2002-01-0875

# Automotive Software Development: A Model Based Approach

**Martin Rappl, Peter Braun**
Munich University of Technology – Department of Computer Science

**Dr. Michael von der Beeck**
BMW Group – Function Development Process

**Dr. Christian Schröder**
Telelogic AB Germany



## ABSTRACT

This article focuses on model based development of electronic control units (ECUs) in the automotive domain. The use of model-based approaches solves requirements for the fast-growing integration of formerly isolated logical functions in complex distributed networks of heavily interacting ECUs. One fundamental property of such an approach is the existence of an adequate modeling notation tailored to the specific needs of the application domain together with a precise definition of its syntax and its semantics. However, although these constituents are necessary, they are not sufficient for guaranteeing an efficient development process of ECU networks. In addition, methodical support which guides the application of the modeling notation must be an integral part of a model-based approach.

Therefore we aim at developing a so-called ‘system model’ which comprises all of these constituents: the modeling language AML its mapping to the Unified Modeling Language (UML) which represents the modeling language standard for object-oriented system development as well as a system of abstraction levels which will help the AML user to achieve a well-structured development process. Within this methodical framework we outline the use of the AML in this article by illustrating a case study which comprises parts of the body car electronics within a car. Used modeling concepts are discussed in detail by showing its correlation to the UML representation and its counterpart in the metamodel.

Long term goal of the project Automotive is to establish the AML as a de facto standard for the specification of embedded systems. The accompanying realisation of an integrated tool chain, comprising the tools Telelogic UML Suite /28/, ETAS ASCET-SD /27/ and Telelogic DOORS /28/ as well as the validation of the methodology by the different project partners facilitate this goal.

## INTRODUCTION

Within the automotive industry model based specification techniques are becoming more and more popular allowing the complete, the consistent, and the unambiguous specification of software and hardware for car specific networks of control units. In this context model based approaches provide methodical support to manage the integration of logical functions as interactive compounds within distributed networks of electronic control units. In addition well founded models are the source for analysis, validation, and verification activities.

The basis for the design of a model based specification methodology is a precise knowledge of the architecture of the targeted system class. In our opinion a good modeling approach regards different kinds of information which are necessary to implement a domain specific system. In Automotive our notion of the system architecture incorporates different kinds of information classes and their relationship among each other. During modeling all this information is stored in an integrated and consistent system model. To cope with the complexity of this model a system of domain specific abstraction levels provides an appropriate structuring mechanism for the specification of networks of control units on different technical levels. Within the system of abstraction levels, the specification of functions on a logical level is counted more and more among the core competences of car manufacturers.

Further a precise notion of the system architecture and its model based representation is fundamental for the transition from textual requirements specifications to first models and on the refinement of these models to an implemented system. The underlying model theory has to provide different kinds of modeling concepts to regard the different kinds of information to be modeled and the model theory has to support refinement relationships between every kind of information. Beyond the notion of

a version and the notion of a configuration has to be an integral part of the theory.

Further the model theory also forms the basis for the use of different kinds of notations. The presented work is related to the work the OMG /15/, the U2 Partner group /24/, and the pUML group /6/ are carrying out. In contrast to their approach we believe, that a more rigorous mathematical theory is necessary for a realistic model based development process. Nevertheless the representation of abstract model concepts is done by specifically adapted notations from the UML 1.3 /16/, the ASCET-SD modeling language /27/, and textual specification techniques which stem from the tool DOORS /28/.

STRUCTURE OF THE PAPER – The paper introduces central terms used in this article and in the context of model based systems development. Thereupon the concepts for a seamless and integrated development methodology with the focus on logical functions and on the functional network are illustrated.

- In the first section the principles of model based development are described. Therefore, central terms like *system model*, *Automotive Modeling Language (AML)*, *metamodel*, *semantics*, *abstraction levels*, *notation* as well as *mathematical foundation* are introduced.

- Following a system of abstraction levels is presented which provides a mean of structuring the system model as well as the development process.

- The subsequent section focusses on two of these abstraction levels to provide insights into the modeling theory. At first the functional level is described. On this level (complete) building blocks of the system under development are modeled. In the subsequent section, the network level is presented which first deals with the tailoring of functions to particular functional variants and second specifies the interaction and communication pattern among them. In both cases the usage of the UML notation artifacts is demonstrated by the application to a case study example from the automotive domain.

- In the final section we draw our conclusions and provide an outlook to future work.

THE PROJECT AUTOMOTIVE – The work presented in this paper represent results of the research project FORSOFT Automotive – Requirements Engineering for embedded systems /25/. The partners of this project are the Munich University of Technology, the tool providers Telelogic and ETAS, the car manufacturers BMW and Adam Opel as well as the suppliers Robert Bosch and ZF Friedrichshafen.

## PRINCIPLES OF A MODEL BASED DEVELOPMENT

In contrast to conventional structured programming methods model based methods provide a comprehensive support for all kinds of issues imposed by modern system development processes. This support comprises activities ranging from the modeling task itself to requirements tracing as well as version, configuration, and change management. Furthermore the produced models constitute the basis for applying formal verfication and validation techniques.

But unlike the existence of the Backus Naur Form (BNF) used for the syntax definition of textual programming languages, there exists no commonly accepted standard for the conceptual design of model based development languages and methods. In Automotive we use an algebraic specification approach for the definition of the model based specification method. In Figure 1 all relevant concepts and their interrelationships are shown. These concepts are constituents for the definition of the automotive specific development process, which is the focus of our research. The highlighted concepts of Figure 1 are discussed in detail in the following sections of this article.

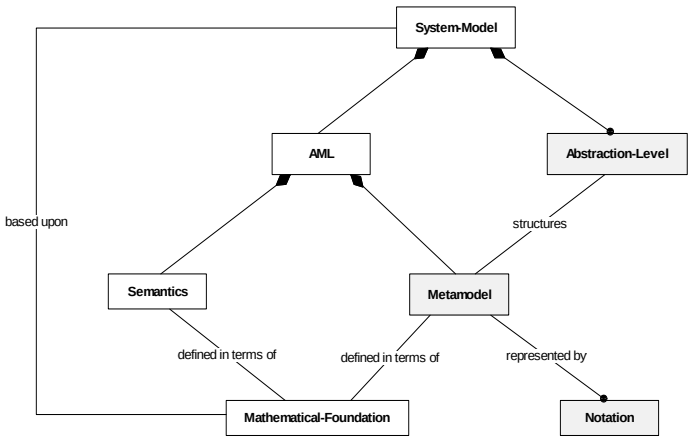


**Figure 1: Basic Modeling Concepts**

**System Model –** The system model contains all theoretical foundations necessary for the development of models within the specific domain of Automotive. The system model itself is based upon a mathematical foundation and consists of the Automotive Modelling Language (AML) and the Automotive specific system of abstraction levels. As this paper presents current work in progress, Figure 1 has to be completed by the specification of a corresponding development process. This affects future work of the project Automotive.

