Model-Based Design of ECU Software – A Component-Based Approach

Ulrich Freund, Alexander Burst

ETAS GmbH,
Borsigstraße 14,
70469 Stuttgart
Germany

Abstract: This paper shows how architecture description languages can be tailored to the design of embedded automotive control software. Furthermore, graphical modeling means are put in an object oriented programming context using classes, attributes and methods. After a survey of typical automotive requirements, an example from a vehicle's body electronics software shows the component based architecture. Introducing the concepts of component and connector refinement provide means to close the gap between system theoretical modeling and resource constraint embedded programming practice, leading to an object-oriented behavior description on the one hand and to a common middleware on the other.

1   Introduction

Around ninety percent of vehicle innovations are mainly driven by electronics. Hence, software- and systems-engineering becomes a crucial discipline which vehicle manufacturers and their suppliers have to conquer. Automotive software runs on so-called Electronic Control Units (ECU). Besides a microcontroller and memory, an ECU consists of power electronics to drive sensors and actuators. The software implementing control algorithms combines the sensor values and calculates some meaningful actuator signals. ECU software- and system-engineering is characterized by the following characteristics:

- The software is embedded which means it directly interacts with sensors and actuators and does not change its purpose during lifetime.

- The software fulfils a dedicated control task, i.e. the performance of the control algorithm imposes real-time constraints on the software to be fulfilled.

- The software is realized as a distributed system. The information of sensors located on other ECUs can (and hence will) be used to improve the control algorithm's performance. This means that the sensor information has to be sent to several other ECUs.

- The development itself is distributed. As a rule, a vehicle manufacturer employs several suppliers to deliver the control algorithms and the ECU. Since both the algorithms as well as the ECUs have to work together, the vehicle manufacturer has to do a lot of integration work to get the vehicle on the road.

Since vehicle manufactures traditionally put their focus on production cost rather than on development cost, the sensors and actuators along with the bare ECU represent almost the whole amount of costs for electronics spent. Though software does not have a direct amount of production cost, it is not for free! The only parameter where software contributes directly to production cost is by memory size. It is therefore a must for a software to be as tiny as possible. Furthermore, there is a direct relationship between the amount of production cost for sensors/actuators and the complexity of the control software in between.

To meet these constraints several ECU-programming techniques have been identified in the last twenty years:

- Establishment of building blocks and sub blocks. Exchange of information between building blocks asynchronously by means of messages (ECU global variables) instead of synchronous procedure calls.

- Call stack minimization due to limited synchronous function calls within a building block.

- Use of message duplication in case of tentative task interruption. Cyclic tasks with well chosen cycle time ensure the meeting of hard real-time constraints.

- To save overall vehicle's weight and wiring harness, bus-systems are preferred for the communication between ECUs.

- In case of bus-interconnected ECUs, cyclic broadcasting mechanisms with collision resolution (e.g. CAN) are the preferred communication means.

- Explicit and static scheduling of functions within building blocks according to the timing requirements keeps track of the memory demands and – more importantly – gives the software engineer the chance to intervene.

Well established analysis and design methods in computer science on the one hand and control engineering on the other are UML [OM99], SA/RT [WM85] as well as static data flow graphs [LP95], the latter are better known as control engineering block diagrams. These methods serve well the analysis phase but clearly lack the design requirements of ECU software:

- In UML class diagrams it is not possible to specify communication constraints in associations.

- Communication between or within an (orthogonal) StateChart [Ha87] is by means of events – the order of event handling has to be specified elsewhere to ensure hardware constraints.

- The strength of static data flow diagrams, the automatic built schedule of functions within building blocks and hence invisibility to the user, is their weakness too. To fit a building block into an ECU explicit scheduling of functions has to done elsewhere.

Even worse, all these methods allow a complex design by employing techniques not suited for a efficient ECU software design (e.g. orthogonal states).

This paper is organized as follows: Section 2 introduces abstraction levels in automotive control software engineering. A typical architecture description language for automotive purposes is described in section 3 and used throughout this paper. This language is further refined by behavioral classes (section 4) and component instances (section 5). Section 6 brings separate functions in a vehicle context.

2   Abstraction Levels in Automotive Control Software

Automotive Control Software can be viewed from different levels of abstraction. During the analysis and design process, new design and implementation information will be added to an analysis model and then transformed into ECU software [SZ02].

