2006-01-1222

An Architecture-Centric Approach towards the Construction of Dependable Automotive Software

D. Wild, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann
Technische Universität München – Fakultät für Informatik

ABSTRACT

In this paper a *model-based design approach* currently developed is introduced to optimize the development process of automotive software. The approach plays special emphasis on a quality-oriented construction of embedded software to shorten the development life cycle and the development costs at the same time. "Quality-oriented" in this context means, that design and implementation decisions may be better traced back to the actual user requirements which are essential for the validation of the system. In contrast to low-level modeling approaches (such as Matlab/Simulink [1] or ASCET-SD [2], which mainly focus on technical aspects of the system), high-level modeling concepts are introduced to represent HW-/SW-architectures within a set of consecutive abstraction levels.

A newly reworked system of *automotive-specific abstraction levels* is presented, where architectures are specified introducing more detail on each level. The system of abstraction levels supports the inheritance of model information from abstract levels down to concrete levels and the refinement of this information at each level. Thus the gap between (informal) requirements and the implementation is reduced. Since the higher levels abstract from technical details, reuse of models will be possible in a very easy way. The abstraction levels will form the basis for the strongly formal definition of an *automotive specific architecture description language* which we call "CAR-DL" (Combined Architecture Description Language).

The presented approach is currently developed within the project "mobilSoft"[1].

1. INTRODUCTION

Though innovative functions are the key potentials to competitive advantage in the automotive domain, their merit will be limited, if their integration into an existing network of control functions, deployed on a highly distributed network of control units, cannot be handled in a managed and predictable way. Architecture Description Languages (ADLs) [3, 4, 5, 6] provide promising concepts to represent central aspects of the SW-architecture to support the designer during the design of functions and during the process of integration. However, in the automotive domain an isolated view on the SW-architecture is not satisfying; rather an appropriate order of abstractions is necessary to interconnect the SW-architecture and the HW-architecture. Each level should support the designer in answering questions with respect to compatibility management, diagnosis, verification and test.

This paper contains an introduction to the automotive-specific modeling language CAR-DL, which incorporates a newly reworked order of abstraction levels. The CAR-DL builds upon prior developments of automotive relevant modeling languages such as AML [7, 8, 9], EAST-ADL [10, 11, 12], and SAE-AADL [13]. It supports the representation of SW-/HW-architectures on subsequent abstraction levels. But in contrast to the preceding modeling languages, the CAR-DL relies on a clearly simplified order of abstractions, it incorporates less but more intuitive modeling concepts, and it supports a seamless top-down information flow.

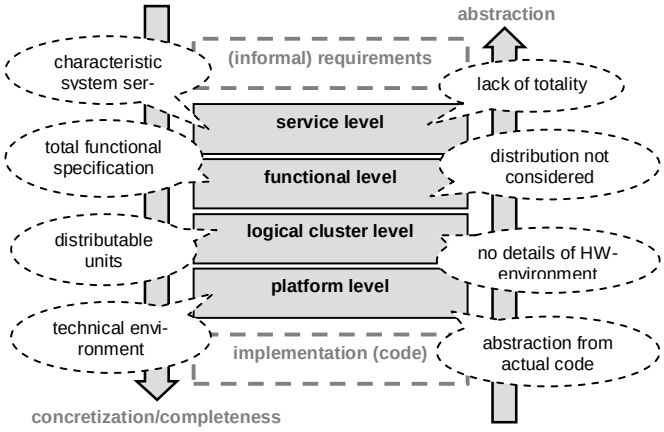


Figure 1: Abstraction and Concretization of Models

[1] The mobilSoft project is funded by the Bavarian Government under grant number IuK 188/001.

The system under consideration is described by a set of domain specific abstraction levels (in the following also called modeling levels) which are built upon each other. Hereby, the modeling levels (resp.) provide self-contained concepts for the representation of the information, which are specific for each level.