**Automotive Modeling Language –** The AML consists of a metamodel part and a semantic part. In the context of the project Automotive the terms *abstract syntax* and *metamodel* are used as synonyms. The AML

defines a language which is specific for the development of embedded automotive systems. The AML itself contains the abstract modeling concepts used in this language as well as their meaning.

**Metamodel –** The definition of a clear and unambiguous metamodel of the AML constitutes a main purpose within the project and is primary addressed in this paper. The metamodel is defined in terms of a mathematical foundation and is structured by a system of Automotive specific abstraction levels. The metamodel represents abstract modeling concepts which are used to build an integrated model. A model contains all necessary information about the functional logic, the distributed network of electronic control units, the actors, the sensors, and the environment. The metamodel structures this information and describes the possible interrelationships. For representing the metamodel of the AML a restricted subset of UML class diagrams is used together with metamodel patterns as a rationale. Besides pure class diagram constraints are defined to ensure a basic consistency level.

**Semantics –** One big advantage of model based development methods shows up when one needs to handle complex system specifications and wants to reason about it. The semantic part describes the static and dynamic semantics of models. It is defined in terms of the mathematical foundation and by using the metamodel, at least theoretically. However, a complete and precise semantics defintion is not a main focus of the project Automotive. Only some fragments of this definition will be worked out within Automotive. Rather for the use of the AML the defined semantics and their mathematical foundation won't be visible.

**Abstraction Levels –** The information gathered during the development process is structured and stored using different levels of detail. The abstraction levels precisely define which sort of information is stored at which level. Beyond this the abstraction levels are used to structure the metamodel which in turn allows to specify dependencies among the information defined in an actual user model. The definition of appropriate abstraction levels and their usage is a core activity of the project Automotive (see next section).

**Notation –** Textual and graphical notation elements are used to model specific parts of a system. The concrete syntax of those notational elements has to conform to the metamodel. Within Automotive a restricted subset of the current UML notation standard is used to represent some parts of a model. The other parts are represented by a subset of notations offered by ASCET-SD, a tool for the design and prototyping of embedded systems. Also textual notations like structured (hyper-)text and tables are used especially in the earlier phases of the system development process.

The main reason for using UML within Automotive is the growing importance of this language as a modeling standard within the area of software engineering. We make use of UML model elements to represent parts of the concrete graphical syntax of the AML. To be aligned to the actual language standard we first define a UML profile. The resulting modeling concepts will be revised with respect to their applicability, efficiency, and expressiveness in the realm of automotive software development.

However, examining the expressiveness of the UML with respect to its application to the development of embedded systems one already realizes fundamental weaknesses which consequently have an impact on the expressiveness of the AML representation as a UML profile /9/. There are several approaches addressing that issue, e.g. the merge of UML and SDL 2000 /2/. These approaches are currently being elaborated and evaluated by the U2 partners group /24/ during the ongoing UML 2.0 standardization.

**Mathematical Foundation –** The mathematical foundation contains formal definitions of some terms which are used to describe metamodels. Furthermore definitions and operations are defined here to be used in the semantic part of the AML. With these definitions in mind also a formal definition of the abstraction levels could be given. The mathematical foundation contains work currently in progress and only some fragments of this foundation will be defined within the project Automotive.

## ABSTRACTION LEVELS OF EMBEDDED SYSTEMS

To cope with the complexity of system models, abstraction levels are introduced. Abstraction levels have been successfully applied in the field of hardware design /13/ and in the field of protocol design (ISO/OSI reference model /23/) For the design of a systems engineering methodology abstraction levels can be utilized in several ways. First, abstraction levels act as means of structuring and integrating large information sets. Thereupon rests the construction of a seamless development processes. Second, abstraction levels structure the definition of activities within the development process and therefore they provide methodical advise. Third abstraction levels can be utilized to structure the model theory in an uniform manner. This facilitates the design of a model based specification methodology. In the next sections the role of abstraction levels is discussed from a mathematical, model oriented point of view as well as from a methodical, process oriented point of view. Subsequently the system of automotive relevant abstraction levels is sketched.

Abstraction levels define restrictive views upon the system model. A view within an abstraction level shows the system on a uniform technical level. For the development of networks of control units six different abstraction levels are defined /Figure 2/.

From a mathematical point of view a system model defines a collection of sets containing all necessary information to derive an implementation for control units. Each abstraction level provides a certain view as well as a certain accessibility on a subset of these information sets. The defined levels build upon themselves. Model information and logical dependencies defined in higher abstraction levels are accessible in the lower ones. The information is successively refined by adding more technical information of new modeling concepts within lower abstraction levels. Those modeling concepts introduce new information as well as relationships to existing information of higher abstraction levels. To assure consistency of the complete system model a set of consistency constraints is defined.

From an engineering point of view, the abstraction levels guide the analyst through the process of requirements elicitation and requirements structuring. They facilitate the process of writing down ideas and visions an engineer (or an other stakeholder) has in mind for the functioning of the system under development. The system of abstraction levels gives assistance for the structuring process of ideas. Rather it follows requirements engineering principles for a successive refinement of informations. Starting with abstract models which just provide a minimum set of technical details, at each abstraction level new, more technical oriented concepts are introduced which explicate the afore modeled information. The specification starts with the definition of signals and their ordering and ends with the specification of the central processing unit, the communication infrastructure, and the realtime operating system /Figure 2/.