An appropriate modeling language will cope well with all levels of abstraction, acting as an information integration tool. Typical abstraction levels are

- the analysis model,

- the design model (functional architecture model),

- the implementation model (physical architecture model)
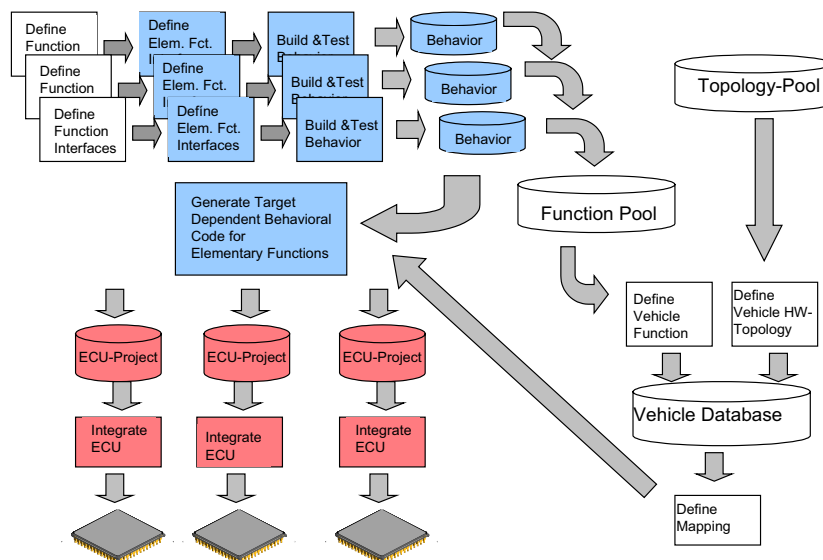
- and the software itself.

Figure 1: Two-stage design process

On every level of abstraction, it is possible to simulate the model, to check the properties of the model and to generate code for an appropriate target, e.g. a PC, an experimental hardware or a series production ECU. Generally speaking, the modeling language is capable of linking the information horizontally (e.g. simulation and code-generation) and vertically (i.e. between different layers of abstraction). This approach forms the basis for the development according to the V-Cycle. However, abstraction levels do not focus on the other side of the pie, namely the interaction between the vehicle manufacturer and its suppliers.

As mentioned above, the vehicle manufacturer is responsible for the overall functionality whereas the suppliers deliver the control algorithms and the hardware. It is the task of the vehicle manufacturer to coordinate the suppliers and let them work as cohesive as possible together. Means for identifying and describing vehicle control functions are necessary; specifying the interfaces is crucial. Currently, the interface description is more or less the communication matrix of a CAN-Bus, i.e. a list of how to link application signals with CAN frames. Due to massive integration problems, it is common understanding among the vehicle manufactures that the bus system is the wrong level of abstraction – some higher level means for integration are necessary.

Provided the appropriate means are available the development of a vehicle control function can be divided in two separate stages. The vehicle project independent development of functions and their tailoring to dedicated vehicle projects.

The vehicle manufacturer identifies vehicle functions, the interaction of elementary functions within the function or with other functions. The vehicle manufacturer asks suppliers to deliver vehicle functions which might be demonstrated by rapid development systems or simulation. The quality and performance of the functions might be assessed – function suppliers might be candidates for future series production projects.

Driven by the market requirements the vehicle manufacturer eventually decides to start a series production project (new vehicle). Instead of reinventing all functions the vehicle manufacturer asks a dedicated supplier to deliver its function for the project. The vital step of identifying the functions has been done independently. All elementary functions are mapped to the ECUs being involved in this project. Needless to say that the communication of the elementary functions between ECUs determines communication matrix. The supplier now receives the mapping system, puts its algorithm in the elementary functions and generates the code for the given ECU.

Figure 1 shows the different tentative roles of the vehicle manufacturer and the supplier. The white boxes show the specification and integration activities of the vehicle manufacturer whereas the rest of design tasks is normally done by the suppliers. The upper left parts leading to the function pool are done independently of a vehicle project. The lower parts are vehicle project dependent.

The next sections presents appropriate means for identifying functions, elementary functions and interfaces. Furthermore, the modeling of the behavior and its mapping to tiny runtime-systems will be described.

3 Component Based Modeling of the ECU-Software Architecture

3.1 Body Electronic Example

A simplified software controlling a window lifting facility shall demonstrate the concepts of architecture modeling and the subsequent refinement steps necessary to run the software on an ECU-network.

The control-software evaluates the state of a switch and drives a motor which opens or closes the window. Of course, opening and closing has to be stopped when the lower or upper limit is reached w.r.t. to the vehicles body. An anti-squeeze-function[1] is omitted due to complexity reasons.

The function offers a normal open/close mode, i.e. the button is pressed during the whole process. A 'tip-mode' opens or closes the windows by pressing the switch only for a very short time. The window opens or closes until either the limit is reached or the button is pressed again. Limit detection is based on measuring the window-lifter's motor current.