The modeling levels are ordered hierarchically starting with (very abstract) high levels and leading to (very concrete) low levels (s. figure 1). During the transition from a higher level to a more concrete level, the model information is enriched; i.e. the completeness of the models - with regard to the implementation of the system - is increased top down. Thereby, the transition has to be correct: in spite of the additional information the specification of higher levels must be obeyed and completely realized. It must be pointed out that no primary decomposition of the system into subsystems or components is made during the transition from one level to another. Instead, the system is enriched with specific classes of information on each level, respectively.

Due to the fact, that each level defines its own aspects which have to be taken into consideration, this system of modeling levels is a basis for a systematic design process of software for embedded automotive systems.

The paper contains a survey of the CAR-DL. In section 2 an automotive specific order of modeling levels is introduced. At each level corresponding modeling constructs of the CAR-DL and their semantics are described in an own section. At the end of the paper a conclusion and an outlook on future work is given.

2. CAR-DL MODELING LEVELS

For the structuring of the development process, four levels were identified, namely:

1. service level
2. functional level
3. logical cluster level
4. platform level

Each level abstracts from specific aspects of the system under consideration. Hence the lowest level – the *platform level* – is the most concrete one; it mainly abstracts just from the actual program code by describing the single *tasks* of the systems. *Clusters* in the next higher level – the *logical cluster level* – do not consider the technical runtime environment. Furthermore *functions* on the *functional level* abstract additionally from questions concerning distribution. Finally, *services* on the *service* level do not require – in general – completeness conditions with respect to all possible inputs and outputs; they only focus on its characteristic input and outputs (s. Figure 1).

Figure 2 illustrates the most important modeling elements which correlate with the order of abstractions depicted in Figure 1. Beyond, essential interrelationships between the modeling elements are shown.

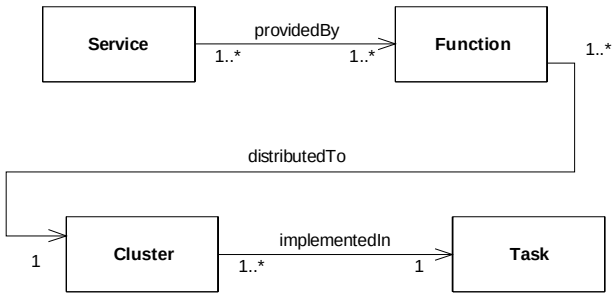


Figure 2: **Interrelationships between Modeling Elements on different Levels**

In the following four sections, the individual modeling levels are described in more detail.

3. THE SERVICE LEVEL

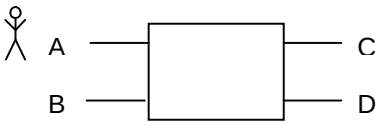
Starting point of the software development process typically is a (more or less) complete collection of requirements being documented in an arbitrary way (e.g. textually or by means of use cases). The requirements are not or only loosely related to each other, i.e. it is difficult to identify unwanted interactions or contradictory requirements. On the service level the majority of the requirements are now formalized consistently and related to each other in order to detect these contradictions and interactions.

The aim of the service level is the consolidation and the precise specification of requirements that describe the characteristic system behavior. Here, characteristic system behavior depicts the behavior that is visible for the user (which does not only refer to a human user but also to another system). Among all requirements that have been acquired so far, those ones describing the characteristic system behavior are chosen, formalized, and related with each other. The formalization of the system behavior by means of services and relationships between services allows for the detection of contradictions and behavioral conflicts.

The design decisions on this level consist of both, the specification of the system boundaries and the specification of the inputs and outputs (= actions and reactions) of the system under consideration. The services can be constructed step by step on basis of composition operators on this level. Please note, that the behavior is specified by a pure black box view. Inputs and outputs are described in a quite abstract way – that is to say: as they are interpreted by the users. However, the representation of data (how it is processed by the realization of the system) is not considered so far. The complete definition

of the overall system behavior is not aim of this level. The system behavior is only partially defined, i.e. the behavior is not defined for each possible input sequence. The totalization is done on the subsequent functional level.

On the service level, it is out of interest which subsystem provides which services; also a decomposition of the system is not done up to now. Furthermore, it is not important which user generates which input and which user consumes which output. Figure 3 represents the system on the service level graphically.



