Reuse of Software in Distributed Embedded Automotive Systems

Bernd Hardung
AUDI AG
I/EE-93
Ingolstadt, Germany
bernd.hardung@audi.de

Thorsten Kölzow
Audi Electr. Venture GmbH
I/AEV-22
Gaimersheim, Germany
thorsten.koelzow@audi.de

Andreas Krüger
AUDI AG
I/EE-93
Ingolstadt, Germany
andreas.krueger@audi.de

ABSTRACT

Until recently, in the automotive industry, reuse of software has entirely been a typical activity of suppliers. They try to reduce the increasing software development costs that stem from rising complexity and size of software in the modern automobile. Lately, also the automotive manufacturers began to develop specific software with competitive relevance. Now they have to deal with the problem of reuse, too. Nevertheless, there is a difference between the manufacturers' and the suppliers' point of view because the manufacturers have to integrate the networked hardware components to one automotive system. Therefore, the manufacturers have to deal with additional problems compared to the supplier. At the beginning of this paper, the specific problems of reuse of software in the automotive domain are shown from the perspective of automotive manufacturers. After that, a framework is proposed to deal with these problems. Moreover, the application of this framework is shown in a realistic application example.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.2.13 [**Software Engineering**]: Software Architecture—*Domain-specific architectures*

General Terms

Design, Standardization

Keywords

Reuse, Automotive, Software, Architecture, Product Line

1. INTRODUCTION

At the beginning of the third millennium, the automotive industry is facing a new challenge. Electronics make 90 % of the innovations, 80 % out of that in the area of software. This fact means a big change for the development of electronics. More and more highly connected functionality has to be brought into series production while development time is getting shorter and shorter. The importance of software in the automotive industry is shown very impressively in a study of Mercer Management Consulting and Hypovereinsbank [16]. According to this study in the year 2010, 13 % of the production costs of a vehicle will be software (Fig. 1).

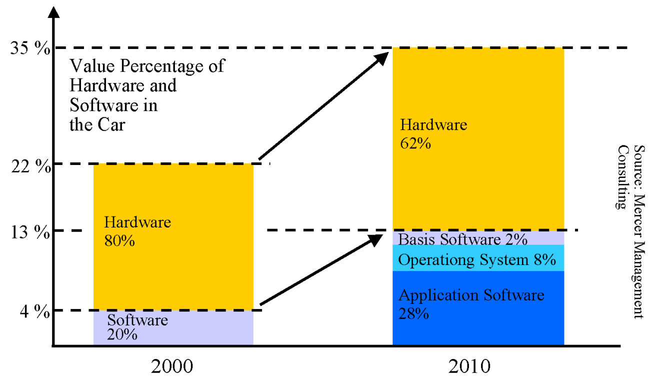


Figure 1: Rise of Importance of Software in the Car

To consider this, a changed development process has to be established and also the methods of developing software for the automotive domain have to be changed. In parts of this field intensive work has been done covering for example requirements engineering, quality of software or model based software development, to name only a few. The goals are to shorten the software development time and to increase the quality of software. Another challenge to reach these goals is the reuse of software. The main requirement for reusing software in the automotive domain is to separate the hardware of an *Electronic Control Unit* (ECU) from the embedded software on it.

Until a few years ago, automotive manufacturers saw the ECU's of a car as single units. They specified and ordered them as black boxes from the supplier. After the delivery of samples, they tested them as black boxes. For the automotive manufacturer, this procedure has the disadvantage that the software has to be newly developed for each new project, if the supplier is changed. This not only causes expenses, but also an increase of development time.

A further important point for the automotive manufacturers is the responsibility for the whole electronic system. This is different to the view of the suppliers who see only their part of the system. The fact of networked units makes it necessary for the automotive manufacturers to have development processes and methods, which allow them to reuse software on the system level. In addition methods for the reuse of software enable the manufacturers to develop competition relevant software on their own in the future.

After this introduction, the recent work in this field is described introducing the electronic systems in a modern vehicle (Sect. 2). Then a framework is presented, which supports automotive manufacturers to reuse software (Sect. 3). In Sect. 4 an example is given in which the framework is used on a realistic application of an automotive electronic system followed by the conclusion (Sect. 5).

2. REUSE OF SOFTWARE IN THE AUTOMOTIVE DOMAIN

To get an idea of the software reuse issue from the view of an automotive manufacturer, an introduction to the electronic systems in a modern car is given here. The innovation in the automotive domain is mainly influenced by electronics. Improvements in economy and power of engines in the last years as well as the existence of driver assistance systems like *Blind Spot Warning System* and *Lane Departure Warning* are not imaginable without electronics.