3.2 Architecture Description Languages

As a general perception, software architecture is often described by means of "box-and-line" diagrams. Though being very popular they have the big disadvantage that their correctness can neither be ensured by construction nor later be checked by formal methods. Architecture Description Languages (ADL) try to keep the advantages of "box-and-line" diagrams, i.e. their simplicity and understandability even by non-computer-scientists, and augment them with means to analyze their correctness. According to [Ga01], a typical ADL consists of

- Connectors

- Components

- Systems

- Properties

- Style

There exist a lot of activities to tailor architecture description languages to automotive needs. For example, in 1994 the TITUS-project [Ei97][Mü99] was started by DaimlerChrysler. This is an interface based approach [Fr00] and resembles in many cases to the ROOM [SR98][Ru99] methodology, but differs considerably in details mainly to make an 'actor-oriented' approach suitable for ECU-Software. A detailed comparison between the TITUS- and the ROOM methodology is given in [HRW01]. Projects focussing on similar aspects are the BROOM methodology of BMW [Fu98], the French AEE research effort [Bo00], and the Forsoft II (Automotive) project [GR00]. The

---

[1] Einklemmschutz in German

latter project expresses ADL concepts by means of standard UML and uses the tool ASCET-SD [ASD01] to bring designs down to ECU software. Last but not least, in spring 2001 the European research project EAST/EEA[2] was started as an ITEA project. One of the main goals of the EAST/EEA project is to develop a standardized ADL for automotive software.

3.3   Components and Service Access Points

The ADL described in this paper is based on the common characteristics of the above mentioned automotive ADL research efforts. Components are the basic building blocks of this ADL. They employ a class/parts/instance concept and can therefore compared with capsules in ROOM. Since instantiation can only be performed when the final runtime-system is known, component instances to come are described as parts. Systems and subsystems only consist of parts and connectors and cannot have own behavior which is different to ROOM.
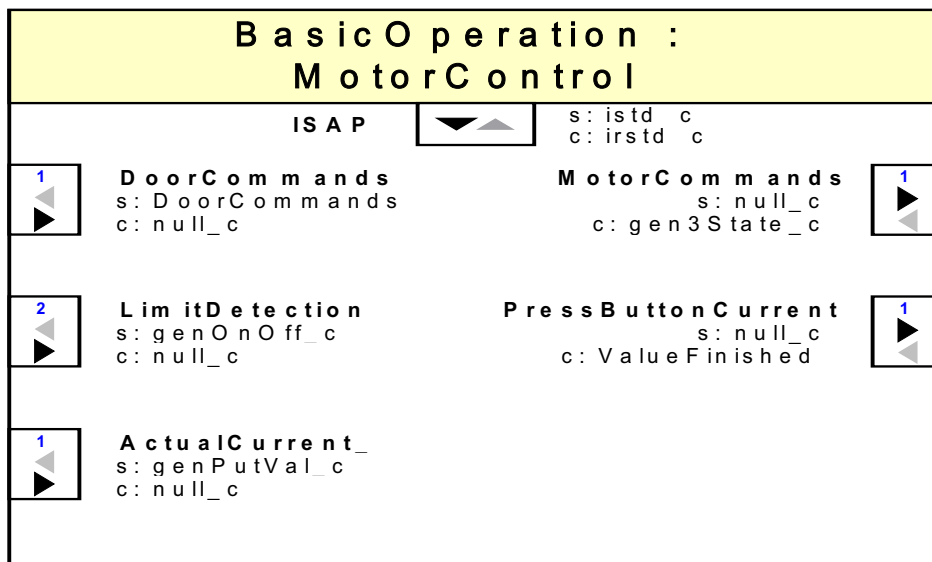
Figure 2: The BasicOperation component with its interfaces

Components are encapsulated from their environment by means of interfaces. In this ADL, interfaces are described by Service Access Points (SAPs) and ports. Interfaces describe what services a component offers to its neighbors as well as which services the component requires from its neighbors. Using a client/server interpretation of components, a SAP providing services is called server-SAP whereas a SAP requiring a service is called client-SAP. However, a SAP's role is associated to the primary communication role since SAPs can describe a bi-directional communication. Thus, every SAP employs a client and a server signal-set complementing each other. This is

---

[2] Embedded Architecture Software Technology/Embedded Electronic Architecture

indicated in Figure 2 by an *s:* prefix (for server) or an *c:* prefix (for client) in front of the signal-set name. If a SAP implements only one role the - non-existing - complementing signal-set is indicated by the `null_c` signal-set. The primary role of a SAP depends on which side of the component the SAP resides: left side for server-SAPs, right side for client-SAPs.

Figure 2 shows the component doing the basic motor control algorithm for one window. It provides services for handling the commands of the switch for a door, appropriate handling if the window reaches the lower limit of the door frame or the upper limit of the roof as well as means to use the actual motor current being measured by dedicated neighbor components. Server-SAPs are drawn on the left side of a component. Consequently, the required services of the component are shown on the right side which are requests to drive the motor in the appropriate direction and delivering a maximum current, measured during a given time. Using SAPs only on the left or right side is one further difference to ROOM[3] and reflects the data flow thinking of automotive control engineers.