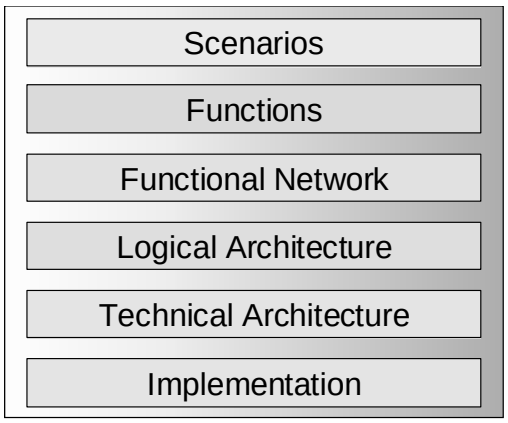


**Figure 2: Automotive Relevant Abstraction Levels**

The division of the system model with the help of abstraction levels determines different semantic properties which are characteristic for each level /Table 1/. These semantic properties depend on the used modeling concept and define the dynamic semantics of the used notation. So for example in higher abstraction levels, systems are event driven and triggered by a global system time frame. In lower, more technical oriented levels systems are clock triggered and data driven. The global time is split into a set of local times each related to a control unit. The same is true for the transition from asynchronous to synchronous communication principles.

| | Execution | Communication | Time |
|---|---|---|---|
| **Semantic Properties** | clocked/ event driven | synchronous/ asynchronous | synchronous/ asynchronous |
| **Scenarios** | event | synchronous | synchronous |
| **Functions** | event | synchronous | synchronous |
| **Functional Network** | event | synchronous | synchronous |
| **Logical Architecture** | data | synchronous | asynchronous |
| **Technical Architecture** | data | asynchronous | asynchronous |

**Table 1: Semantic Properties of the System Model**

In the following paragraphs each abstraction level is described separately to show its role in the development process. In the subsequent sections, the functional level and the functional network level are explained explicitly in terms of concepts, UML representation and the underlying metamodel.

**Scenarios –** The abstraction level scenarios contains model information about the system with the lowest amount of technical details. At this level there is a big universality in enriching the model with technically oriented information. The core modeling concept at this stage is the concept of a signal and operations on signals (actions). This concept provides in addition to an ordering mechanism enough modeling power to describe scenarios. Scenarios are ordered sequences of actions which are necessary to achieve a determined goal in a certain context. Scenarios describe exemplary use cases, ‘not use cases’ showing how a system must not be used, possible exceptions or test cases.

One intuitive way to interpret this model is to observe the occurences of actions. There is no mechanism to restrict the visibility of signals and there is no need for exchanging actions. These also holds for the time. Therefore both, the communication mechanism and the (global) time can be regarded as synchronous.

**Functions -** Functions define building blocks of the to be developed system. In contrast to scenarios functions describe complete sequences. The view upon the function at this high abstraction level is independent of later used implementation techniques or target languages. Particularly functions are considered which behave as abstractions of the later used control units, actors, sensors or the environment.

The set of functions is structured to allow easy navigation between the usually great number of different functions of a system. Therefore several different hierarchical structures have been defined to locate functions for a deeper understanding and easier modification. Each structure shows at least a clipping of the functions of the whole system. Different hierarchies of functions result from different levels of experience or different views upon a system by different developers.

As mentioned above each function divides the systems behavior into manageable pieces. For reuseability each function restricts the scope of signals and requires a mechanism for exchanging signals. By exchanging signals the overall systems behavior is achieved through interaction. A communication mechanism can be realized without great expense by a synchronous communication pattern between subfunctions. Due to the high abstraction level at this stage, a global time ensures the synchronous execution of the observed functions.

**Functional network –** The purpose of the functional abstraction level is to define generic and reusable building blocks so that they can easily be adapted to different system development contexts. The formed functions serve as a framework for the definition of functional variants. At the abstraction level of the functional network, the tailoring of functions to specific functional variants is considered. Therefore required functions are selected and variants are formed. This generally leads to an enlargement of the namespace. The different variants are connected to gain information about their signal dependencies. Mutual dependencies between different functions within these variants are displayed at this stage.

The introduced modeling concepts at this stage facilitate the use of functions in different contexts and do not add new technical information. The introduced modeling concepts do not influence the semantic properties inherited from the functional level.

**Logical architecture –** By developing the logical architecture the model is enlarged with information belonging to orthogonal modeling concepts. Concurrently gained information from the functional network is now refined. The new, orthogonal modeling concepts deal with the logical distribution (information about logical control units) and the information about actors and sensors of the environment. The different variants defined at the functional network level are instantiated and deployed on the different logical control units.

At this stage the system is treated to be clocked instead of being event driven. For the communication of architectural entities a signal exchange has to be assured at each clock. Furthermore each clock enables the system to perform operations on the state space. The actions of the higher abstraction levels can now be interpreted as an operation on the state space at each step under the premise of the occurrence of the action. It is important that time is split into a real time part (universal time) of the environment and into a system time part which has its own clock. The adjustment of those different times is crucial for proper operation of the system. The adjustment is not done continuously. Instead it takes place at certain peaks. The peaks depend on the tolerance which is valid in weak real-time systems. Hard real-time systems do not permit tolerances at all.

Due to the abstraction of technical informations at this level, the communication among different logical control units and the communication within function instances deployed on logical control units are regarded as synchronous. At a more technical level this perfect synchrononity has to resolved by an asynchronous communication model.

**Technical architecture –** The technical system architecture offers further modeling concepts describing concrete technical information. So the model is enriched with information about concrete bus-realisations and conrete control units. Further information describes control unit specifications and operating system descriptions. So it is possible to gain first performance estimates.

Comprehensive experience from the development of electronic control units reveals that a clear separation within a logical system architecture level and a technical system architecture level is very helpful when it comes to the partitioning of functions on electronic control units. On the logical architecture level only a subset of partitioning criteria is applied in order to achieve a clear view of the functional structure - without identifying the set of functions which constitutes an ECU. However, finally the complete set of partitioning criteria (e.g. also those which consider geometric requirements) has to be applied. This will be done on the technical architecture level and results in the identification of the set of electronic control units. To each of them a set of to be realized functions is assigned.

Since control units as well as a detailed technical model is regarded at this stage, the synchronous communi-cation mechansim is replaced by a asynchronous one. This is done due to the fact of the model robustness. As each control unit can be replaced by a physical one, the failure of a control unit must be considered at this stage. With a synchronous communication model the overall system would be blocked or would malfunction.

**Implementation –** At the implementation level the realization of the model is done in hard- and software. This goes beyond the first prototypes which could be generated from the models gained above. For example often code has to be optimised for performance and size reasons. Beside this often adoptions to existing systems have to be made.

## THE FUNCTIONAL LEVEL

CONCEPTS – The core modeling concept at this abstraction level is the concept of the function /Table 2/. A function provides an encapsulation of behavior at a logical level. Functions act as basic building blocks for further development activities such as tailoring of variants, instantiation, deployment and so on. Each function can be regarded as a single service which has to be fulfilled by the whole system. For structuring functions, uniform structuring rules are applied to ensure a uniform decomposition process. Therefore, modeling concepts like signal delegation resp. propagation and coordinator functions are introduced.

