INVITED PAPER

Engineering Automotive Software

For the gigabyte of costly software that will be used in intelligent cars of the future, new system and software development techniques and tools are required.

By Manfred Broy, Ingolf H. Krüger, Alexander Pretschner, and Christian Salzmann

ABSTRACT | The amount of software in cars grows exponentially. Driving forces of this development are the availability of cheaper and more powerful hardware, as well as the demand for innovation through new functionality. The rapidly growing significance of software and software-based functionality is at the root of various challenges in the automotive industries. These concern their organization, definition of key competencies, processes, methods, tools, models, product structures, division of labor, logistics, maintenance, and long-term strategies. This paper pinpoints the idiosyncrasies of the domain, characterizes the essentials of automotive software, and discusses the challenges of automotive software engineering.

KEYWORDS | Road vehicle electronics; software engineering; systems integration

I. INTRODUCTION

Just 30 years ago, the automotive industry witnessed the first deployment of tiny little bits of software in cars. It was used to control the engine and, in particular, the ignition. The first software-based solutions were strictly local, functionally and technically isolated, and did not relate to one another. These independent and unconnected pieces of software used to run on single dedicated controllers Electronic Control Units (ECUs). A typical node ran a few kilobytes of software, and there were roughly a dozen nodes. Only a minimum of abstraction was applied, and the focus was on minimum resource consumption. Applications were designed directly in machine code or the programming language C.

The basic architecture in cars was hence formed by ECUs for dedicated tasks together with their sensors and actuators. In order to optimize electrical wiring, bus systems were conceived and implemented. In the beginning, they mainly connected ECUs to sensors and actuators.

Given this infrastructure, it was not long before ECUs themselves were interconnected as well and were thus able to exchange data (e.g., [13]). As a result, the car industry started to introduce functions distributed over several ECUs, connected by bus systems as the underlying communications infrastructure. Somewhat obviously, such functions were built bottom-up, adding more and more ECUs to the appropriate bus, as the need arose. In contrast, communications architectures [58]–[60], [72] found in today's cars are very sophisticated, integrating multiple different bus technologies for the various safety and comfort systems [6], [42].

Within only 30 years, the amount of software has evolved from zero to tens of millions of lines of code. A current premium car, for instance, implements about 270 functions a user interacts with, deployed over about 70 embedded platforms. Altogether, the software amounts to about 100 MB of binary code [35]. The next generation of upper class vehicles, hitting the market in about five years, is expected to run up to 1 GB of software. This is comparable to what a typical desktop workstation runs today. Today, more than 80% of the innovations in a car come from computer systems; software has thus become a major contributor to the value of contemporary cars—but software has also become an increasing cost factor [73].

One reason for this trend simply is that software enables the implementation of functionality deemed impossible just 20 years ago. Indeed, the concept of "intelligent vehicle" is being promoted to assist the driver perform a wealth of activities [62], including traffic monitoring [27],

Manuscript received May 15, 2006; revised September 19, 2006. This work was supported in part by the University of California Discovery Grant, in part by the Industry-University Cooperative Research Program, and in part by the California Institute for Telecommunications and Information Technology (Calit2).

M. Broy is with Software and Systems Engineering, Institut fur Informatik, Technische Universität München, D-85748 Garching, Germany (e-mail: broy@in.tum.de).
I. H. Krüger is with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093-0404 USA (e-mail: ikrueger@cs.ucsd.edu).
A. Pretschner is with Information Security, Department Informatik, ETH Zürich, CH-8092 Zürich, Switzerland (e-mail: pretscha@inf.ethz.ch).
C. Salzmann is with BMW AG, 80788 Munich, Germany (e-mail: Christian.salzmann@bmw.de).

Digital Object Identifier: 10.1109/JPROC.2007.888386

and parking assistance [99]. Addressing driver-distraction [17], [57] and fatigue [40] have become important areas of development and use of software-based systems.

Another reason is that electronics in cars help reduce gas consumption and increase performance, comfort and safety, as indicated by today's numbers of increasing traffic with decreasingly many serious accidents [37]. Information processing technology cuts across all aspects of the car [64] and is a persuasive, sophisticated, and differentiating value addition to the product. Furthermore, software enables the car manufacturers (which we will refer to as "OEMs" in the following) and suppliers to tailor systems to particular customers' needs. In other words, software can help differentiate between cars. At least in principle, it is the software that also allows hardware to be reused across different cars. Other than hardware, software has an almost negligible replication cost, which is a further incentive to bet on software as a potential tool in cost-reduction, while on the other hand, the development costs of software are increasing dramatically.

The worldwide value creation in automotive electrics/electronics, including software, amounts to an estimated 127 billion Euros in 2002 and an expected 316 billion Euros in 2015 [30]. Software makes up an estimated 38% of this value creation by 2010 ([50], referencing a 2001 Mercer study): software engineering for automotive systems obviously is of utmost relevance. This is also exemplified by Hansen's observation that embedded software has been an important automotive electronics topic since the early 1990s—yet, tools supporting the automotive software development process are exploited only to 10% of what they could deliver [47]. The engineering of systems as complex as found in today's cars in a reliable and efficient way is a very challenging task [53], [54], among others, because the requirements profile for this system class is diverse [46], [104]; it integrates demanding economic and process challenges with hard technical challenges.

In this paper, we characterize a specific software engineering discipline that takes into account the idiosyncrasies of the automotive domain: the market, the interplay of OEMs and suppliers, the heterogeneity of the software involved, namely, infotainment as well as embedded software, and the multidisciplinary nature of the field. The purpose of this characterization is the identification of the most pressing challenges, both in research and practice of software engineering. As an update to and consolidation of our earlier work [19], [23], [69], [90], this paper presents the combined understanding of the problem space we have gained in various collaborations in the automotive domain.

The remainder of this paper is organized as follows. Salient nontechnical features of the automotive (software) domain are described in Section II. We present the most important characteristics that have direct impact on software engineering activities. These include division of labor, long life in spite of short innovation cycles, unit-based cost models, the market's demand for many different variants, obstacles to reuse, and a brief discussion of the activities in the software development and maintenance processes. In Section III, a more technology-centered perspective is taken. We describe different types of automotive software, discuss specific requirements on reliability, safety, and security, and explain the complexity of technical infrastructures in cars. The characterization of the domain, as provided by Sections II and III, then leads us to a description of trends, challenges and prospects in Section IV. Our summary in Section V highlights systems of systems integration as the key challenge and future innovation area.

II. DOMAIN PROFILE: ORGANIZATION AND ECONOMICS

A. Organization of the Development Process

Traditionally, the car industry has been organized in a highly vertical manner (or, to use software engineering terminology, modularly). Mechanical engineers worked hard for over 100 years to render the various subsystems in cars independent. This facilitated independent development and production of the parts and gave rise to a highly successful division of labor: today, an estimated 25% of the value is created by the OEM who tends to concentrate on the engine, integration, and marketing of the brand.

Enabled by the car design's modularity, suppliers have always taken care of a considerable part of the engineering task, the development, and also the production. Suppliers can use synergies in development and production, because they usually produce similar systems for different OEMs. This, in turn, also keeps the unit cost low for the OEMs. For functions that do not differentiate between the different OEMs this synergy is greatly exploited at the suppliers' side. As a negative side effect, development is geographically distributed and communication gets more complicated. The large number of parties involved in itself is a reason for unstable requirements. One result are frequent changes and revisions of the requirements during the development process.

In the past, the "ideal" of automotive development was that the parts of cars are produced by a chain of suppliers and more or less only assembled by the OEM. Thus, a large proportion of the engineering and production activities are outsourced. Cost and risk distribution can be optimized.

A car is (or better was) considered a kit of parts that are merely assembled by the OEM. With software becoming a major force of innovation, the OEM's responsibilities have evolved from the assembly of parts to system integration. Traditionally unrelated and independent functions (such as braking, steering, or controlling the engine) that were freely controlled by the driver suddenly related to one another, and started to interact. A telling example for the increased interaction among previously unrelated subsystems is the central locking system (CLS) as found in most modern vehicles: it integrates the pure functionality of locking and unlocking car doors with comfort functions (such as adjusting seats, mirrors, and radio tuners according to the specific key used during unlocking), with safety/security functions (such as locking the car beyond a minimum speed, arming a security device when the car is locked, and unlocking the car in case of a crash), and with human-machine-interface functions, such as signaling the locking and unlocking using the car's interior and exterior lighting system. Many of these functions are realized in subsystems that are still distributed according to the major mechanical breakdown of the vehicle into engine, drivetrain, body, and comfort systems. The car has thus turned from an assembled device into an integrated system. Phenomena-like unintentional feature interaction have become issues. Feature interaction [15], [110] is the technical term created in the telecommunication domain for intentional or unintentional dependencies between individual features, which refer to distinct functions. Feature interactions are visible at the user level as specific dependencies between distinct functions.