Whereas SAPs describe the functional interface, ports are used for navigation purposes between components. Especially, the number of ports per SAP indicate how many clients (or servers) can use the provided service. The service itself is the same for all ports. From this point of view, ports can be interpreted as instances of a SAP. Every SAP must have at least one port. Subsequent sections will show how ports relate to the middleware and SAPs to the behavior of a component.

For example, the `DoorCommands` SAP of the component `BasicOperation` has one port, indicated by the number at the top of the SAP symbol. The `LimitDetection` SAP below the `DoorCommands` SAP has two ports, one for the upper- and one for the lower-limit. Both ports will convey the signals of the signal-set `genOnOff_c` depicted in

Figure 3.

A port of a client-SAP can be connected to more than one component, but it depends on the communication mode whether all connected components will receive a request or only dedicated components. Two communication modes are possible:

- peer-to-peer, meaning that the port has to be selected separately or,

- broadcast, where all connected classes will receive a request.

The broadcast mode is typically used for signals being used by several clients in the vehicle, e.g. the vehicle voltage, the clamp-state or the vehicle's actual velocity. As a rule, the peer-to-peer mode is used to drive several devices of the same type, e.g. an LED.

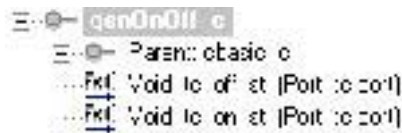[3] The ISAP service access point on the top of Figure 2 is only used initialization purposes

Figure 3: Signal-set consisting of the on and off method

3.4 Kinds of components

Components are elementary in the sense that they do not have further decompositions. Components can be distinguished into client/server and firmware components. Firmware components are directly connected to sensors and actuators. Their behavior can be described by means of C-code. Since firmware components directly incorporate HW-drivers, they are bound to specific ECUs. Client/server components are independent from hardware and can be assigned to arbitrary ECUs in a mapping step described later. An exception are client/server components next to firmware components performing adaptations w.r.t. to sensor and actuator peculiarities. All other client/server components are called monitors.

3.5 Connectors

In this ADL, services are described by means of methods having as a rule no return values. Therefore, the methods have the same meaning as ROOM-signals. The aggregation of all methods (or signals), offered at a server-SAP or being incorporated at a client-SAP, establishes the signal-set being transferred by a connector. Signal-sets can be structured hierarchically using a single inheritance mechanism. A "NULL" signal-set containing no methods represents the root of the signal-set tree. The functionality of a signal-set can be extended by creating a child set and adding new methods.

Figure 4 shows the signal-set `DoorCommands`. Its parent is the `null_c` signal-set set having no methods whereas the `DoorCommands` signal-set consists of the methods `off()`, `open()`, `close()`, `tipopen()`, `tipclose()` and `stop()`.
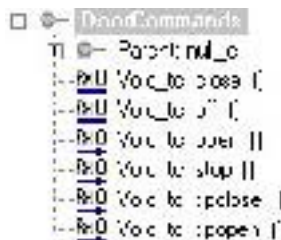

Figure 4: The DoorCommands signal-set

A connector employs two signal-sets having their roles indicated at the connected SAPs by the prefixes *s:* or *c:*. It is mandatory that a server signal-set of a component's SAP has its counterpart as client signal-set at the connected SAP of the neighbored component and vice-versa.
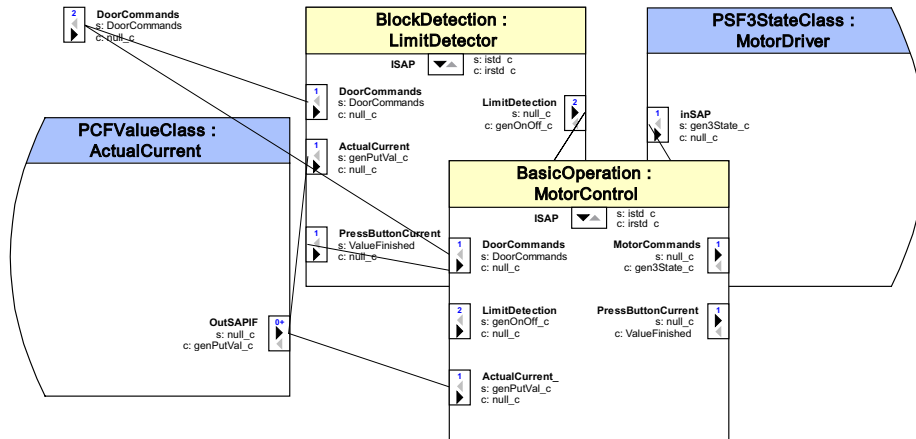

Figure 5: Inner View of the Window Lifting subsystem.

