Center for Pervasive Computing
Report Series

Working Note
CfPC

An Architectural Style for Closed-loop Process-Control

Author(s): Henrik Bærbak Christensen and Ole Eriksen

Version: 1.0

Status: Closed

Abstract

This report describes an architectural style for distributed closed-loop process control systems with high performance and hard real-time constraints. The style strikes a good balance between the architectural qualities of *performance* and *modifiability/maintainability* that traditionally are often in conflict.

Contents

1 Introduction **5**
- 1.1 Terminology . . . 5
- 1.2 State versus Event information . . . 6

2 An Example **7**

3 Architecture **8**
- 3.1 Modular View . . . 8
- 3.2 Deployment View . . . 9
- 3.3 Execution View . . . 9

4 Distributed Store **13**
- 4.1 Store synchronization . . . 13

5 Name Service **14**

6 Analysis **15**
- 6.1 Strict Control of Program Flow . . . 15
- 6.2 Strict Control of Task Sequencing . . . 15
- 6.3 No Need for Synchronization . . . 15
- 6.4 Encapsulation of Bus Characteristics . . . 15
- 6.5 Bus Packet Optimization . . . 15
- 6.6 Location Transparency . . . 16
- 6.7 No Event Information Handling . . . 16
- 6.8 Fixed Boundary for Communication . . . 16
- 6.9 Scheduling Timeline . . . 16

Version	Date	Changes
0.1	27-8-2002	Initial version
0.2	5-9-2002	OE changes
0.3	23-9-2002	Comments by Danfoss and Ole E.
1.0	01-12-2003	Closed

1 Introduction

Process control systems [6] are often subject to severe performance and hard real-time constraints. The quality of control is determined by the sample time and execution frequency. Control systems are often embedded systems that have minimal processing power and memory in order to provide competitive, cost-effective, solutions. Thus the software in these systems have been designed and programmed with a very strong focus on these performance issues, often at the expense of other issues.

The software architecture research teaches us that any software system has inherent *quality attributes* [1] and that performance is only one of a large number of qualities. One other important quality is *modifiability/maintainability* that measures the relative cost of introducing a change in the software system. It is a well known fact that performance and modifiability are opposing forces in system design: maximal performance is often achieved by programming directly towards the functional requirements and utilizing the computing hardware directly. Modifiability, in contrast, is achieved by creating abstractions, encapsulating the hardware, and the use of delegation as is evident in most design patterns [4]—that all comes at a cost in terms of performance.

Computer science has traditionally focussed much on designs that are “modifiable” and “reusable” which may explain why findings from for instance architecture research is not often seen applied in process-control and embedded systems.

In this paper we describe an architectural style for the handling the steady-state aspects of closed-loop process-control system that have both good performance characteristics and achieves a high level of modifiability and maintainability.

1.1 Terminology

We will generally use the terminology used by Shaw et al. [6]. It is reproduced here as a convenience.

- **Process variables.** Properties of the process that can be measured; several specific kinds are often distinguished as outlined below.
- **Controlled variable.** Process variable whose value the system is intended to control.
- **Input variable.** Process variable that measures an input to the process.
- **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.

1.2 State versus Event information

Kopetz distinguish between *event information* and *state information* in a distributed embedded system [5, §2.1].

Event information is external events that alter the operation of the process control, for instance hitting the “stop” button, changing the way set points are handled, or chaning the source of input values. Event information is characterized by that they appear “seldom” over time (compared to state information); they must not be lost; they must be processed exactly once; and they must be processed in the order they appear in time.

State information is input values that represent the state of the process and must be acted upon by the process control. State information is characterized by happening frequently (typically once in every control interval); state information overwrites the previous value of the variable; and that to some extend the control can survive if a value occasionally is lost.

The described architectural style focus on handling state information in the process control. We do not consider it suitable for handling event information — some other mechanism must provide this handling.

2 An Example

Before we describe the architecture in more detail, let us set the stage by considering an example of the type of system our architectural style addresses.

Consider a process control system that controls the torque of an motor with very high precision. The motor may run an elevator, a conveyour belt for a brewery's bottling system, or the hook of a crane.

The motor control is a closed-loop algorithm that requires one or more set point values (sometimes also denoted *reference values*) and an input value (sometimes also denoted *feedback value*) that report the current state of the motor.

