Incremental Development for Automotive Software in AutoMoDe[*]

Andreas Bauer[1] Jan Romberg[1] Bernhard Schätz[1] Peter Braun[2]
Ulrich Freund[3] Pierre Mai[4] Dirk Ziegenbein[5]

[1]Technische Universität München [2]Validas AG
[3]ETAS GmbH [4]PMSF IT Consulting [5]Robert Bosch GmbH

Abstract

Automotive software development is inherently complex and involves different stakeholders, phases, and disciplines. The AutoMoDe approach to automotive software development defines distinct levels of abstraction for integrated development. To facilitate the design and evolution of heterogeneous automotive software, suitable views for each level are supported, targeting development steps like instantiation, clustering, or deployment of functions. Analysis and synthesis steps enabling a consistent development process across these areas are integrated. The techniques described have been integrated into the tool prototype AutoFOCUS.

1. Introduction

The growing number of functionalities offered by embedded systems and the increased need to combine these functionalities into inter-operating networks has drastically increased the complexity of automotive software. Applying design and implementation techniques for monolithic embedded software for these rather different class of systems has lead to a rising number of problems especially during the integration phase.

Here, costly redesign cycles can be avoided by using models for inter-operating functions, covering early modeling as well as deployment, and supporting a modular development across phases and organisations.

AutoMoDe is based on the AutoFOCUS tool-based approach [4] targeting the modular development of reactive, component-oriented systems. The AutoMoDe approach specifically addresses the development of embedded control software. Therefore modular views of the system under development for functional, logical, and technical architectures, from early design to deployment are introduced. Each of these architectural levels is tailored toward the specific purposes associated with the level, like functional analysis, functional design, or component clustering. By adding automated analysis and synthesis techniques, like clock analysis or component construction, both within and between these levels, a consistent and efficient development process is supported.

Section 2 starts with a short overview of the AutoMoDe development process. Section 3 describes the operational model which is the base for our work in AutoMoDe. Section 4 briefly presents the notations together with the levels of abstraction which we use. Our approach allows seamless modeling of structure and behaviour of automotive software across three levels of abstraction: from the capture of functional dependencies, through a complete behavioural and platform-independent representation, to a more deployment-oriented structure. In Section 5 a simple example for modeling a traction control system is sketched. Based on this example the necessary steps for the translation and further refinement into an ASCET/INTECRIO model is described in Section 6. Section 7 gives a concluding summary.

Related work

For scanning related work, we note that AutoMoDe's contribution roughly fits in two categories: Firstly, a *methodical* framework for automotive control systems development is established, involving specific abstraction levels, transformation steps, and an embedding in the automotive development process. Secondly, AutoMoDe provides *technical* contributions in the area of modeling languages and semantic foundations, transformation languages for CASE tools, and distributed implementation of synchronous dataflow programs.

[*]This work has been supported by the German Bundesministerium für Bildung und Forschung (BMBF) under reference number 01ISC08.

Methodology. There are a number of related methods for model-based design of automotive software [13][14][20][24]. Besides some differences in detail, all of the cited approaches use a number of defined abstraction layers and supporting tools/notations for incremental design of automotive control software, comparable to AutoMoDe. On the other hand, using several tools and notations at the design level, these previous approaches typically do not achieve a tight unification of syntax and semantics across different system views and phases, as this is clearly difficult in a heterogeneous setting. In AutoMoDe, using a unified and semantically founded domain model supported by the AutoFOCUS tool framework [4], allows the integration of novel technical contributions, such as the use of transformation languages for refactoring designs, or semantics-preserving refinement steps, in an integrated automotive design method.

AutoMoDe is specifically based on results from both the Automotive [24] and the EAST-EEA [20] projects and addresses some prevailing deficits: compared to Automotive project, instead of UML 1.x, AutoMoDe uses the AutoFOCUS notation, featuring an explicit notion of components and their interfaces for the description of the structures of embedded systems, and appropriate support for modeling control algorithms, such as data flow diagrams. AutoFOCUS is tightly related to selected UML 2.0 concepts, so possible conformance to the UML standard is not regarded as a critical issue. Compared to UML 2.0, which is a generic standard and thus intentionally adaptable in many respects, AutoFOCUS has a well-defined and unambiguous semantics without semantic variation points. The notational and semantic choices inherent in AutoFOCUS have been extensively validated in a number of industrial case studies. Compared to both EAST-EEA and Automotive, AutoMoDe also puts a stronger emphasis on adequate ways of modeling and preserving *behavioural* aspects of embedded control systems, as opposed to merely modeling *structural* aspects of a design.

Since the structual part of AutoMoDe is based on a component-based paradigm, it can be seamlessly integrated with the "virtual functional bus" approach as defined in AUTOSAR [17].

Technical contributions. The usage of explicit *operational modes* for high-level decomposition of embedded systems, and the design and semantical foundation of appropriate languages, has also been brought forward by other authors, for instance [11]. In addition to the idea of using explicit notations for operational modes, our approach employs such mode representations across several levels of abstraction, especially for coarse-grained structuring of systems, and investigates in particular transformations between different mode representations suited for different abstraction levels.

The concept of expressing frequencies and event patterns as Boolean expressions (*clocks*), along with an accompanying framework for checking and inferring clocks, originates from the field of synchronous programming languages [3]. Our clock checking and inference procedure makes some different tradeoffs in detail than the known clock calculi to combine computational tractability of the procedure with adequate expressivity of the modeling language.

Distribution of synchronous (dataflow) programs is an active area of research. Related publications in the area of semantics-preserving implementation of synchronous programs by preemptively scheduled tasks are, for instance, [1] and [22], again with different tradeoffs in detail.

The use of the *ODL transformation language* to translate between different representations and views of model artefacts is reminiscent of the current development of a transformation language standard in the Object Management Group's MDA [12] framework, and related approaches for model transformation [9].

2. Overview

The AutoMoDe approach covers distinct aspects of a model-based software design method:

Domain model An integrated domain model for the development of automotive embedded software is defined.

Notations The domain model comprises modeling concepts which are syntactically combined in a number of problem-oriented graphical and textual notations.

Abstraction levels For structuring the development process several *abstraction levels* are proposed.

Toolchain The approach is supported by a toolchain for editing, analysing and transforming models at various abstraction levels.

