Using Software Architectures for Designing Distributed Embedded Systems

Henrik Bærbak Christensen
University of Aarhus
Department of Computer Science
DK–8200 Aarhus N, Denmark
Tel.: +45 8942 3188
E-mail: hbc@daimi.au.dk

Abstract

In this paper, we outline an on-going project of designing distributed embedded systems for closed-loop process control. The project is a joint effort between software architecture researchers and developers from two companies that produce commercial embedded process control systems. The project has a strong emphasis on software architectural issues and terminology in order to envision, design and analyze design alternatives. We present two results. First, we outline how focusing on software architecture, architectural issues and qualities are beneficial in designing distributed, embedded, systems. Second, we present two different architectures for closed-loop process control and discuss benefits and reliabilities.

1. Introduction

In many companies that envision, design, and produce embedded systems, software has historically not been the main point of attention. The challenges have been of an electrical and hardware oriented nature and has naturally called for people with expertise in these fields. Early systems did not have much processing power or memory and consequently the onboard programs were small and often designed, implemented, and tested by hardware engineers. However, embedded systems of today are becoming very powerful with respect to both processing power and memory allowing much more flexible and adaptable products to be designed and produced. Flexibility combined with the larger computational power skews the focus of embedded systems design towards the problems and challenges of designing large distributed software systems. The software must perform well and at the same time have the qualities advocated in software engineering.

In this paper, we describe an effort of designing high quality architecture for a distributed, embedded, system. The system is a closed-loop process control system [11] with hard real-time constraints and severe performance requirements. The main point of the exercise has been to strike a good balance between achieving the harsh performance goals while at the same time achieve the architectural qualities of *modifiability* and *maintainability* of the design and implementation. The motivation is that a high performance design that is difficult to maintain and too inflexible to accommodate new features is not cost-effective.

Our project group consists of computer scientists as well as software developers from two companies that develop embedded control systems. One company develops high-precision motor control systems that are flexible in terms of the number and quality of the option-cards (analog and digital input cards and various cards for interfacing legacy bus systems), the other develops wind mill control systems.

The researchers in the team have their expertise in *software architecture* and *software engineering*, not in embedded systems design. Our main contribution is to present how software architecture concepts can be used to clarify design goals and outline interesting design alternatives. We present how these alternatives form basis for evaluation from both an analytical as well as an experimental standpoint. In the theoretical analysis, we assess the impact of the architectural constructs used in the alternatives on performance and maintainability. In the experimental work we construct *architectural prototypes* that run on real hardware and networks. These prototypes are programmed to run a set of typical, but demanding, scenarios under different operating conditions. The prototypes are evaluated both by direct performance measurements as well as source code reviews to assess qualities of maintainability, readability, and abstraction level.

We reserve the term *architecture* in the rest of the paper to denote the software architecture as opposed to the architecture of the hardware or network topology, unless explicitly stated.

2. Problem Domain

Our problem in closed-loop process control is to control a continuous process. The process is optimal when measurable properties of the process's state have some desired values, called the set points. The process is indirectly controlled, like for instance by controlling a motor or a brake that influences the process. Closed-loop means that the control is based upon measuring variables, called input variables, of the process and react accordingly.

As an example, consider cooling a room to a certain temperature based upon controlling the speed of a ventilator. Temperature values are constantly measured in the room and feed to the process control that then tries to ensure a set point temperature in the room by controlling the speed of a ventilator.

Embedded systems design often distinguishes between *event information* and *state information* [9]. State information is concerned with the steady state of the system typically the input variables of the process while event information is typically user commands, parameter changes, etc., that change the operational mode of the control. In the architectural discussion below, we restrict ourselves to the handling of state information and ignore event information. We also exclusively focus on the input side of the control, i.e. ensuring the timely delivery of input variable values to the control algorithm. This does not pose a problem for the validity of our approach, as the major performance requirements are indeed concerned with the state information part of the system.

Hardware requirements dictate that input variables are measured on one set of computational nodes while the control algorithm is located on a separate node. This hardware architecture achieves the maximal flexibility in terms of configuring the sources of input variables and provides room for future upgrading of the system as a whole. It also minimizes the cost of cabling in a windmill, as only a communication network has to be fitted between measuring nodes and control node(s). Thus the hardware architecture is a driving marketing force.