The accuracy of the process variables, set point and input, is vital. To ensure high motor performance the following performance requirements must be met:

- The set point and input values must be sampled frequently and timely arrival of the values to the control algorithm must be ensured (hard real-time). In the case studies we have the control loop require freshly sampled values to arrive at intervals measured in a few milliseconds. We denote this time interval the *control interval* $\mathcal{I}$. The control interval must be fixed and constant.
- The input values must be guarantied to be synchronized. I.e. if a feedback value is correctly sampled in a given control interval but the corresponding reference values is somehow lost then the feedback value must be ignored.
- The exact samling time must be known to the control algorithm with high precision to avoid aliasing effects. For instance, if the algorithm assumes an input value has been sampled at time $t = 0.2\mathcal{I}$ but the actual sampling time is $t = 0.5\mathcal{I}$ the resulting motor control variables will be wrong.

Our case study systems are distributed systems i.e. input variables are sampled on one set of computational nodes and sent using some communication bus to a central computational node holding the control algorithm. Both studied systems rely on the CAN bus [3].

3 Architecture

In this section, we describe the static and dynamic aspects of the architectural style. Note that we only consider the steady state operation of the process control. Another aspect is handling commands that alter the operational mode of the control algorithm.

We use architectural views to organise our presentation. Views focus of different aspects of an architecture. The views we have found useful are the *modular*, the *deployment*, and the *execution* view.

The modular view focus on the static nature of the architecture: the basic concepts and their relationships. It is described by a UML class-diagram and the respective responsibilities of the interfaces/classes.

The deployment view focus on the physical executables, the nodes they run on, and the transport mechanism they use to communicate.

The execution view focus on the run time objects/threads, on their run-time interplay with respect to flow of control and data, and on their scheduling.

3.1 Modular View

The static aspects of the architecture is shown in fig 1 as an UML class diagram. It is expressed in terms of interfaces that partition the design into three main abstractions—or *mini frameworks* as Gurp and Bosch term it [7]. Concrete classes then join the mini frameworks by implementing several of the interfaces as is the case with the class `StandardStore`. The three mini-frameworks are in separate parts in the figure, as shown by the lines.