In order to fulfill the increased communication needs of these electronic systems, the ECU's communicate via different bus systems. Common automotive bus systems are for example *Controller Area Network* (CAN) [3], *Local Interconnect Network* (LIN) [14], *Media Oriented System Transport* (MOST) [17] and *Flexray* [10]. An example for the complexity of such a system is the network topology of the Audi A8, shown in Fig. 2.

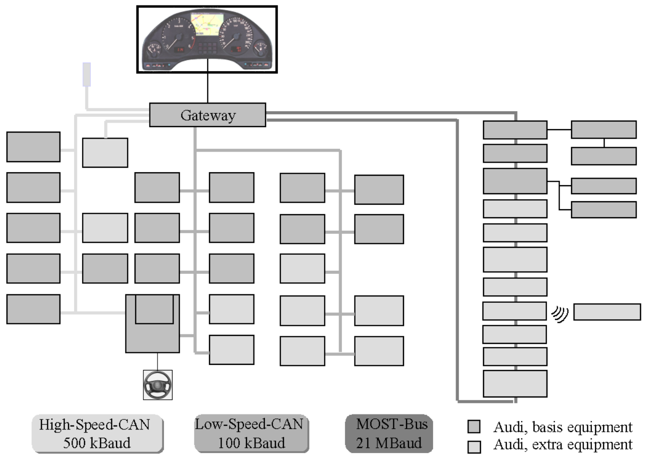


Figure 2: Electronic System of the AUDI A8

It was already pointed out in the introduction that the car manufacturers see the software increasingly independent from the hardware. This view is necessary to enable the car manufacturers to reuse software. The reuse of software thereby has a variety of different aspects and unresolved problems. In [8] requirements for the reuse of software within the automotive range are presented as follows:

- Reusable application software components must be *hardware independent*.
- Interfaces of the software components must be able to exchange data both locally on an ECU and/or via a data bus.
- Developing reusable software, future requirements to the function have to be considered.
- *Code size* and *execution time* of the software components must be minimized during the development of reusable software. Both resources are expensive due to mass production of parts in the automotive industry.

Additionally, further aspects have to be considered. The individual software modules must have an optimal *modularity*. That means that a functionality (for example central door looking system or exterior light) might consist of different individual sub components. Building sub components improves the reusability, since a functional change can imply only changing a single sub component.

The partitioning of functionality into sub components however can cause repetitions of code. For example, multiple variable declarations lead to a higher memory consumption of all sub modules together in comparison to a module developed as single unit. In addition, the execution time might be worse.

The interface definition of the software modules must be specified once, that means statically. It should not change during the reuse in a new car model or on a new micro controller[1]. The interfaces must be maintained in a database over all type series.

A further requirement is the existence of a database in which the individual reusable software components and/or sub-components are stored. A *uniform data format* is thereby the requirement for the data exchange of the different software development departments of a manufacturer and for the data exchange with the supplier. In order to fill and use the information and the reusable software components from the database, *processes* must be defined, which enable a standard development process. In particular this processes must describe the integration process of software of different sources and the role allocation between manufacturer and suppliers.

For the support of the processes a *tool chain* as seamless as possible is necessary. An important aspect thereby is the use of *uniform modeling guidelines*. Likewise, standards should be defined between suppliers and the manufacturer to facilitate the exchangeability of software modules effectively. From the view of a manufacturer, there exist *different kinds of reuse*, depending on the usage in an electronic system. According to the kind of reuse of a software component – on the same type series or over different type series – different aspects must be considered.

Of course, the safety aspect must be taken into consideration. In [23] requirements for the development of safety critical functions are stated. Simonot-Lion gives an overview on several aspects, for example verification process, time triggered architectures, and software architecture models.

3. FRAMEWORK FOR THE REUSE OF APPLICATION SOFTWARE COMPONENTS FOR AUTOMOTIVE MANUFACTURERS

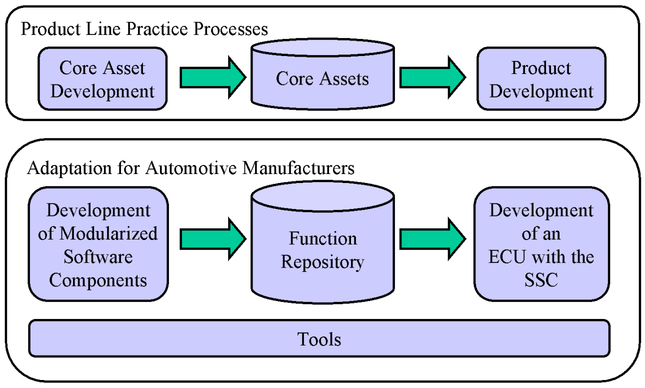
In this section, a framework for the reuse of application software in the automotive domain is introduced. It is based on the process model named *Product Line Practice* (PLP) [4] developed by the Software Engineering Institute of the Carnegie Mellon University in Pittsburgh.