The role of these concepts is similar to the concepts of order, request and inquiry known from the CARTRONICS /10/ approach. A more general view is given regarding the concepts of the recursive control pattern /19/. This architectural pattern can be applied in the following context: a complex system shall be divided into subsystems containing a significant amount of control. The recursive control pattern provides the following solution: Control parts and functional parts are decoupled within each component. More concrete, within each hierarchically structured function *f* a coordinator function exists which gathers all signal flows

- outside *f* before each of them is delegated to one of the (other) subfunctions of *f*, or
- of the (other) subfunctions of *f* before they are propagated outside *f*.

The first case is called *delegation*, whereas the second case is called *propagation*. Both cases are subsumed under the term *vertical communication*.

In contrast to another well-known architectural pattern used in software-engineering technology (namely the subsystem controller pattern /20/) the recursive control pattern allows that two subfunctions of a hierarchically structured function can be directly connected by signal flows in order to achieve horizontal communication. However, we do not require the application of the recursive control pattern for each hierarchically structured function.

We now come to the concepts for structuring functions and displaying the mutual data dependencies between two of them. At this point we follow recent work on architectural languages which recommend the use of ports and connectors.

| **Abstraction Level:** *Functions* | |
|---|---|
| Hierarchically structured functions | **Diagram Type:** *Class Diagram* |
| **Modeling Concept** | **UML Representation** |
| **Functions** | Classes |
| **Hierarchical Structure** | Decomposed Classes |
| **Environment** | Stereotyped classes |
| **Inports** | Interface classes resp. Lollipops |
| **Outports** | - no appropriate construct available - |
| **Signal Dependency** | Dependency arrow |
| **Propagation** | Dependency arrow between function and environment |
| **Delegation** | Dependency arrow between environment and function |

**Table 2: UML Representation of Functional Concepts**

**Functions –** Functions are basic building blocks structuring the total amount of behavior into manageable pieces. In UML functions are represented as classes. For clarity reasons classes are shown in a folded state. Hence this kind of class visualization hides information about attributes and methods.

**Hierarchical Structure –** Each defined function can be decomposed into a set of subfunctions. To each decomposed function a coordinator may be bound. A coordinator resolves functional dependencies between functions on the same hierarchical level. Furthermore the coordinator function has some responsibility for controlling the signal flow via vertical communication /Figure 3/. For representing the functions in UML we use hierarchically structured classes. For the representation, one clearly has to distinguish between an external view and an internal view. The external view shows a function from the outside. An internal view shows all included subfunctions and the decomposed function as a class with the stereotype <<environment>>. This enables one to represent all vertical communication relations in symmetry to the horizontal communication relations.

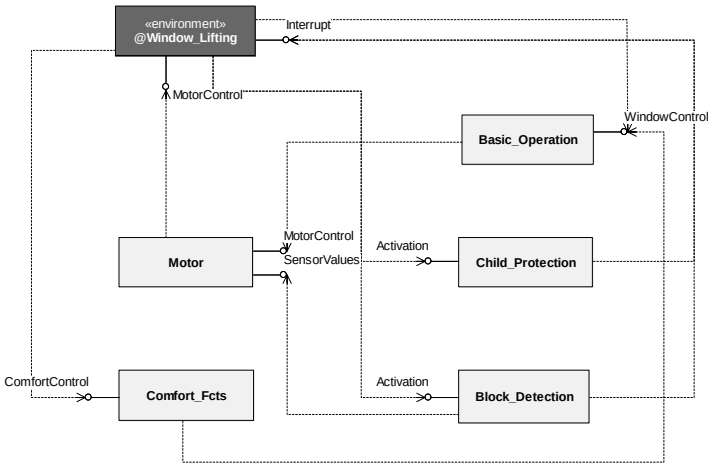


**Figure 3: Hierarchically Structured Functions**

**Environment –** In contrast to the listed concepts /Table 2/ the environment takes up an exceptional position. The environment represents the compliment of the function itself. That means it contains all specified functions except the function itself and all composd subfunctions. The environment is used for the definition of the vertical signal flow. It is represented in UML as a regular class. That ensures the binding of an interface

class (lollipop). By that a perfect symmetry can be reached between the vertical and the horizontal signal flow. In the metamodel the environment does not have a counterpart. That's due to the fact, that all informations can be calculated from the external view. In the UML the environment is displayed as a class, attached with the <<environment>> stereotype. For qualification reasons the name is prefixed with an '@' character.

**Inport –** The communication between functions is handled via signal exchange. A set of signals is combined to a port. The set of specified and used ports can be classified into inports and outports. Inports aggregate signals that the function is offering to other functions to provide some services. Outports in contrast aggregate signals that the function is requiring from other functions for operation. The ports themselves are represented as lollipops which are attached to classes. In the UML the lollipop is a shorthand for an abstract class which contains interface information and is inherited to the class, where the interface is realized. In the automotive methodology the definition of the signals is done by interface classes /Figure 4/. This ensures a reuse of ports. Each attached interface class represents an instance of a port definition.

**Outport –** In the Automotive methodology outports are not stated explicitly. This limitation depends on the expressiveness of the current UML standard which is not offering an appropriate modeling construct. Of course we could use colored lollipops to distinguish between in- and outports, as some work suggests /17/, but in our opinion this leads to a lot of confusion by misinterpreting the meaning of familiar symbols. For a further discussion of an appropriate representation of outports please refer to the documentation of architecture description languages like UML/RT /19/ or AutoFocus /21/.

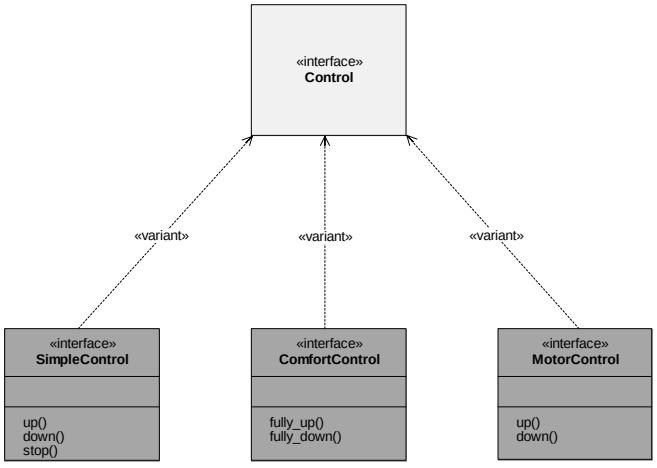


**Figure 4: Exemplary Signal Definition**

**Signal Dependency –** Within the modeling concepts of the AML a horizontal and a vertical signal flow are differentiated. The horizontal signal flow denotes the communication of functions within one functional level. The vertical signal flow denotes the communication between two levels of functions (as stated above). Due to the symmetry of modeling concepts and due to the introduction of an element depicting the environment, the horizontal and vertical flow of signals can be identically represented as UML dependency arrows.