During the refinement phase, the methods have to be described by graphical or textual models (which are translated to C-code later on). This is achieved by behavioral modeling tools (BMT) or pure C-Code. In simple applications the whole functionality can be expressed within the methods, whereas in more complex designs they act as glue for the input vector of a finite-state-machine.

3.6 Systems

Systems serves the need for hierarchy. They can include components or further systems as parts and offer services at SAPs. Systems use services of other sub-systems or components. Since the SAP of a system represents the SAP of a component, the connection between the system's SAP and the component's SAP is called binding. Connected SAPs between subsystems are called bindings too.

The top level system describes the entire structure of the application. Since in automotive software resources are always allocated statically and dynamic instantiation is not used, all connectors are already resolved at compile time.

The example (sub-)system in Figure 5 shows the interface to the outside world in the upper left corner, i.e. the commands of the door-switch. The 'half-rounded' component on the left is used for sensing the motor's current whereas the rightmost component is used to drive the motor directly. The elementary components in the middle are the basic motor control and the limit detector. The door command signals are evaluated by both

elementary components. The same holds for the actual motor current. The measured maximum current is sent from the basic motor control component to the limit detecting component.

3.7 OSEK-based Remote Procedure Call

Using an event driven style, communication between components is asynchronous and explicit. To have control over the timing behavior of two components residing on remote ECUs, it is necessary for the designer to be aware of the traffic a remote procedure call will generate. For example, a `getValue()` procedure call has to be modeled with no return value. Whereas in the classical remote procedure call (RPC) world calling a getValue method of a (tentative stateless) server and expecting the result at a later (unspecified) point in time, the `getValue()` method in the OSEK[4]-based RPC world (or ORPC for short) can only set a flag at the server. The server will then notice the set flag and calls the `putValue(real result)` method of the client. The result will be sent as the actual parameter of the method, thus using the secondary roles of the SAPs' complementary signal-set.

This none-stateless interpretation of a remote procedure call under automotive constraints, hence OSEK-based Remote Procedure Call, not only makes the timing implications explicit to the designer but furthermore encourages a clear design based on pure interfaces. Components support this design approach.

4 Means of Behavioral Modeling

The behavior of a vehicle control function describes the functionality of the system. The system behavior will be measured against the performance criteria. During the analysis and design phase, the performance criteria of a control algorithm has to be checked by means of analysis, simulation and experiments.

Since vehicles are safety critical systems, it is wise to use system theoretic modeling means like finite state machines or control engineering block diagrams to design control algorithms. To bring the design down to an ECU, a more software oriented modeling is crucial, e.g. to make use of a class/instance concept while keeping the advantages of a graphical behavior description. A tool providing this dedicated automotive control software view is ASCET-SD.

---

[4] OSEK means "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" and is a standard for automotive embedded operating systems
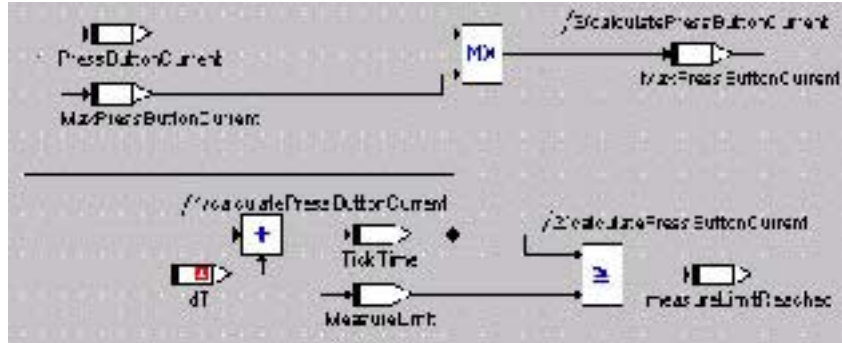
Figure 6: Data-Flow graph method for the maximum current detection

4.1 Using Behavioral Classes as Component Refinement

Component refinement means to add behavior to the components. The component's interfaces have to match the input- and output signals of the control algorithm. The class/part/instance concept of the above described ADL can only be kept during the step of component refinement by using the class/part/instance concept for behavioral modeling too. Furthermore, the method-like signals of the ADL's signal-set should have a direct counterpart in the control-algorithm. Thus, the behavioral class concept described below forms the conceptual basis for component refinement.

4.2 Behavioral Classes

A behavioral class captures control algorithms by means of methods and attributes. Inheritance and associations are omitted. Inheritance is covered by means of variants of a component. A behavioral class is a prototype and can have multiple instances somewhere in the control algorithm. It may use other classes by means of aggregation.

Within an aggregate of an object, the communication is done by means of synchronous method calls, i.e. by calling a method of the aggregated class. In a behavioral class the execution sequence of statements is given by the order of the statements. A method is a collection of statements realizing Boolean and arithmethic expressions as well as method calls to aggregated objects. Control structures like loops and selections constitute a powerful programming language.