Another issue is the architecture view onto embedded software systems where we have to understand how the separated software components that are deployed on different ECUs interact to provide the distinct user functions. Architecture verification aims at the correctness of the designed architecture. Bringing together the implemented software components and making sure that they interact correctly according to the designed architecture is called integration.

While the integration of subsystems is always a challenging task for a complex system, the situation in automotive software engineering is even worse, because suppliers usually have a lot of freedom in how they realize individual solutions. (In business IT the client would often strongly constrain the technologies that may be used by a supplier to facilitate integration and maintenance. The situation is, however, similar to the relationship between vendors of desktop operating systems and manufactures of devices and device drivers. What is different here is that the desktop operating systems market is close to being a monopoly.) Furthermore, the OEM usually only has black-box specifications of the subsystems to be integrated, and it is difficult or impossible for the OEM to modify parts of the subsystems. This, of course, complicates the localization of errors. While the value of systematic software development processes and process models—such as the capability maturity model integration (CMMI) [1], [93] and the software process improvement determination (SPICE) [34] is increasingly recognized by players in the automotive industry [48], major challenges remain at the interface between OEMs and suppliers. Critical issues here are the quality and precision of requirements documents, as well as the (limited) transfer of IP among OEMs and suppliers [102].

B. Life and Innovation Cycles

A car model is usually produced for about seven to eight years. The customer expectation of a long lifetime is reflected in the OEM's duty to offer service and spare parts over at least 15 years after the purchase of a vehicle (compare this to an estimated lifecycle of, say, four years for an average workstation program, with several hot fixes during this period). The life cycle of the hardware components, like CPUs, however, is much smaller, say less than five years. Software may be changed at much shorter intervals, typically several times a year. In particular, the comparatively short CPU life cycles enforce changes in a vehicle's software/hardware system during the production period and maybe also during the development phase.

A further reason for changes is the short innovation cycle in the automotive domain. Already, during the production time of a vehicle new features become available in other models and have to be ported to vehicles already in production. The most obvious example originates in the infotainment domain. Here, consumer electronics trends like mp3 players or new mobile phones have to be integrated into the vehicle's infotainment network soon after entering the market.

Because code is today mostly written and directly optimized for specific individual processors (essentially motivated by the unit-based cost structure, see Section II-C), it is difficult to port the code to another processor. Thus, keeping pace with processor life cycles is hindered. The integration of new functionality is made more difficult or even impossible if memory size of the ECUs was optimized too much during the development process. Long vehicle life cycles and the huge number of vehicle variants require efficient handling of compatibility, which conflicts with the desire to optimize the unit-based cost.

C. Cost Model

The automotive industry operates in a highly competitive mass market with strong cost pressure. Here, the rules of business of scale—how many units of a product are sold—prove to be crucial. Depending on the market segments targeted by an OEM, competition occurs over product price, product quality, product image, and differentiating product features. Competition over price requires permanent optimization. Competition by differentiation requires innovation and a strong brand profile.

Traditionally, for all decisions to realize car functions, the cost per unit produced plays a decisive role. A consequence of the large quantities produced, production and material cost by far outweighed engineering cost for classical, not software-centric vehicle parts. The classical argument is as follows. A vehicle component may be produced over seven years or more with, for instance, 500 000 units per year. A hardware cost reduction of Euro

1 for 20 such components (including one processor each) in each car would then lead to an overall cost reduction of Euros 70 million over the production period.

For vehicle software, this argument continues to be used as a motivation to keep the cost per unit low. As a consequence, engineers concentrate on reducing the amount of required memory and computation power. There are two consequences of this approach to optimize up to the limit. First, such optimization requires that the software be very closely tuned towards the processors' characteristics. Second, trying to squeeze the code into as little memory as possible requires a further set of code optimizations. The negative results are as follows. First, it is very difficult to add any functionality to the system later on. Second, it is very difficult to change parts of the code or to fix defects. Third, the code is more complex (for instance, in terms of strong coupling between modules) than necessary. Fourth, some defects in the code may be a result of the optimization itself and finding bugs may become even more difficult, since now application logic issues are obscured by optimization. Fifth, making required changes to the code becomes very difficult. Sixth, reusing this code in future car models or on other processors is almost impossible.

In sum, exclusively thinking in terms of unit-based costs with the associated need for optimizations makes the software complex and difficult to handle. Thus, premature optimization has a negative effect on many classical quality attributes for software. Time-to-market, maintenance costs, and the risk of not finishing a development project in time are substantially increased.

We do not argue that unit-based costs are unimportant for software-based functions, simply because they are not, as the above numbers show. However, they are but one factor. For the future, the automotive industry needs decision and cost models that also take into account rising development costs, maintenance cost, project risk, and time-to-market. It is likely that such models require a more transparent cooperation between suppliers and OEMs (cf. the collaborative development process in Section II-F).

D. Variants

The need for differentiation in the mass market motivates the desire for customized items. The authors of [25] calculate for a simplified power train control application 3488 possible component realizations by instantiating different algorithms and their variants. A premium car typically has about 80 electronic fittings that can be ordered depending on the country, etc. Simple yes/no decisions for each function yield a possible maximum of roughly 2^{80} variants to be ordered and produced for a car. Besides variants due to market demand, a second source of variants is the long product life cycle. Several changes in the software/hardware system are usually made over the time a vehicle series is produced. This means that there are various versions for each piece of software in a car. When defective ECUs have to be replaced or when a software update is performed as part of vehicle maintenance, configurations containing a mixture of "old" and "new" software can be created.

Thus, we have a huge amount of variants and configurations that must be handled technically and organizationally. Technically, desirable configurations must be identified and their correctness has to be established. Obviously, an elaborate design and test methodology is required for this. Organizationally and economically, it is indispensable to reuse large parts of the software and even produce the software in a way that it is future-proof for coming new variations and compatible with old configurations.

E. Reuse

Typically, functionality changes only to a small amount from one vehicle generation to the next. Most of the old functionality remains and can be found in the new car generation, as it was in the old one. From one car generation to the next, functionality differs mostly not more than 10%, while much more than 10% of the software is rewritten. Nevertheless, today the process of software reuse is not systematically planned between OEMs and suppliers, as required, say, for software product lines [103]. In addition, the reuse objectives of OEMs and suppliers may be in conflict with each other.

Furthermore, too strong an optimization of the software towards the hardware can make reuse in the form of porting it to new hardware impossible or very expensive. This is an enormous loss of investment. The development cost of the electrical parts in cars today amounts to 300 million Euros and more. Roughly one or two thirds of that is software. Therefore, if we managed to keep and reuse only 50% of the software in the next car generation, this would save up to 100 million Euros in development (compare this to the abovementioned 70 million Euros of potential savings as a result of using cheap hardware). This may not be the decisive argument for vehicle series that are produced in huge quantities. However, especially for optional equipment that only has a low take rate and, therefore, much lower production numbers, we enter a region where the software development cost per unit produced is higher than the amount needed for more powerful CPUs with more memory such that more of the software can be reused.

With the increasing importance of software, it is only a question of time until it becomes more economical to use more generous hardware structures and to stay away from low-level code optimization. In the end, a lot of the code could be generated from high-level models, which can be reused in product line approaches (see [2], [69], and [86]). In fact, a considerable amount of collaboration projects between academia and industry is exploring opportunities for auto code generation [29], including [97].

F. Consequences for Development and Maintenance

It is obvious that there is a need for a suitable development process that reduces complexity, enables innovation, saves cost, is transparent, and addresses outsourcing. We will now gloss over the main activities of the software development process and show how they are impacted by the characteristics of the domain, as explained in the previous sections.