**Delegation/Propagation –** Depending on the communication direction the vertical signal flow is called delegation resp. propagation. Delegation denotes the flow of signals from a function down to the composed subfunctions, whereas propagation denotes the opposite direction of the signal flow.

EXAMPLE – In Figure 3 a simplified model of the window lifting functionality is sketched. This function is part of the body car electronics within a car. It is modeled by the element *Window_Lifting* and is further decomposed into five subfunctions. The diagram shows an internal view of this function. An external view of the function *Window_Lifting* is included in Figure 6.

The class *@Window_Lifting* attached with the stereotype <<environment>> denotes the environment of the function *Window_Lifting*. The environment is composed of all specified functions except the function itself and its composed subfunctions. The role of the model element environment is to visualize the vertical signal flow between the environment and the function resp. subfunctions. So for example the interface *MotorControl* of *@Window_Lifting* shows the communication of the function *Window_Lifting* with some functions outside of *Window_Lifting*, by exchanging signals which are compatible with the interface specification of *MotorControl*. By using an explicit model element for the representation of the environment, a symmetric port concept can be realized within the AML.

The inports attached to the specified functions are defined explicitly by interface classes. Figure 4 shows the interface *Control*, which is called a signal in terms of the AML. The definition comprises a set of signals which can be exchanged. *Control* is the composition of the signals: *up*, *down*, *stop*, *fully_up* and *fully_down*. Note that those signals are not shown explicitly in the picture. Figure 4 shows some variants of *Control*, which use subsets of these signals. So for example the signal *MotorControl* includes the signals *up* and *down*. The definition of those signals and their corresponding variants is used for a complete specification of functions. *Basic_Operation* /Figure 3/ uses the interface *MotorControl* of the *Motor* by sending signals to it.

METAMODEL – As stated above the metamodel provides abstract modeling concepts of the AML. Again the system of abstraction levels is used at the meta level as a structuring mechanism. Figure 5 shows the static part of the metamodel at the functional level. For the representation of the metamodel a restricted subset of the UML class diagram notation is used. For the description of the metamodel used keywords are written in *italic* to ease reading and navigation.

To provide a more fine granular and uniform structuring mechanism, metamodel patterns are used for the definition of modeling concepts on every abstraction level. The metamodel of the AML is defined by the help of three different patterns. A very simple pattern is the hierarchy pattern. The hierarchy pattern is used for the definition of arbitrary hierarchical structures. In Figure 5 this pattern is applied for the definition of the metamodel element *Function*. The other two patterns are called variant pattern and instance pattern. Their application is shown at the network level.

The introduction of hierarchical structures, variants and instances as modeling concepts implies a general mechanism of how to proceed with any hierarchical structures. The process itself starts with the definition of hierarchical structures, continues with the tailoring of variants, and ends with the instantiaton of variants. The instances serve as an input for the lower abstraction levels. At the functional level /Figure 5/ *Instantiable-Signals* are taken as an input of the higher abstraction level scenarios. *InstantiableSignals* denote the set of signals which have passed the variant formation and instanciation process. Within the AML this mechanism is applied at several abstraction levels. Its application to the functional structure is explained in the network level section in more detail.

For a better understanding we have chosen different colors to distinguish between model elements, variants and instances. Classes which define model elements on the modeling level are shown in yellow[1]. Classes which define variants on the modeling level are shown in red[2]. Classes which define instances on the model level are shown in cyan[3].

The modeling concept of the function is represented by the class *Function* including the attribute *Name*. A *Function* composes an arbitrary number of *Subfunctions*. In Figure 3 the function *Window_Lifting* represents an instance of a function. This function is hierarchically structured and can be decomposed into five subfunctions. The modeling concept of the function is represented as a class in UML. The class name shows the name of the function.

Corresponding to UML classes, AML functions have local attributes. In terms of the AML, these attributes are called *Variables* /Figure 5/. A *Variable* is a specialization of an *InstantiableSignal* and therefore inherits its name and type. The metamodel element *InstantiableSignal* is defined in the upper metamodel part belonging to the abstraction level scenarios. The related diagram is not sketched in this paper. *InstantiableSignal* is used to represent a constant, a UML class-attribute, a UML class-method, or a UML interface class. The AML concept of a signal facilitates to model even more complex structures which go far beyond the simple definition of attributes and methods. But note, in UML these structures can not be specified adequately. Next to the definition part of signals, the metamodel of the upper section also provides concepts for tailoring signals to variants whereas another part deals with the formation of instances. At the functional level only the set of *InstantiableSignals* is taken as an input containing instances of signals.

While taking a look at the functions shown in Figure 3 we can see the explicit specification of their interfaces. In UML-class diagrams the port concept is represented by interface classes (lollipops) to show at least an unidirectional interface specification which is provided by