The term *product line* is thereby defined as follows [5]:

"A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of *core assets* in a prescribed way."

[1]This applies only if the software component is not hardware independent.

The terms and the processing model of the PLP are applied on reuse of software by an automotive manufacturer. Figure 3 shows the components of the presented framework. As it can be seen, the framework is divided into five parts. In the process part (Sect. 3.1), the general processes of the PLP are explained. This is followed by an explanation how to perform the modularization of the core assets (Sect. 3.2). Thereby the core assets are stored in a database called *Function Repository* (Sect. 3.3). The last part of the process is to develop products out of the Function Repository by using a standard software core (Sect. 3.4). The tools are necessary to support the process (Sect. 3.5).



Product Line Practice Processes
Core Asset Development
Core Assets
Product Development
Adaptation for Automotive Manufacturers
Development of Modularized Software Components
Function Repository
Development of an ECU with the SSC
Tools

Figure 3: Modules of the Framework for the Reuse of Software Components for Automotive Manufacturers

3.1 Processes

According to [4] the process in a product line is divided into three different areas:

- *Core asset development*: Within the development of core assets, first a list of the products that are desirable from today's view is created. This list is defined as the *product scope*. Thus, it contains also products which may be realized in the future and which are not a goal of the current development. The list of products represents a boundary, which should be thought over very carefully: If the scope of production is spread out too much, many of the core assets can be used only once. This can be equated with the conventional development. If the product scope is chosen too small the future variety of product is limited.

- *Product development:* In addition to the *product scope* and the *core assets*, there are also product specific requirements. With the core assets, the development of a new product within the product scope is equal to combining some of the core assets. The description of the process of combination is called the *production plan*. The production plan is a general description and the product development should fulfill the product specific requirements. Depending on the accuracy of the production plan the product must be developed under consideration of *variation points*. In Fig. 4 [4] core assets are shown as rectangles and the correspondent processes as triangles. String together these processes result in the production plan, from which the product can be developed.

- *Management:* The *management* is divided into a technical and an organizational part. The organizational management must provide the right form of organization and the needed resources (this includes also the training of the employees). The technical management is responsible for the realization of the core asset development and the product development.

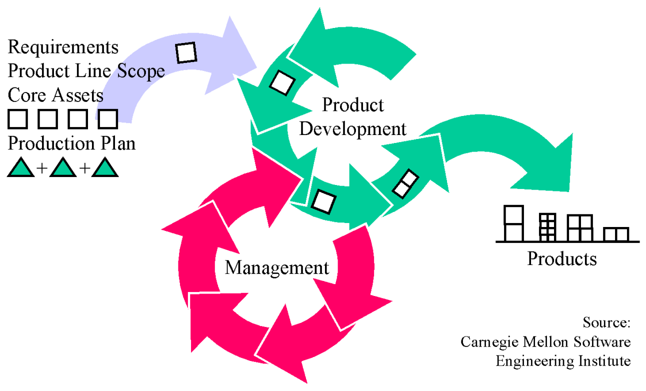


Figure 4: Product Development in the Product Line Practice

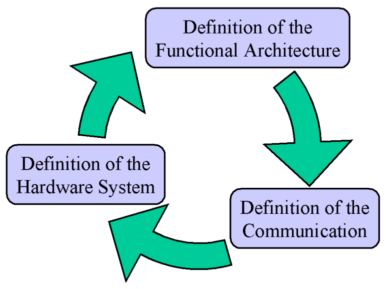
The engineering tasks of the processes within the presented framework contain on the one hand the production and archiving of reusable software components into a database. On the other hand it contains the use of this software components from the database to develop new products.

In [25] a practical application example of the automotive domain is discussed. In the article of Thiel and Hein a method is described, which allows controlling the variability of a product within the product development process.

The focus of the work described in this paper is put on distributed and networked applications, which is not covered yet. Furthermore here the focus is on the modularization of the functional software and the connection to the standard software core.

3.2 Development of Modularized Software Components

Figure 5 shows the process for developing the complete electronic system of a new type series[2].



Definition of the Functional Architecture
Definition of the Communication
Definition of the Hardware System

Figure 5: Product Development Process for a Electronic Automotive System

Thereby the three main elements must be gone through iteratively, since each element depends on the other one. Therefore, a distribution of the software modules must take place on the ECU's

[2]The term new type series here also covers a new model of the same type series.

for example after a definition of the hardware system. However, this depends on the memory and processor capacity of the ECU's exactly the same and also on the question whether the necessary signals between the individual software modules can be transferred across the bus system due to bandwidth limitations.