Whereas textual description languages focus on statements, graphical descriptions emphasize the system theoretic aspects. Hence, the methods are described by either a finite state machine or a data flow graph. The methods of the SAP `DoorCommands` are shown in Figure 7. If one of these methods is called, the appropriate the enumeration type `SwitchState` will be set to the appropriate value. The method names have the form **/1/off** for the `off()` method. The number in front of the name shows the sequence number. A method calculating the maximum measured current within a given

time is depicted in Figure 6. It is named `calcPressButtonCurrent` and realizes a sequence of three graphical statements.
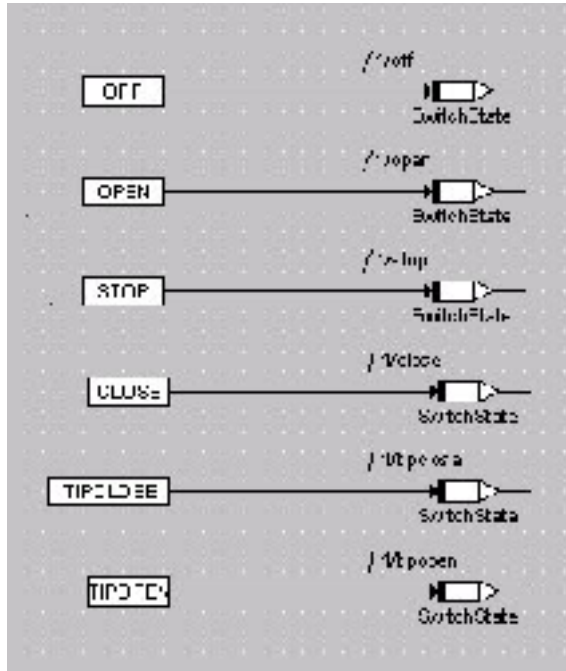

Figure 7: Data flow graph of the DoorCommands methods

Figure 8 shows how the enumeration attribute `SwitchState` is evaluated by the finite state machine diagram of the method trigger. The method `calcPressButtonCurrent` is in invoked in a refined diagram of the hierarchical state `MotorDown`.

4.3   Mapping of Signal-Sets to Methods of Behavioral Classes

The methods of a behavioral class correspond to the signals in a signal-set. Since a component maintains its signal-sets by means of service access points, the behavioral class has to implement all signal-sets in the context of a SAP. Method templates can be generated out of a component description of this ADL.

5   Means of Runtime-System Modeling

As stated in the introduction, automotive control software is tailor-made to a series production vehicle. Whereas the hardware consists of interconnected ECUs nowadays an ECU will employ tiny operating systems. An assessment of the runtime properties of an automotive control function requires its resource allocation scheme which can only be

evaluated on instance level. A component instance combines middleware aspects with instances of behavioral classes. The latter can be derived by means of component refinement whereas the former is the result of connector refinement described in the next section.
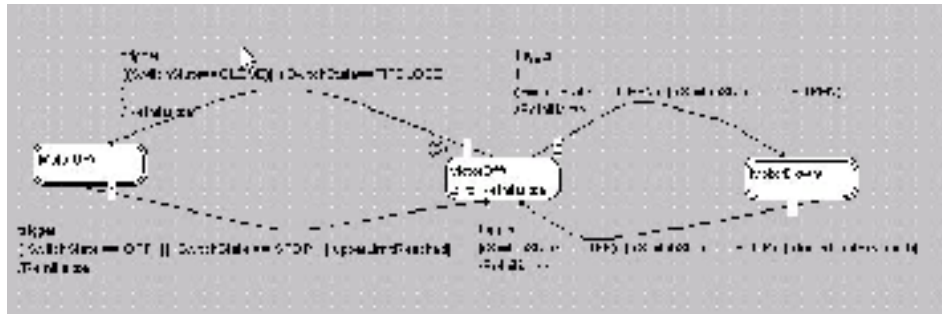

Figure 8: State Transition Diagram of an (Extended) Finite State Machine Behavioral Class
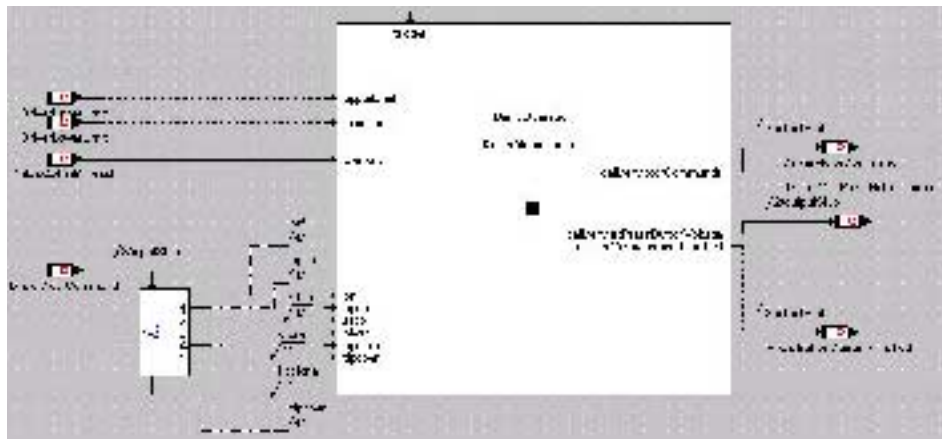