Actions: A,B,C,D
Services: A AND B => C; C AND B => D

Figure 3: Black-Box-View of the System on the Service Level

Information Flow within Modeling-Levels

- Functional Level :
  The services specified on the first modeling level are realized by the interplay of the functions on the functional level. Between functions and services there is a n:m-relationship, i.e. one or more services can be provided by one or more functions (in latter case, the interplay of several functions provides the service).

Modeling Elements

Figure 4 depicts a metamodel of the service level. The main important modeling elements are:

- Service:
  Services are finite, causal sequences of *Actions*. They own *QualityAttributes* which for example can describe timing properties or reliability requirements. Services are hierarchical constructs, i.e. they can be further decomposed into services again.

- Action:
  An *Action* can be either an *Event* (with no time specification) or an *Activity*. An *Activity* has a well-defined *Duration*. Actions are elementary and expose visible system interfaces to the environment of the system.

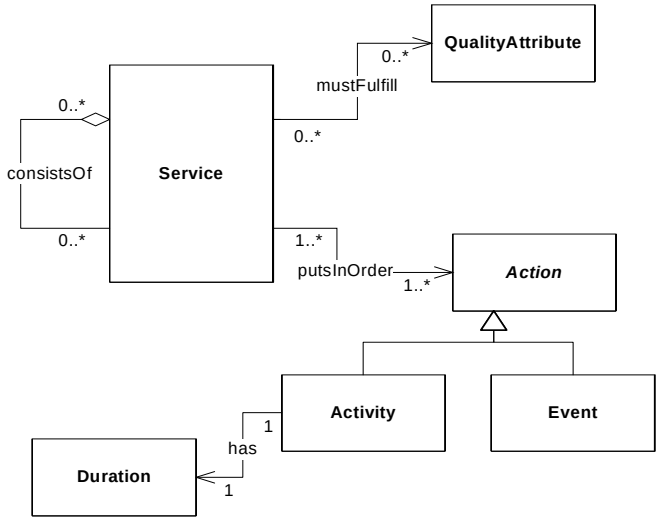


Figure 4: Metamodel of the Service Concept

Benefits of Service Level

- The detection of contradicting requirements is possible.
- Due to the formalization, the characteristic system behavior can be simulated.
- Due to the formalization of the requirements (and their simulation) the accordance with the stakeholders' opinion may be easily validated.
- The reuse of services – i.e. of characteristic system properties – is supported.

4. THE FUNCTIONAL LEVEL

So far, the focus lied on a consistent, but not necessarily complete, description of the characteristic behavior. On the functional level, the behavior will now be totalized and defined completely and deterministically (i.e. the system provides a well-defined predictable output for each possible input sequence). Additionally, a decomposition into smaller pieces – again: functions – is done on the functional level (= hierarchical decomposition).

The design decisions on this level are limited mainly to the identification of functions, the determination of their granularity and the mapping of actions to abstract data types. The functions have to be specified in a way, that their interplay realizes all services which have been defined on the service level. The system is now presented as a network of (hierarchical) functions.

On this level the assumption holds that the complete system does not have to be distributed to different runtime environments. It is ideally assumed that there exists a monolithic runtime model, i.e. one single synchronous and undistributed runtime model. The execution is controlled by global clock and the communication is assumed to consume no time. Aspects regarding the distribution of functions are not considered up to the logical cluster level. Figure 5 shows a schematic diagram of the functional decomposition of a system.

Information Flow within Modeling-Levels

- Service level:
  One or several services on the service level are related to one or more functions on the functional level, i.e. that a service can be established by several functions, or a function can provide several services. The actions of the service level are related to abstract data types of the functional level.

- Logical cluster level:
  On the logical cluster level, several functions of the functional level are grouped to clusters.