This process requires modularized software components to separate hardware dependent and independent parts. The modularization of software components according to definitions of movability and reusability also enables reuse in this process. To support the classification of software according to their movability and reusability, the following terms shall be introduced: *firmware, basic software*, *adaptation software* and *function software.*

- *Firmware* is the hardware dependent part of the ECU software. Examples are the communication drivers.
- *Basic software* is software that is independent from application and hardware. Examples are interaction layers which are not dependent from the hardware since they are using the *Application Programming Interface* (API) of the communication drivers.

The two groups basic software and firmware are reused already nowadays. They are called standard software although the partitioning of the components is not only made due to the reusability (Sect. 3.4). Another reason for decomposing the standard software is the possibility to fulfill the needs of different classes of ECU's by removing standard software components that are not needed.

Here the definition for the remaining two classes of software:

- *Adaptation software* is the application specific part of the software that adapts the function software to the type series and builds the connection from the function software to the firmware and the basic software. In general it is
- *Function software* is the function specific part of the software that is not dependent from the type series where it is used in.

With these new terms, the types of components can be separated according to the type of movability and reuse. Considering this abstract model three terms of reuse can be instituted.

- Reuse over different ECU's
- Reuse over different micro controller platforms
- Reuse over different type series

The classification of the types of software and the according types of reuse are shown in Tab. 1. This classification gives guidance how the modularization can be performed in order to get the maximum movability and reuse effect.

Table 1: Overview of the Classified Types of Software

Reuse over different	ECUs	μC-Platforms	Type Series
Firmware	+	-	+
Basic Software	+	+	+
Adaptation Software	-	+	-
Function Software	-	+	+

3.3 Function Repository

The *Function Repository* contains the *core assets* (Fig. 3). The question, what the core assets are, is essential for reuse of software. In [4] information is given which items in general form the set of core assets. In the following, the set of the core assets for a Function Repository of an automotive manufacturer is discussed considering the special requirements as described in Sect. 2. It has to contain more than only descriptions about software, but also hardware and bus system description.

- *Software components*: An important content of the Function Repository are the reusable *software components* themselves. It is necessary to differentiate the software according to the different kinds of software and the different kinds of languages they are implemented with.
  A meaningful distinction of the kinds of software can be made between firmware, basic software, adaptation software and function software as described in Sect. 3.2.
  In addition, the characteristics of the software components must be stored in the Function Repository. For example, the code size and the worst case execution time of the individual software components and subsystems are important information for a proper real time integration in an operation system. The worst case execution times could be determined for each supported hardware platform of the software component. This could be done for example with model based automatic approaches like in [12]. Another possibility is to get this value during the integration process. This procedure has the advantage that the environment can be taken in account[3]. This approach in general will derive worst case execution times which are tighter to the real execution time.
  In order to further optimize the software reuse specifications, test plans and test cases should be stored as well.
- *Interfaces*: The *interfaces* must be part of the Function Repository. In an automotive system, the communication of the software components between each other can take place by a data bus or internally on the ECU. The interfaces of the software components must be defined globally. Possibly, new software components must use the available interface definitions. If done so, a compatibility of the interfaces of different software components can be achieved.
  The number of needed interface is very huge in a modern vehicle. To be able to manage the complexity it is necessary to specify interface types. The interface instances can be derived from them.
  The description of interface types is thereby not only meant as storage of the bit size and the name. Also further descriptions have to be stored. One point is the supported communication types that allows the system designer to map the signals on communication busses correctly. Once the interface instances are mapped to software components, timing requirements have to be specified which also influence the decision if a certain communication bus system is suitable.
  The semantics of the interface types are an additional very important point. For example, once a user presses the remote key the signal of the type remote key is ON as long as the key is pressed but at least 50 ms. Further research should be performed on the description of this semantics in a formal way.

[3]For example in a certain type series some function will never be used and therefor the worst case execution time gets smaller.

- *Functional network*: The *functional network* combines the software components with each other. A support of hierarchical decomposition is recommended in order to be able to reduce the complexity.
  An add-on for the interface definition of software components is the *error matrix*. This error matrix can help with the trace of errors in the automotive system after production in the field. The error matrix contains information which output of a software component depends on which input. With that, it is easier for the service to find software errors. A more detailed description can be found in [22].

- *Hardware platform and bus system description*: The complete software can not been seen independently from the hardware although this is necessary for reuse. The execution of software depends on the type of processor, the amount of memory, the clock cycle or even the used compiler. Therefore, a *hardware platform description* must be stored in the Function Repository.
  As mentioned before, parts of the communication between the software components in functional networks use bus systems. These bus systems can be distinguished in different ways, for example speed of data transfer, used protocol, and so on. Thus for the development of a new type series it is necessary to store also the relevant data in the *bus system description* of a Function Repository.