Figure 9: DriverMotorControl component instance with its surrounding middleware

5.1 Connector-Refinement

As mentioned above, the interfaces of a component are described by means of SAPs and ports. Methods of a signal-set employed at a SAP form the template for the methods of a behavioral class thus being the conceptual basis for component refinement. Ports are a template for the IPC-buffers of the middleware and therefore establish the conceptual basis for connector refinement.

5.2 IPC-Buffers

Asynchronous communication between tasks in a real-time operating system is realized by the mailbox principle[5]. Since automotive control systems have a static structure, the mailbox has dedicated entries for each communication connection realizing a connector in a typical architecture description language. Figure 9 shows an example associating the behavioral class instance `DriverMotorControl` with several mailbox-entries (IPC-buffers) on both sides of the component instance. IPC-buffers being read from are shown on the left side whereas IPC-buffers being written to are shown on the right side. The behavioral class instance in the middle has connections from its methods to the IPC-buffers. These connections are called stub-routines and may contain operators. Typical operations are endian conversions or 'method number'-interpretation described in the next section. Therefore, all graphical elements of Figure 9 not belonging to the behavioral class instance constitute the middleware contribution of the component. The ensemble of middleware contribution and behavioral class instance is called a Module in ASCET-SD. On ECU level the IPC-buffers are typically realized as global variable which might be duplicated in case of a tentative interruption by a higher priority task.
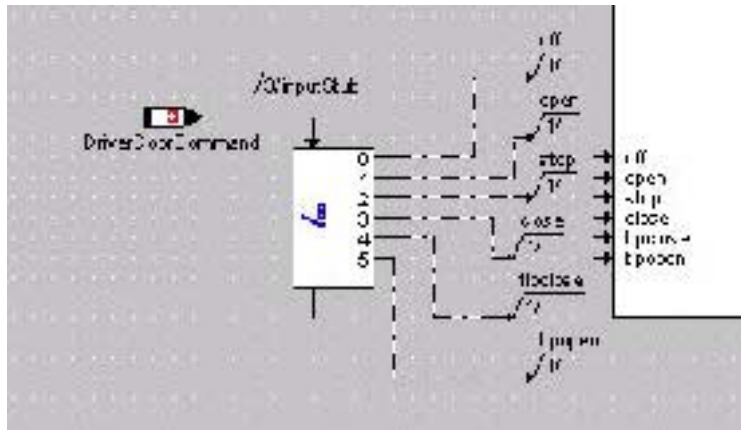

Figure 10: DoorCommands stub-routine

5.3 Stub-Routines

Reading from and writing to mailbox entries is performed by so-called stub-routines. It is their task to read values from the input mailbox entry and call the method of a behavioral class, interpreting the just read values as actual parameters for the methods of the behavioral class. In a signal-set, every method has an associated number starting with 0 for the first method. Depending on the type of the runtime-system, tentative formal

---

[5] The mailbox might be organized as a queue.

parameters of a method can either be stored in separate IPC-buffers or concatenated to the bits reserved for the method number.

The stub-routine for the `DoorCommands` signal-set in the lower left part of Figure 9 is depicted in more detail in Figure 10. The stub-routine `inputStub` reads the method number out of the IPC-buffer `DriverDoorCommand` and calls the appropriate method of the instance `DriverMotorControl`. Remember that the methods of the `DoorCommands` signal-set do not have formal parameters.

As written in section 3, the `LimitDetection` SAP of the component `DriverMotorControl` has two instances in form of ports. Whereas Figure 2 shows only the number of employed ports at the top of the SAP symbol the corresponding IPC-buffers are made explicit on the middleware level. The middleware contribution of the `LimitDetection` SAP is shown in detail in Figure 11.

Figure 11: IPC-buffers of the LimitDetection SAP

To summarize, a component instance consists of :

- An instance of a behavioral class

- Mailbox-entries related to the SAPs of a component

- Stub-routines

From an ECU-centric point of view, the ensemble of all mailbox-entries and stub-routines defines its middleware and hence is part of the runtime-system. The middleware of an ECU, i.e. mailbox-entries and stub-routines, can be automatically generated by using the system model built in the above described ADL. Furthermore, the instantiation process is performed automatically too.