To frame the description of both the individual abstraction levels and the employed notations and concepts, we shall briefly describe the parts of the AutoMoDe toolchain relevant to this paper, shown in Fig. 1. The relation of tools to abstraction levels is straightforward: For the more *abstract* modeling levels, we use the AutoFOCUS tool, which incorporates the modeling notations used for the more abstract levels, along with several options for model transformation and refinement which shall be discussed in the

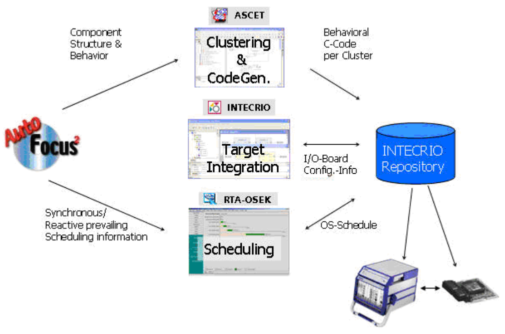


Figure 1. AutoMoDe refinement toolchain

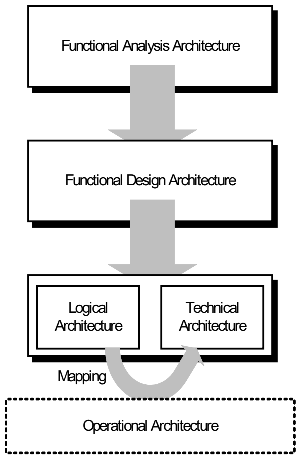


Figure 2. AutoMoDe abstraction levels

remainder of the paper. Towards more *concrete* and *implementation-oriented* modeling, the toolchain uses the ASCET [6], RTA OSEK Planner [10], and INTECRIO [7] tools, which are established commercial modeling tools used for development of automotive embedded software.

Fig. 2 shows the abstraction levels incorporated by AutoMoDe. The upper three levels are supported by the AutoFOCUS tool:

Functional Analysis Architecture (FAA): This level describes structural aspects of functions and their communication dependencies. The FAA description is intended to give a good overview of the functionality and communication: The hierarchy is typically structured in a user- and purpose-oriented way, and ignores realisation constraints. The FAA-level description concentrates on core application-level algorithms: auxiliary functionality such as sensor/actuator preprocessing, diagnosis- and monitoring-related algorithms are not considered.

Functional Design Architecture (FDA): The FDA contains a complete description of the software with respect to both structure and behaviour. The focus is on behavioral validation of the software, and on identifying reusable units. Compared to the FAA structure, the FDA-level software component structure is somewhat realisation-oriented: for instance, for a vehicle dynamics controller, a user-oriented partitioning into "longitudinal" and "lateral" dynamics may be abandoned in favor of a restructured design that considers technical, organizational, and reuse-oriented factors. Some implementation-specific aspects such as clustering, detailed execution timing, and implementation types will not considered on the FDA level

Logical Architecture (LA): In the AutoFOCUS-related part of the logical architecture, the software components of the developed system are clustered into deployable parts, and technical informations such as execution timing or implementation types are fully specified. The purpose of the LA is thus to prepare the design for the final steps of refinement towards the Operational Architecture. Unlike the FDA-level structure, which may consider implementation-independent criteria such as conceptual coherency or reuse concerns, LA-level clusters are typically grouped according to technical criteria stemming from the implementation, such as frequency of activation, priority, or criticality. Consequently, functionality running on the same ECU and within the same task is grouped.

The lower two levels are supported by the commercial tools ASCET, INTECRIO, and RTA OSEK Planner:

Logical Architecture (LA): Beyond the information available in the AutoFOCUS model, further implementation details about OS schedules and processes are specified in ASCET.

Technical Architecture (TA): Necessary information about the platform, such as available ECUs, I/O devices and communication buses, is specified in the TA.

Operational Architecture: Code for the embedded targets is generated and deployed.

INTECRIO is used for integrating several clusters of a developed software system with the used I/O hardware and for deploying these clusters onto several targets. ASCET deals with single clusters and the generation of C-Code for specific targets. RTA-OSEK is used to refine the logical time to a real-time OS schedule.

3. Operational Model

AutoFOCUS employs a message-based, discrete-time communication scheme as its core semantic model [5]. AutoFOCUS designs are built from networks of *components* or *blocks* exchanging *messages* with each other and with the environment via explicit *interfaces* (message ports) and *connectors* (message channels) between interfaces. Messages are time stamped with respect to a global, discrete time base. This computational model supports a high degree of modularity by making component interfaces complete and explicit. It also provides a reduced degree of complexity: Because the discrete time base abstracts from implementation details such as detailed timing or communication mechanisms, the use of timing information below the chosen granularity of observable discrete clock ticks is avoided. Examples for such detailed assumptions include the ordering of message arrivals within one time slot, or the precise duration of message transfer. Real-time intervals of the implementation are therefore abstracted by logical time intervals. Note that the message-based time-synchronous communication model does cater for both periodic and sporadic communication as required for a mixed modelling of time-triggered and event-triggered behaviour.

Modelling Real-Time Behaviour. In AutoFOCUS, modelling of real-time behaviour and dependencies occurs via dedicated *sampling operators* and the introduction of dedicated *unit delays*. Basically, a sampling operator relates messages streams with different frequencies, whereas a unit delay realises a delay of one logical time interval in the communication between two components or blocks.

In the AutoFOCUS notation, the various frequencies and aperiodic event patterns of streams are represented in terms of *clocks* [3]. Each message stream in AutoFOCUS is associated with a clock. The clock for any given stream is a Boolean condition describing the frequency or event pattern. At run-time, a clock evaluates to *true* whenever the associated message stream has a non-absent value.

4. Abstraction Levels and Views

The different system abstractions and their supported views on the system (see Fig. 2) are central to the model-based approach of AutoMoDe. The system abstractions chosen are similar to those defined in [20], but are adapted to match the model-based AutoMoDe development process. The respective abstraction levels and their corresponding use of the AutoFOCUS notations are introduced in the following.

4.1. Functional Analysis Architecture

The *Functional Analysis Architecture* (FAA) is the most abstract level considered in AutoMoDe. The FAA provides a system-level abstraction representing the vehicle functionalities and their dependencies.