- *Implementation*: For the implementation (or *behavior description*) of the software components, model based methods and tools can be used. From these models C code can be derived. This C code normally has performance disadvantages in comparison to hand-written C code. The advantage of the model based software components is the ability to more easily adapt to different hardware platforms. For this reason, it could be useful to be able to store both variants in the Function Repository depending on the kind of software.

3.3.1 Standards for the Storage of Data

All the mentioned data for the Function Repository has to be stored in a database. Therefore, a standardized data model has to be used.

Within the EAST-EEA project [7] an *Architecture Description Language* (ADL) [26] has been developed in order to be able to specify functions separately from the hardware. An ADL allows the description of the structure (and not necessarily the behavior) of software in a functional approach as described in this section.

A further example for such a data model is MSR MEDOC [18, 9]. It defines how the data has to be stored in the data model. The data model is specified in an *eXtentable Markup Language Data Type Definition* (XML DTD).

Another promising project is *Automotive Open System Architecture* (AUTOSAR) [1]. This project is currently defining a modular software architecture. Thereby the aspect of integration of software components from different software suppliers is considered as well.

3.4 Development of an ECU with the Standard Software Core

For the product development, based on the modularized software components and the other stored assets, a *Standard Software Core* (SSC) is used. Beside the application, the micro controller must fulfill further tasks. Controlling the hardware drivers, recognize and store errors and controlling the network connection can be named as examples. These functions are realized in separate, reusable modules and are called *standard software*. The sum of all standard software components represents the *standard software core*.

It supports not only a hardware independent interface, but also a complete infrastructure to the actual application at the micro controller. This makes it possible to develop the application independently from the used micro controller platform.

The interface between SSC and the function software is not unique. The link to the SSC modules is typically performed through *Application Programming Interfaces* (API). An example therefore is the OSEK COM [19] specification. The Function Repository in the described form contains all necessary information to configure the API's of standard software modules.

Thus, the combination of standard software modules represents the basis for an efficient reuse of the application as already explained in Sect. 3.2.

Figure 6 shows the structure of the standard software core of the VOLKSWAGEN AG. A more detailed explanation can be found in [13].

The standard software core is the part of the ECU software, where reuse is already performed nowadays – even once the ECU supplier changes. The degree of reusability of a software component is dependent from the kind of software it is (Sect. 3.2). Since the architecture of the standard software core is not only made due to reuse aspects, the reuse effect can still be improved. Nevertheless, they play an important role for reusing function software and have to be taken into account for further developments.

3.5 Tools

Obviously, for all these processes tool support is necessary. Nevertheless, the view changes if the concept of the Function Repository is taken into account. In past days the goal of tool users was to get a seamless tool chain by developing conversion filters between them. This approach cannot work. For every pair of tools (and version) used somewhere in a company or department a filter has to be developed which translates the data from the database to the tool and vice versa. Since the language coverage is different for every tool, there is also a loss of information coming with every conversion. That is no issue as long as going along the V model [2] in one direction only. As soon as the development takes place in different locations of the V model at the same time (simultaneous engineering), the data cannot be kept consistent automatically anymore. The solution for this problem is a standard data model as required in Sect. 3.3. The tools are then the editors for the data in the Function Repository. With this understanding a tool chain is a set of tools working on a common database whereby no manual step is necessary.

3.5.1 Tools Required for the Core Asset Development

There is a need for tools supporting different levels of system descriptions in the field of the core asset development:

- *Requirements* are the textual description of what the user expects from the vehicle. In order to manage the requirements it makes sense to use tools like `DOORS` [24], which provide features like automatic requirement key generation, an extended search engine and document generation possibilities.

- *Interfaces* are all points within an application or module where information is coming from or going to outside of it. A tool is needed to manage these external interfaces. Also internal data might be interesting for debugging purposes. With the `Data Dictionary` [21], it is possible to manage external interfaces and internal data in one tool.