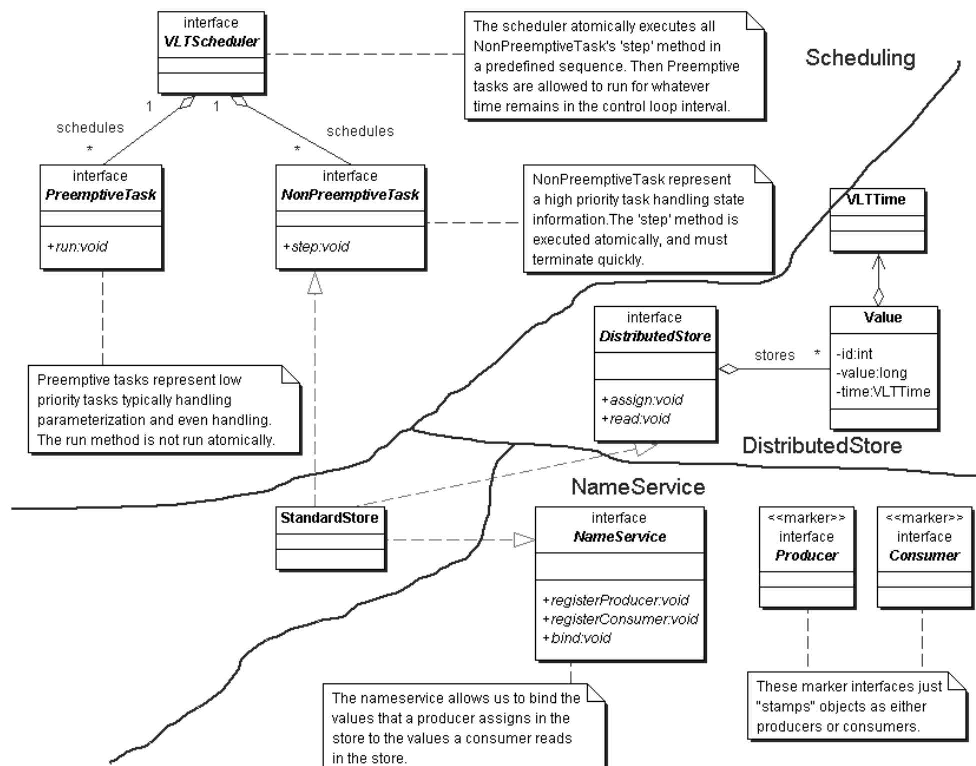

Figure 1: UML class diagram of the process control architecture

The three main frameworks are

- *Scheduling* defines the scheduling abstraction and the two types of tasks that can be run in it. Scheduling is described in detail in subsection 3.3
- *DistributedStore* defines a shared memory abstraction across computational nodes in the system. Any runtime component in the system may assign and/or read a variable in the distributed store. The store is described in detail in section 4.
- *NameService* defines a system wide name service / lookup abstraction. The name service allows producers and consumers of data values to be bound together and ease rebinding; for instance changing the data producer for a given consumer. The name service is described in detail in section 5.

3.2 Deployment View

Diagram 2 shows a simple deployment scheme where one computational node, `Analog Input Card`, is responsible for sampling a set of input values, and another computational node, `uController` is responsible for the control loop and motor control.

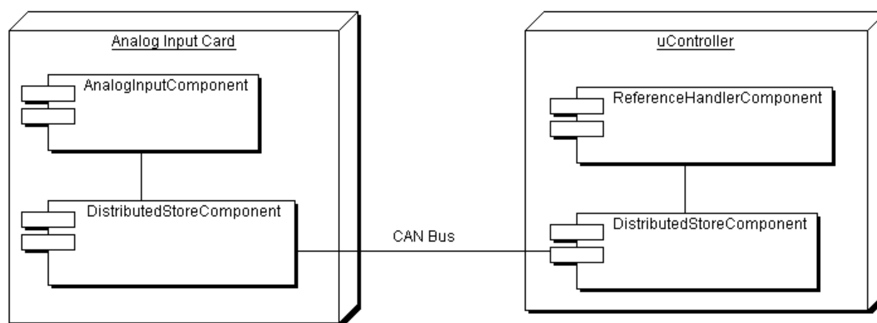

Figure 2: Deployment on two nodes

All computational nodes have an instance of a distributed store running locally. All distributed stores in the system are connected using the CAN bus (or another bus) and are ensured to be synchronized as well-defined points in the control interval. How this is done is described in the next section.

3.3 Execution View

In the collaboration diagram 3 is shown three active objects that are representive of a typical steady state situation. AnalogInput samples an input variable from the environment using an A/D converter when its `step` method is invoked by the scheduler (not shown) and assigns the value to the variable in the distributed store. Later the referencehandler (a consumer of the value) reads it in its own `step` method.

As described above, the distributed store is only conceptually a single active object; in real deployment it is two active objects that keeps their shared memory synchronized over the bus communication medium.

Scheduling of tasks is done by the VLTScheduler. Each computational node has one instance of the VLTScheduler running. A global clock must ensure that all VLTSchedulers work in synchronization. It means that the scheduler regards the time as divided into control intervals, each lasting N ms (where N is small e.g. 1). The control

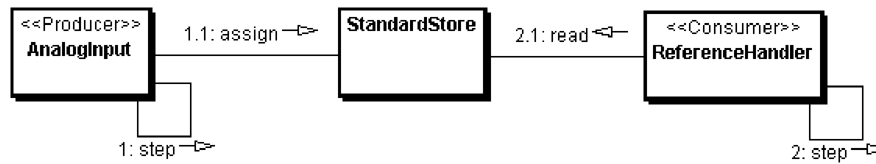

Figure 3: Steady state collaboration

intervals are enumerated and for at given input value X we can talk about the values as x_1, x_2 etc. generated by the input in the sense that x_i is the value of X in control interval i. In running mode, the output value can be described as a function of the input values.

The synchronization constraint is that output value number n must be a function of input values with index n. It means that all input values must be present in the same control interval at the node computing the output value. The consequence is that in one control interval we have to collect the input values, perhaps transform them properly, send them via the network and compute the output value.

In fact in small amount of time it is enough if the system fulfills a weaker synchronize constraint: the output value must be a function of input values with equal indices.

The scheduler is responsible for the sequencing of task computation. As shown in figure 1 the scheduler maintains a list of NonPreemptiveTask and PreemptiveTask.