System Structure Diagrams. The dominating notation used on the FAA level is called *System Structure Diagram* (SSD). SSDs are used for describing a high-level architectural decomposition of a system, similar to UML 2.0 component diagrams [15]. They consist of a network of components, shown as rectangles, with statically typed message-passing interfaces (ports), shown as black and white circles. Explicit directed connectors (i. e., channels) connect ports and indicate the direction of message flow between components. A component can be either be *hierarchical*, so that it is recursively defined by another SSD, or *atomic*, so that it is defined by a behaviour description in one a number of specifically suited notations (see Sec. 4.2). On the FAA level, being strongly focused on structural modeling, it may be perfectly adequate to leave the detailed behaviour unspecified. An example FAA-level SSD is shown in Fig. 3.

Suited for the description of the structural aspects of both components and functions, the SSD formalism uses delayed communication at the component level. Discrete-time models require delays both for avoiding causal cycles, and in order to be implementable in later phases. Introduction of delays through SSDs is thus a prerequisite for refining designs with reduced revalidation effort. Note that SSDs are not unique to the FAA, but will be used throughout the AutoMoDe approach on other abstraction levels as well (see also Sec. 4.2 and 4.3).

4.2. Functional Design Architecture

The AutoMoDe system abstraction *Functional Design Architecture* (FDA) is a structurally as well as behaviourally complete description of the software part of the system or a subsystem. The description is in terms of actual software components that can be instantiated in later phases of the development process.

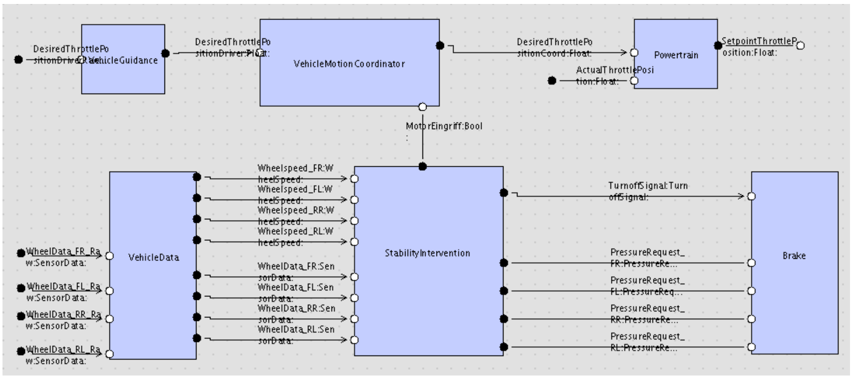

DesiredThrottlePositionDriver:Float
VehicleGuidance
DesiredThrottlePositionDriver:Float:
VehicleMotionCoordinator
DesiredThrottlePositionCoord:Float:
Powertrain
SetpointThrottlePosition:Float:
ActualThrottlePosition:Float:
MotorEingriff:Bool
Wheelspeed_FR:WheelSpeed:
Wheelspeed_FL:WheelSpeed:
Wheelspeed_RR:WheelSpeed:
Wheelspeed_RL:WheelSpeed:
VehicleData
StabilityIntervention
TurnoffSignal:TurnoffSignal:
Brake
WheelData_FR_Raw:SensorData:
WheelData_FL_Raw:SensorData:
WheelData_RR_Raw:SensorData:
WheelData_RL_Raw:SensorData:
WheelData_FR:SensorData:
WheelData_FL:SensorData:
WheelData_RR:SensorData:
WheelData_RL:SensorData:
PressureRequest_FR:PressureRe...
PressureRequest_FL:PressureReq...
PressureRequest_RR:PressureRe...
PressureRequest_RL:PressureRe...

Figure 3. Example SSD component network on the FAA level

Typically the structure at the FDA is more realisation-oriented than at the FAA, and more auxiliary functionality is included.

In contrast to FAA-level functionalities, atomic SSD components in the FDA are required to have a well defined behaviour. Behaviour specifications of atomic components are allowed in terms of Data Flow Diagrams, which specify algorithms in terms of blocks communicating through data flows, Mode Transition Diagrams, which decompose the component's behaviour into distinct operational modes, or State Transition Diagrams, which specify reactive, event-driven behaviour in an automaton style.

Data Flow Diagrams. *Data Flow Diagrams* (DFD) define an algorithmic computation in a structure-oriented manner. Graphically, DFDs are similar to SSDs (see Fig. 4): DFDs are built from individual blocks with ports connected by channels. Typing of ports is dynamic, using type inference properties of operators. A block may be recursively defined by another DFD. The behaviour of atomic DFD blocks is given either through a Mode Transition Diagram (MTD), through a State Transition Diagram (STD), or directly through an expression (function) in AutoFOCUS's base language [8]. For example, block `Difference` in Figure 4 is defined by the function `ReferenceSpeed - WheelSpeed`, where `ReferenceSpeed` and `WheelSpeed` are port identifiers for input ports of block `Difference` (identifiers not shown). It is thereby possible to define adequate block libraries for discrete-time computations with this mechanism.

In contrast to the delayed composition primitives in SSDs, the semantics of DFD composition is "instantaneous", in the spirit of synchronous languages [3]. In the AutoFOCUS tool, instantaneous commu-

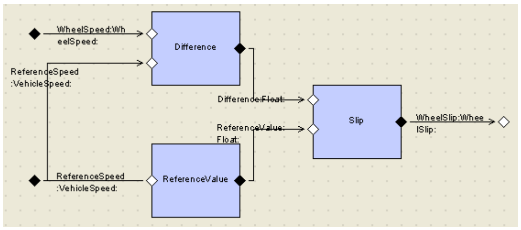
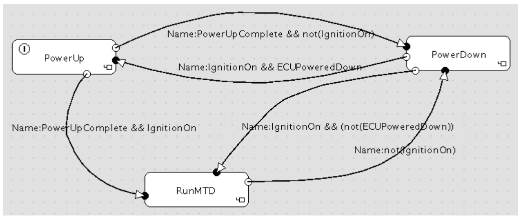


Figure 4. Example DFD for slip determination

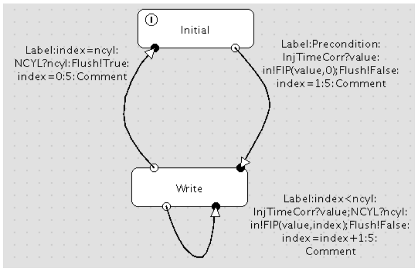
nication primitives are accompanied by a causality check for detecting instantaneous loops. Note that computations "happening at the same time" in FAA-, FDA- or LA-level models are perfectly valid abstractions of sequential, time-consuming computations on the level of the *Operational Architecture* (OA) if the abstract model's computations are observed with a delay, such as the delays introduced by SSD composition. The duration of the delay then defines the deadline for the sequential computation on the OA level.