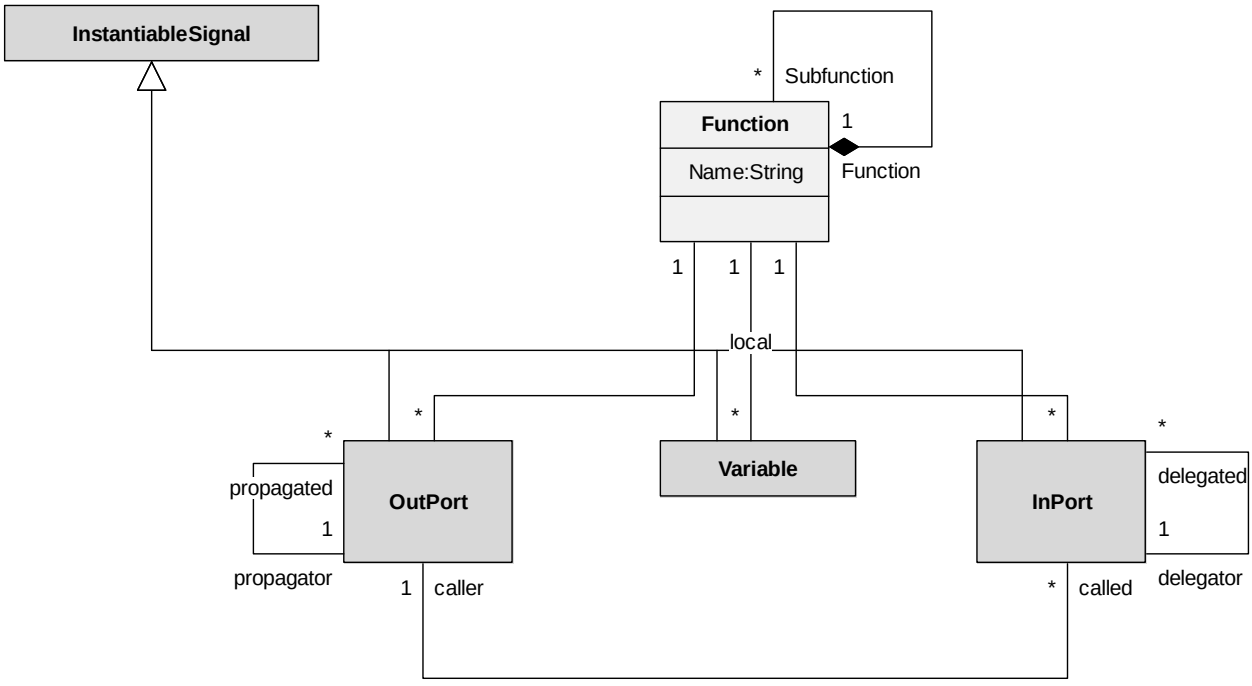


**Figure 5: Metamodel for Abstraction Level Functions**

each function. So for example the function *Motor* provides *MotorControl* and *SensorValues* as *InPorts*. An *InPort* is an *InstantiableSignal*. In Figure 4 the definition of *MotorControl* is derived from the more complex *Control* signal by the formation of a variant. In the model /Figure 3/ *MotorControl* already acts as an instance. This facilitates the direct use of *MotorControl* on the functional level. The process of building variants and instances can be shortened in order to restrict the namespace and of course to keep the diagrams lean.

In the metamodel the call relation as well as the delegation and the propagation of signals are modeled by one-to-many associations. For representing mutual functional dependencies on the same hierarchical level, dependency arrows are used to model call relations between the function and ports. For example the *MotorControl* interface is called by the *BasicOperation*. As described above delegation and propagation are call relations between functions at different hierarchical levels. For representation dependency arrows are taken again. For example the *WindowControl* signal of *Window_Lifting* is delegated to *Basic_Operation*.

## THE NETWORK LEVEL

CONCEPTS – The set of defined functions is building up a framework for further development activities. Within this framework, the elaborated functions are basic building blocks which can be put together in various combinations. Due to the plurality of similar control activities within a car, a great number of variants exists for each single function. These are made up of various combinations of their subfunctions. When defining variants, new signal dependencies arise /Table 3/. These signal dependencies emerge from the specified variants and are not contained in the model of the previous abstraction level. After tailoring functions to functional variants we can construct their instances. These instances can be deployed on logical control units at lower abstraction levels.

| **Abstraction Level:** *Functional-Network* | |
|---|---|
| Definition of functional variants | **Diagram Type:** *Class Diagram* |
| **Modeling Concept** | **UML Representation** |
| **Functions** | Classes |
| **Variants** | Classes |
| **Tailoring** | Functional Structure (see Table 2) |
| **Inports** | Interface classes resp. Lollipops |
| **Outports** | - no appropriate construct available - |
| **Signal Dependency** | Dependency arrow |

**Table 3: UML Representation of Functional Network Concepts**

**Variants –** Using variants of general functions allows to specialize common functional behavior according to the particular subsystem needs /Figure 6/. From a methodological point of view the relation between functions and variants allows to manage complexity by abstracting specialized behavior to general needed mechanisms. In contrast to the concept of inheritance in object orientation building variants just means to select subfunctions from the function pool of general functions. A Variant has a name and consists of a list of Variant Interfaces (similar to Function Interfaces).

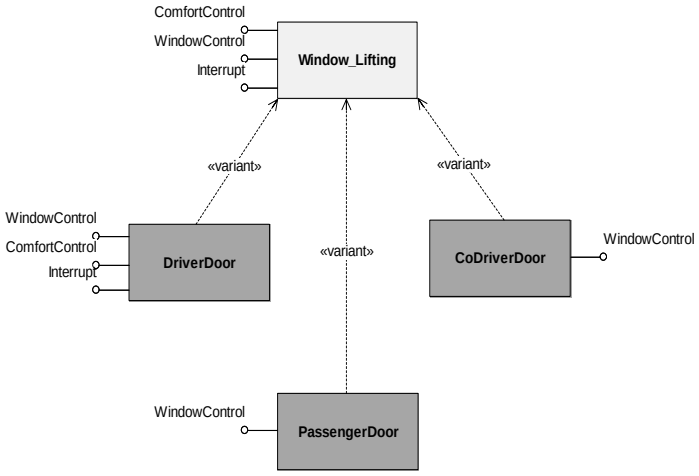


**Figure 6: Definition of Functional Variants**

**Tailoring –** The process of tailoring functions to variants takes place within two steps. In step 1 the set of variants associated with a function is stated explicitly. In step 2 subfunctions and subvariants are selected for each specific variant. Variants are displayed in UML as arbitrary folded classes marked in red color. These variants are bound to the derived function by a dependency arrow attached with a <<variant>> stereotype. For the specification of the selected subfunctions and the selected subvariants the same modeling concepts are used as they are used for the specification of subfunctions at the functional level. Therefore all specified concepts such as inports, signal dependencies (horizontal + vertical communication) are used at this stage as well.

**Inports** – Analog to the representation of functions, inports are attached to variants. Here again interface classes are used. The collection of inports attached to variants primarily depend on the selected subfunctions resp. subvariants. As variants select a subset of functions of the original functional decomposition, the collection of inports attached to a variant is a subset of the original inport collection as well. In Figure 6 three variants with differing interface specification are stated.

**Signal Dependency –** At the functional network level new signal dependencies arise between different variants. These signal dependencies are displayed in the same way as on the functional level by using dependency arrows. The associated diagram is called functional network in terms of the AML.

EXAMPLE – As described above tailoring of functions to variants is one essential step. Figure 6 shows the definition part of three variants of the function *Window_Lifting*. Each of the defined variants selects a certain subset of subfunctions. The diagram also shows the interface provided by a variant. The set of offered ports of each variant is a subset of the ports attached to the corresponding function. The same is true for the subfunctions of *Window_Lifting*. For example the *DriverDoor* doesn't include the *Child_Protection*. The explicit declaration of subfunctions of a variant looks similar to the diagram shown in Figure 3.

On the network level inter-functional dependencies between the defined variants will be described. Figure 7 shows an example where the dependencies between the variants of Window_Lifting are described. In this example the DriverDoor has some advanced functionality for the remote operation of other windows. The meaning of the dependency arrows is exactly the same as it is at the functional level.