1) Requirements Engineering: One of the prevalent problems in automotive software engineering is a tailored requirements engineering process [46]. That this is essential is quite evident because in addition to the many functions that carry over from one car model to another, there are many completely new and innovative functions in newly developed cars. When introducing new functions, of course, we have no experience with them at first. What is the best way to work out the detailed functionality, what are the best man-machine dialogs to access the functions, what are the best reactions of the systems? The massive use of software results in a much larger design and solution space when compared to earlier cars with less software-enabled integration between subsystems. Several challenges to expressing, managing and reusing requirements in concrete automotive projects have been identified [104]. Therefore, requirements engineering is one of the crucial issues [39], [44], [79] discusses a weighted requirements framework for automotive software—albeit with heavy vehicles (such as for construction, forestry, or combat) in mind; more work in the direction of gaining a deep understanding of the automotive systems engineering requirements space is needed.

In fact, some of the requirements engineering has to be done inside the OEMs, to capture or establish, say, intellectual property at the function, systems and integration level [102]; a further incentive for the OEM to engage in requirements engineering is also to support systems integration tasks, such as validation, verification and adaptation. Supplementary requirements engineering has to be performed by the suppliers, which usually implement the functionality. Communication between OEMs and suppliers has to be organized via the requirements documents, which nowadays tend to be neither sufficiently precise nor complete. This gives rise to the issue of distributed concurrent engineering. The more complex systems become, the more important it is to use good product models that support integrity of the information exchange between the different parties. Because many automotive software functions have strict quality and quality-of-service requirements, this integrity is essential.

A requirements study [49] conducted in the domain of heavy vehicles (trucks, say) indicates another area of concern: in this domain, much of the ECU software is written assuming it is safety-critical even though it is not. Because of its limited scope this study does not generalize to the automotive domain in its entirety; however, the diversity in requirements among, for instance, power train (extremely short cycle times, high throughput networks) and body functions (typically longer cycle times, reduced throughput demands) needs to be properly reflected in a comprehensive automotive domain model *and* corresponding deployment infrastructures.

Finally, getting hold of the huge variety of versions and variants, as mentioned in Section II-D, directly impacts requirements engineering activities.

2) Platform and Software and Hardware Architecture Design: Designing the architecture of an automotive IT system requires determination of the *hardware architecture* that consists of ECUs, bus systems, and communication devices, of sensors, actuators, and of the (hardware-related) man-machine interface, or MMI. The *software infrastructure* is then based on this hardware structure. It includes the operating system, bus drivers, and additional services. This infrastructure software, together with the hardware, forms the *implementation platform*. These architectures obviously need to be described. Architecture description languages (ADLs) [78] have long been used to describe design and deployment architectures of software systems. An example of a study describing the application of an ADL to the design of an automotive vehicle appears in [75], based on the EAST ADL [41]. While this points into the right direction, important automotive topics, such as failure management are left unaddressed by existing ADLs.

Finally, *application software* (i.e., application code—the structure of all applications forms the *application software architecture*) is based and executed on the implementation platform. This demonstrates the significance of the platform for many typical software engineering goals such as separation of concerns, portability, reusability, etc. In today's cars, synergies and opportunities for reuse are realized mainly at the level of bus drivers and hardware-communication infrastructure, as well as by means of operating systems for individual ECUs.

Platform software reuse is becoming common, being explicitly supported by the Automotive Open System Architecture (AUTOSAR) [5]. Application-level reuse is only in its early stages. Examples include the multiple occurrences of voice-recognition implementations across different subsystems with MMI components, including navigation, phone, air conditioning, and entertainment subsystems. There is a trend to make such common functionality available as a "service" via the supporting infrastructure software [5], which requires a comprehensive software and systems engineering approach that includes identification of common functionality across subsystems [69] and facilitates its implementation and deployment via the implementation platform (see also Section IV).

3) Coding: Coding means the actual production of the program code of application or platform software.

Suppliers carry out most of the coding, today. Only in extraordinary cases the OEM produces code for some of the infrastructure as part of the platform (such as bus gateways or communication backbones) or in innovative applications (such as advanced driver assistance).

A lot of the code is still written by hand, although some tools generate good quality code from models [33], [38]. Code generation is often considered not efficient enough to exploit the ECUs in the optimal way. However, as we have argued above, highly optimized code makes reuse and maintenance difficult and is economically disadvantageous in many circumstances. Example approaches to addressing these problems for code generation from models appear in [9], [38], and [100]. Experience with model and code reviews in that context are documented in [97].

4) *Software and Systems Integration:* Examples such as the CLS (see Section II-A) illustrate the "scattering of functionality" across automotive subsystems: what is offered as a cohesive locking/unlocking functionality to the customer in reality emerges from the interplay of multiple components distributed over the entire vehicle infrastructure. This scattering of functionality presents a major challenge for the OEM in its role as the integrator to ensure the cohesive functionality expected by the customer. Since today, by their very design, architecture and the interaction between subsystems are not precisely specified, and since the suppliers implement the subsystems in a distributed process, it is not surprising that integration is a major challenge.

First, a virtual integration and architecture verification is not possible today, a result of the very lack of precise specifications. Second, in turn, the subsystems delivered by the suppliers do not always fit together smoothly, and integration may fail. Third, due to the missing guidelines for architecture development, there is no guiding blue print to make the design consistent when errors are corrected.

5) *Quality Assurance:* A critical issue is, of course, system quality. For software there is a rich spectrum of quality aspects (see [20]). Examples are reliability, maintainability, reusability, portability, and many more.

The car industry is highly cost-aware. As a result, to save engineering effort, quality issues are not always observed in a way advisable for software systems (see [20]). On the long run, this proves counterproductive for the overall cost structure. The focus on quality and the established certification processes and extensive software and systems redundancy to achieve error tolerance in the avionics industry are, by the way, the reasons for airplanes' reliability outmatching cars' reliability by far.

6) *Maintenance:* A further critical challenge is that cars are in operation over two or three decades. This means we have to organize long-term maintenance. Today, an OEM's vehicle fleet is predominantly maintained by vehicle dealers following prescribed semi-automated procedures. With the increasing amount of software in a vehicle, this vehicle more and more inherits the characteristics of a complex IT system. There is a difference with the desktop software market, however. It is likely that future maintenance of on-board software will continue not to be delegated to the users. Among others, this is a consequence of the OEM's desire to create an overall brand "experience" to which the customer can relate as a "package." Therefore, new software maintenance processes are needed that enable effective maintenance, also without direct vehicle access by the OEM and without software professionals at the dealer's office.

a) *Compatibility:* Updating software in cars in itself is a challenge. Today new versions of software are brought in during maintenance by "flashing" techniques for replacing the software of an ECU. When doing this, one has to be sure that the new versions correctly interoperate with the old version. In other words, we have to answer the question whether the new version is *compatible* [8], [96] with the one we had before. Because of the substantial scattering of functionality in cars today, this is difficult. A lot of the problems we see today in the field are indeed compatibility problems.

b) *Defect Diagnosis and Repair:* An interesting observation states that today more than 50% of the ECUs that are replaced in cars are technically error-free (they are replaced when the customer brings the car to a garage to fix a problem of maintenance). They are replaced simply because the garage could not find better ways to fix the problem. However, often the problem is not rooted in broken hardware but rather ill-designed or incompatible software.

This clearly demonstrates that we need much better adapted processes and logistics to maintain the software of cars. Understanding how we do a further development of the software architecture in cars, understanding the configurations and version management, and making sure that not only extremely well-trained people in garages really can handle the systems, is a major challenge.

c) *Changing Hardware:* Hardware has to be replaced in cars if it is broken. Moreover, over the production time of a car model, which is about seven years, not all the ECUs originally chosen are available in the market over the whole production period. Some of them will no longer be produced and have to be replaced by newer types. Already after the first three years of production, 20%–30% of the ECUs in the car typically have to be replaced by newer ECU models due to discontinuation of an ECU's specific technology. As a result in the current state of affairs, software has to be reimplemented, since it is tightly coupled with the ECU. Thus software quality attributes like portability and reusability become increasingly important for the car industry.

III. DOMAIN PROFILE: TECHNOLOGY

A. Software Application Domains (and Required Skills)

Automotive software is very diverse, ranging from entertainment and office-related software to safety-critical real-time control software. It can be clustered according to the application area and the associated nonfunctional requirements. The following five clusters are usually distinguished.

