the hardware and software controlling the motor in the "modes of steady-state operation". In the software case the processing is handled by periodically activated software, driven by periodic interrupts - since no real continuous (analogue) processing can happen in software.

Where we tried to apply it, we found that it was fruitful to use the process-control model [Shaw&Garlan96] to describe each of the *"modes of steady-state operation"*. When we tried to make a use case model for our system, we had initially problems with the specification of the use cases for the continuous processing part of it. Later on we discovered that the description of these use cases could be based on the process-control model.

Some of the control problems we have are simple open-loop problems, but most of them are closed-loop problems, either feedback or feed-forward.

Looking at the internals of the control algorithms (the controller part of each of the process control problems), we have found, that this algorithm normally has a simple dataflow structure, which can be described using the "Pipes and Filters" style [Shaw&Garlan96].

A constraint, that the continuous processing part must handle, is when replacing one control/output algorithm with another, there normally is a demand for "bumpless transfer", i.e. as smooth a shift as possible from one control algorithm to another, in terms of disturbances on the motor shaft.

The event-controlled part is responsible for high-level motor control, i.e. how to safely bring the motor from one steady situation to another – both situations that the user recognises as steady – based on the events it receives. It is important to assure that the system is safe, i.e. no obscure combination of events will make it unsafe.

Fan control revisited

To make it concrete we will illustrate the two-part model with the fan-control application described above.

The process-control model - in this case a closed loop control – is shown on Figure 3 with the relevant terms for this application. Figure 4 gives some useful definitions.

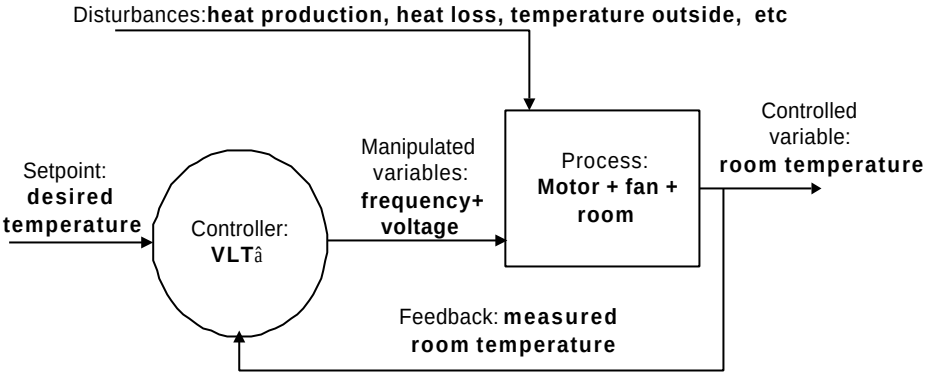

Figure 3. Closed loop feedback process control of room temperature

---

Process variables. Properties of the process that can be measured; several specific kinds are often distinguished.

Controlled variable. Process variable whose value the system is intended to control.

Input variable. Process variable that measures an input to the process.

Manipulated variable. Process variable whose value can be changed by the controller.

Set point. The desired value for a controlled variable.

Open-loop system. System in which information about process variables is not used to adjust the system.

Closed-loop system. System in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.

Feedback control system. The controlled variable is measured and the result is used to manipulate one or more of the process variables

Feedforward control system. Some of the process variables are measured and anticipated disturbances are compensated for without waiting for changes in the controlled variable to be visible.

Figure 4. Process Control Definitions – from [Shaw&Garlan96]

The responsibility of the software required for this application, i.e. the internal structure of the *"Controller: VLT"* "bubble", is represented in Figure 5 as a block diagram where the Pipes-and-Filters structure can be seen with each processing block representing the filter component.

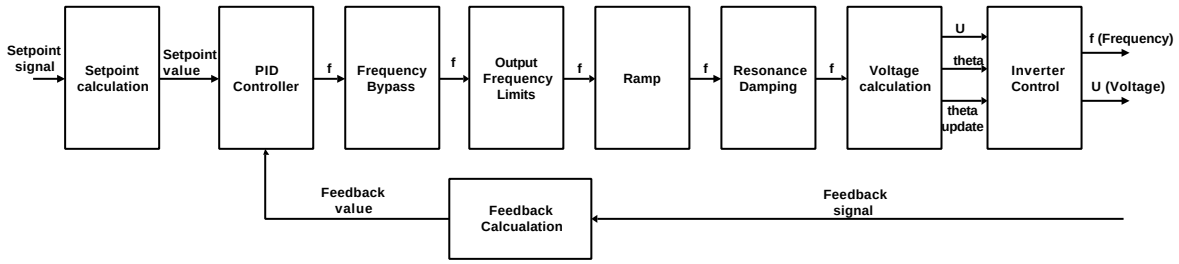

Figure 5. Block diagram of the internals of the controller in Figure 3

Let us add a few comments to some of the processing blocks on Figure 5. The four blocks in the section from *Frequency Bypass* to *Resonance Damping* can change the desired output frequency, based on configuration values provided by the user. For ex-

ample, the *Ramp* block will limit the changes (slope) in the output frequency to a certain value, set by the user. *Inverter Control*, implemented in the ASIC, is responsible for the 'continuous' production (PWM generation) of a voltage vector, which has the desired amplitude and frequency.

For the "*catch spinning motor*" case, a diagram similar to Figure 3 (a feed-forward diagram however) and also a block diagram similar to Figure 5 can be made.

We are now ready to examine the two-part model introduced above by looking at a very simplified "picture" (Figure 6) of the internals of a frequency converter.