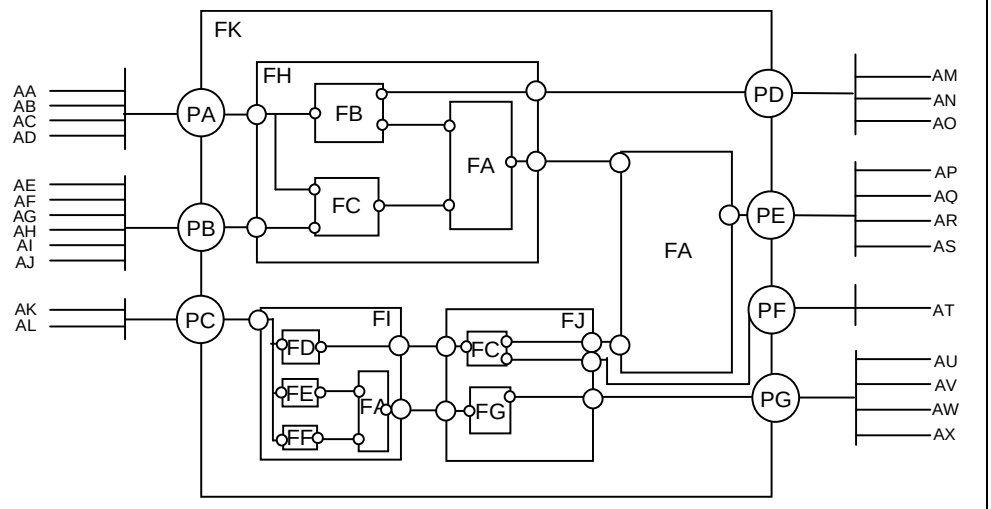
Modeling Elements

The main elements on this level – as shown in the metamodel of Figure 6 – are:

- Function:
  The *Function* is the central concept of this level. A *Function* can again consist of *Functions* (= *hierarchical* function). A *Function* which is not decomposed any further is called an *elementary* function.

- Variable:
  Variables are related to *Functions*. On the one hand, they can describe the interface (*FunctionPort*), on the other hand they can represent the internal state of the function (*State*) due to their assignment.

- Type:
  Each *Variable* has a specific *Type*.



Px and ○ : Ports

Fx: functions

(Ax: actions)

Figure 5: Functional Decomposition of a System

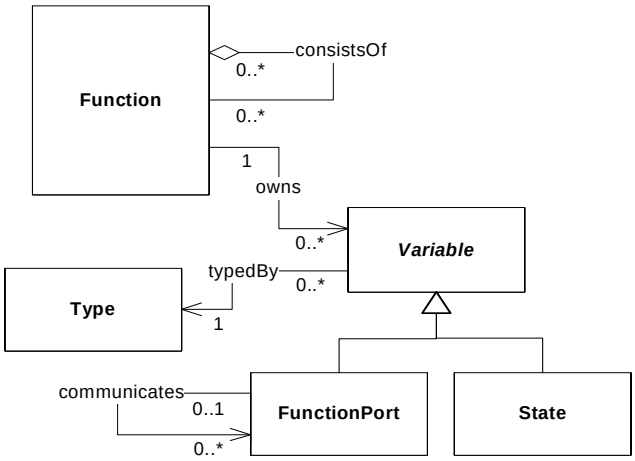


Figure 6: Metamodel of the Functional Level

Benefits of the Functional Level

- The behavioral specification is totalized.
- The system can be simulated.
- The reuse of functions, in particular the integration of legacy, is possible.

5. THE LOGICAL CLUSTER LEVEL

Software that typically is deployed in embedded automotive systems runs in asynchronous tasks which are administrated by an operating system. Additionally, the software is typically distributed over different runtime platforms (CPU including the operating system). However, up to the functional level (included), the assumption holds that the system runs on a synchronous runtime platform as a whole, i.e. there exists a global system clock which allows a totally deterministic, parallel – or alternatively sequential – execution.