1) Multimedia, telematics, and MMI software: typically soft real-time software which also has to interface with off-board IT, dominated by discrete-event/data processing.
2) Body/comfort software: typically soft real-time, discrete-event processing dominates over control programs.
3) Software for safety electronics: hard real-time, discrete event-based, strict safety requirements.
4) Power train and chassis control software: hard real-time, control algorithms dominate over discrete-event processing, strict availability requirements.
5) Infrastructure software: soft and hard real-time, event-based software for management of the IT systems in the vehicle, like software for diagnosis and software updates.

As the five vehicle software clusters suggest, automotive software engineering requires skills from various disciplines. Since the software/hardware systems of the five clusters are distributed, consisting of a high number of separated functions and processes that exchange information over several communication links, they show all the details and complexities of distributed computer networks. Hence, computer science and computer engineering skills are required.

On the other hand, the systems are connected to sensors and actuators that input and output physical data in real-time and where the software/hardware systems have to respond to these inputs in a classical control theory manner. As a result, control theory knowledge is required. For some functions, like power train control software, an understanding of mechanical engineering is also important.

In any case this means that knowledge from different engineering disciplines is required for automotive software engineering.

Historically, the methods and approaches as well as the models of control theory are quite different from the models of information processing. In control theory, traditionally tools like MATLAB/Simulink [4], [105] are used to model the control theory equations. In contrast, in business information processing engineers are increasingly eager to use UML-like [83] models and all kinds of other discrete data flow or state machine models. In cars, these separate engineering cultures have to be united to reflect all relevant aspects of the different engineering domains. What is needed in the end is a comprehensive model of the car where some of the issues are mapped into control theory and others are represented by discrete models of data processing. On a limited scale, these problems have been addressed in the past, notably in the synchronous programming community. The Esterel language [10], [14], for instance, separates control from data processing, and is widely being applied to industrial designs in airplanes, trains, and cars [56]. Esterel has a formal semantics [11] and an extensive set of tools [36] that is certified to preserve correctness of programs, which is particularly important in avionics. As more government mandates and standards arise in the automotive domain, similar certifications will become necessary here as well. LUSTRE, another synchronous language [26], [51], is based on the same principles as Esterel. LUSTRE has a suite of tools to convert high-level specifications into Simulink and to make them amenable to the formal analysis and code generation techniques developed for synchronous languages [26].

B. Reliability, Safety, and Security

It does not come as a surprise that reliability and safety concerns are important for all functions relevant to driving, from engine control and passenger safety functions to forthcoming X-by-wire [107], [109] functions where mechanical transmission is replaced by electrical signals, e.g., drive-by-wire (steering signals are electronically transmitted to the wheels) or brake-by-wire (braking signals are electronically transmitted to the brakes). However, with the development of infotainment functionality, the car is becoming an information hub where functions of cell phones (UMTS [65], Bluetooth [12]), Laptops (Wireless LAN [28], [31]), and PDAs are interconnected via and with the car information systems. Increasingly, the on-board electronics systems together with the customer's mobile phone establish communication links beyond car boundaries; this results in the car transitioning from a pure information hub to an information/communication hub. This transition brings with it all the potentials and challenges of both local and wide-area networking combined with specific automotive applications, including remote diagnostics, access, global positioning, and emergency services as they are offered in vehicles already today. Therefore, personalization and the related privacy and security issues are becoming most important, notwithstanding usability issues. These topics are still under research even in their more traditional home grounds of Internet-enabled business information systems; their inherently crosscutting nature make them particularly challenging to address in automotive architectures with a high degree of scattered functionality.

A goal of a software engineering discipline for automotive systems must be to differentiate between the various software domains in the car (i.e. infotainment, driving functionality, etc.) and offer the proper reliability, safety, and security techniques, both for the software infrastructure and the development process.

C. Hardware and Technical Infrastructure

Today's cars exhibit a very complex technical infrastructure. We find, for instance, five bus systems and more as communication platform for ECUs. We find real time operating systems, and a lot of system-specific technical infrastructure on which the applications are based. Because of the resulting scattering of functionality, relatively simple applications get highly complex: they have to be distributed, and they have to communicate over a complex infrastructure.

One of the problems of this infrastructure is the significant amount of multiplexing on the bus level to efficiently support communication among the dozens of ECUs for thousands of software-enabled tasks in parallel. The same holds for the individual ECUs where there are tasks and schedulers to manage (virtual) parallelism. We can thus find all the problems of distributed systems, in a situation where physical and technical processes have to be controlled and coordinated by the software, some of them being highly critical and hard real-time.

What creates the crucial problems due to the multiplexing on the bus-level? Among other things, the transmission time of messages exhibit jitter so that systems appear to be nondeterministic. In many cases, timing deadlines cannot be guaranteed. On one hand the reliability of the bus systems today is not good enough, on the other hand, due to the unpredictable load time guarantees cannot be provided. Therefore, a lot of interesting potentials for improvement looking at drive-by-wire systems are not realized so far. Time-synchronous bus systems like TT-CAN [60], [67], [74], or FlexRay [80] are current attempts at solving these problems. In the long run, a combination of deployment technologies, supporting both loose and tight coupling with respect to time and messages/data, is necessary to reflect the coordination and quality requirements of all automotive application clusters identified above [42].

IV. CURRENT TRENDS AND PROSPECT

The increase of software and functionality in cars is not close to an end. Just to the contrary, we expect a substantial growth in the future, and see the future development to be driven by the following trends:

1) high demand for new innovative or improved functionality;
2) quickly changing platforms and system infrastructures;
3) rapid increase in development cost in spite of a heavy cost pressure;
4) demand for higher quality and reliability;
5) shorter time-to-market;
6) increased individualization.

These trends are likely to impact the fields as explained in the following paragraphs.

A. Likely Future Functionality

Software will remain the innovation driver in cars for the next two decades, leading to many new software-based functions in cars in the future. Each new software-based function entering the vehicle enables several further features. This accelerates the development.

1) Crash Prevention, Crash Safety: Already today the safety standards in cars are very high. Statistics [37] show that in spite of increased traffic, the numbers of fatalities, injuries, and accidents have continually decreased, and there is some general agreement that this is at least in part due to software-based functions in cars. Nevertheless, there is potential for future improvement. New generations of crash prevention, pre-crash, and crash mitigating functions are in preparation.

2) Advanced Energy Management: Hybrid cars are only in their infancy. In future cars, we can expect a technical infrastructure that takes care of many in-car issues like energy consumption, car calibration, and management of the available electrical energy.

3) Advanced Driver Assistance: The complexity of software systems in cars is perceived to be rather high for their drivers, passengers, but also for maintenance. Various driver assistance functions at all levels can help, supporting instantaneous driver reactions but also providing short-term driving assistance in, for instance, lane departure or tour planning. With increased technical potential for inter-car and car-to-infrastructure communication, opportunities for distance "look-ahead" and coordinated driving through congested or dangerous intersections will be realized. This will also have an impact on the MMI (see below) to alert the driver of detected dangers, and to offer means for addressing these dangers in the full spectrum from manual to completely automatic intervention.

New pixel-light approaches are based on the principle of a video-projector that is controlled by a complex software system. Such headlights can be used to realize revolutionary new driving assistance systems, where for example people or animals are detected and the according region is high beamed where at the same time the rest of the lighted area is low beamed. It is also possible with this approach to include navigation signs (arrows, etc.) into the light beam.

4) Man-Machine-Interfaces (MMIs): Cars get more complex also due to software—but they also get more safe and convenient due to software. In order to get easy access to this convenience, we have to offer those functions to drivers and passengers such that they do not have to experience and operate all this complexity explicitly. Adaptive context-aware assistance systems that grasp the situation and are able to react within a wide range without too much explicit interaction by the driver or the passengers can lead to a new quality of MMIs.

One problem results from the sheer number of functions. To reduce the number of buttons and instruments, new concepts are needed. Today's MMI concepts are very much influenced by the interaction devices of computer systems like mice or touch pads we are used to today in controlling complex computer systems. The challenge here is not so much, how to avoid buttons, but how to organize the huge number of functions in a way easy to understand for the user.

But quite obviously, these are merely first steps. Multifunctionality of cars needs flexible ways of addressing and operating and interacting with all those functions. What makes the situation more difficult than in classical computers is, of course, that car drivers cannot pay as much attention to secondary tasks as computer users can. Because they must concentrate on the traffic and driving, drivers should be confronted with a user interface which allows them to deal with the many functions in a car in a way that takes not too much of their attention compared to attention given to the traffic-in other words, road-safety concerns demand reduced driver distraction.