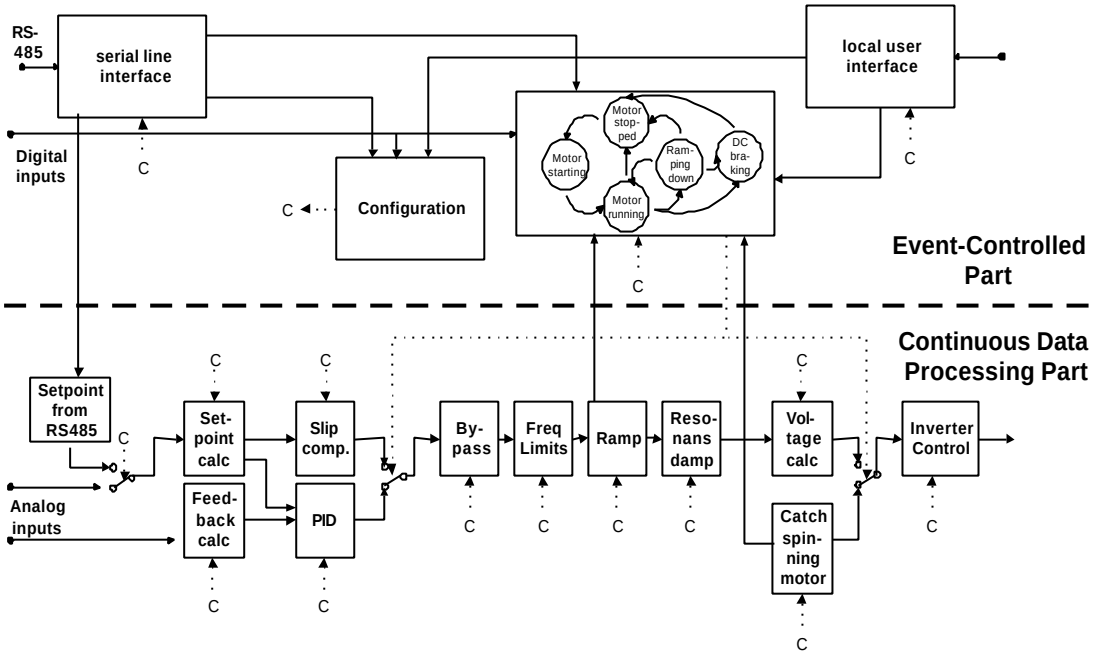

Figure 6. Architecture view of frequency converter software - very simplified

The part *below* the dashed line represents the part of the system that is responsible for continuous data processing, such as process control and measurement. In a given situation, determined by certain setting of the 'switches', the data is flowing through a subset of the blocks. Some of the characteristics of this part are that it has a data flow-oriented architecture and is normally driven by periodic interrupts.

The part *above* the dashed line represents the part of the system responsible for handling events occurring asynchronously with the continuous data processing. State machines have become important design tools for this kind of design, and configuration data will typically also be held in this part.

The event-controlled part not only configures the blocks in the continuous data processing part, represented with the arrows marked "C" to the blocks, but also determines the signal path in this part, shown by the arrows to the switches. On the other hand, the continuous data processing part can produce events that have to be handled in the event-controlled part, shown by the arrows leading back to the upper half of the diagram.

To illustrate the two-part architectural model we will show, how elements from a frequency converter can be mapped onto this model.

To show the mapping of frequency converter-software required for the ventilator application onto the two-part architecture we will revisit the scenario above: Initially, the motor coasts, with no power applied to the motor, and the ventilator is wind milling. When the event-controlled part of the drive receives a *start* command, it sets the switches in the continuous part to perform an algorithm that is called *"catch spinning motor"*. Whenever this algorithm detects the motor speed and adjusts its output to the motor, we have reached a situation where the drive is controlling the motor. The algorithm sends an internal event to the event-controlled part with this information. The event-controlled part then changes the settings of the continuous part to execute a closed-loop control algorithm.

We get a similar model by looking at an application (Figure 7) where the speed, e.g. of a conveyer belt, is controlled without speed feedback. Instead, the speed is estimated by measuring the motor's current consumption. In this case, the process control algorithm is an example of a closed-loop feed-forward speed control and the algorithm is called *"speed open loop"*.