5.4 Scheduling of Component Instances

Scheduling elements are implemented as void/void C-functions being called from OS-tasks. The order of the scheduling elements within an OS-task determines its priority within the task. The calling sequence stub-routine/method-call of a behavioral-class instance will be implemented in a void/void C-function and thus forms a scheduling element. Hence, the activation time of the OS-task determines, via its associated scheduling elements, the timing behavior of the control algorithm realized by behavioral classes.

6   The Vehicle Perspective: ECU-Networks

The above sections have described how a typical architecture description language can form the backbone of component- and connector refinement. The result of the refinement steps is a list of component instances per automotive control function. The ensemble of all component instances to be used in a vehicle determines the software architecture. In an allocation step, the component instances are mapped to ECUs, i.e. forming a deployment diagram. After this mapping, some signals have to be exchanged via a PDU[6] (e.g. a CAN-Frame) of the ECU-network. All PDUs are defined w.r.t. the vehicle. Mapping of every connector to a single PDU is not feasible in an automotive environment because of resource- and timing constraints. Hence, it is common sense to map several signals of connectors to a single PDU provided that they own the same timing properties. Addressing is not an issue because the CAN-bus uses broadcasting mechanisms. The list of signal-sets to be transmitted over the communication medium is given by the distribution of the components to the ECUs. The communication system can be validated by means rate-monotonic-analysis and simulation. Furthermore, the communication software can be configured automatically out of a component instance/ECU mapping description.

7   Summary

To enhance the productivity in embedded automotive control software design, a clear software architecture is indispensable. A typical ADL forms the backbone of a vehicle's software architecture. Components are refined by means of behavioral classes whereas connectors are realized by well established ECU-programming means. Hence, an appropriate refined ADL constitutes the conceptual basis for model-based design of distributed ECU-software.

References

[ASD01]   ASCET-SD User's Guide Version 4.2; ETAS GmbH; Stuttgart; 2001.

[Bo00]   Boutin, S.: Architecture Implementation Language (AIL); 1er Forum AEE; Guyancourt; March 2000; http://aee.inria.fr/forum/14032000/SB_Renault.pdf.

[Ei97]   Eisenmann, J. et al.: Entwurf und Implementierung von Fahrzeugsteuerungsfunktionen auf Basis der TITUS Client Server Architektur; VDI Berichte (1374); pp. 309 – 425; 1997; (in German).

[Fr00]   Freund, U. et. al.: Interface Based Design of Distributed Embedded Automotive Software - The TITUS Approach. VDI-Berichte (1547); pp. 105 – 123; 2000.

---

[6] Protocol Data Unit

[Fu98]   Fuchs, M. et al.: BMW-ROOM An Object Oriented Method for ASCET-SD; SAE Paper 98MF19; Detroit; 1998.

[Ga01]   Garlan, D.: Software Architecture; in Wiley Encyclopedia of Software Engineering,; J. Marciniak (Ed.); John Wiley & Sons, 2001.

[GR00]   Gebhard, B.; Rappl, M.: Requirements Management for Automotive Systems Development; SAE 2000-01-0716; Detroit; 2000.

[Ha87]   Harel, D.: StateCharts: A Visual Formalism for Complex Systems; Science of Computer Programming 8(3); pp. 231- 247; 1987.

[HRW01]   Hemprich, M.; Reiser, M.O.; Weber, M.: Die TITUS-Modellierungsnotation und ihre Zuordnung zu UML/RT; OBJEKTspektrum 2/2001; pp. 32 ff.; 2001. (in German).

[LP95]   Lee, E.A.; Parkes, T.: Dataflow Process Networks; Proceedings of the IEEE; vol. 83; no. 5; pp. 773-801; 1995.

[Mü99]   Müller, A.: Client/Server-Architektur für Steuerungsfunktionen im KFZ; it+ti Volume 41; Issue 5; pp. 41 ff. Oldenbourg-Verlag; 1999; (in German).

[OM99]   OMG: UML Unified Modeling Language Specification. Version 1.3; 1999; http://www.omg.org.

[Ru99]   B. Rumpe et al.: UML + ROOM as a Standard ADL; Proc. ICECCS'99 5th Int. IEEE Conf. on Engineering Complex Computer Systems; pp. 43 - 53; F. Titsworths (eds.); IEEE Computer Society, Los Alamitos; 1999.

[SR98]   Selic, B.; Rumbaugh, J.: Using UML For Modeling Complex Real-Time Systems; 1998; http://www.rational.com/media/whitepapers/umlrt.pdf.

[SZ02]   Schäuffele, J.; Zurawka, T.: Automotive Software Engineering – Current Situation, Perspectives and Challenges; Automotive Electronics I/2001; pp. 10 - 21Vieweg-Verlag; Wiesbaden; 2002; (in German).

[WM85]   Ward, P.; Mellor, S.: Structured Development for Real-Time Systems. Prentice-Hall, 1985.