Mode Transition Diagrams. *Mode Transition Diagrams* (MTDs) are used to represent explicit system modes and alternate behaviours within modes (see Fig. 5). MTDs consist of modes and transitions between modes. Transitions are triggered by certain combinations of messages arriving at the MTD's component. The behaviour of the component within a mode is then defined by a subordinate DFD or SSD associated with the mode, which may be further decomposed: Consequently, MTDs can be used up to the highest levels in the model hierarchy. MTDs thus provide a valuable means of architectural decomposition specifically suited for embedded control systems.



PowerUp
PowerDown
RunMTD
Name:PowerUpComplete && not(IgnitionOn)
Name:IgnitionOn && ECUPoweredDown
Name:PowerUpComplete && IgnitionOn
Name:IgnitionOn && (not(ECUPoweredDown))
Name:not(IgnitionOn)

Figure 5. Example MTD for operational modes of a gasoline engine



Initial
Write
Label:index=ncyl: NCYL?ncyl:Flush!True: index=0:5:Comment
Label:Precondition: InjTimeCorr?value: in!FIP(value,0);Flush!False: index=1:5:Comment
Label:index<ncyl: InjTimeCorr?value;NCYL?ncyl: in!FIP(value,index);Flush!False: index=index+1:5: Comment

Figure 6. Example STD for injection timing control of a gasoline engine

State Transition Diagrams. *State Transition Diagrams* (STD) are extended finite state machines with states and transitions between states (see Fig. 6). STDs are similar to the popular Statecharts notation, but with some syntactic restrictions, such as no AND-states, no inter-level transitions, and restricted primitives for preemption. Through the chosen restrictions, semantic ambiguities allowed by some standard Statecharts dialects [23] are avoided.

Though MTDs and STDs both specify control flow and look similar at first glance, the two notations use notably different syntax, and have largely orthogonal purposes. STDs write to outputs and access local variables *directly*, while MTDs switch between different subordinate behaviors, which in turn (*indirectly*) determine their local states and the MTD's outputs. According to preliminary experiences, the former seems to be better suited for sporadic, event-triggered computation, while MTDs are easier to reconcile with switched periodic computation. Other than MTDs, STDs are *not suitable for high-level decomposition* according to control flow.

4.3. Logical and Technical Architecture

The *Logical and Technical Architecture* (LA, TA) is the most implementation-oriented abstraction level supported by AutoFOCUS. For the transition from

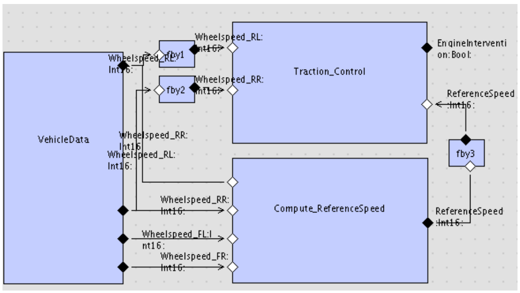


Figure 7. Simplified example CCD for traction controllerr

FDA to LA, FDA-level components are instantiated and grouped into clusters at the LA level. The TA represents hardware and platform components (ECUs, communication buses, message frames) used to implement the system. A cluster can be thought of as a "smallest deployable unit" in a software system. Consequently, several clusters may be mapped to a given operating system task at the OA level, but a given cluster will not be split across several tasks.

While we use AutoFOCUS to demonstrate the AutoMoDe method, AutoFOCUS supports only some aspects of the LA and TA level. Therefore we use two typical commercial tools, ASCET and INTECRIO, for the descriptions at these levels.

Cluster Communication Diagrams. The notation used for top-level definition of the LA structure is called *Cluster Communication Diagram* (CCD). Syntactically, CCDs are largely equivalent to DFDs, but are named differently according to their specific methodical purpose. Unlike general DFDs, CCDs underly certain restrictions: blocks (*clusters*) have statically typed interfaces, and may not be recursively defined in terms of other CCDs. Based on the clocks and the implementation strategy, unit delays are enforced at certain cluster boundaries [2]. The type system at the LA level is extended by implementation types which capture the more or less platform-related constraints associated with the implementation. Basically, an implementation type is the concrete realisation of an abstract type, such as `Int8` being the 8-bit realisation of the type `Integer`. Fig. 7 shows an example CCD for the traction control system.

CCDs can be implemented by sets of communicating real-time tasks. Typical automotive implementation platforms may use preemptive scheduling of tasks: this poses some challenges for implementation of deterministic, logical-time models inherent in the CCD description. [2] outlines a method for implementing multirate CCDs based on fixed-priority, preemptive scheduling. The method uses a deadline-monotonic mapping from clocked clusters to prioritised tasks: the cluster with the smallest inter-event arrival time corresponds to the task with the highest priority. Inter-cluster communication is achieved by a wait-free inter-process communication (IPC) mechanism based on double buffering for data-consistent communication from low-frequency to high-frequency tasks [16]. The real-time, implementation-level delays correspond to delays on the level of logical time in the CCD's semantics: with the AutoMoDe implementation scheme, the zero-delay communication in logical time can only be implemented in certain situations. On the level of logical time, this corresponds naturally to the delay constraints inherent in CCDs. See [2] for further details of the implementation scheme.

5. Example: Traction Control System

The incremental design of an automotive control system with the AutoMoDe method will be demonstrated by means of a Traction Control System (TCS).

Overview. A traction control system compares the wheel speeds of the drive wheels of a two-wheel-driven vehicle with the actual vehicle velocity. Wheel speeds above the actual vehicle velocity indicate slip. This slip will usually result from excessive engine torque in relation to the given road conditions. In the case of slip, two typical actions are taken.

1. If just one of the two drive wheels slips, the brake calliper will be actuated.

2. If both drive wheels slip, the engine torque will be reduced. In a gasoline engine, this is typically achieved by manipulating the engine's spark advance through its ignition system, or by reducing throttle throughput. The case study examines the latter option.

A TCS system typically interacts tightly with an Antilock Braking Systems (ABS). The TCS and ABS systems both use wheel/vehicle speed signals, and both interact with brake callipers.