The control systems under consideration are both high precision products that achieve a competitive edge by providing a very short response time to process variations. Thus the architecture is subject to a number of severe performance requirements.

- *Delivery deadlines.* The input variables must be sampled frequently and timely arrival of the values to the control algorithm must be ensured (hard deadline guaranties). In our case studies, the control loop requires freshly sampled input values to arrive at intervals measured in a few milliseconds. We denote this time interval the *control interval* $\mathcal{I}$. As the control algorithm adjusts the output variables (like motor torque or brake force) once in every control interval, this time span is the essential "heart beat" of the system.

- *Synchronization.* All input variables must be guarantied to be synchronized. By this, we mean that all input variables have been sampled within the same control interval as the control algorithm calculates the resulting output value. For instance, if a measured input variable is correctly sampled in a given control interval but the corresponding set point value is somehow lost then the full set of measured variables within this control interval must be discarded and the synchronized values of the previous control interval used.

- *Aliasing.* The exact sampling time must be known to the control algorithm with high precision to avoid aliasing effects. For instance, if the algorithm assumes that one input value has been sampled at time $t = t_0 + 0.2\mathcal{I}$ but the actual sampling time is $t = t_0 + 0.5\mathcal{I}$, the resulting output variable will be wrong and the process will be incorrectly controlled.

- *Network congestion.* In some setups the control algorithm draws upon a lot of input variables, which stresses the network. On top of that, the bus of course also carries other types of traffic not considered within this paper: global clock synchronization, events for changing operational modes, feeding input values to other output sources like displays, slave controllers, etc. We must avoid that the bus is congested to the point that it endangers the deadline delivery requirements.

- *Memory footprint:* Ideally a flexible system serves customers with a wide range of needs: from the customer that wants a cheap system with a few fixed possibilities to the customer that wants full flexibility and is willing to spend the money to get it. This can be achieved by offering plug-in cards to the system that vary with respect to processing power and memory capability—however the software architecture is required to be able to interoperate with all types and perhaps even dynamically adjust to the specific setup at runtime.

In both our case studies, the companies have chosen to use the CAN bus as communication medium [2]. CAN is a broadcast protocol that ensures all-fail semantics of packet delivery: either all or none CAN controllers receive a sent packet. .

3. Desired Architectural Quality Attributes

Performance is the fundamental driving requirement and quality. However, it is not the only quality: Bass et al. list a number of *quality attributes* that any software system have [1]. They distinguish between two types: qualities observable via execution and qualities not observable via execution. Performance, security, and availability are examples of the former category while maintainability, portability, and testability are examples of the latter.

Maintainability can be defined as

> ***Maintainability:*** *measures the relative cost of modifying the architecture to accommodate new functionality*

Maintainability is largely a function of the locality of any change: making consistent changes to many components is more costly and error-prone than making changes to just a single component. The abstraction level of the software also greatly influences maintainability: a design with natural abstractions that encapsulate tedious details helps comprehension and lowers the number of errors introduced.

The challenge we set out to investigate was the following

> *We want to maximize the maintainability quality of the architecture without severely compromising performance.*

In other words, to produce an architecture that is comprehensible and easily modifiable for the programmers in order to speed up development and lower error rates.

4. A Software Architecture Based Design Process

Our project's goal was find the best possible architecture for the aforementioned domain and with focus on the described qualities. Our path to fulfill this goal contains two steps. In the first step, we envision, design, and analyze a number of alternative software architectures. In the second step, we build *architectural prototypes* of the alternatives.

At the time of writing, we have mostly completed the first step, and are some way in the second step. We have envisioned and described a small set of viable architectures for the process control system, as outlined and briefly discussed in the next section.

We are at the moment in the process of constructing the architectural prototypes. Architectural prototypes are "skeleton systems" that focus on architectural high-risk issues and largely ignore concrete functionality. In our domain, this means designing and implementing systems that are capable of sampling the set of input variables and making them available for the control algorithm under the sketched performance constraints, but none of the components actually do anything useful with the values.

Once the skeleton systems have been build, we use them as an experimental basis. They are stressed until the point where they cannot perform properly. The systems are stressed along a number of dimensions: the bus is flooded with transmissions of dummy packets to simulate that other information besides steady state input information is present in the system; the CPUs are forced to execute computationally heavy tasks (that simply waste processing power) to simulate the load of the control algorithm, household tasks, etc. This way we can insight into the core of the performance problems that will feedback to the architectural design and how various issues have been resolved. In parallel, the architectural prototypes are evaluated with respect to modifiability and maintainability by code inspections and architectural analysis methods like SAAM [8].