This assumption is now abandoned on the logical cluster level. Here, we switch to an asynchronous runtime model. Thereby more flexibility for the implementation of the software is yielded; the asynchronous runtime platform corresponds to a communication structure as it is typical for embedded automotive systems. Due to the fact that the synchronous runtime platform is a special case of the asynchronous one, the model of the cluster level can be implemented on synchronous communication structures, too. Similarly, realistic conditions under which the software has to operate are taken into consideration; i.e. the fact, that resources such as computing time and memory are not available in an unlimited quantity is made allowance.

The aim of the logical cluster platform is to partition the system into distributable units (= clusters), i.e. to structure the functions specified at the functional level, and to combine them to clusters. This has to be done in an adequate manner so that these can be a usable basis for the physical distribution which takes part on the subsequent level (namely the platform level).

Each cluster is a self-contained unit with well-defined communication interfaces. The functionality within these clusters is not of major interest as this has been done on the functional level. Instead, more attention is drawn to interface specifications to obtain distributable units. These units stem from a combination of functions by considering the question when these units have to be executed. Furthermore, the required resources (i.e. available machine time and memory consumption) have to be specified in a way that can easily be transformed into the concrete tasks of the respective runtime platform (on the logical cluster).

Major design decisions made on this level are on the one hand, at which points the catenation between functions has to be disconnected. And on the other hand, the determination of the timings defining the scheduling of units is a further design decision. Alternatively, it has to be stated when the inputs have to be available from other units. The formation of clusters can be influenced by the following criteria, for example:

- Existent hardware: hardware architecture, connected sensors, actuators, bus systems (bus topology)
- Performance / CPU-load / bus load
- Safety and reliability
- Reuse for other systems (→ product lines)
- Legacy which has to be integrated
- Maintainability
- Placing of orders to suppliers
- Scalability

Depending on which criteria have the highest priority, the clusters turn out differently concerning both granularity and the kind of formation.

On the logical cluster level, the system is represented as a network of distributable units which we call *clusters*. This view on the overall system can also be seen as "logical deployment".

Figure 7 describes schematically the transition from the functional level to the logical cluster level.

Information Flow within Modeling-Levels

- Functional level:
  The functional network that was determined on the functional level is clustered in a new manner by considering additional criteria. Functions are grouped to clusters; function ports are related to cluster ports.

- Platform level:
  The clusters are related to individual tasks. Several clusters can be related to one task.

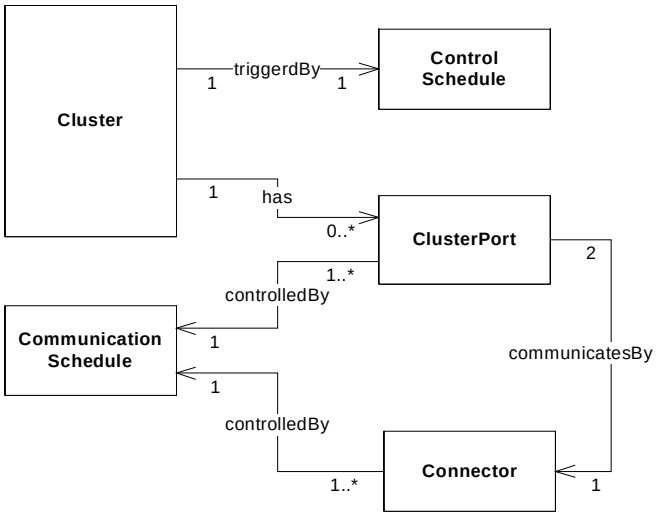


Figure 8: Metamodel of the Cluster-Level

Modeling Elements

Figure 8 shows the metamodel, which defines all modeling elements on the logical cluster level. These are:

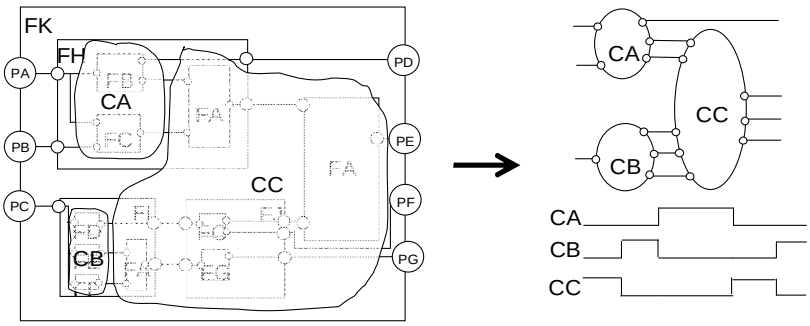
- Cluster:
  Clusters are elementary units on this level. They do not have a hierarchical structure. In a *Cluster*, grouped functions are executed sequentially. A cluster has a *ControlSchedule* which determines which cluster has to be called after an occurring event, or alternatively, after a detected timeout.