The TCS model. As the behavioural definition of the `StabilityIntervention` component shown in Fig. 3, the DFD in Fig. 8 defines the essential blocks for general stability control and vehicle dynamics intervention. Beyond the core `Traction_Control` block, the `StabilityIntervention` component is

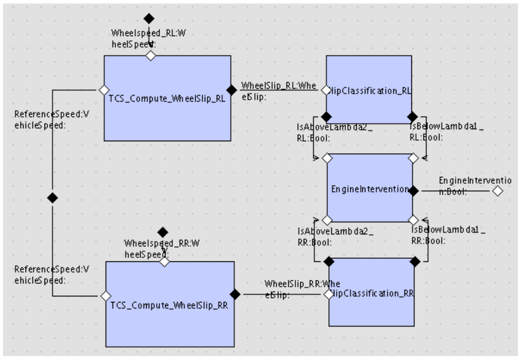


Figure 9. DFD for `Traction_Control`

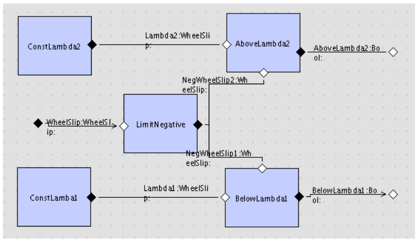


Figure 10. DFD for `SlipClassification_RX`

comprised of a reference speed determination, the traction controller, a brake slip controller, an acceleration controller, as well as a coordination of the actuation requests for the brakes. The result of the brake actuation request coordination is a pressure request to the hydraulic valves which manage the fluid supply to the brake callipers. The figure schematically illustrates the frequency of execution for the different blocks, which is specified by clocks on corresponding message streams (not shown in the diagram).

Figure 9 shows the inner view of the `Traction_Control` block. From the wheel speed signals a reference velocity is calculated against which the actual wheel speeds are compared. The resulting slip value is normalized w.r.t. the vehicle velocity and then classified. The classification yields, for each rear wheel, whether the value is above a given threshold value (`AboveLambda2`), or below a second threshold value (`BelowLambda1`). The classification is used to trigger the throttle manipulation algorithm in case of slip.

The slip determination is shown in Figure 4 while the slip classification is shown in Figure 10. The EngineIntervention block in Figure 9 will influence the current throttle position by a certain amount if the wheelslip of both wheels is outside the limits.

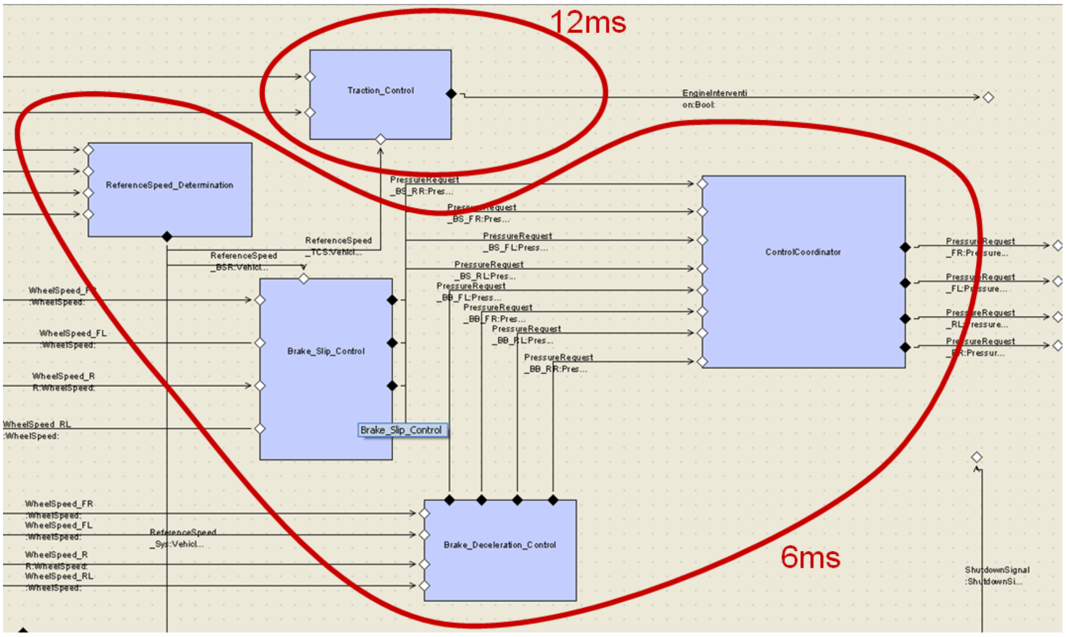

12ms
Traction_Control
EngineInterventi on:Bool:
ReferenceSpeed_Determination
ControlCoordinator
Brake_Slip_Control
Brake_Deceleration_Control
6ms
WheelSpeed_FR :WheelSpeed:
WheelSpeed_FL :WheelSpeed:
WheelSpeed_R R:WheelSpeed:
WheelSpeed_RL :WheelSpeed:
ReferenceSpeed _BSR:Vehicl...
ReferenceSpeed _TCS:Vehicl...
ReferenceSpeed _Sys:Vehicl...
PressureRequest _BS_RR:Pres...
PressureRequest _BS_FR:Pres...
PressureRequest _BS_FL:Press...
PressureRequest _BS_RL:Pres...
PressureRequest _BB_FL:Press...
PressureRequest _BB_FR:Pres...
PressureRequest _BB_RL:Pres...
PressureRequest _BB_RR:Pres...
PressureRequest _FR:Pressure...
PressureRequest _FL:Pressure...
PressureRequest _RL:Pressure...
PressureRequest _RR:Pressur...
ShutdownSignal :ShutdownSi...

Figure 8. DFD for `StabilityIntervention`

The amount of throttle actuation is calculated in the throttle control block which is part of the Powertrain component shown in top leftmost block of Figure 3. The throttle valve is itself a dynamic system which is controlled by a PID controller computing a setpoint for the throttle motor.

Corresponding to the different temporal resolution of sensors and actuators, the different parts of the traction control system are executed at various frequencies. For example, the wheel and vehicle speed determination might run in a 6ms task, while the traction control might run in a 12ms task. Throttle control is done every millisecond (Fig. 8). In AutoFOCUS the rates are described as different clocks while in ASCET or INTECRIO, the differently clocked clusters will be assigned to disjoint tasks, each task triggered at a rate corresponding to its clock in the model.