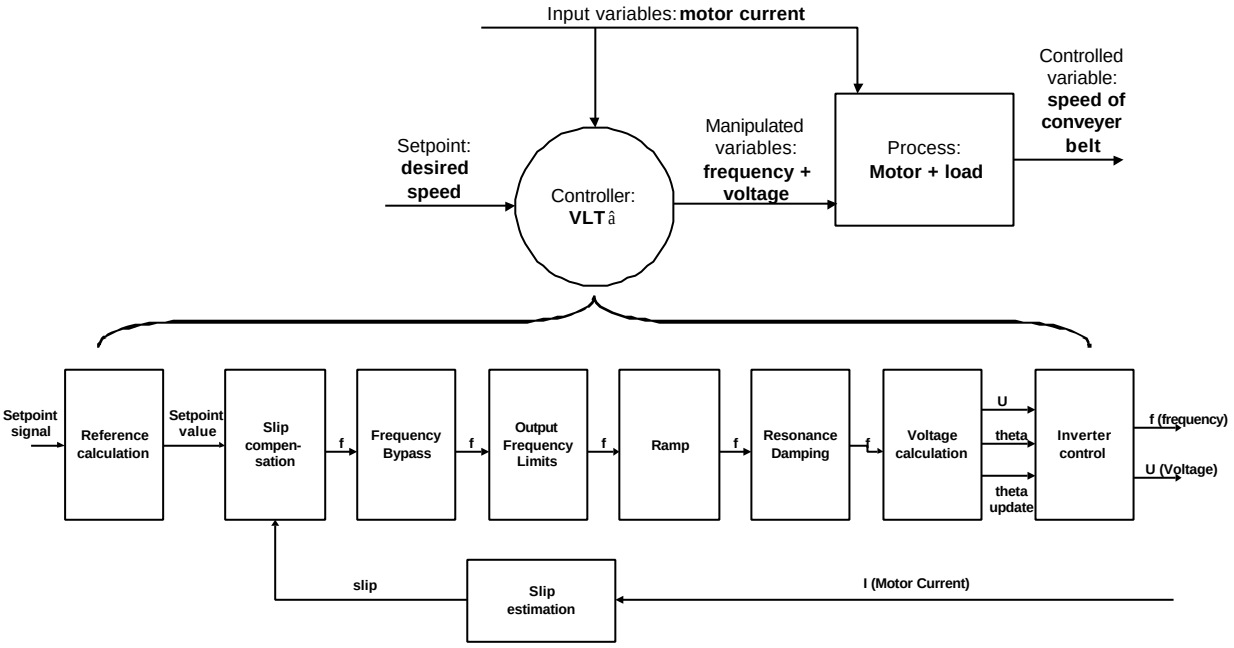

Figure 7. Process model and block diagram of a VLT's algorithm *"Speed open loop"*, actually a closed-loop feed-forward algorithm

Comparing Figure 5 and Figure 7 shows that some of the processing components are common for the two shown processing modes.

Architectural styles in the two-part model

The two-part model described above builds on and encapsulates several architectural styles. The continuous processing part contains multiple continuous data processing algorithms, and in case they are control loop algorithms they can be described accord-

ing to the *process control* architectural style. Further each of these algorithms can be divided in components that are connected according to *pipes and filters* style. Finally the event-controlled part encapsulates a *state-based control* style.

Why not rely just on the classical process control architectural style? The process control architectural style describes a single process control situation, and does this very well. However many systems must be able to handle a number of these process control situations, and when shifting between these, there often is a demand for "bumpless transfer". The two-part model tries to cover multiple controllers and bumpless transfer – handled by continuous processing part. Further "continuous data processing" does not have to be process control. Measurement is another example so other types of continuous data processing are also covered.

The classical process control architectural style contains handling of discrete events. We have however chosen to place the handling of events external to the process control algorithms. The reason for this is that we believe this gives us a better overview of the system.

OBJECT-ORIENTED DESIGN OF THE TWO-PART ARCHITECTURAL MODEL

After introduction of the pilot project domain and a high level description of the two-part model, the following sections will describe how this model was realised with an object-oriented design with extensive use of Design Patterns [Gamma95] and documented using the UML notation [UML97].

Figure 8 shows a UML package diagram of the two-part model for the informal architectural diagram presented on Figure 6. The Stick-man notation represents the external actors connected to the system.

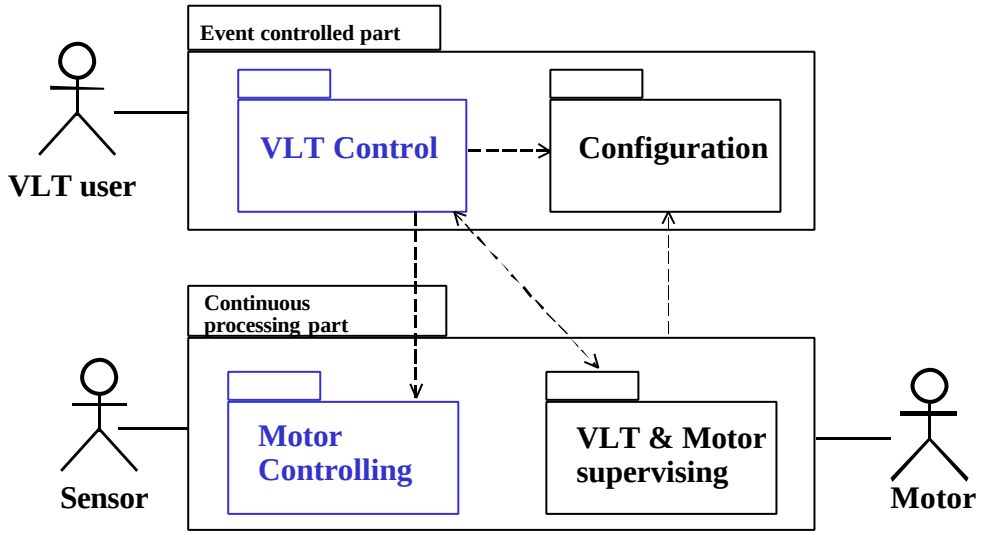

Figure 8. UML package diagram showing the two-part model

The event-controlled package is further divided into a VLT control package and a configuration package both affected by discrete events or commands from the VLT user. The continuous processing package is divided into two different continuous-processing packages. One is used to control the motor and the other is used for supervision purposes. Both are dependent on a configuration package.

The continuous processing part

The continuous processing part is designed as a combination of the Strategy Design Pattern [Gamma95] combined with the Pipes-and-Filters architecture pattern [Shaw &Garlan96, Buschmann96]. Figure 9 shows the general structure for the Strategy Pattern. In this pattern, the Context object delegates a part of its algorithm to its connected strategy object. This solution makes it possible for the Context object to change its algorithm by changing the ConcreteStrategy object pointed to by the Context object.

The actual implementation of the Strategy Pattern in the pilot project is shown on Figure 10. The dashed ellipse is a UML collaboration symbol used for showing the roles played by the actual classes in realisation of the Strategy design pattern. The different strategies in this implementation correspond to the different VLT operation modes, such as speed open-loop processing or closed-loop processing. The *AlgorithmInterface* function in the Strategy Class is here called *generateOutput* and here only defined and implemented in the *OutputController* superclass.