METAMODEL – Figure 8 shows the modeling concepts for the network level. This part of the metamodel deals with the definition of variants and instances of functions. As stated above we use patterns to build up the AML metamodel. At the functional network level we make use of the variant and the instance pattern. We instanciate them to provide modeling concepts for the formation of variants and for the definition of instances. The particular combination of elements, variants, and instances is not only used at this stage it is also used in other parts of the metamodel, e.g. within the definition of signals or in more technical oriented levels.

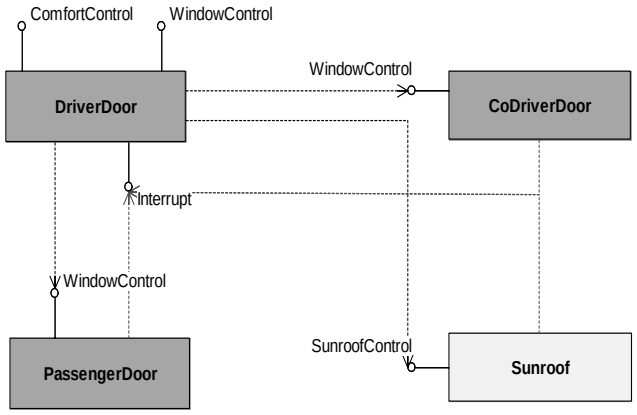


**Figure 7: Signal Dependencies Between Variants**

For the definition of functional variants the set of functions which stems from the functional level is taken as an input. In the instance of the variant pattern each *Function* can be tailored by many *FunctionVariants*. For example the function Window_Lifting is tailored by the functions *DriverDoor*, *CoDriverDoor* and *PassengerDoor* /Figure 6/. *FunctionVariants* may not exist without the *Function* they are tailored from. Each *FunctionVariant* selects functionality provided by its corresponding *Function*. For example the *DriverDoor* selects all subfunctions of *Window_Lifting* except *Child_Protection*, as the *DriverDoor* provides no such functionality. This select relation is only valid for direct subfunctions of the tailored *Function*. *FunctionVariants* can be defined hierarchically that means a *FunctionVariant* can contain further variants of functions. On the other hand *Functions* may be used within the definition of variants at the same time. This is a necessary 'shortcut' to describe

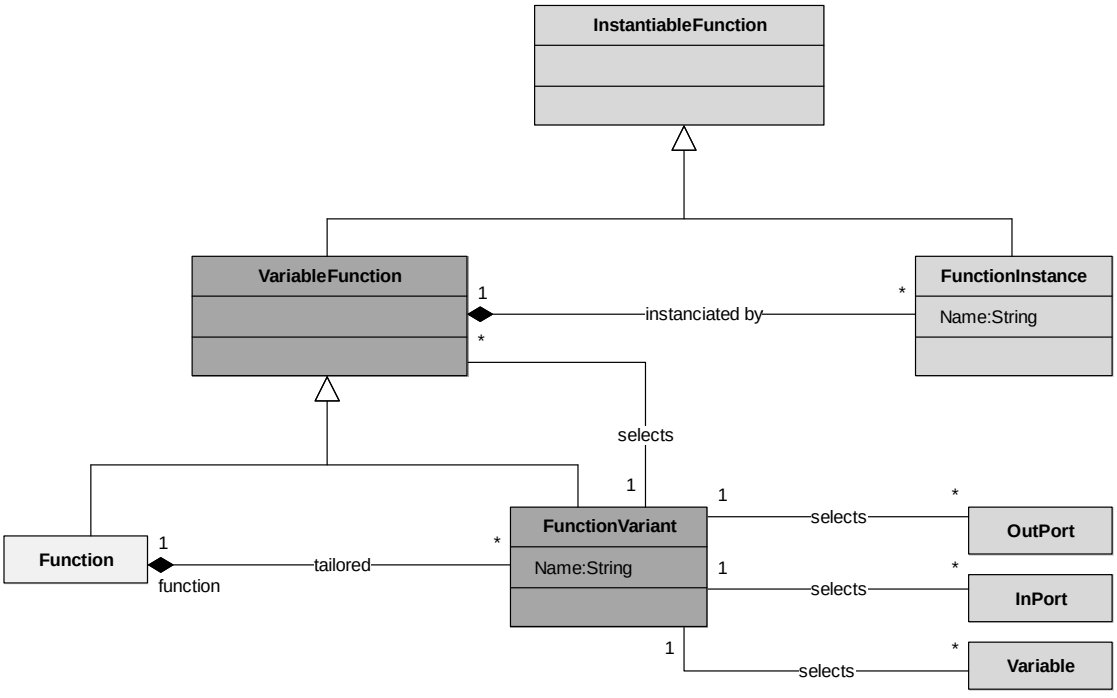


**Figure 8: Metamodel for Abstraction Level Functional Network**

the situation that every function may be used as a variant itself holding all of its subfunctions. To avoid a blow-up of the namespace we make use of the superclass *VariableFunction*. In this context a function may play the role of a variant. Besides the selection of functions, also the *OutPorts*, *InPorts*, and *Variables* of the tailored *Function* have to be selected. So for example the PassengerDoor only provides the interface *WindowControl* of all possible interfaces of *Window_Lifting* /Figure 6/.

The last pattern used for the construction of the metamodel /Figure 8/ is the instantiation pattern. Every *VariableFunction* can be instantiated in an arbitrary number. Taking *VariableFunction* as an input for the instanciation pattern, variants as well as functions can be instantiated. Here again a 'shortcut' is introduced to ease the specification. *InstantiableFunction* is the superclass of *VariableFunction* and *FunctionInstance*. So *Function* and *FunctionVariants* act as *Function-Instances*. Again no auxiliary name has to be introduced for the instantiation of a variant resp. function.

## CONCLUSION

Motivated by the aim to meet the challenges of developing complex networks of heavily interacting electronic control units (ECUs), we have presented a system model comprising all necessary constituents for a model-based software development in the automotive domain. Three essential ingredients of this system model have been presented in this article:

- A system of abstraction levels to structure the metamodel as well as the user model and
- the meta model resp. abstract syntax of the AML (Automotive Modeling Language) as well as
- the representation of the AML by means of the UML aligned to the current UML standard.

Whereas the AML establishes the basis for an adequate modeling of software in the automotive domain - especially for ECU networks – the system of abstraction levels additionally provides means for structuring the development process according to different domain-specific categories. Our application of these concepts to a common realistic example, a window lifting control system, reveals the benefits of applying our approach.