- ClusterPort:
  Each cluster can have an arbitrary number of *ClusterPorts*.

- Connector:
  Connectors connect two *ClusterPorts*, respectively, and describe – together with the *ClusterPorts* – the *CommunicationSchedule* that takes place between these *ClusterPorts*.

Benefits of the Logical Cluster Level

- It provides the basis for accurate calculations concerning time and memory consumption at known runtime platforms.
- Easy reuse in other systems is possible because of the platform independence.
- Optimal partitioning/structuring of the functions with regard to the distribution.



○ : Ports

Cx : Cluster

Figure 7: Clustering of System Functions

6. THE PLATFORM LEVEL

On the previous levels, the system was developed widely independently from the hardware. The hardware was only taken into consideration if it influenced the behavior crucially. This was motivated by reuse issues. On the platform level the hardware independence is abandoned and the system is adapted to the real hardware. The units detected on the logical cluster level are embedded in the hardware environment which is abstracted by software (= drivers).

Design decisions that are typically made on this level comprise the mapping of clusters to controllers or tasks, and their scheduling and the realization of the communication between the clusters. This realization must be done according to the required resources, stated on the logical cluster level on the one hand, and the availability of resources on the platform level on the other hand. The concrete technical data types, the signals and bus messages are defined and the interfaces between clusters and hardware are adapted accordingly. The system now consists of communicating hardware components. The behavior of each component is determined by clusters and drivers.

In Figure 9 a view of a system on the platform level is shown.

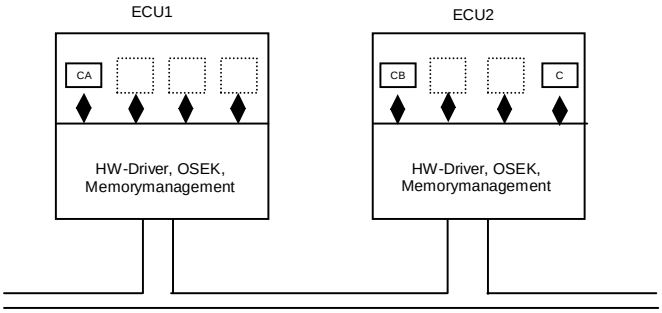


Figure 9: Platform View of a System

Information Flow within Modeling-Levels

- Logical cluster level:
  The logical clusters on the logical cluster level are related to the tasks on the platform level. Here, the *ControlSchedules* of the logical cluster level have to be observed strictly.