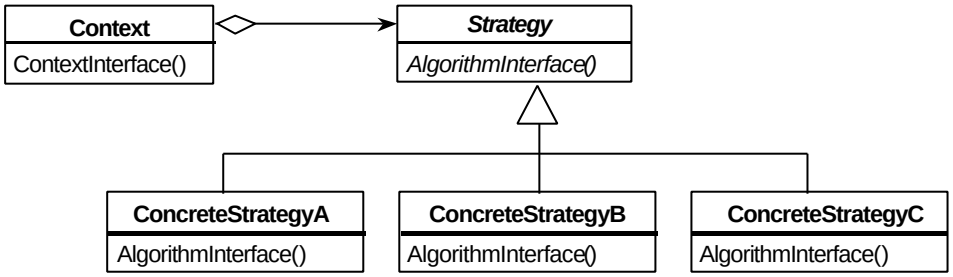

Figure 9. Structure for Strategy pattern

The function *generateMotorOutput* in the Class *MotorOutputGenerator* is activated continuously by a timer interrupt every 2ms.

```
MotorOutputGenerator::generarateMotorOutput()
{
      theActiveOutputController->generateOutput();
}
```

This function delegates the generation of motor output data to the current active output controller object's *generateOutput* method, where each of these objects will use the inherited *generateOutput* operation defined in the superclass.

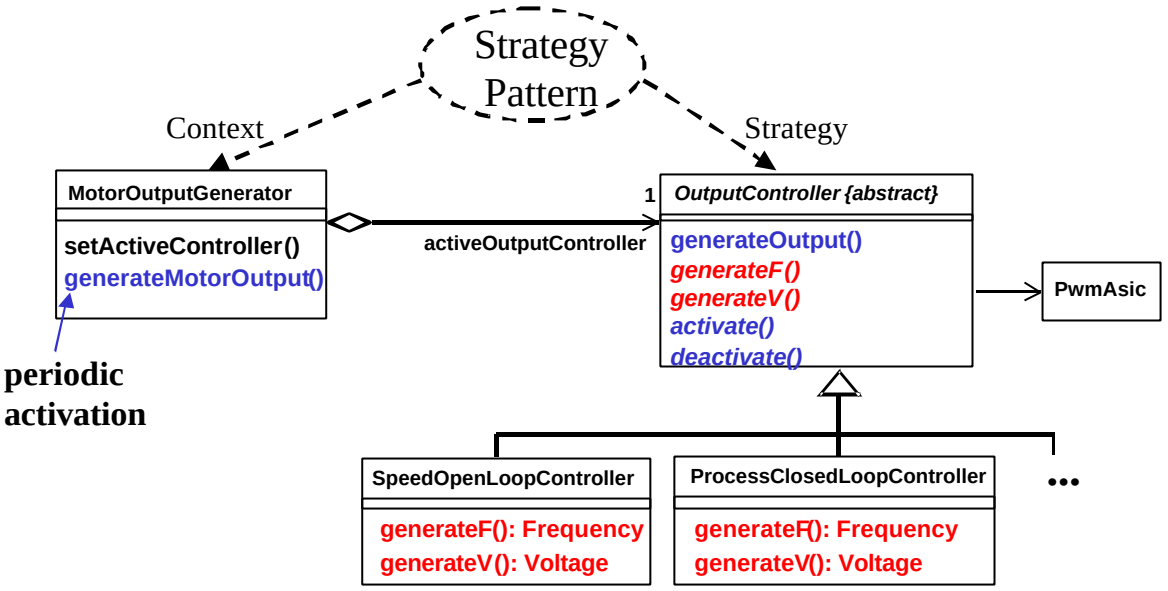

Figure 10. Class diagram showing realisation of Strategy Pattern

The *generateOutput* method calculates the control parameters and sends them to the ASIC controlling the motor:

```
OutputController::generateOutput()    // template method
{
      frequency = generateF(); // pure virtual function
      voltage = generateV(); // pure virtual function
      thePwmAsic->output(frequency,voltage);
}
```

In this implementation, the strategy method *generateOutput* is also a Template Method [Gamma95], where the two functions *generateF* and *generateV* are pure virtual (C++) functions that must be implemented with code in the concrete subclasses of the *OutputController* superclass.

The VLT system requires as smooth a shift as possible from one type of controller to another. This is called bumpless transfer. The solution to this requirement was to enhance the standard Strategy Pattern with functionality to enable this bumpless transfer between the operation modes. This is implemented with the addition of two virtual operations *activate* and *deactivate* in the *OutputController* class. These operations are defined in the actual *OutputController* subclasses.

The configuration of the "*strategy*" is performed by calling the operation *setActiveController* in the class *MotorOutputGenerator*, with the new *OutputController* object as a parameter. The current controller is deactivated and the returned information is used to initialise the new controller object, which starts with the same conditions as the previous controller:

```
MotorOutputGenerator::setActiveController
                        (OutputController* newController)
{
      controllerInfo=
      theActiveOutputController->deactivate();
      theActiveOutputController = newController;
      theActiveOutputController->activate(controllerInfo);
}
```

The different VLT operational modes require identical processing components for the controlling parts of the overall algorithm. Use of the Pipes-and-Filters architecture pattern fulfils this requirement. It results in a flexible design where the algorithm components can be reused and easily configured to the actual needs of a particular operational mode.

The Pipes-and-Filters architecture divides an algorithm into several sequential processing steps, which are connected by data flows. The output of one step is the input to the next step. Each processing step is implemented by a *filter* component. Each filter component is connected to another filter component by a *pipe* mechanism. Each filter processes its input data and forwards it to the next filter via the pipe.