NonPreemptiveTasks have a single method `step` that defines the task's computation. Objects of this type runs in a stepwise, nonpreemptive, way and with highest priority. They run in an order that is completely defined by the programmer. Thus the programmers can fully specify the sequencing of computation and does not need to worry about synchronization issues as these tasks are run atomically. The order can be defined either by the position of the NonPreemptiveTask object in the scheduler's list: the first task is executed first, then the next, and so on; *or* the task can identify the exact time that it wants to be executed, expressed as some fraction of the control interval.

The other type is called PreemptiveTask—they are run preemptive and with lower priority. They are executed when the scheduler has exhausted executing its list of NonPreemptiveTasks.

The interface of the scheduler may look like this: interface Scheduler

```
{
  void add(NonPreemptiveTasks c);
  void add(NonPreemptiveTasks c, TimeAmount t);
  void add(PreemptiveTask t);
  void start(TimeAmount t);
  void pause();
}
```

The scheduling is shown in somewhat more detail in the two sequence diagrams below.

In sequence diagram 4 is shown the sequence of invocations that takes place for the analog input handler to sample a value and assign it to the store—as representative of a typical producer. Note that in the analog input's `step` method the A/D hardware is told to sample the value. Later the A/D hardware will interrupt the analog input task to deliver the value and only then is the store assigned. Later the store's `step` method is invoked which is responsible for synchronization of store contents in all stores.

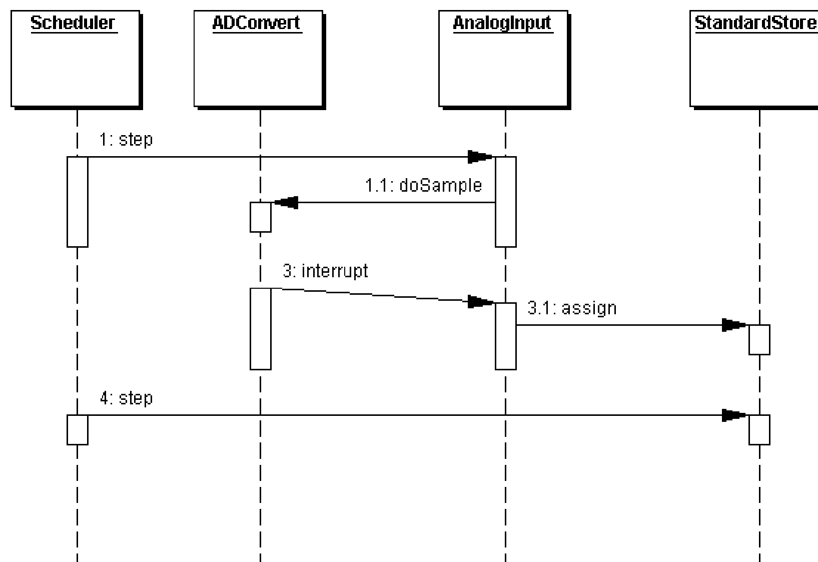

Figure 4: Scheduling of input handling

The consumer side is even simpler as shown in diagram 5. There is a problem in this