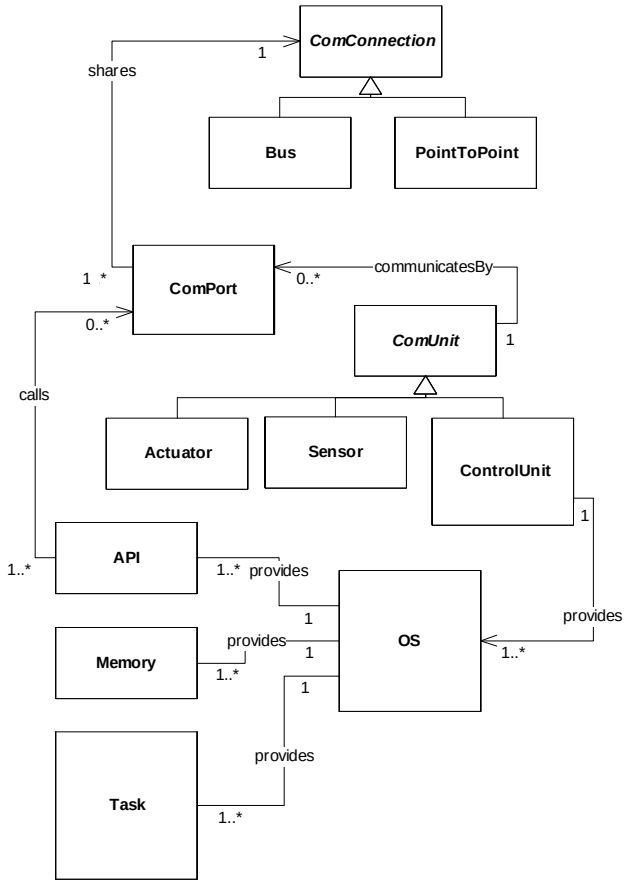


Figure 10: Metamodel of the Platform Level

Modeling Elements

Modeling elements at the platform level are shown in Figure 10 and described as follows:

- ComUnit:
  Communication units (*ComUnits*) represent the most important hardware parts of an embedded system. These can be *Actuators*, *Sensors*, and *ControlUnits*.
- ComConnection:
  The communication connections (*ComConnection*) connect the communication units with each other. Hereby, a distinction between buses and point-to-point connections is made.
- OS:
  OS represents an operating system that can run on one controller. The operating system takes care of the memory management, executes the tasks, and poses an interface to the communication ports by means of an API.
- ComPort:
  A *ComUnit* can communicate with another *ComUnit* through a *ComConnection* by means of a communication port (*ComPort*).

Benefits of the Platform Level

- The systematic consolidation of independently developed software and hardware can be done mainly automatically.
- Exact guidelines for the technical implementation of software functions into program code are given up to now. The specification is detailed enough for automatic code generation.

7. CONCLUSION

The design approach introduced in this paper aims at a systematic structuring of different design decisions. Each of the four modeling levels – *service*, *functional*, *logical cluster* and *platform level* – facilitates methodically specific design decisions which have to be concerned in the respective level. The designer's main advantage using such an approach of design methodology lays in the well coordinated stepwise refinement of the system construction.

The defined abstraction levels build the basis for the definition of our architecture description language called "CAR-DL". The formal definition of this design language is the main issue in our ongoing research within the project "mobilSoft". Special emphasis lays on a formal description of the transition from one level to the subsequent one. Thus the design language itself contains and defines constructs and dependencies for an integrated design methodology.

With today's development processes we often face a gap between the (usually) textually performed requirements engineering and the model-based design. Within the "mobilSoft" project we are aiming at a seamless transition between these two development tasks. The services on the service level seem to be a promising approach to tackle this problem. Together with the requirements engineering competence center of our chair, we are working on a continuous (model-based) development process.

Moreover, safety and reliability are of highest priority for automotive software systems. Validation and verification activities of these systems are therefore a major task during the development process. However, the validation and the verification of an implementation must be traced back to the actual (user) requirements to really confirm the accurate behavior of the system under consideration. As shown above, the introduced modeling levels comprise a stepwise refinement of requirements, starting with the definition of services. Furthermore, each level specifies requirements which must be fulfilled on the following level(s). The integration of techniques for test and verification in the introduced design approach is therefore another major consideration for our future work.

We believe, that a well structured and incremental design approach as introduced above will lead to more dependable systems and also reduce over-all life-cycle costs of embedded automotive systems. The well defined design steps will increase quality since smaller design steps may be less error-prone. An integration of validation techniques in the early phases of development – e.g. on service and functional level – will lead to an early detection of errors. Finally the separation of concerns in different abstraction levels simplifies the reuse of design models, for example if in a new line of production the hardware environment of the system has changed.

ACKNOWLEDGMENTS

We are much obliged to our colleagues of the "mobilSoft" project for many fruitful discussions; in particular Stefan Kuntz, Markus Bechter, Benno Stützel, Michael Blum, and Hendrik Dettmering. We thank Manfred Broy for directing this research. This work has been partially funded by the Bavarian Government within the "High-Tech-Offensive Zukunft Bayern" [14].