5) The Programmable Car: Equipped with various actuators and sensors, as premium cars are today, we get already close to a point where we can create new functions for cars by merely introducing new software. Examples range from simple functions such as allowing remote control of the vehicle via cellular networks (as it is used already today for remote unlocking or vehicle location tracking) to more complex ones such as downloading updates to car software to change the "feel" of a car by means of modified suspension and other car characteristics. This comes close to the vision of the programmable car.

6) Personalization and Individualization: A promising issue is personalization and individualization of cars. Drivers are quite different. When cars get more and more complex, of course, it is crucial to adapt the ways cars have to be operated to the individual demands and expectations of the users. An example here is adaptation to individual braking habits to ensure timely reaction before an accident occurs.

7) Interconnected Car Networking: Another notable line of innovation is the networking of on-board and off-board systems. Using wireless connections, in particular, peer-to-peer solutions, we can connect cars, which gives many possibilities to improve traffic safety or to find new solutions in the coordination of traffic far beyond the classical road signs of today. For instance, in the long-term future when we can imagine that all road signs are complemented by digital signals between cars, we can have a completely different way of coordinating traffic.

8) X-by-Wire: While it is true that potential X-by-wire [107], [109] applications were marketed more aggressively some time ago, this does not mean that the arguments for this technology—reduction of massive material and hence weight; increase in flexibility—have turned wrong. Instead, the challenges behind this technology have turned out to be greater than expected, and we expect to see a renaissance when a better software engineering discipline materializes.

B. Integrated Comprehensive Data Models

Currently, in cars a highly distributed, rather uncoordinated data management takes place. Each of the ECUs contains and manages its own data. A substantial amount of this data is exchanged via the various bus-systems present in the vehicle. Overall, however, the available vehicle-data is not organized as a distributed database with some means to achieve data integrity, for instance. Instead, all the different ECUs and functions keep a significant portion of their data separately. This can lead to the schizophrenic situation where some ECUs, according to their local data, define the car to be moving, while others define the car to have stopped.

We consider it an interesting exercise to design the architecture of a car in a way that there is an integrated interfunction data model that includes sensor fusion and overall car data management. With the increasing push towards sensor networks and corresponding data models and infrastructures *outside* the vehicle today it is very likely that a similar approach will find its way into automotive systems development.

C. Architecture

In premium cars, we find up to 2000 software-based functions (the 270 *user* functions mentioned above are a result of combining these 2000 *software* functions, which are not all directly user interaction functions). Those functions address many different issues including classical driving tasks but also other features in comfort and infotainment and many more. Most remarkably, these functions do not stand alone, but exhibit a high dependency on each other; we have demonstrated this scattering of functionality using the CLS in Section II-A. In fact, many functions are very sensitive with respect to other functions operated at the same time.

So far, the understanding of these feature interactions between the different functions in the car is still in its infancy. We hope to develop much better models to understand how to describe a structured view on multifunctional systems like those found in cars.

Due the multifunctionality and all the related issues we need a sophisticated structural view onto the architecture in cars that addresses all the aspects that are relevant. In such an architecture (see [106]), we distinguish ingredients that we briefly explain in the following. A modeling approach has to be expressive enough to deal with all the

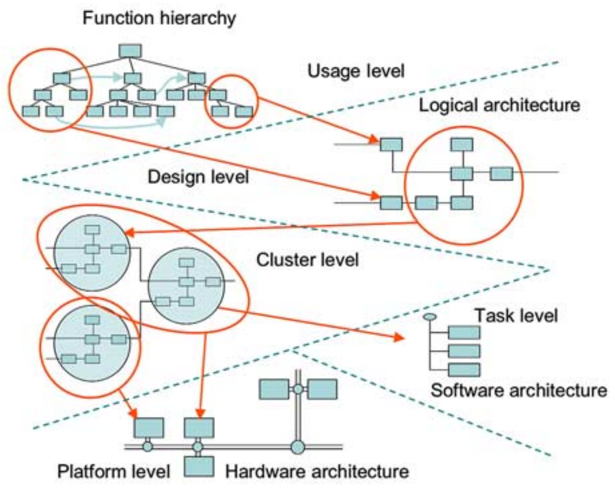

Fig. 1. ***Different levels of abstraction in architecture [19].***

mentioned aspects of architecture. The different levels are shown in Fig. 1.

1) Functionality Level-Users View: The usage view aims at capturing all the software-based functionality offered by the car to the users. Users include not only drivers and passengers but also garage and maintenance staff, perhaps even the people in production and many more. We call this the user or functionality level (see [89]).

The functionality level provides a specific perspective on the car that captures its family of services and aims at understanding how the services are offered and how they depend on and interfere with each other. This can be modeled by so-called feature or function hierarchies. If data- or message-flow among services can already be identified, techniques such as Message Sequence Charts [61] can be used to capture services and their interactions [2].

2) Design Level-Logical Architecture: The design level addresses the logical component architecture. In a logical architecture, the functional hierarchy is decomposed into a distributed system of interacting components.

At the design level, we describe the distributed architecture of a system, independent of whether the components are implemented by hardware or software, and also independent of how many different components implement the function. The logical architecture can be described by a number of interfaces for communicating state machines with input and output that realize the functions in the system. Via their interaction, they realize the observable behavior described at the functionality level. In fact, these interactions can be used to *define* the system's decomposition into services [2], [22]. The logical architecture describes abstract solutions and to some extent the protocols and abstract algorithms used in these solutions. This promotes *conceptual reuse* for services that differ only in the way they are deployed from one vehicle line to another; it also opens the potential for identifying common services needed across the vehicle infrastructure [69].

3) Cluster Level: In the clustering level, we rearrange the logical architecture in a way that prepares the deployment and the step towards the software architecture. This means that we further decompose software components until a sufficiently fine-grained granularity is reached and then rearrange them into clusters. The clusters form the units of deployment.

4) Software Architecture: The software architecture consists of the classical division of software in platforms like operating systems and bus drivers on one side and the application software represented by tasks, which are scheduled by the operating system, on the other side. This software has to be deployed onto the hardware.

The high-level software architecture is derived quite straightforwardly from the logical architecture. In some sense, it is a representation of the logical architecture by programs. How easy and directly this can be done and how difficult the encoding is depends very much on the programming language used. If appropriate tools are available, the code can be generated to a large extend from the logical architecture.

Another viewpoint is the target code in terms of the tasks and processes, as well as the scheduling. This is derived from the high-level software architecture and closely related to the software infrastructure such as the operating system and, in particular, the scheduler [2], [88].

5) Hardware Level-Hardware Architecture: The hardware architecture consists of all the devices including sensors, actuators, bus systems, communication lines, ECUs, MMI, and many more. We can separate between the macroscopic view of the hardware architecture, being the network of busses (mostly, CAN, MOST, Flexray, and LIN), the ECUs, the gateways that route and adapt the signals between several domains and busses, and the sensors and actors that are connected within the network. Within one ECU, we can define the microscopic hardware architecture that exists of the processor, the hardware I/O is the memory layout, etc.

Note that the specific automotive requirements on hardware must be specified and satisfied in the hardware architecture, such as electromagnetic compatibility, temperature tolerance, or the packaging (physical construction space) of the device.

6) Deployment-Software/Hardware Codesign: Finally, we need a deployment function that relates hardware to software. The hardware/software and the deployment function together have to represent a concrete realization of the logical architecture that just describes the interaction between the logical components. Many studies and experiments deal with design and analysis from a system-level perspective [66]. The design of automotive systems has became a driver for electronic design automation (EDA) and co-design research as the boundary between hardware and software and the distributed nature of computing in a car becomes very similar to that found in chip design [45], [112]. Furthermore, there have been numerous efforts in building frameworks to link control theory with implementation languages for both hardware and software [32]. Any deployment architecture has to take into account the network topologies and technologies as different functionalities have different communication requirements [6].

D. Model-Based Development

In contrast to classical business IT, the automotive domain has a history of model-based approaches: in order to integrate one software unit into the car, a supplier must design, integrate, and test against the units of another supplier. Since the code inside the units (e.g., ECUs) is the intellectual property of the suppliers, the other suppliers (or the OEM) often will not get the code of the others' units. As a consequence both have to build up some kind of "black box model" which they code/integrate/test against. This black box model of an ECU usually exists as a description of the ECU's messages on the bus and a semiformal description of the behavior of the ECU, namely, when and in which context the messages are sent.