5. An Abstract Architecture

We set out to define an *abstract* architecture that could serve as basis for concrete architectural designs. One has to be careful not to introduce ideas of a concrete architecture in this kind of description. For instance describing components in terms of their interfaces is not possible because any interface in terms of method signatures already locks the architectural choices. Instead we found the concept of *component responsibility* attractive because it allows us to express quite precisely the role of a given component without making assumptions about communication protocols and control flow.

The abstract architecture that we found is quite simple, but provides room for radically different concrete architectures. A conceptual/logical view of the architecture is sketched in diagram 1 and component responsibilities are outlined below. Components shown are logically entities that exist at run-time as independently executing processes.

The diagram notation is adopted from Hofmeister et al. [7] which have been inspired by ROOM [10]. Components are shown as boxes, and connectors as diamonds with lines attached. Components interact with the connectors over *ports*. A port is not simply an interface but states both provided as well as used services. A port also specifies a protocol for the interaction.

The component responsibilities are as follows:

- A **Continuous Input** component is A) responsible for sampling/collecting a single input value from the environment and B) make this value available to other components. A continuous input component may be associated with an A/D converter hooked up to some measuring instrument or the value may arrive from a field bus.

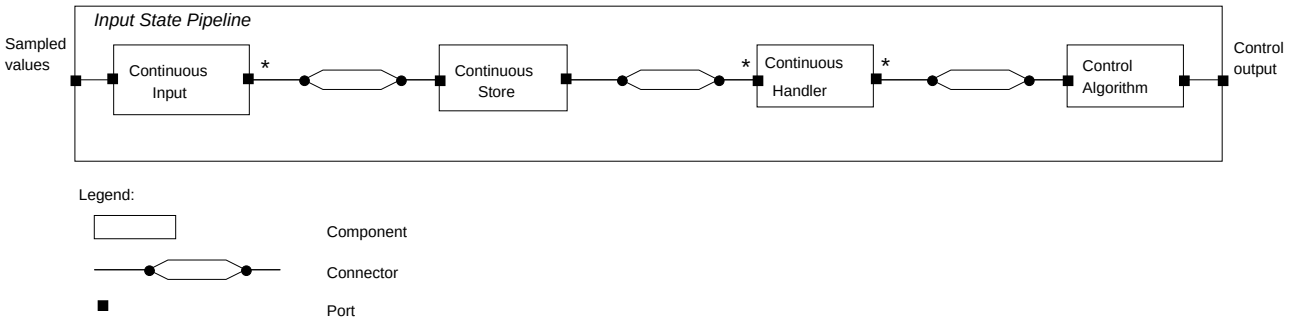

Figure 1.

- A **Continuous Store** component is A) responsible for storing/caching input values from a set of associated Continuous Input components, B) store the identity of the source of the value so its origin can be determined, and C) make the set of values available to other components.

- A **Continuous Input Handler** component is A) responsible for processing a set of continuous input variable values in a way proper for the control algorithm and the controlled process, and B) provide this resulting value to the control algorithm. Handlers typically process a set of inputs to provide a single output, like the average, maximum, or sum of the input values.

- A **Control Algorithm** component that encapsulate the complex calculations that leads to the output variables that ultimately control the process by some means: motors, heaters, brakes, etc.

The domain is inherently a data flow: data is created at the input components and flow through the store component(s) to the handler component(s) and on to the control algorithm.

The presence of the store components in the architecture is mandated by the fact that other components need access to input data besides the control algorithm—for instance for showing on a display. It also allows decoupling in time between the producers and consumers of state information: the producer may deposit a value and the consumer can fetch it at a later time.

Note that components described here are *logical* entities: in a concrete design/implementation the store components may be designed as part of the input component, as a separate component, or even as part of the handler components.

5.1. Architectural Variability

The abstract architecture defines a class of concrete architectures. The concrete architectures arise when we make specific decisions on a number of open issues in the abstract descriptions: these issues we denote *architectural variability points*.

We have identified three variability points.

- *Architectural style.* An architectural style defines the component and connector styles and the semantic constraints on them. We have found that the presented abstract architecture can be realized by either a *pipeline* style or a *repository* style. These are explained in detail in the next sections.