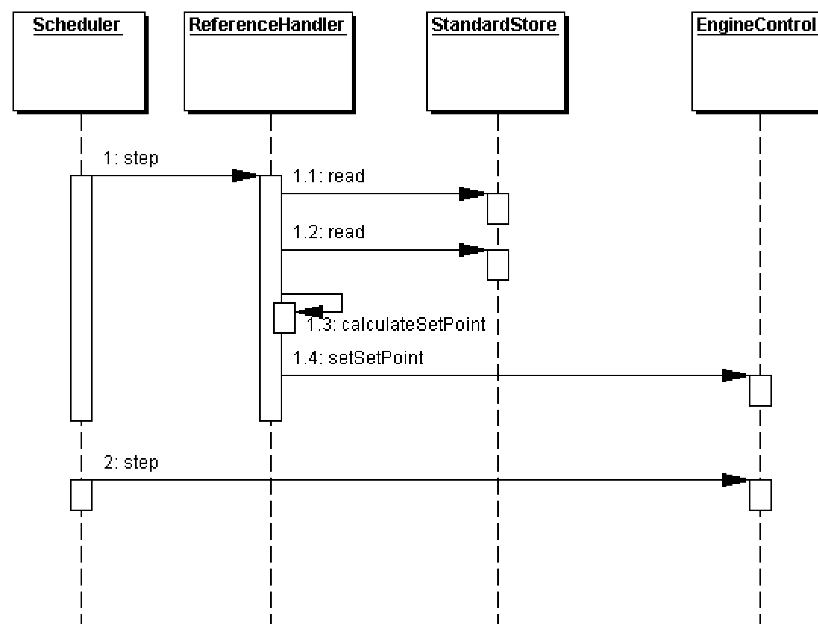

Figure 5: Scheduling of input variable processing

scenario, however. The StandardStore instance in the two diagrams are deployed on different computational nodes and thus we must globally ensure that the sampled values have *all* been a) sampled b) stored c) and distributed to the store on the reference handler code *before* we can rely on the value read from the store on the reference handler node!

At the present, we find that this can only be handled by keeping strict control of the timing budgets of the computations on the involved nodes. I.e. global analysis of the time spent on each node is used to define the time when the consumer NonPreemptiveTasks are allowed to perform their computation on the consumer node(s).

4 Distributed Store

The distributed store provides an abstraction of a single, global, memory of variables throughout the distributed system. I.e. any component may define a set of variables in the store and then later assign values to the variable and/or read back values from the variable.

At present it is envisioned that the distributed store only provides variables of a single, predefined, type (Value in diagram 1).

Variables are identified by some means. The case studies use different classic approaches such as predefined integer constants and strings values.

Assignment: The traditional assignment in Pascal syntax

```
windspeed := 12.3;
```

can then be expressed like

```
#define WINDSPEED  645; // unique identity of WINDSPEED

store.assign(WINDSPEED, 12.3);
```

(In practice the value part of the assignment will be an object.)

Simularly, reading a value in Pascal

```
controlVariable := f(windspeed);
```

where `f` is the control algorithm function that calculates the value to assign to the control variable.

This statement will become (using the definitions from above):

```
windspeed = store.read(WINDSPEED);
controlVariable := f(windspeed);
```

4.1 Store synchronization

The distributed stores must be kept in synchronization. More specifically the architecture must ensure the following:

> *Given n computational nodes ,$N_1..N_n$ in the system, each having a distributed store instance, $D_1..D_n$. Then it must hold that for a variable, v, then all components deployed on a node N_i must read the same value of v:*
>
> $\forall D_i, D_j \quad D_i.read(v) = D_j.read(v)$

In practice this is not possible.

Instead we adopt to ensure that the above statement is true (at least with high probability—"shit happens" in the real world) at a well-defined time in the control interval $\mathcal{I}$. As shown in the sequence diagrams in the preceeding section the statement is not true until all distributed store instances in the system have all executed their 'step' method, the bus communication has been effected and the store instances have reacted on the incoming packets.

5 Name Service

As described in the preceeding section, variables must of course be identified. A simple scheme is to use global identities.

In many situations, however, a more flexible identification scheme is desirable in a distributed system.

To illustrate this, consider the problem of reconfiguring the control algorithm to accept input variables from one source to another. I.e. in control interval i the control algorithm reads a temperature from a AnalogInput that measures it from a real temperature sensor but in control interval $i+1$ it reads it from a source that has received the value over a ProfiBus.

One solution is to augment the control algoritm with additional methods that can be invoked to change the input source. This introduces a new responsibility into the control algorithm component which is always dangerous—it quickly leads to “The Blob” antipattern, where objects take on too much responsibility. Even worse is that there may be more consumers of the temperature value than just the control commponent—we thus have to reconfigure several components accept temperatures from a new source—and guaranty that they are reconfigured simultaneously.

However, as all process variables must pass the distributed store it is better to augment this one abstraction with the responsibility of binding produced values to consumed variables.

We have defined this rebinding responsibility into a separate interface, `NameService`.

The name service in essense decouples data producers from data consumers. The name service thus defines these roles by means of two marker interfaces “Producer” and “Consumer”. Producers and consumers respectively register names that they want to user for their datavalues in the nameservice component, which is a run-time object that implements the NameService interface.