6. Refinement to ASCET/INTECRIO

Abstract modeling based on the message-based, discrete-time semantics schema of AutoFOCUS, and validating control algorithm at the FDA and LA levels, is just one side of the coin in embedded automotive software development. For retaining the properties established on the abstract levels, it is imperative that the translation to a real-time system preserves the model's semantics. Real-time executables synthesized with ASCET consists of tasks, which in turn call `void(void)` routines (corresponding to the *process* concept in ASCET) written in a sequential language, such as C. Communication between ASCET processes and tasks is either through *global variables*, or through inter process communication messages (*IPC messages*. ASCET *modules* group processes and messages, and ASCET *projects* group modules.

The behavior of ASCET processes can be defined either by *state machines*, a C-like language (*ESDL*), detailed *C code*, or a *block diagram* notation. In order to preserve the rich information in graphical AutoFOCUS models, the refinement tool chain makes extensive use of the block diagram option: each ASCET block diagram consists of elementary *operators*, *variables*, and a total order of assignments for the given process, indicated by enumerated *sequence calls*. Using a sequential, programming-centric semantics, ASCET block diagrams are thus very close a sequence of assignments and corresponding right-hand-side expressions in a sequential programming language, and have a notably different syntax and semantics than AutoFOCUS DFDs.

In addition to the software described in an LA model, there are plenty of hardware-related interfaces to consider, such as drivers for I/O devices or interrupt service routines (ISRs).

To translate AutoFOCUS models to real-time software in ASCET, a refinement tool chain was developed in AutoMoDe, incorporating an AutoFOCUS refinement plugin, and the ETAS tools ASCET [6], RTA-OSEK [10], and INTECRIO [7]. This refinement chain is shown in Fig. 1. We shall summarize the entire refinement procedure briefly before describing it in more detail: The refinement tool translates the clusters defined in message-based, discrete-time model of AutoFOCUS to real-time modules, processes and messages in ASCET, shown in the top middle of Fig. 1. The ASCET code generator translates the clusters to C code: the clusters are then integrated on a rapid development target (lower right hand side of Fig. 1). Target integration comprises the implementation of an OS schedule: The schedule is directly derived from the clocks inherent in the AutoFOCUS design. The hardware related interfaces, such as interrupt routines for driving the I/O devices, are added manually to the synthesized design.

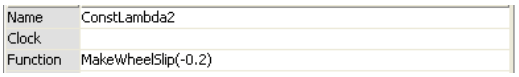
In following, the applied transformation steps are described in detail: *Module Identification*, *Sequence Call Generation*, *Cluster Definition*, *Software System Construction*, *Target Integration*, and *OS Configuration*.

Module Identification and Sequence Call Generation. The AutoMoDe refinement algorithm transforms CCD clusters to ASCET modules, each module containing one ASCET process. As described in Section 4.3, the clusters correspond to the "smallest deployable units" of a design. Each CCD cluster, in turn, is defined by a behavioral description in AutoFOCUS, such as a (possibly hierarchical) DFD.

Each cluster corresponds to a flat block diagram in ASCET[1]. Within the hierarchically structured DFD, the atomic AutoFOCUS blocks, whose behavior is specified through expressions in a textual language, are transformed into a number of operators, interconnections, IPC messages, and corresponding sequence calls in ASCET. The sequence call numbering can be inferred from the causality inherent in the AutoFOCUS semantics.

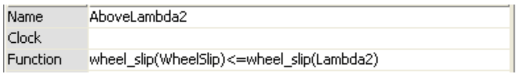
As a very simple example of block diagram translation, we consider a limit check from `SlipClassification_RX`, where `RX` stands for either `RL` or `RR`, and whose DFD is shown in Fig. 10. It is checked whether the actual slip is above the limit `lambda2`. The block `ConstLambda2` provides upper the limit `lambda2`: the block defined by a simple textual constant (constructor function), see Fig. 11. The block `AboveLambda2` compares its two inputs, `Lambda2` and `WheelSlip`, and returns the result. The selector function `wheel_slip` is needed

[1]Translating the AutoFOCUS model hierarchy to corresponding hierarchy features in ASCET is technically feasible, but is not realised in the current version of the refinement chain.



Name	ConstLambda2
Clock	
Function	MakeWheelSlip(-0.2)

Figure 11. Properties of the `ConstLambda2` Elementary Block



Name	AboveLambda2
Clock	
Function	wheel_slip(WheelSlip)<=wheel_slip(Lambda2)

Figure 12. Properties of the `AboveLambda2` Elementary Block

because the type `WheelSlip` is a "wrapper" data type in AutoFOCUS, defined as: `data WheelSlip = MakeWheelSlip(wheel_slip:Float)`. The translation of these two blocks into an ASCET block diagram is shown in Figure 13.

The complete ASCET block diagram translation of the `SlipClassification_RX` cluster from Figure 10 is depicted in Figure 14. The `SlipClassification_RX` DFD is translated into a block diagram with three sequence steps of one process. The number of necessary ASCET processes is derived from the number of different clocks used. In the example of `SlipClassification_RX`, only one clock is modelled, so only one ASCET process is needed. The assignment of sequence call numbers follows from the causality inherent in the AutoFOCUS DFD. In our example three sequence steps are needed. Firstly, the `NegWheelSlip` variable is assigned as part of an ASCET IF statement (`/5/process`), then `BelowLambda1` and `AboveLambda2` are assigned (`/7/process` and `/8/process`).

The translation is optimised such that only a minimal number of internal help variables representing the channels is needed. In this example the internal variable `NegWheelSlip` is introduced. This variable stores the result of the limited negation of `WheelSlip` DFD block, and is subsequently read for the computation of `BelowLambda1` and `AboveLambda2`.

Each external port of a cluster is translated

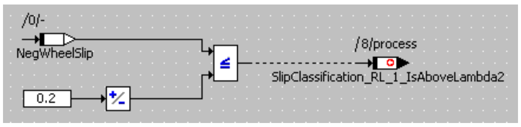


Figure 13. ASCET block diagram for `AboveLambda2`

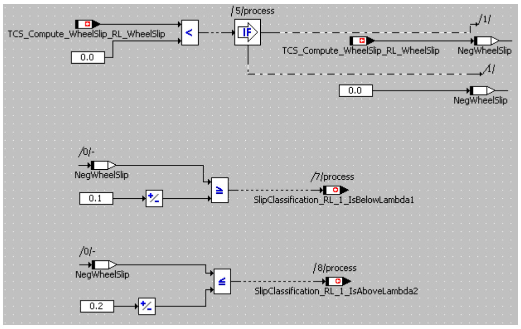

TCS_Compute_WheelSlip_RL_WheelSlip
0.0
/5/process
IF
TCS_Compute_WheelSlip_RL_WheelSlip
NegWheelSlip
/1/
0.0
NegWheelSlip
/0/-
NegWheelSlip
0.1
/7/process
SlipClassification_RL_1_IsBelowLambda1
/0/-
NegWheelSlip
0.2
/8/process
SlipClassification_RL_1_IsAboveLambda2