- *Network communication abstraction.* In the distributed system, part of the inter-component communication must pass over the network and a variability point is how to realize this. We have identified at least three possibilities: direct, low-level, communication; communication based upon some commercially available protocol, or remote method invocation using either a third-party object-request broker or one developed in-house.

- *Conversion component deployment.* Sampled values must be converted to process related values, like for instance converting a sampled analog number to a temperature in Celsius. This conversion can be made in a subcomponent deployed in any of the three components shown in diagram 1: the conversion can be made in the input, store, or handler component. Each decision has its own benefits and liabilities, especially with respect to the distribution of the conversion parameters: conversion late in the pipeline means parameters do not have to pass the network for instance.

In the following we will not discuss the conversion variability point further but concentrate primarily on the implications of using different architectural styles.

5.2. Concrete Architecture: Pipeline

Data that flows from a source towards a destination undergoing various transformations on its way makes a *pipeline architectural style* an obvious choice [1, §5].

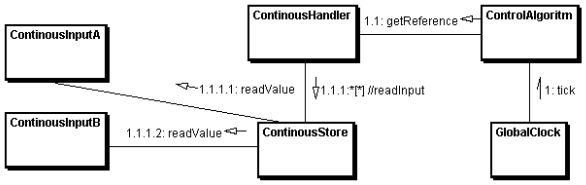

Figure 2. A pulled pipeline architecture

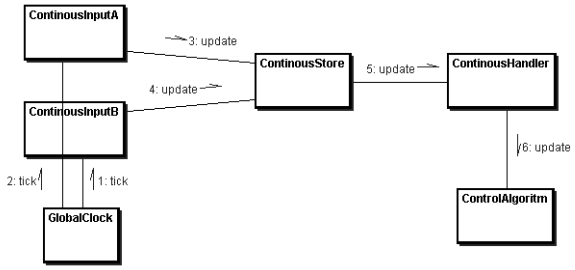

Figure 3. A pushed pipeline architecture

The pipeline architectural style defines two component types: pipes and filters. Data flow between filters in pipes, the filters modifies data while pipes are simply the transport medium between filters. The UNIX shell is the standard example of a pipeline architecture. Thus in our abstract architecture the input, store, and handler components are filters and the connections between them the pipes.

Any pipelined architecture must decide the direction of data flow as well as the mechanism used: either pull or push [4]. We have identified three concrete pipeline architectures based upon a pull model, a push model, or a combined model.

5.2.1. Pulled Pipeline In the pulled data flow model the receiver component requests the data from the supplier. In the UML collaboration diagram 2 a plausible implementation of a pulled pipeline architecture is outlined. Runtime components are shown as active objects—here we have shown two input components but there may be more. The control flow originates from the control algorithm that receives a "tick" from the global clock indicating that a new *control interval* must begin. It then reads values, using a method call with blocking semantics, from the continuous handler component, which again reads from the store etc. Thus, data is pulled from the input sources by read requests all the way down the pipeline.

The above diagram does not show the deployment aspect. Input components are deployed on a set of input nodes; the control algorithm component is deployed on the central node. The store and handler components may be deployed on either the input nodes or on the central node. (If the store components are deployed on the input nodes, then the above diagram must of course be changed to show multiple store components.) Thus at least one of the method calls shown in the diagram must be substituted with communication over the network using one of the mechanisms described in section 5.1..

This architecture has several nice properties. First, it is simple and thus easy to comprehend and program. Second, timing and synchronization of data values are straight forward as the control algorithm actively "pulls" all data needed. The design also has some inherent liabilities. The receiver of data must directly query the supplier(s), thus the resulting code easily ends up being hardwired to a certain setup. This is not in line with the wish for a flexible, open, architecture on the hardware side. Another problem is that of consistency of values—if two different components request a value, how do we ensure that they actually get the same value within the same control interval?

5.2.2. Pushed Pipeline In the pushed data flow model, the sender pushes data onto the receiver. A plausible architecture is shown in UML collaboration diagram 3. The flow of data is organized around the observer design pattern [6] so all receivers of data have to register with the data suppliers as data observers before the process control is started. The control algorithm registers itself as observer of the handler data; the handler in turn registers itself as observer of store data; etc. During normal operation, the clock notifies the inputs that the control interval must start. They react by sampling/fetching the input values and then sends an update (containing the values) to all registered observers—here the store. The store also notifies its registered observers and this way data is moved through the pipeline. This is essentially an event-triggered architecture [9]: state changes trigger the control.