REFERENCES

1. MathWorks: Matlab/Simulink, online informations http://www.mathworks.com/, 2005.
2. ETAS: ASCET-SD, online informations, http://de.etasgroup.com/, 2005.
3. David Garlan: Software Architecture; in Wiley Encyclopedia of Software Engineering; J. Marciniak (Ed.); John Wiley & Sons, 2001.
4. David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek: Reconciling the needs of architectural description with object-modeling notations. In Science of Computer Programming 44, P. 23-49, Elsevier, 2002.
5. Nenad Medvidovic, David Rosenblum, David Redmiles, and Jason Robbins: Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, Vol. 11, No .1, P. 2-57, 2002.
6. Nenad Medvidovic and Richard N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transactions on Software Engineering, vol. 26, no. 1, January 2000.
7. Peter Braun, Michael von der Beeck, Ulrich Freund, and Martin Rappl, "Architecture Centric Modeling of Automotive Control Software", SAE Transactions Paper, Number 2003-01-0856, World Congress of Automotive Engineers, 2003.
8. Peter Braun, Michael von der Beeck, Martin Rappl, and Christian Schröder, "Automotive UML", UML for Real, Luciano Lavagno, Grant Martin, and Bran

Selic (eds.), Kluwer Academic Publisher, ISBN-1402075014, 2003.
9. Manfred Broy ++: The Design of distributed Systems, An introduction to FOCUS – Revised Version, Technical Report, TUM-I9202, Technische Universität München, 1993.
10. S. Boutin: Architecture Implementation Language (AIL); 1er Forum AEE; Guyancourt; March 2000; http://aee.inria.fr/forum/14032000/SB_Renault.pdf.
11. J. Eisenmann et al.: Entwurf und Implementierung von Fahrzeugsteuerungsfunktionen auf Basis der TITUS Client Server Architektur; VDI Berichte (1374); pp. 309 – 425; 1997; (in German).
12. U. Freund et. al.: Interface Based Design of Distributed Embedded Automotive Software - The TITUS Approach. VDI-Berichte (1547); pp. 105 – 123; 2000.
13. SAE: SAE-AADL, online informations, http://www.aadl.info/, 2005.
14. High–Tech-Offensive Zukunft Bayern, online informations, http://www.bayern.de/Wirtschaftsstandort/IuK/High-Tech-Offensive/hto.html, 2005
15. Manfred Broy, Michael von der Beeck, Peter Braun, and Martin Rappl: A fundamental critique of the UML for the specification of embedded systems.
16. Bernd Gebhard and Martin Rappl: Requirements Management for Automotive Systems Development; SAE 2000-01-0716; Detroit; 2000.
17. David Harel: StateCharts: A Visual Formalism for Complex Systems; Science of Computer Programming 8(3); pp. 231- 247; 1987.
18. Derek Hatley and Imtiaz Pirbhai: Strategies for real time system specification, Dorset House Publishers, New York, 1988.
19. Object Management Group: OMG Unified Modeling Language Specification, Version 1.4, September 2001.
20. Object Management Group: OMG XML Metadata Interchange (XMI), Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF), OMG Dokument, Oct 20th, 1998.
21. Bran Selic, Garth Gullekson, and Paul T. Ward: Real-Time Object Oriented Modeling, John Wiley, 1994.
22. Desmond F. D'Souza and Alan C. Wills: Objects, Components and Frameworks with UML – the CATALYSIS approach, Addison-Wesley, 1998.

CONTACT

Dr. rer. nat. Martin Rappl
Technische Universität München
Fakultät für Informatik
Lehrstuhl IV: Software & Systems Engineering
Boltzmannstraße 3
D-85748 Garching bei München
rappl@in.tum.de

Dipl. Inform. Doris Wild
Technische Universität München
Fakultät für Informatik
Lehrstuhl IV: Software & Systems Engineering
Boltzmannstraße 3
D-85748 Garching bei München
wildd@in.tum.de