Figure 14. ASCET block diagram for `SlipClassification_RX`

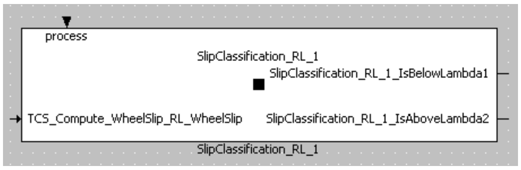

process
SlipClassification_RL_1
SlipClassification_RL_1_IsBelowLambda1
TCS_Compute_WheelSlip_RL_WheelSlip
SlipClassification_RL_1_IsAboveLambda2
SlipClassification_RL_1

Figure 15. ASCET module for `SlipClassification_RL`

into an IPC message. In the example the cluster `SlipClassification_RX` is translated into a module as shown in Figure 15. CCDs are not explicitly supported by the AutoFOCUS graphical editors, so in the current translation clusters are AutoFOCUS components marked with the stereotype `<<cluster>>` (not shown in the diagrams). In the example one "receive" IPC message for the wheel slip value and two "send" IPC messages of ASCET type `logic` are used.

These clustering algorithms as well as other model transformations are defined in AutoFOCUS by the help of the Operation Definition Language (ODL) [18]. The ODL is a first-order logic language which can be used for the definition of checks and transformations. Within an ODL expression user interaction is possible. So for example a clustering algorithm may ask the user to provide a clock and some components which are using this clock. Afterwards the algorithm could transform the AutoFOCUS model, so that a CCD cluster is inserted and the communication is rerouted appropriately.

Cluster Definition. For the CCD of Fig. 7 comprising clusters `VehicleData`, `Traction_Control`, and `Compute_ReferenceSpeed`, the translation yields three ASCET modules as shown in Figure 16. In the example also the connectors between ASCET modules are shown. These connectors represent ASCET messages.

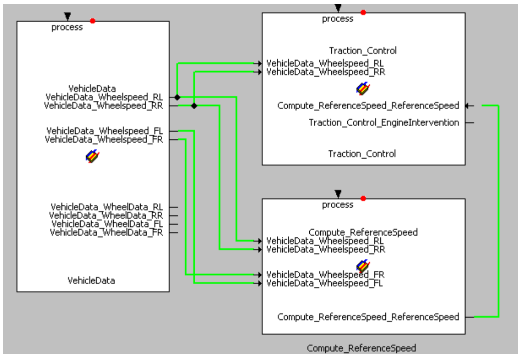

process
VehicleData
VehicleData_Wheelspeed_RL
VehicleData_Wheelspeed_RR
VehicleData_Wheelspeed_FL
VehicleData_Wheelspeed_FR
VehicleData_WheelData_RL
VehicleData_WheelData_RR
VehicleData_WheelData_FL
VehicleData_WheelData_FR
VehicleData
process
Traction_Control
VehicleData_Wheelspeed_RL
VehicleData_Wheelspeed_RR
Compute_ReferenceSpeed_ReferenceSpeed
Traction_Control_EngineIntervention
Traction_Control
process
Compute_ReferenceSpeed
VehicleData_Wheelspeed_RL
VehicleData_Wheelspeed_RR
VehicleData_Wheelspeed_FR
VehicleData_Wheelspeed_FL
Compute_ReferenceSpeed_ReferenceSpeed
Compute_ReferenceSpeed

Figure 16. ASCET Modules representing three AutoFOCUS clusters

The next refinement step is the grouping of ASCET modules to ASCET projects. In ASCET, modules grouped in one project run on the same ECU, so the module-to-project grouping activity follows from the mapping of LA clusters to ECUs from the TA. Given a suitable such mapping, the step of forming ASCET projects could be easily performed automatically. Because it is a lightweight activity, the current refinement chain leaves this operation to the user.

Software System Construction. When all ASCET modules have been grouped to ASCET projects, the ASCET code generator will be used to generate C code conforming with INTECRIO. For every cluster respectively ASCET module this will result in C code as well as a code description using the SCOOP-IX format. Both descriptions establish an INTECRIO module (not to be mistaken for an ASCET module). For software system construction in INTECRIO, all clusters are then imported as INTECRIO modules to INTECRIO and might be further clustered by INTECRIO functions. The result of all clustering steps is shown in Figure 17. Sensor and actuator modules are shown on the left hand and right hand sides of the INTECRIO diagram. For example, the wheel speed calculation is done by edge detection. The ports at the exterior of the diagrma interface the modules to the I/O boards of an ES-1000 rapid prototyping system.

The wheel speed sensors do edge detection and trigger a counting process as ISR. Later on, this triggering will be the only information exchange between the I/O devices and the wheel speed sensor module so that no additional transformations to the synthesized software is necessary.

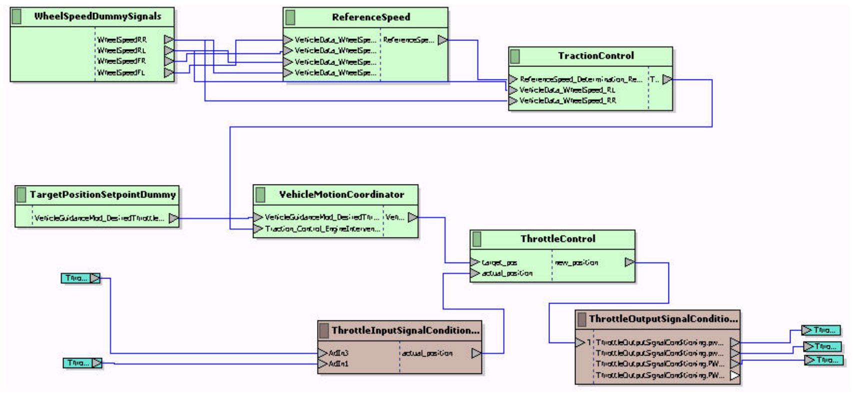


Figure 17. The Traction Control System in INTECRIO

Target Integration. The ETAS rapid development system ES-1000 is a VME bus-based rapid prototyping target. For the traction control example it consists of a microprocessor board (ES 1135), an A/D converter board (ES 1303), and a PWM board (ES 1330). The software interface representation of the A/D converter board as INTECRIO module has to be added so that the PWM signals for the hydraulic valve interaction and the throttle motor are connected to the TCS system.