Figure 11 shows an example of a UML object diagram for a speed open-loop controller with the different filter components modelled as objects. Compare it with the block diagram in Figure 6. This is an example of a situation where a UML object diagram is very useful to illustrate the runtime situation where the corresponding UML class diagram only shows the static code organisation.

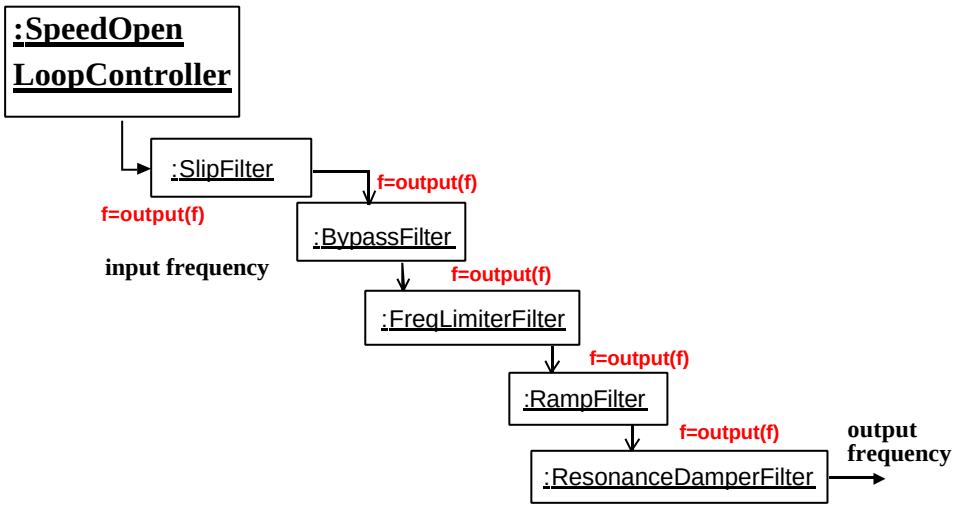

Figure 11. Object diagram for “Speed Open Loop” operation mode controller

The corresponding object diagram for a closed-loop controller, shown in Figure 5, will be exactly the same, except that a *PidFilter* object that encapsulates a PID algorithm will replace the first object.

Figure 12 shows the static structure of the corresponding class diagram implementing the Pipes-and-Filters pattern.

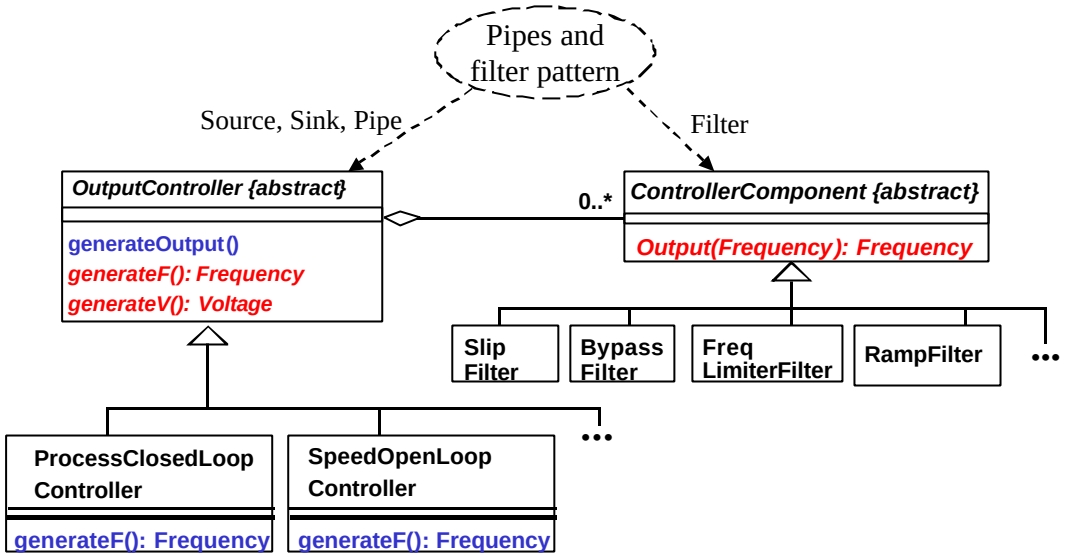

Figure 12. Class diagram, showing realisation of Pipes and Filter pattern

The class *OutputController*, the strategy class, is an aggregation of different filter classes from the Pipes-and-Filters architecture pattern. Each of these filter classes implement parts of the final algorithm and, during runtime, it is possible to combine them with other filter objects to obtain the desired functionality for a given controller. So, a subclass of class *OutputController* actually implements a certain setting of the 'switches'.

The actual implementation of the Pipes-and-Filters patterns is very simple as the pipes are implemented in the code by a certain call sequence of the filter functions. The following C++ code shows the implementation for the *generateF* operation in the SpeedOpenLoopController class shown on Figure 12:

```
Frequency SpeedOpenLoopController::generateF()
{
      frequency = theSlipFilter->output(frequency);
      frequency = theBypassFilter->output(frequency);
      frequency = theFreqLimiterFilter->output(frequency);
      frequency = theRampFilter->output(frequency);
      frequency = theResDamperFilter->output(frequency);
      return frequency;
}
```

Figure 13 shows the UML object-collaboration diagram with the different collaborating objects involved in the continuous-processing loop, activated every 2ms by calling the *generateMotorOutput* method. The use of polymorphic operations results in fast processing as the execution path through the code is set up in advance by the event controlled part (by a call to *setActiveController*). The traditional non-OO approach in the C programming language is to have a substantial number of C switch statements in the code, where each switch is calculated in the continuous processing loop.