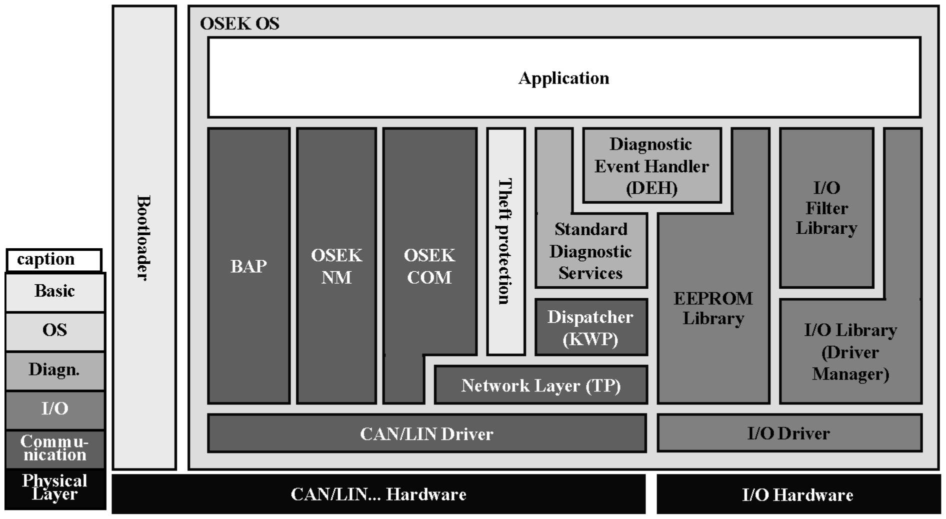


Figure 6: Example of a Standard Software Core (SSC)

- *System architecture* is the functional network, the hardware architecture plus the mapping between themselves. The system architecture can be described for example with UML tools or automotive specific tools like `DaVinci` [8].

- The *behavior description* can be either C code or models in various modeling languages like UML. Another option very common for the automotive industry is the use of `Matlab/Simulink/Stateflow` [15]. `Targetlink` [6] could be used for generating C code out of the model even if the target platform does not support floating point code.

3.5.2 Tools Required for the Product Development

In addition, for the product development there is a need for tools. Most of the standard software components like network drivers for the CAN bus [11] or the OSEK Operating System [20] have to be adapted to the specific network node and hardware platform.

For a tool supported component configuration, information about the communication between the several control units is needed. Each component needs specific information, which is important for the function of the component. The variety of nowadays configuration tools results in the fact that every tool expects its information in a specific data format. For the system integrator, as the only one who knows about the overall communication between the control units, this means a great effort in handling the several data formats and a potential risk of failures. For this reasons, a unique database for the communication information, like in Sect. 3.3 has to be created.

This adaptation is done by configuration tools with integrated code generators. Here a weakness of today's standard software cores can be seen. There is no unique configuration interface for the whole standard software core, but many component specific configuration tools. As the result, it is only possible to optimize the software components, but there is no tool support for optimizing the overall system. This optimization depends on the experience and knowledge of the system developer. Nevertheless, there are already ongoing efforts to integrate the several tools in an open framework with an unique interface.

4. APPLICATION EXAMPLE

In this section an example is given, how the modularization of software can enable the reuse of software components. Some highlights of the process of modeling the function architecture and implementing it on hardware shall be demonstrated using the function exterior light. It was chosen because it is quite challenging to reuse this function over type series without changing the interfaces.

The requirements of this function are taken out of existing specifications. Apparently, there are differences in the partitioning of the rear light of cars even in different type series of a single car manufacturer. Fig. 7 shows how the rear lights behave while combinations of turned on light and usage of the brake. Thereby one car model has one bulb and the other one two bulbs per side. The brightness differs in the modes normal driving with light on and braking.

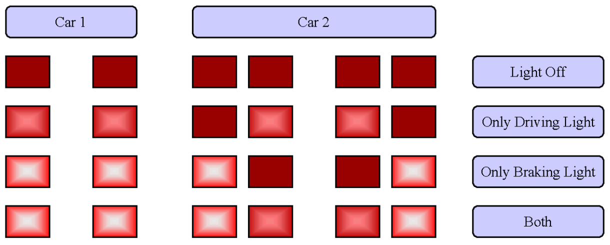


Figure 7: Example for Different Arrangements of Rear Lights

The application exterior light is controlling the bulbs over interfaces from the standard software core. These are on the one hand *rear_light_left_outer*, *rear_light_left_inner*, *rear_light_right_inner* and *rear_light_right_outer* and on the other hand only *rear_light_left* and *rear_light_right*. Additionally the third brake light also can be accessed. The input interfaces are supported from the underlying layers as well. Some of them come via network from other ECU's and some are directly connected to the same ECU, where the exterior light function is seated. In Fig. 8 two not reusable software components can be seen.

In Fig. 9 the software component holding the logic of the light control in it is separated (Exterior Light Common). An adaptation

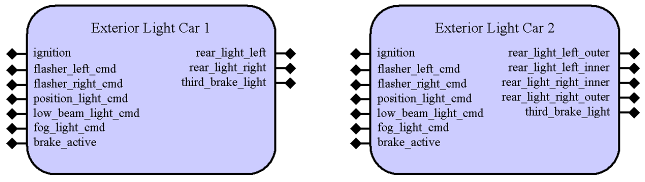


Figure 8: Not Reusable Software Components

software component for each car model is used to connect to get the same interfaces as in the first case.