Consider the example mentioned about. The AnalogueInput registers an input variable in the name service under the name “Input03” which, accidentially, is an measured process input variable like a temperature. The consumer that controls a fan in a similar vein registers that it wants to be able to read values under the name “Temp01”. Of course these two names have to be bound in order for the consumer to read the proper input value. The “bind” method is used for that. It simply takes two parameters, a name of a producer variable name and the name of a consumer variable name, like “bind(“Input03”, “Temp01”)”. Hereafter when the consumer reads the value denoted Temp01 it will get the last value written by any producer that has written a value to “Input03”. If another producer is producing values on the ProfiBus under the name “Input04” then changing the input process variable to this one is simply a matter of invoking the bind method again with the correct parameters: “bind(“Input04”, “Temp01”)”.

6 Analysis

In this section, we outline some of the benefits and liabilities the proposed style has. You should consider each of these carefully before you commit to using this style for your particular process control.

6.1 Strict Control of Program Flow

Individual components only communicate about steady state process variables in terms of the distributed store abstraction—that is basically assignment and reading of a variable. Variable assignment and reading has no side effects in contrast to accessor methods on objects.

This has the extremely important property when it comes to reasoning about timining and deadlines. A programmer simply cannot by accident invoke a computationally heavy method in another component as is the case with an RMI based approach. This means that timing analysis becomes local to each component.

Enhances the modifiability property and reduces the cognitive requirements of performing deadline analysis.

6.2 Strict Control of Task Sequencing

The order in which tasks are executed is deterministic and explictly controlled by the programmers using a simple abstraction—namely the ordered sequence of non-preemptive tasks whose `step` methods are simply executed atomically in turn.

6.3 No Need for Synchronization

Non-preemptive tasks are executed atomically and there is therefore no need for costly synchronization between concurrent tasks.

Enhances both performance and modifiability property.

6.4 Encapsulation of Bus Characteristics

Only the distributed store access the bus and thus all bus specific programming is isolated to this component.

Enhances the modifiability property.

6.5 Bus Packet Optimization

Because the broadcast of the shared values are performed at a specific time (the 'step' method) the set of values assigned to the store since the last broadcast can be precisely determined. Thus multiple value assignments can be grouped into fewer packets on the bus than if a broadcast was done on every assign. This lowers the bandwidth demands.

Enhances the performance property.

6.6 Location Transparency

Location transparency is described by for instance Buschmann [2, §?]. In a distributed setting you do not want to hardcode your communication implementation to a specific location of components.

The distributed store ensures location transparency. As long as producers and consumers of values agree on the identity, they do not know on which node in the system each other is located—they only need to know the existance of the store instance (which is deployed locally on all nodes).

6.7 No Event Information Handling

The described architectural style does not deal with event information. Thus some other mechanism must be implemented to handle this; or perhaps to adapt the presented style to handle event information also.

This forces both designers and maintainers to understand *two* different styles and communication paradigms (this one for state information and some other for event information) which can lead to obvious errors, like developers using the wrong style for a pieces of information.

Two mechanisms may also potentially require more code in the middleware layer having a negative impact on memory footprint.

Negative impact on modifiability and performance (memory footprint).

6.8 Fixed Boundary for Communication

Compared to a RMI based approach for communication between components, the proposed style dictates the boundary over which communication happens: namely within the distributed store component. Thus developers have no way of re-deploying components to other nodes based on the computational abilities of the specific system's hardware architecture.

Negative impact on modifiability and perhaps performance.

6.9 Scheduling Timeline

The design must be calibrated to ensure that the consumers of values does not query the store *before* all producers are guarantied to have stored their values, the stores have distributed them, and have stabilised. This require either very strict analysis of the time budget on each computational node and/or trial-and-error calibration of the system.

However, if the system is significantly altered this calibration is potentially wrong. It also hampers the ability to make run-time re-deployment of components or the ability to add new computations at run-time.

Negative impact on modifiability (as entering a computation into the scheduling queue may alter system behaviour significantly and thus strict control of the time budgets must be made) and potentially large impact on performance.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture*. John Wiley and Sons, 1996.

[3] Controller Area Network CAN, an In-Vehicle Serial Communication Protocol. In *SAE Handbook 1992*. SAE Press, 1992.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

[5] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[6] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[7] J. van Gurp and J. Bosch. Design, implementation and evolution of object-oriented frameworks. *Software: Practice and Experience*, Sept. 2000.