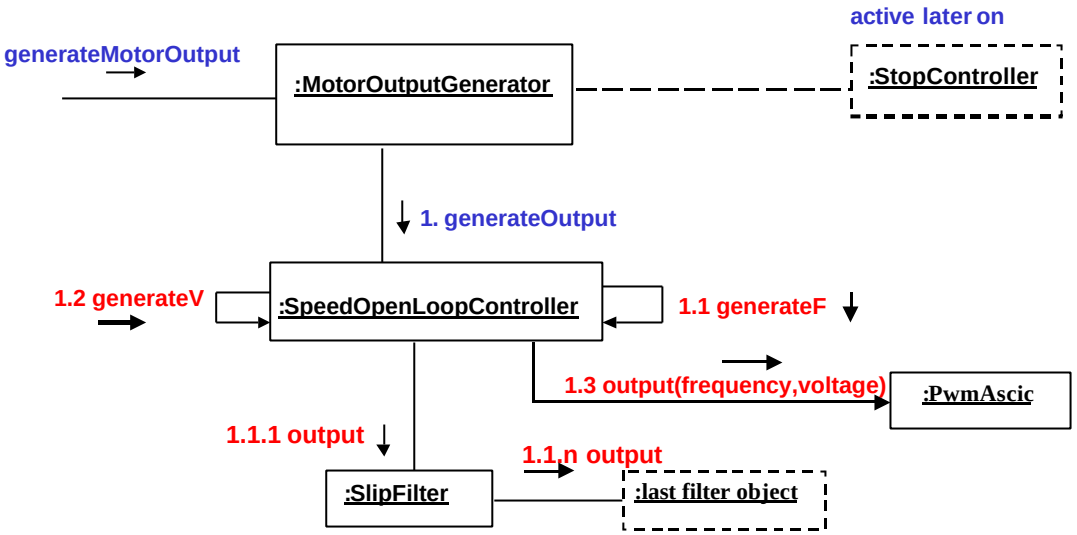

Figure 13. Object collaboration diagram for the continuous processing

The event-controlled discrete part

This part is activated by external and internal events, which may occur at any discrete point in time. Examples of user-initiated events are a start command to the system, a change of the current operation mode or a change of configuration parameters. Another source of events is the different monitoring functions supervising the VLT and the connected motor and generating internal events when some predefined situations occur. An example could be the detection of overheating in the motor, resulting in an internal event to the event-controlled part, which should stop the motor.

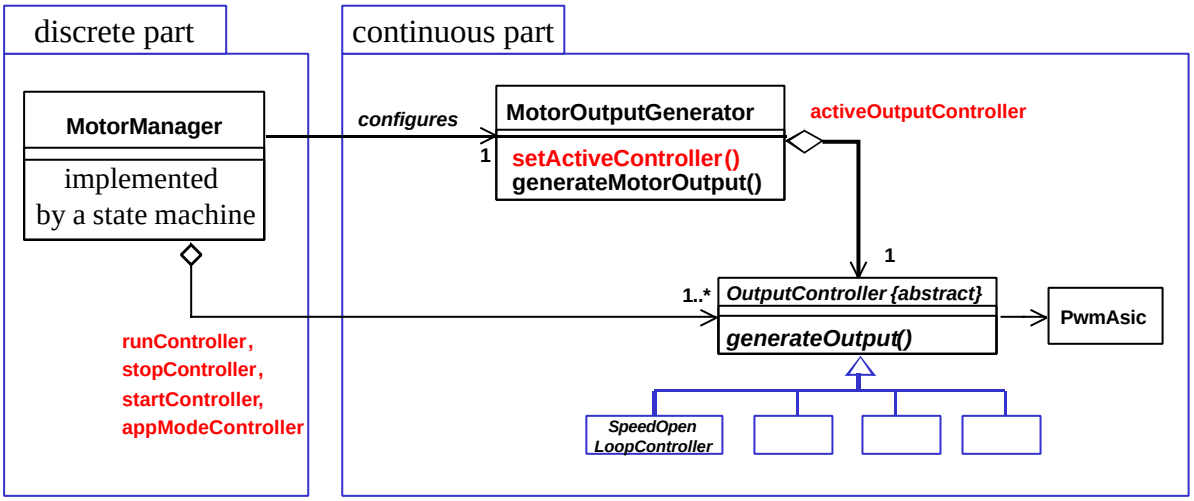

Figure 14. Class diagram with the configuration of MotorOutputGenerator

Figure 14 shows a *MotorManager* class, with the responsibility of controlling the motor by configuring the continuous part with the right controller. The *MotorManager* class is designed in the traditional way as a state machine, here modelled with a UML statechart.

Our experience shows that compared to the modelling of state machines in structured analysis and design, an object-oriented method seems to result in a larger number of smaller and more understandable state machines. There are two reasons. First, the Unified Modelling Language (UML) statecharts are more expressive than Structured Analysis/Structured Design (SA/SD) statecharts. Second, the emphasis on the responsibility of each class tends to move the state machines from the level of the system to that of the class.

Figure 15 shows a part of the statechart for the *MotorManager* class. The configuration of the continuous-control part is performed either on a state transition or when a new state is entered. The object pointers *runController* and *stopController* are initiated to point at the actual run and stop controller objects in another part of the program depending on the configuration parameters. The actual run controller could be one of the two previous described controllers “Process Closed loop” or “Speed Open Loop”.

```
runController= theConfiguration->AppModeController());
```