This is the major reason for the success and the popularity of model-based approaches in today's automotive software domain. To integrate more complex systems that go beyond today's applications, we will need more detailed specifications to augment the black box models and give us more information on internal states and behavior to be able to guarantee compatibility and reuse of units. We, therefore, expect the architecture of the systems, including detailed models of black-box and white-box behaviors, to become increasingly important.

The high degree of interaction between OEMs and suppliers makes the need for clear interfaces and specifications evident. Models that take into account the static and dynamic aspects of the subsystems appear as an attractive way to specify the subsystems' architecture, syntactic interfaces, and behavior [7], [102]. We work with a chain of models for the classical activities.

1) Requirements Modeling.
2) Design Modeling.
3) Implementation Modeling.
4) Modeling Test Cases.

An attractive vision is an integrated modeling approach that captures the relationship between all the models, and where parts of some models are generated from the models used in previous activities.

Today models and model-based development are applied only at particular stages of the development process. So a lot of its benefits get lost that could be exploited if we had integrated model chains. Since the models are only semiformal and the modeling languages are not formalized, a deeper benefit is not achieved. Typical examples are consistency checking or the generation of tests from models [21], [87]. Another issue is that models could help very much in the communication between the different companies such as OEMs and first and second tier suppliers; also here, models are used only in a limited way so far.

Pragmatic approaches such as UML (see [90]) and other modeling approaches used in industry are not good enough. Such approaches are not based on a proper theory and sufficient formalization. As a result, tool support is ineffective, possibilities to do analysis with models are weak, and the precision of the modeling is insufficient.

Unfortunately, what we can find in modeling today does not directly improve the quality of the development processes and of the software intensive products. This is because it does not give precise and uniform views on systems.

There is a need for dedicated modeling languages for both the structural and the behavioral (functional) aspects of a system, where the behavior can be of a discrete, continuous, or mixed discrete-continuous nature. These modeling languages should obviously be sufficiently domain-specific to cater for recurring patterns in the description of the related systems. For instance, graphical notations can be linked to formal models to perform schedulability analysis [63], [95], relying mostly on the abstract semantics. Needless to say, this applies to both OEMs and suppliers. For various reasons, there also seems to be some reluctance to using technology and concepts that have been around for quite some time. An attempt at generating code for scheduling and analysis that separates the supplier and integrator, such that corresponding tests can be done on the composite system has been proposed in [55] and more subsequent insights on the fundamental optimization versus composition tradeoff are available in [76].

The domain profile (Sections II and III) as well as the architectural layers we have introduced above indicate the set of structural and behavioral properties that need to be captured in adequate automotive software models. In particular, many properties of relevance are crosscutting in nature: fail-safety is a prominent example.

Of course, the desired degree of completeness of these models must be defined. The increasingly popular sequence diagrams, for instance, are intuitive as a representation for *partial* interaction patterns. Models in the form of state machines, for instance, can capture complete behaviors succinctly; however, because of their higher complexity, are harder to build, understand, verify, and maintain. Of course, with the proper semantic foundation, extended sequence diagrams can be used to describe complete behaviors—and state machines can be used to describe partial behaviors as well. However, there is a tradeoff between the strengths and weaknesses of the respective description techniques, depending on the types of properties they are intended to capture. The "home ground" for sequence diagrams is the depiction of the partial participation of distributed entities in a collaboration; the home ground for state machines is the representation of complete behaviors of individual entities. To support a comprehensive, architecture-centric development process for automotive systems, different viewpoints on the overall architecture should be supported and used. Seamless combinations of structural, state- and interaction-oriented description techniques can help adjust the abstraction level to the desired degree in capturing both per-entity and crosscutting properties. Modeling notations such as the UML are a far cry from achieving this kind of seamless model combinations [18]; in particular, they lack any notion of service/function hierarchy that could support the upper layers of the automotive architecture we have outlined in Section IV-C.

Suitable description techniques are just a small part of the problem; the more difficult problem is related to identifying the kinds of properties that need to be considered and the appropriate level of abstraction that is used when models are built, regardless of the language-this defines the level of automation we can obtain in verification and validation, as well as in code generation [29].

In this vein, research must be performed to find out the cost-effectiveness of using models. With models, code must be built and maintained, but also all costs that relate to code will be multiplied by a constant factor between 1 and 2 because these costs also relate to the models. (Because models are simplifications, we would argue that maintaining a simple artifact is less costly than maintaining a more complex artifact). This clearly involves tool-related issues (see below).

Finally, a union of control theory and data processing models, in particular, discrete event processing, has not been fully achieved yet. It is also not clear which engineering discipline is the critical one for the architecture of the software/hardware systems in cars. It is possible that there are small control loops in the "leaves" of the software/hardware systems of cars, and that these separated control processes are connected at the level of data processing in classical software architectures. However, this is mere conjecture so far. More work is needed to find the right architectural patterns of heterogeneous systems of that kind. A starting point in obtaining these patterns is the collection of a comprehensive automotive *domain model*.

E. Cost Model

To make benefits of modular and reusable software clear, a comprehensive model of the costs of software over the entire life cycle of a product must be available. The status quo of cost models—the basis for any software development and the respective software architecture—focuses on the early stages (e.g., development) of the software. It does not fully include the later phases, such as maintenance, and their relationship to modularity and reuse.

A cost model, which contrasts the efforts in the development phase with the benefits in the maintenance phase and the saving in reuse within other precut lines, clearly does not yet exist for the automotive world. This is somewhat surprising since only by reasoning on the basis of such a model [43], we can make plausible the necessary investments in research and engineering.

F. Processes

A key issue is process-orientation and software development processes. So far, the processes in the car industry are not entirely adapted to the needs of software intensive systems and software engineering.

The development and engineering processes are distributed due to the heavy involvement of first and second tier suppliers. In addition, they are concurrent due to pressing time to market needs.

Consequent process orientation would, in the long run, require a deep understanding of the product models and their relationship. The product data of course need a comprehensive coherent and seamless model chain. Here, we find an exciting dependency between engineering support software and embedded on board software.

Despite the UML's shortcomings overall, some of its notations have proven quite useful, especially for communicating interaction scenarios (and their variants) between the system and the environment, among the OEM and suppliers [52], [85]. Furthermore, work on the formalization of requirement specification graphs aims at establishing increased requirements traceability by providing a visual language to guide the safety and risk analysis process [71]. Further analyses based on these techniques enable the engineers to reason about hostile scenarios such as the interaction of car electronics with intruder attempts, in a way that is modularizable with the rest of the system [3]. These powerful techniques help the car companies to manage crosscutting concerns of product lines [101].

G. Model-Based Middleware

The heterogeneity in the car brings with it a huge amount of complexity. To manage this complexity and to increase the reuse of software in the embedded automotive domain a clear separation of concerns [84] is needed that supports a uniform way of designing the application towards a uniform software platform.

The application logic, for instance, must be independent from the underlying communication infrastructure, all applications should, system-wide, react uniformly to exceptional events (such as physical read/write errors on the bus), etc. Especially this is a challenge in the automotive software domain, since in today's premium cars up to five different bus systems exist, with each bus having its own characteristics and protocols.

Separation of concerns can be reached on several levels, including architecture, design, and implementation (language dependent). Popular techniques, well known from business IT, are middleware layers and the corresponding component orientation.

An automotive middleware, however, must fit other requirements than middleware known from business IT (such as the common request broker architecture, CORBA [82], and web services [92], [108]). Not flexibility at runtime, but flexibility at design time is needed, since the car is mostly a static system. The following issues need to be considered for automotive middleware.

- *Resource optimization*: due to the unit-based cost structure, modularity must not be overly expensive with respect to resource consumption.
- *Adaptability to different domains*: real-time and non-real-time, safety-critical, and non-safety-critical software is integrated within one system.
- *Optimizability to hardware but also transferability from one hardware platform to another*: due to the lifecycle gap and the hardware–software correlation, the software must be transferable from old platforms to new ones, but still optimizable towards the hardware.
- *Extensibility*: again, due to the lifecycle gap, it must be possible to upgrade a system during its lifetime and extend it with new features.

A promising approach is model-based middleware for embedded systems. This includes a component model, a middleware that separates the application from the communication logic, and a model that is independent of the latter implementation language.