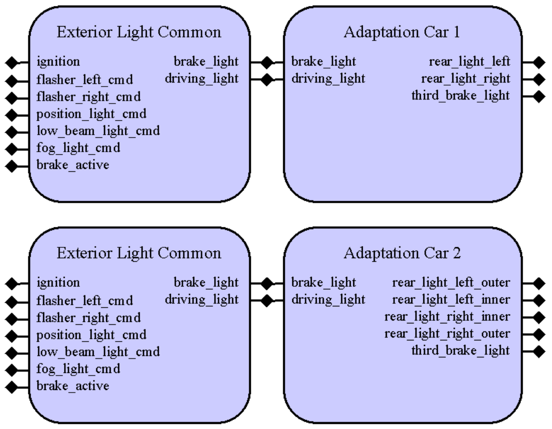


Figure 9: Reusable Application Software Component by Adaptation

Beside the software also the hardware has to be considered. Fig. 10 shows that the hardware can differ significantly in different scenarios. In the left hand scenario car 1 controls the bulbs with digital input and output ports. In the right hand scenario the third brake light is still controlled by a discrete wire. The rear lights are so called smart actuators which are connected via LIN bus. This makes sense as the number of wires can be reduced in that cases.

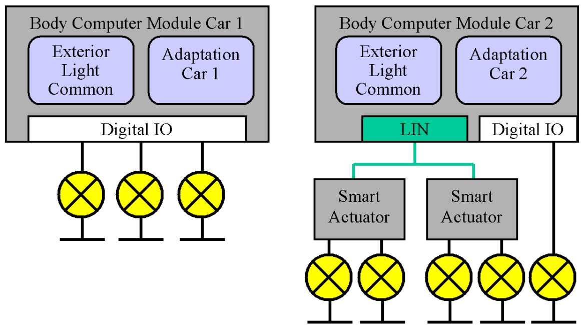


Figure 10: Different Hardware Mappings

This example is simplified since a more detailed example would extend the scope of this paper. The method is transferable to more complex applications. To get an impression of the real complexity of the systems handled in the automotive industry the example of the backup light shall be explained shortly. In Fig. 11 it can be seen that 9 ECU are involved to figure out, if the backup lights can be switched on and which other actions are to undertake after the use of the reverse gear.

5. CONCLUSIONS

This article gives an overview on the challenge of reusing software in the automotive domain from the perspective of an automotive manufacturer. It explains the challenge of reuse explaining the electronic system in cars and pointing out the difference between reuse for suppliers and manufacturers. The process model product line practice is used to collect requirements derived from the reuse of function software. Additionally, a method is introduced, which allows classifying software to the possible kind of reuse. This is a pre-requisite to go from the process of ordering black box control units to a process where software components are reused even after a change of supplier in several control units. In conclusion, an application example is shown applying some parts of the presented framework.

6. REFERENCES

[1] AUTOSAR. Automotive Open System Architecture. http://www.autosar.de, Apr. 2004.

[2] Bundesministerium des Inneren. Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell. Kurzbeschreibung. Bonn, 1997.

[3] CAN in Automation. Controller Area Network (CAN) - an overview. http://www.can-cia.de/can/, Mar. 2003.

[4] Carnegie Mellon Software Engineering Institute. Software Product Lines. http:// www.sei.cmu.edu/plp/product_line_overview.html, Feb. 2003.

[5] P. Clemens and L. Northop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, Boston, 2002.

[6] DSpace. TargetLink – Automatic Production Code Generation for Target Implementation. http://www.dspace.deU/ww/en/pub/products/targetimp.htm, Apr. 2004.

[7] EAST-EEA. Embedded Electronic Architecture. http://www.east-eea.net, Apr. 2004.

[8] B. Hardung, M. Wernicke, A. Krüger, G. Wagner, and F. Wohlgemuth. Development Process for Networked Electronic Systems. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles, Baden-Baden*, pages 77 – 97, Sept. 2003.

[9] J. Hartmann, S. Huang, and S. Tilley. Documenting Software Systems with Views II: An Integrated Approach Based on XML. *Proceedings of the 19th Annual International Conference on Computer Documentation, Santa Fe, New Mexico, USA*, pages 237 – 246, Oct. 2001.

[10] H. Heinecke, A. Schedl, J. Berwanger, M. Peller, V. Nieten, R. Belschner, B. Hedenetz, P. Lohrmann, and C. Bracklo. FlexRay – ein Kommunikationssystem für das Automobil der Zukunft. *Elektronik Automotive*, pages 36 – 45, Sept. 2002.