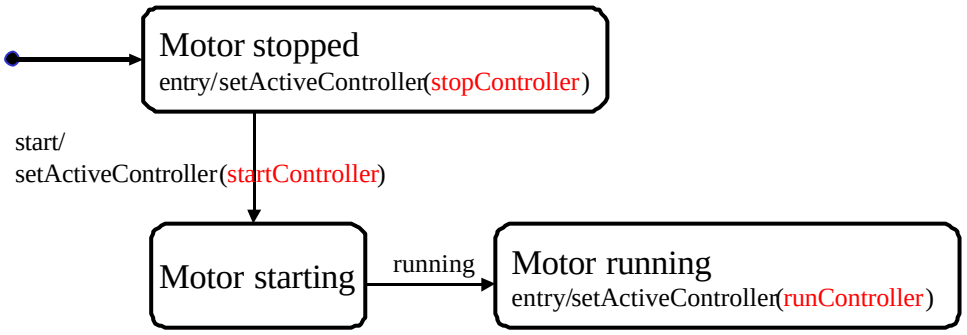

Figure 15. Part of the State Machine for the MotorManager class

The state machine implementation is realised by using the State design pattern [Gamma95] working in concert with the Command design pattern [Gamma95]. In the State design pattern, each state of the actual state machine is modelled as a class with the triggering events as operations.

Each external event is transferred to a command object, which is subsequently sent to the *MotorManager* class for processing. This solution resulted in a simple interface for the *MotorManager* class, obtained by the method call:

```
handleCommand(Command* theActualCommand);
```

Figure 16 shows three steps of the execution of a *start* command issued with the *MotorManager* in the *MotorStopped* state. The solution is based on a double dispatch. The first dispatch is called in the body of the *handleCommand*, where the *execute* operation is called on the command object received as a parameter. The second dispatch is performed in the body of the command object, in this case the *StartCommand*, where the *start* operation is called on the actual state of the state machine.

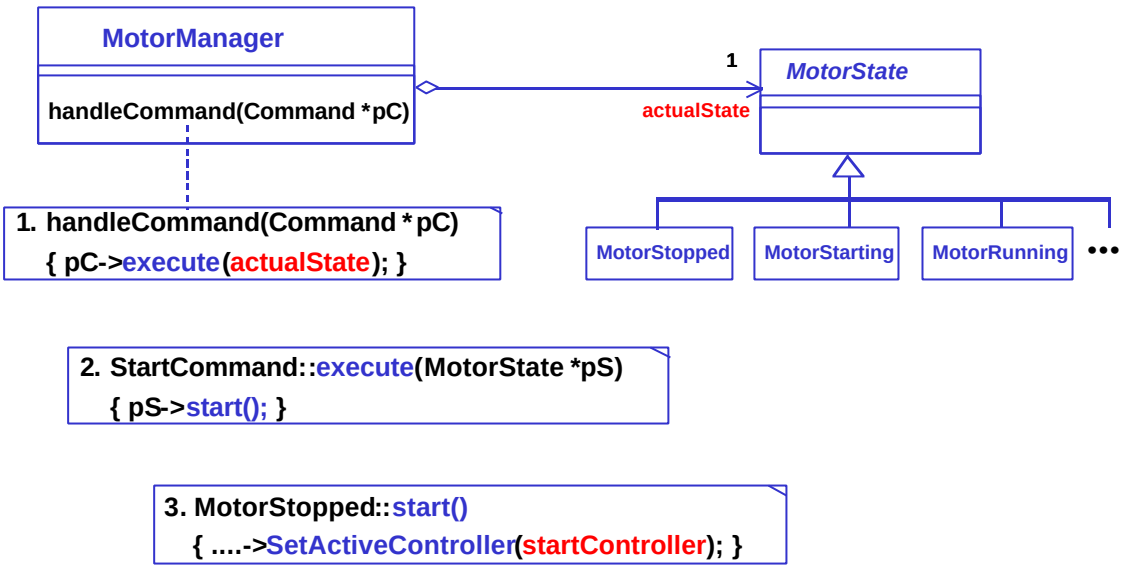

Figure 16. Outline of the implementation of the Command + State Pattern

METHOD CONSIDERATIONS

This section will describe how the two-part model approach can be used as a constructive step in the development process, starting with a requirements specification based on the Use Case technique [Jacobson92].

Use Cases

Use Cases were employed in the pilot project to specify the VLT functionality. Use Cases for this system type can be divided in two different groups. One group is used for the event-controlled part of the system functionality and the other group represents the continuous-processing parts.

The Use Case diagram in Figure 17 shows two examples of event-initiated Use Cases, where the Use Cases are activated by events from external actors to the system, such as the VLT user. The lower part of the figure shows two examples of the second group of Use Cases, corresponding to the continuous processing parts, where the Use Cases are continuously activated by the system.

Our experience was that the system-initiated, continuous group of Use Cases was the most difficult group to identify and describe. The reason is probably, that the Use Case technique was invented primarily for actor-initiated Use Cases. To overcome the difficulties with the continuous Use Cases we now suggest that each of the modes of steady-state operation should give raise to a Use Case. This makes the identification easier. Further we suggest, that the description of each of these continuous Use Cases if appropriate should be based on the Process Control model. This approach makes it possible that all the functional requirements can be documented in the Use Case model - a fact that we believe is very important.

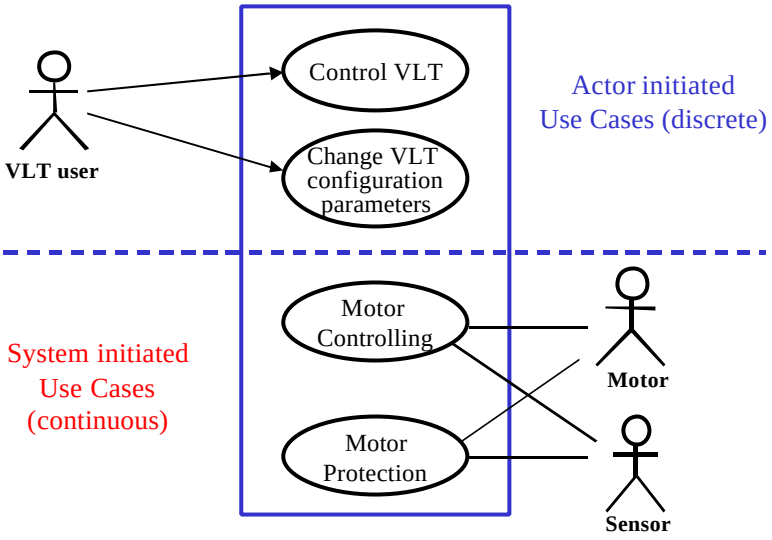

Figure 17. A part of a VLT Use Case diagram

Another Use Case specification problem was that the VLT is a very general device with many operational modes that can be used in very different applications.