This architecture has the nice property of decoupling the suppliers from knowledge of identity of the receivers and also their number. For instance, if input values are also to be displayed then the display component simply register itself with the store component and is then also feed with updated values. More important, however, is that this architecture guaranties synchronization—all observers registered on a supplier of values are guarantied to receive the same value.

The architecture, however, also has a serious performance drawback. In the diagram two input values are sampled which lead to *two* complete update sequences throughout the pipeline. Thus all processing within the pipeline is performed twice. Consider for instance that the handler component calculates a reference for the control algorithm by adding the two input values, and assume that at the end of control interval $\mathcal{I}_n$ the input values are v_n^1 and v_n^2. At the next control interval $\mathcal{I}_{n+1}$ the first input node samples a

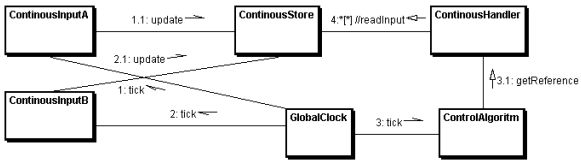

Figure 4. Mixed push/pull pipeline architecture

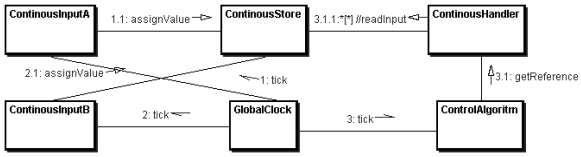

Figure 5. Repository architecture

new value v^1_{n+1} which starts an update chain leading to the handler calculating a new reference $r = v^1_{n+1} + v^2_n$. This is of course wrong and in conflict with the requirement of synchronization of values from different sources.

5.2.3. Mixed Push/Pull Pipeline One can also consider a pipeline architecture that mixes the push and pull. In diagram 4 a design is sketched where the input side uses push to update the store component and the handler component then pulls values from the store.

Note that this architecture decouples producers and consumers in the time dimension. A producer may deposit a value, but does not require the consumer to read it. Similar, a consumer can read a value but does not require a producer to sample a value. Thus, this type of architecture must rely on proper scheduling/timing of tasks to ensure that producers deposit values before consumers read it—a time-triggered activation [9]. This of course complicates the programming task (and lower maintainability) because proper operation depends on task timing, which is often not very visible in the source code.

This architecture has some similarities to the repository architecture discussed next.

5.3. Concrete Architecture: Repository

Another viable architecture is the *repository* architecture [1, §5]. In the repository architecture there are two types of components: the *repository* that stores shared data and a set of *clients* that are separate threads of control. The well-known client-server architecture is an example of a repository architecture.

In diagram 5 is shown an plausible design based upon the repository architecture. Here input components store the values of input variables in a *single centralized store component* that the handler components then read the input values from. Thus the single store component acts as the repository while input and handler components act as clients.

If we compare diagram 4 and 5 they are quite alike. The difference in architecture stems from two important changes in responsibility: A) there is conceptually only a single store component serving all input and handler components and B) the store component encapsulate all network communication. We achieve point B) by designing the store component as a distributed database. All computational nodes has a local store component deployed, and these keep each other synchronized with respect to their shared contents over the network. For efficiency we find that a complete replication of at least state information must be maintained at each local store. The CAN bus is particularly well suited for implementing this type of distributed database.

This architecture also has the important property that several input components may assign/change values in a given store before the store broadcasts the values to the other stores in order to make them consistent. This way we can easily combine several changes into a single (CAN) telegram and this way lower the traffic on the network.

6. Discussion and Status

The two different architectural styles offer quite different abstractions to the programmers, which affect the maintainability quality. In the pipeline style, programmers are required to think and program the state information handling in terms of *flows of data and control between components*. In the repository style, in contrast, the main abstraction is that of *variable assignment and read*, and thus becomes the means of inter-component communication. These are fundamentally different mechanisms, each with benefits and liabilities. Flow of data and control can be implemented by asynchronous or blocking (remote) method invocations, which facilitates a strong object-oriented programming style. It can, however, also make the resulting deadline and timing analysis rather complex, especially in a push architecture.