In contrast to classical middleware initiatives, such as CORBA or the Java 2 Enterprise Edition (J2EE) [98], model-based middleware differs in being much less dynamic. The goal is not to change system or component structure at runtime, but to have more flexibility for the distribution of software components over different ECUs, while designing the board net (the set of implemented functions and their mapping to hardware, including communication dependencies) of a car. This certainly is a restriction; yet, it enables us to perform an approach that is suitable to fit the lean resource consumption that is required in the embedded environment.

To achieve the goal of little overhead, a code generation framework is used that generates a specific, optimized middleware layer for a specific board network, based on a model-based specification of the board net and its dependencies.

In contrast to other middleware approaches, there is no single instance in the system that handles the communication dynamically (like an object request broker, or ORB, in CORBA), but the middleware layer is generated statically for this special configuration of the system. This leads to lean, highly optimized middleware portion inside every ECU that minimizes the overhead that comes along when using middleware.

The consequence is that middleware can be only as flexible and optimized as allowed by the expressiveness of the underlying model. Therefore, a powerful meta or domain-model is needed that allows us to express all required aspects of the system, including, but not limited to communication variants, timing aspects, safety, and redundancy.

The goal of a uniform, lean middleware layer that manages communication and exception aspects in a system-wide uniform way is only possible by increasing the expressive power of automotive modeling approaches towards formal, model-based system specifications supporting code generation in an optimized and validated way. The abovementioned AUTOSAR [5] partnership, consisting of various OEMs and suppliers, is a promising step towards an open architecture that features such a model-based middleware layer. It provides a basis and an enabler for further aspects such as timing and redundancy aspects that can be included in the metamodel to further increase expressiveness.

H. Reuse

Reuse comes in different forms. At the level of programming or modeling languages, recurring patterns of behavior in a domain can be encapsulated into concise language constructs. In terms of research, this necessitates the analysis of domains where such patterns can be identified, and then the definition of these patterns. The tradeoff between the benefits of general-purpose languages on the one hand and the benefits of domain-specific languages on the other hand has to be evaluated. Domain-specific design patterns must be defined in places where it makes no sense to encode recurring patterns into dedicated language constructs.

Reuse is also facilitated by standardized architectures [5] that allow for coordinated and standardized interfaces. At the level of requirements engineering, there definitely is a need for further research into product lines. Research remains to be done to cleanly relate feature graphs to artifacts of the design activities, as well as to identify means for efficient exploration of architectural variants [66], [68]. The organizational structure of the development process, with its interplay between OEMs and suppliers and the resulting conflicting desires for reuse, must also be taken into account, and could, in a second step, possibly be reshaped (trends are described in a recent study [30]).

Of course, reuse at an even finer granularity must also be supported, and certainly so at the level of code. Along with well-designed libraries, this asks for research in the area of programming languages that (also for embedded systems) naturally support: 1) reuse, as provided by object-oriented languages, by particularly emphasizing run-time performance problems and late binding in terms of safety; 2) efficient exception handling; and 3) worst-case execution time predictability.

Research into reuse must be based on studies of the cost effectiveness, and hence related to research into cost models. It is unclear to date to what extent and where at least *ad hoc* reuse occurs today, and how OEMs and suppliers profit from different forms of reuse [2].

I. Tool Support and Integration

Today we find a huge family of tools in use, but unfortunately these tools are often not integrated. Therefore, there are a number of attempts to create pragmatic tool chains by connecting the tools in a form where the data models of one tool are exported and imported by the next tool. Other efforts aim at product line reuse of functionality and architectures [50].

Unfortunately, this does not help if there is no semantic understanding of how the different tools could use joint concepts [25]. On the other hand, if such a semantic integration is established, model-based tool chains can help in modeling, automatic generation, validation and verification of automotive software and its crosscutting properties [2], [24], [81], [111]. The specification and verification framework described in [91] integrates model-based engineering with model checkers and theorem provers such that properties can be proved about a design. The framework uses control descriptions in MATLAB, Simulink, and Stateflow [4] and translates these artifacts to the appropriate format for formal analysis.

J. Improved Quality and Reliability

1) Number of ECUs: The car of the future will certainly have many fewer ECUs in favor of more centralized multifunctional multipurpose hardware, fewer communication lines, and fewer dedicated sensors and actuators. Having gotten to more than 70 ECUs in a car today, the further development will rather go back to a small number of ECUs by keeping only a few dedicated ECUs for highly critical functions and combining other functions into a small number of ECUs, which then would be rather not special purpose ECUs, but very close to general-purpose processors. Such a radically changed hardware would allow for quite different techniques and methodologies in software engineering. Because of the reduced complexity, this in itself will help with improving quality. However, other steps are likely to be taken, as discussed in the following.

2) Automatic Test Case Generation: The amount of quality assurance in cars is enormous. Today the industry relies very much on hardware and software as well as system-in-the-loop techniques (HIL/SIL). subsystems are executed under simulated environments in real-time. However, this technology is coming to its limits because of the growing combinatorial complexity of software in cars. More refined test techniques can help here [16], where tests are generated from models and test execution is automated [21], [70], [87].

3) Architecture and Interface Specification: A critical issue for the precise modeling of architectures is the mastering of interface specifications. So far, interface specification methods in software engineering, in general, are not good enough. This fact is, in particular, particularly problematic for the car industry, because of its distributed mode of development.

The OEM today has to identify the required functionality and has to build the logical architecture, and then distributes the development of the components that correspond to the components of the logical architectures to the suppliers. The suppliers do the implementation, even supply the hardware and bring back pieces of hardware and software that then have to be integrated into the car and connected to the bus systems. Since the interfaces are not properly described, the integration process becomes a real challenge. A lot of configuration, testing, and experimentation has to be performed, a lot of change processes are needed to understand and finally work out the integration process.

4) Error Diagnosis and Recovery: Today the amount of error diagnosis and error recovery in cars is limited. In contrast to avionics, where hardware redundancy and to some extent also software redundancy is a standard technique, in cars we do not find an extensive error treatment. In the CPUs some error logging takes place, but there is neither any consideration nor logging of errors at the level of the network and the functional distribution; there is neither a comprehensive error diagnosis nor any systematic error recovery beyond individual CPUs.

There are some fail-safe and graceful degradation techniques found in cars today, but a systematic and comprehensive error treatment is missing.

One result of this deficiency is the before-mentioned maintenance problem. Today, as mentioned above, more than 50% of the hardware devices that are replaced in garages are physically and electrotechnically without defects. They are changed, since a successful diagnosis, error tracing, and error location did not work, and thus by replacing the CPU software is replaced, too, such that the error symptom disappears—but often further, different errors show up later.

5) Reliability: Today the reliability of software in cars does not reach the high level of avionics software where a reliability of 10^9 hours mean time between failures and more is state-of-the-art. In cars, we do not even know these figures. Only adapted quality processes can improve the situation, with Toyota being quite successful with reviews based on failure modes, for instance.

The high reliability in the avionics field is of course, to a large extent, due to the use of very sophisticated error tolerance methods and redundancy techniques, as well as a sophisticated way of error modeling [like Failure mode effect analysis (FMEA)] [77], [94] also heavily applied in the automotive industries, but rather not at the level of software). In cars today, errors are merely captured within processors in so-called error logging stores. A consequent combination of modeling techniques with FMEA is a promising line of research.

What we need in the long run are comprehensive error models in cars, and also software for the detection and possibly mitigation of errors. On such models, we can base techniques to guarantee fail-safe and graceful degradation and, in the end, also error avoidance by the help of redundancy. Another issue is error logging to arrive at better error diagnosis for maintenance.

V. EVOLUTION, STATE-OF-THE-ART, AND OUTLOOK

Automotive electrics and electronics have come a long way since the introduction of electrical headlights. The requirements portfolio for automotive systems is increasingly driven by software-enabled features-think of the possibilities for flexibly integrating entertainment devices into today's cars, as compared with the birth pains experienced when trying to integrate cellular phones well into the 1990s.

As we have laid out in the organizational/economical and technical domain profiles of Sections II and III, respectively, the requirements space for automotive software is unique in its combination of innovation and cost-driven mass-market characteristics with high demands at safety, reliability, usability, and a wide spectrum of other quality properties. Customer demands and government regulations lead to increased demands for flexible addition and modification of car features—this is possible only through software.