OS Configuration. On the level of AutoFOCUS design, the traction control model employs message streams running on different clocks. From the model-based point of view, the throttle control algorithm runs 6 times faster than the anti-lock braking algorithms and 12 times faster then the traction control algorithm.

This clocked design will be translated to a real-time system where the throttle controller cycle time is set to 1ms. In combination with the clock schemes, there will be a 1ms, a 6ms and a 12ms task. The processes of the INTECRIO modules will be allocated to the appropriate tasks.

As explained in more detail in Section 4.3, [2] describes a correct-by-construction method for implementation of time-synchronous AutoFOCUS programs based on rate-monotonic scheduling. The approach uses the aforementioned double buffering technique for communication from low-frequency to high-frequency tasks. For analysing this default configuration in the context of the traction control example, we use the planner feature of the tool RTA-OSEK [10] which implements algorithms described in [21].

In the case that the simple top-down, rate-monotonic approach of [2] is not sufficient for a particular situation, the algorithms described in [1] can ensure in a bottom-up fashion that the multiple clock scheme is appropriately implemented by some given real-time OS schedule. The basic idea of this algorithm is to check whether all signals are read in the appropriate cycle and that a writer is not overtaken by the reader. Beyond the usage of checking algorithms, it is common automotive design practice to support this analysis by measurements on the real executing system [19] using dedicated measurement and tracing tools.

7. Summary and Outlook

The AutoMoDe approach to model-based automotive software development is based on integrated design and modeling techniques with system views on various levels of abstraction. In this paper, we have shown that by using application-specific modeling notations and (semi-)automated analysis and synthesis steps, one can on the one hand establish *abstraction* and a *separation of concerns* based on different views and abstraction levels, and maintain overall *consistency* on the other hand. We believe this to be a central necessity for the future automotive software development process: while abstraction and separation of concerns cater to the needs of the numerous stakeholders in the automotive development process, automated consistency ensures efficiency and tractability of the used abstractions and views.

The development process designed within the AutoMoDe project was evaluated in three automotive case studies: the described traction control system, an engine management controller (containing the throttle valve actuation), and a power windows controller. As part of the physical prototyping of the traction control case study, a real throttle valve was actuated based on real-time input to demonstrate physically that the AutoMoDe approach scales down to the implementation level.

Part of the described case study is the refinement of an abstract AutoFOCUS model into a technical model how it is built today in practice. The technical model is represented in ASCET and INTECRIO which are examples for state of the art modelling tools. By translating the abstract model and the further refining of the technical model we showed which information could be generated automatically and which part of the model has to be refined manually. While AutoFOCUS shows the potential of a model-based development, the refinement into ASCET and INTECRIO shows that such a approach is principally applicable in practice. An important result of this case study was, that the AutoMoDe approach can lead to an efficient realization.

References

[1] M. Baleani, A. Ferrari, L. Mangeruca, A. L. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff. Correct by construction transformations across design environments for model-based embedded software development. In *DATE 05*, 2005.

[2] Andreas Bauer and Jan Romberg. Model-based Deployment in Automotive Embedded Software: From a High-Level View to Low-Level Implementations. In *Proceedings of the 1st International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, Hamilton, Ontario, Canada, June 2004.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. LeGuernic, and R. De Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[4] M. Broy et al. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, (14(3)):121–134, 1999.

[5] M. Broy and K. Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement.* Springer, 2001.

[6] ETAS GmbH Stuttgart. *ASCET User Guide V 5.1*, 2005.

[7] ETAS GmbH Stuttgart. *INTECRIO User Guide V 1.0*, 2005.

[8] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[9] A. Königs. Model transformation with triple graph grammars. In *Workshop on Model Transformations in Practice, MODELS*, Montego Bay, Jamaica, 2005.

[10] LiveDevices York. *RTA-OSEK User Guide V 4.0*, 2005.

[11] F. Maraninchi and Y. Raymond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[12] S. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled: Principles of Model Driven Architecture*. Addison-Wesley, 2004.

[13] K. D. Müller-Glaser, G. Frick, E. Sax, and M. Kühl. Multiparadigm Modeling in Embedded Systems Design. *IEEE Transactions on Control Systems Technology*, 12(2):279–292, March 2004.

[14] M. Mutz, M. Huhn, U. Goltz, and C. Krömke. Model Based System Development in Automotive. In *Proceedings of the SAE World Congress*, Detroit, MI, 2003. Society of Automotive Engineers.

[15] Object Management Group OMG, www.uml.org. *Unified Modeling Language: Superstructure. Version 2.0. OMG Adopted Specification ptc/03-08-02*, 2004.

[16] S. Poledna, T. Mocken, J. Scheimann, and T. Beck. Ercos: An operationg system for automotive applications. In *SAE International Congress*, 1995.

[17] T. Scharnhorst, H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, P. Heitkämper, J. Leflour, J.L. Mate, and K. Nishikawa. Autosar – challenges and achievements. In *Elektronik im Kraftfahrzeug 2005*. VDI, October 2005.

[18] B. Schätz et al. Checking and Transforming Models with AutoFOCUS. In *12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*. IEEE Computer Society, 2005.

[19] J. Schäuffele and T. Zurawka. *Automotive Software Engineering*. Vieweg Verlag, Wiesbaden, 2003.

[20] Thurner et al. Das Projekt EAST-EEA – Eine middlewarebasierte Softwarearchitektur für vernetzte Kfz-Steuergeräte. In *VDI-Kongress Elektronik im Kraftfahrzeug*, number 1789 in VDI Berichte, Baden-Baden, 2003.

[21] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming - Euromicro Journal*, 40:117–134, 1994.

[22] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication under static-priority or edf schedulers. In *ACM Intl. Conference on Embedded Software (EMSOFT)*, 2005.

[23] M. v. d. Beeck. Comparision of statecharts variants. In *Third Int'l Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 128–148, 1994.

[24] M. von der Beeck, P. Braun, M. Rappl, and C. Schröder. Automotive UML. In B. Selic, G. Martin, and L. Lavagno, editors, *UML for Real: Design of Embedded Real-Time Systems*, number ISBN 1-4020-7501-4. Kluwer Academic Publishers, 2003.