Variable assignment is simple and does not dictate the flow of control. On the other hand, care must be exercised in the timing of tasks to ensure that values assigned to the store are available before the consuming nodes/components starts executing within each control interval. Variable assignment does not have side-effects which we find can reduce the risk of programming errors: if components only interact via variable assignment instead of via method invocations then analysis of execution time is greatly simplified because the thread of control never escapes out of the component itself! Variable assignment is, however, also a weak form of communication. Besides state information, event messages are also needed in process control, for instance to encapsulate user interactions that alter the opera-

tional mode. Thus the repository style cannot stand alone, there is still need for some event mechanism.

In both architectures, *scheduling* is important. Given the performance requirements it is absolute necessary that all sampled input values are properly propagated through the pipeline/repository to the control algorithm within the control interval—and that the individual tasks on the nodes are timely ordered so no task is executed too early for the necessary information to have arrived.

While the discussion above is an relevant, initial, analysis of the studied architectural styles, it is also representative of the type of discussions, we have had in the project team of researchers and industrial developers. Instead of focusing on performance tuning on the language near level, how to set the bit-patterns of the CAN identifiers, or the oddities of a given communication protocol, the terminology and concepts of software architecture, styles and qualities, have provided a strong basis for discussing and analyzing the real issues involved. The concept of architectural qualities has also ensured that we throughout the process kept *both* qualities in mind, performance *and* maintainability, instead of blindly striving for maximal performance out of fear of the harsh timing requirements. We thus find that it has help us keep the development teams attention tuned to all aspects of the system, not just a single one. We are confident that our further work will lead us to an architecture that is both performing well and easy to understand, develop, and maintain, and is flexible and open to future changes.

We are presently in the process of constructing *architectural prototypes* for further experimentation. The prototypes are based on the EVM-CAN evaluation board from Europe Technologies [5]. Prototypes are built both in the university laboratory as well as in the labs of the industrial partners.

At present, we have built a small RMI framework that allows normal blocking method invocations as well as asynchronous invocations over a communication network. The qualities that we strive for in this work are performance and small memory footprint on the execution side, and location transparency and the provision of the object-oriented method invocation paradigm on the maintainability side. The RMI framework is generic in the sense that the network protocol is defined as an adaptation of the framework, and presently the framework runs on both TCP/IP and CAN. The RMI framework will be used as implementation basis for the pipeline architecture prototypes.

In parallel, we are developing a distributed database component to be used as the store component in the repository style architecture. The store component uses the CAN broadcast protocol to send update messages to the other store components deployed on the nodes in the system. This implementation will of course become the heart of the repository architecture prototypes.

7. Conclusion

We have presented an approach to designing distributed embedded systems for closed-loop process control. The approach is used in a joint project between a team of software architecture researchers and development teams from two companies developing embedded systems. The approach is highly focused upon architectural issues and combines the terminology and techniques of software architecture research with that of embedded systems development. We find this combination fruitful and viable.

The concept of architectural styles is important in focusing the discussions and analysis on the real issues involved instead of beginning to design communication network and language near optimizations prematurely. The concept of architectural qualities is import in focusing the discussions on *all* the qualities of the system, instead of overemphasizing performance, and is instrumental in ensuring a flexible, maintainable, system that performs well.

We have outlined two viable architectural styles for closed-loop process control, and briefly discussed their respective benefits and liabilities. However, the main point is that these beneficial discussions were possible because the architectural terminologies and issues were understood and actively used by all members of the development team.

Furthermore, we have briefly outlined how the two architectures are used to construct architectural prototypes that will be used to study the implications in detail, both with respect to performance and with respect to maintainability. We hope to be able to report the results of these studies at a later date.

Acknowledgements

This research has been funded by the Center for Pervasive Computing [3]. We would like to thank the participants of the working group for many interesting discussions: Peter Andersen, Ole Eriksen, Finn Overgaard Hansen, Hans Peter Jepsen, Ole Lehrmann Madsen, Jesper Schmidt, and Henning Larsen Schmidt.

References

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

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

[3] Center for Pervasive Computing. www.pervasive.dk.

[4] R. DeLine. A Catalog of Techniques for Resolving Packaging Mismatch. In *Symposium on Software Reusability SSR 1999*, Los Angeles, CA, May 1999.

[5] Europe technologies: Evm-can evaluation board. http://www.europe-technologies.com/.

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

[7] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley Object Technology Series, 2000.

[8] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, Nov. 1996.

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

[10] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

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