[11] HIS – Hersteller Initiative Software, Volkswagen AG. HIS / Vector CAN-Driver Specification V1.0. http://www.automotive-his.de, Aug. 2003.

[12] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. *Proc. 14th Euromicro International Conference on Real-Time Systems, Vienna, Austria*, pages 31 – 40, June 2002.

[13] A. Krüger, G. Wagner, N. Ehmke, and S. Prokop. Economic Considerations and Business Models for Automotive

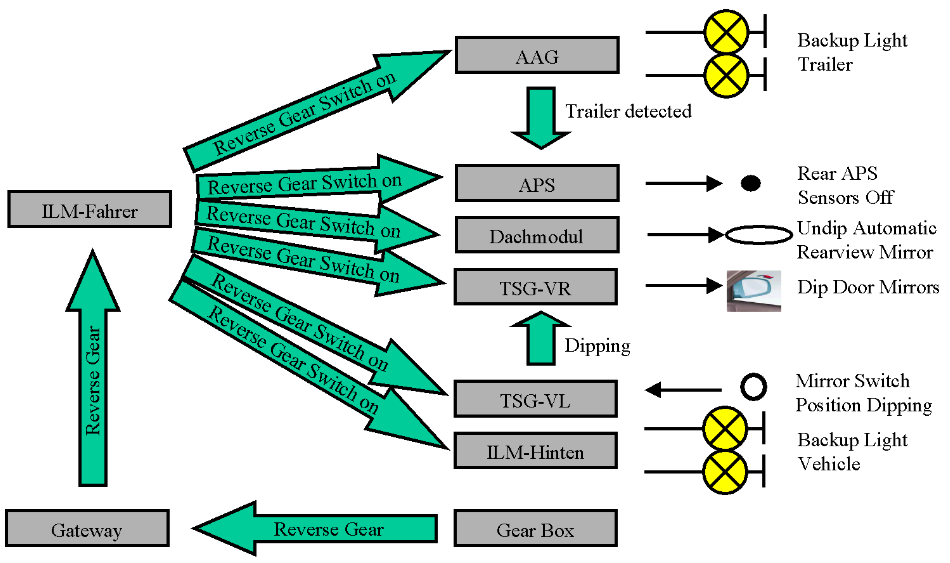


Figure 11: Example Reverse Gear

Standard Software Components. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles, Baden-Baden*, pages 1057 – 1071, Sept. 2003.

[14] LIN Consortium. LIN Specification Package Revision 2.0. http://www.lin-subbus.org, Sept. 2003.

[15] MathWorks. MATLAB and Simulink for Technical Computing. http://www.mathworks.com, Apr. 2004.

[16] Mercer Management Consulting and Hypovereinsbank. Studie, Automobiltechnologie 2010. München, Aug. 2001.

[17] MOST Cooperation. MOST Specification Rev., 2.2. http://www.mostnet.de /downloads/Specifications/, Nov. 2002.

[18] MSR. Development of Methods, Definition of Standards, Subsequent Implementation. http://www.msr-wg.de, Apr. 2004.

[19] OSEK/VDX. Communication Version 3.0.1. http://www.osek-vdx.org/mirror/com301.pdf, 2003.

[20] OSEK/VDX. Operating System Version 2.2.1. http://www.osek-vdx.org/mirror/os221.pdf, 2003.

[21] C. Raith, F. Gesele, W. Dick, and M. Miegler. Audi Dynamic Steering as an Example of Distributed Joint Development. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles, Baden-Baden*, pages 185 – 205, Sept. 2003.

[22] J. Schuller and M. Haneberg. Funktionale Analyse – Eine Methode für den Entwurf hochvernetzter Systeme. Vortrag, VDI-Mechatronik-Konferenz, Fulda, May 2003.

[23] F. Simonot-Lion. In Car Embedded Electronic Architectures: How to Ensure their Safety. *5th IFAC International Conference on Fieldbus Systems and their Applications - FeT'2003, Aveiro, Portugal*, pages 1 – 8, July 2003.

[24] Telelogic. Telelogic Doors Overview. http://www.telelogic.com/products/doorsers/doors/, Apr. 2004.

[25] S. Thiel and A. Hein. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software*, pages 66 – 72, July 2002.

[26] T. Thurner, J. Eisenmann, U. Freund, R. Geiger, M. Haneberg, U. Virnich, and S. Voget. The EAST-EEA project – A Middleware Based Software Architecture for Networked Electronic Control Units in Vehicles. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles, Baden-Baden*, pages 545 – 563, Sept. 2003.