Future work will cover the following two directions:

- First of all, we plan to complete the automotive specific system model: On the one hand this comprises the formal and complete definition of up to now informally and insufficient described parts. For example consistency dependencies between two adjacent abstraction levels have to be defined in an unambiguous way. On the other hand the concrete development process has to be defined based on the system of abstraction levels by providing rules and heuristics, how to use these levels.
- Second, the tool supported transformation from non-executable models to executable, target-dependent code will be explored in order to achieve the long-term objective of a seamless, complete software development process for automotive applications.

## ACKNOWLEDGMENTS

We thank Jianjun Deng for carefully reading draft versions of this paper. We are much obliged to our colleages of the project Automotive for many fruitful discussions and we thank Manfred Broy for directing this research. This work has partially been funded by the Bayerische Forschungsstiftung (BayFor) within the Forschungsverbund für Software Engineering II (FORSOFT II).

## REFERENCES

1. Jesper Andersson: Die UML echtzeitfähig machen mit der formalen Sprache SDL, OBJEKTspektrum 3, 1999.
2. Morgan Björkander: Graphical Programming Using UML and SDL, IEEE Computer, Vol. 24, 2001.
3. Peter Braun and Martin Rappl: Model based Systems Engineering - A Unified Approach using UML, Systems Engineering - A Key to Competitive Advantage for All Industries Proceedings of the 2nd European Systems Engineering Conference (EuSEC 2000), Herbert Utz Verlag GmbH, München, 2000.
4. Peter Braun, Martin Rappl and Jörg Schäuffele: Softwareentwicklungen für Steuergerätenetzwerke – Eine Methodik für die frühe Phase, VDI-Berichte, Nr. 1547, S. 265 ff., 2000.
5. Michael von der Beeck, Peter Braun, Martin Rappl and Christian Schröder: Modellbasierte Software-entwicklung für automobilspezifische Steuergeräte-netzwerke, VDI-Berichte, Nr. 1646, S. 293 ff., 2001.
6. Brodsky, Clark, Cook, Evans and Kent: Feasability Study in Rearchitecting the UML as a Family of Languages Using a Precise OO Meta-Modeling Approach, The pUML Group, 2000.
7. A.-P. Bröhl and W. Dröschel: Das V-Modell – Der Standard für die Softwareentwicklung mit Praxis-leitfaden, R. Oldenburg Verlag München Wien 1993
8. Manfred Broy ++: The Design of distributed Systems, An introduction to FOCUS – Revised Version, Technical Report, TUM-I9202, Technische Universität München, 1993.
9. Manfred Broy, Michael von der Beeck, Peter Braun and Martin Rappl: A fundamental critique of the UML for the specification of embedded systems.
10. Pio Torre Flores ++: Integration of a Structuring Concept for Vehicle Control Systems into the

Software Development Process using UML Modeling Methods, SAE Technical Paper Series 2001-01-0066, Detroit, 2001.
11. Maximilian Fuchs and Dieter Nazareth: Ein BMW-Experiment zur Verbesserung des Steuergeräte Entwurfprozesses mit ASCET-SD, In Proceedings Industrielle Software Produktion,13/14 November 1997, Messe Stuttgart International, Stuttgart,1997.
12. Bernd Gebhard and Martin Rappl: Requirements Management for Automotive Systems Development, SAE Technical Paper Series 2000-01-0716, Detroit, 2000.
13. C. Hansen, O. Bringmann and W. Rosenstiel: A VHDL Component Model for Mixed Abstraction Level Simulation and Behavioral Synthesis, Proceedings of Forum on Design Languages (FDL'99), Lyon, Frankreich, September 1999.
14. Derek Hatley and Imtiaz Pirbhai: Strategies for real time system specification, Dorset House Publishers, New York, 1988.
15. Object Management Group: OMG Unified Modeling Language Specification (draft), Version 1.3 alpha R5, March 1999.
16. OMG: XML Metadata Interchange (XMI), Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF), OMG Dokument, Oct 20th, 1998.
17. Bernhard Rumpe and Andy Schürr: UML + ROOM as a Standard ADL?, Engineering of Complex Computer Systems, ICECCS'99 Proceedings, IEEE Computer Society, 2000.
18. Christian Schröder and Ulf Pansa: UML@Automotive - Ein durchgängiges und adaptives Vorgehensmodell für den Software-entwicklungsprozess in der Automobilindustrie, Praxis Profiline, IN-CAR COMPUTING, 1. Auflage 2000, Vogel Verlag, ISBN 3-8259-1909-9, 2000.
19. Bran Selic, Garth Gullekson and Paul T. Ward: Real-Time Object Oriented Modeling, John Wiley, 1994.
20. Desmond F. D'Souza and Alan C. Wills: Objects, Components and Frameworks with UML – the CATALYSIS approach, Addison-Wesley, 1998.
21. Oscar Slotosch ++: Tool supported Specification and Simulation of Distributed Systems, Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems, pp. 155-164, IEEE Computer Society, Los Alamitos, California, 1998.
22. Richard Stevens ++: systems engineering – coping with complexity, Prentice Hall Europe, Hertfordshire, 1998.
23. Andrew S. Tanenbaum: Computer-Netzwerke, Wolfram's Verlag, 2. Auflage, 1992.
24. U2 Partner Group (http://www.u2-partners.org): Unified Modeling Language 2.0 Proposal, Initial Submission to OMG RFP ad/00-92-02, 2001.

## URLS

25. Homepage Automotive (FORSOFT): http://www.forsoft.de/automotive/
26. Homepage BMW AG: http://www.bmw.de/
27. Homepage ETAS GmbH: http://www.etas.de/
28. Homepage Telelogic AB: http://www.telelogic.de/

## CONTACT

The authors' addresses:

Dipl. Inf. Dr. Michael von der Beeck
BMW Group – Function Development Process
Knorrstr. 147
D-80788 Munich
michael.beeck@bmw.de

Dipl. Inf. Peter Braun
Munich University of Technology
Department of Computer Science
Chair of Software und Systems Engineering
Arcisstr. 21
D-80333 Munich
braunpe@in.tum.de

Dipl. Inf. Martin Rappl
Munich University of Technology
Department of Computer Science
Chair of Software und Systems Engineering
Arcisstr. 21
D-80333 Munich
rappl@in.tum.de

Dipl. Phys. Dr. Christian Schröder
Telelogic AB - Germany
Gadderbaumerstr. 19
D-33602 Bielefeld
christian.schroeder@telelogic.de

## KEYWORDS

Abstraction Level, Automotive, Electronic Control Unit (ECU), Metamodel, Requirements Engineering, Systems Engineering, System Model, Unified Modeling Language (UML)