The functionality specified by the Use Cases in each of these two groups maps into a number of classes in the corresponding parts of the two-part architectural model. This means that the functionality in the two-part model can be identified and structured with Use Cases already in the specification phase by categorising the Use Cases into these two groups.

Task modelling

The two-part model will, as a first approximation, lead to a task design with a minimum of two tasks. One task will control the discrete event-controlled part and the other will take care of the continuous-processing part. In practice, there will be more than one task in the continuous part as there is more than one type of processing.

For example, controlling and monitoring with different timing requirements leads to a minimum of two tasks in this example shown on Figure 18. The configuration data is in this example encapsulated in a class realised as a "monitor" class implementing a critical region for the configuration data.

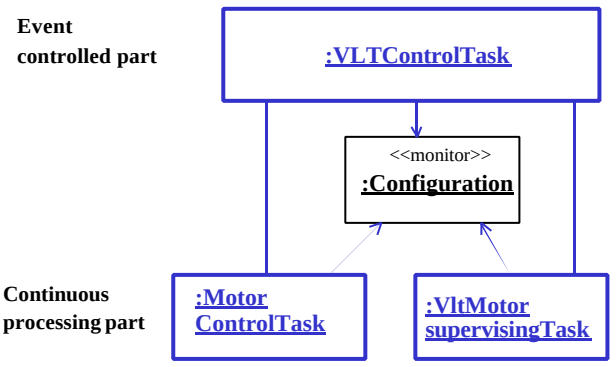

Figure 18. Task model example

Design Patterns

The knowledge and use of some of the standard design patterns described in several of the references [Gamma95, Bushmann96, Shaw&Garlan96] have been a great help in developing the actual object-oriented architecture resulting in two general guidelines. The first is that the continuous processing part can be designed and implemented by combining the Strategy pattern with the Pipes-and-Filters design pattern. The second general guideline is that the discrete processing part can be designed as a state machine and implemented by combining the State with the Command pattern. The use of design patterns in the actual project has resulted in a flexible design model, which is currently evolving into a framework for frequency converters.

Other experiences with Object Technology

The OO design model is extensively based on the use of polymorphic operations, such as C++ virtual functions, an approach that avoids computing the switch arguments in a series of switch statements at runtime. Another advantage of using polymorphic operations is that the code is much easier to modify and extend without having to modify a lot of switch statements in different parts of the program. Another outcome was that the state machines for the new OO based design were considerably smaller than in previous designs based on the Ward & Mellor SA/SD method.

CONCLUSION

It is our experience, that the two-part model compared to the former approach is beneficial with respect to several of the design evaluation criteria mentioned in [Shaw95], namely “separation of concerns and locality”, “perspicuity of design” and “abstraction power”.

In the former software the event handling and output computation were intermixed. One of the drawbacks was that it was difficult to understand and test the individual output control algorithms. In the new architecture the focus on “the system’s operational modes and the conditions that cause transitions from one state to another” [Shaw95] has been localized in the event-controlled part. Separated from this and localized in the continuous processing part is the focus on the behaviour of the individual output control algorithms. This also leads to a higher degree of reuse, since the output control algorithms tend to be unchanged in different products utilising the same inverter control hardware.

When looking at a specific functionality, the clear cut between event-handling and continuous behaviour has made it easy to determine in which part of the model, the functionality should be placed.

The two-part model has given raise to a vocabulary (OutputController, Setpoint, MotorManager, etc) which makes it easier for the developers to understand and make additions to our architecture.

The use of object-oriented technology in the VLT pilot project has been very successful and demonstrated the applicability of object technology to this kind of application.

The use of OO techniques has made it possible to build a framework that currently covers two different product lines within the same design model.

We believe that the design principles shown in this paper are useful for building systems and frameworks for other kinds of industrial systems that include continuous-processing parts combined with configuration and control by external actor initiated events.

REFERENCES

[Bushmann96]:
A System of Patterns: Patterns Oriented Software Architecture
Frank Bushmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stahl, John Wiley & Sons, 1996

[COT]:
The Centre for Object Technology (COT), www.cit.dk/COT

[Gamma95]:
Design Patterns: Elements of Reusable Software
Eric Gamma, Richard Helms, Ralph Johnson, John Vlissides
Addison Wesley Longhamm, 1995

[Jacobson92]:
Object-Oriented Software Engineering – A Use Case Driven Approach
Ivar Jacobson, Magnus Christerson, Pattrik Jonsson, Gunnar Övergaard
Addison-Wesley, Revised fourth printing 1993.

[Shaw95]:
Comparing Architectural Design Styles
Mary Shaw. IEEE Software, November 1995.

[Shaw&Garlan96]:
Software Architecture: Perspective of an Emerging Discipline
Mary Shaw and David Garlan, Prentice-Hall, 1996

[UML97]:
Unified Modeling Language (UML)
Grady Booch, Ivar Jacobson , James Rumbaugh, et. al.
An industry standard – standardized in Nov. 1997 by
Object Management Group (OMG), www.omg.org