The automotive industry has adapted to the demand for value-added functionality—initially, by means of a purely (hardware) component-oriented approach, driven by communication technologies, and the corresponding networking protocols and layouts. This has led to a fixed electrics/electronics vehicle architecture, which has actually *hindered* exploitation of the flexibility of software—many constraints are forced upon the automotive software engineer, not out of technical necessity, but to accommodate design decisions about hardware layout and functionality distribution made early on in a multiyear development cycle. This is still state-of-the-art across a wide industry segment.

The true revolution in the automotive domain happening now is driven by the increasing scattering of functionality, as well as by the recognition of software and its traditional benefits as a major asset for future growth of this industry segment. In fact, efforts such as AUTOSAR show that there is an industry-wide understanding that the automotive platform needs to become software-friendly.

With this recognition comes the need and opportunity to exploit the advances of software engineering to manage automotive software complexity-from the development process to deployment to maintenance. The approaches we have discussed as current trends and prospects in Section IV are the beginning of an era of the software-defined vehicle.

A thorough understanding of automotive software and systems requirements based on the overview we have provided here is the first step towards a comprehensive automotive domain model. This domain model, in turn, is the basis for a systematic development approach that decouples functional from deployment aspects, thus liberating both OEMs and suppliers from technology lock-in, promoting *conceptual* reuse, and enabling the rich set of analysis, synthesis, verification and validation, and maintenance techniques that has emerged over the past few years in Internet-wide business intelligence systems. The convergence between technologies, systems, and development processes between technical, embedded automotive applications and rich Internet-based business applications is on the horizon.

VI. CONCLUSION

Software is playing an increasing role in an economically highly interesting market. We have highlighted some of the salient features of software engineering for automotive systems and shown how they impact current software engineering. All features, taken on their own, can be found in other domains as well, but the combination makes for a unique discipline. The heterogeneity of the process and its players—OEMs and suppliers-as well as that of the product—control and infotainment software—demand tailored solutions in the areas of integration and evolution, including reuse.

In order to argue for new methodologies, tools, and techniques, we have emphasized the need for a comprehensive domain and cost model that augments the existing unit-based perspective to the more comprehensive views of the life cycle of a vehicle, a vehicle series, or multiple vehicle series.

We have provided ample evidence that software engineering for automotive systems provides an exciting platform for researchers with lots of open problems to be solved. We believe that model-based approaches with precise interface specifications at the levels of systems, architectures, and single units are a particularly promising avenue for future research and development. ■

Acknowledgment

The authors would like to thank colleagues from various companies from the automotive industry for enlightening discussions and colleagues from the project Mobilsoft for input. The authors are grateful to F. Doucet, T.-J. Huang, and M. Meisinger for their help with this manuscript.

REFERENCES

[1] D. Ahern, A. Clouse, and R. Turner, "CMMi distilled—An introduction to multi-discipline process improvement," in *SEI Series in Software Engineering*, Reading, MA: Addison-Wesley, 2001.

[2] J. Ahluwalia, I. H. Krüger, M. Meisinger, and W. Phillips, "Model-based run-time monitoring of end-to-end deadlines," in *Proc. Conf. Embedded Syst, Softw, (EMSOFT)*, 2005, pp. 100–109.

[3] I. Alexander, "Misuse cases-use cases with hostile intent," *IEEE Software*, pp. 58–66, Jan. 2003.

[4] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfahrth, *Matlab-Simulink-Stateflow*. München, Germany: Oldenbourg Verlag, 2003.

[5] AUTOSAR consortium, 2006. [Online]. Available: www.autosar.org

[6] J. Axelsson, J. Fröberg, H. Hansson, C. Norström, K. Sandström, and B. Villing, "A comparative case study of distributed network architectures for different automotive applications," MRTC, Tech. Rep. 478, 2003-01-28, 2003.

[7] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein, "Auto-MoDe-notations, methods, and tools for model-based development of automotive software," in *Proc. SAE 2005 World Congr.*, Detroit, MI, Apr. 2005, pp. 33–42.

[8] M. Bechter, M. Blum, H. Dettmering, and B. Stützel, "Compatibility models," in *Proc. 2006 Int. Workshop. Softw. Eng. for Automotive Syst.*, Shanghai, China, May 2006, pp. 5–12.

[9] M. Beine, R. Otterbach, and M. Jungmann, "Development of safety-critical software using automatic code generation," presented at the Proc. SAE World Congr., 2004, paper 2004-01-0708, publication SP-1852: In-vehicle networks and software, electrical wiring harnesses, and electronics and systems reliability.

[10] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[11] G. Berry, "The Constructive Semantics of Pure Esterel, Draft Book," Jul. 2, 1999, ver. 3.

[12] Bluetooth CIG, Specification of the Bluetooth System, ver. 1.1, Feb. 22, 2001. [Online]. Available: www.bluetooth.com

[13] Robert Bosch GmbH, CAN Specification Ver. 2.0, 1991.

[14] F. Boussinot and R. de Simone, "The ESTEREL language," INRIA, Tech. Rep. 1487, Jul. 1991.

[15] T. Bowen, F. Dworack, C. Chow, N. Grifeth, G. Herman, and Y. Lin, "The feature interaction problem in telecommunication systems," in *Proc 7th Int. Conf. Softw. Eng. Telecommun. Switching Syst.*, 1989, pp. 59–62.

[16] E. Bringmann and A. Krämer, "Systematic testing of the continuous behavior of automotive systems," in *Proc. 2006 Int. Workshop Softw. Eng. Automotive Syst.*, Shanghai, China, May 2006, pp. 13–20.

[17] C. Brooks and A. Rakotonirainy, "In-vehicle technologies, advanced driver assistance systems and driver distraction: Research challenges," in *Proc. Int. Conf. Driver Distraction*, Sydney, Australia, 2005.

[18] M. Broy, "Architecture driven modeling in software development," in *Proc. 9th Int. Conf. Eng. Complex Comput. Syst.*, Florence, 2004, pp. 3–14.

[19] ——, "Challenges in automotive software engineering," in *Proc. 28th Intl. Conf. Softw. Eng.*, Shanghai, China, 2006.

[20] M. Broy, F. Deißenböck, and M. Pizka, "A holistic approach to software quality at work," in *Proc. 3rd World Congr. Softw. Quality*, Munich, Germany, 2005.

[21] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, "Model-based testing of reactive systems," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2005, vol. 3472.

[22] M. Broy and I. Krüger, "Interaction interfaces—Towards a scientific foundation of a methodological usage of message sequence charts," in *Proc. Formal Engineering Methods (ICFEM'98)*, J. Staples, M. G. Hinchey, and S. Liu, Eds., 1998, pp. 2–15.

[23] M. Broy, A. Pretschner, C. Salzmann, and T. Stauner, "Software-intensive systems in the automotive domain: Challenges for research and education," presented at the Proc. SAE World Congress, 2006, paper 2006-01-1458.

[24] J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova, "Towards verified automotive software," in *Proc. 2nd Int. Workshop Softw. Eng. Automotive Syst. SEAS'05*, 2005, pp. 1–6.

[25] K. Butts, D. Bostic, A. Chutinan, J. Cook, B. Milam, and Y. Wang, "Usage scenarios for an automated model compiler," in *Proc. EMSOFT 2001*, vol. 2211, *LNCS*, T. A. Henzinger and C. M. Kirsch, Eds., 2001, pp. 66–79.

[26] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to SCADE/lustre to TTA: A layered approach for distributed embedded applications," in *Proc. 2003 ACM SIGPLAN Conf. Language, Compiler, and Tool for Embedded Syst.*, 2003, pp. 153–162.

[27] C. Chan and B. Bougler, "Evaluation of cooperative roadside and vehicle-based data collection for assessing intersection conflicts," in *Proc. IEEE Intell. Vehicles Symp.*, 2005, pp. 165–170.

[28] B. P. Crow, I. Widjaja, L. G. Kim, and P. T. Sakai, "IEEE 802.11 wireless local area networks," *IEEE Commun. Mag.*, vol. 35, no. 9, pp. 116–126, Sep. 1997.

[29] K. Czarnecki and U. Eisenecker, *Generative Programming*. Reading, MA: Addison-Wesley, 2000.

[30] J. Dannenberg and C. Kleinhans, "The coming age of collaboration in the automotive industry," *Mercer Manage. J.*, vol. 18, pp. 88–94, 2004.

[31] A. Dornan, *The Essential Guide to Wireless Communication Applications: From Mobile Systems to Wi-Fi*. Upper Saddle River, NJ: Prentice-Hall, 2002.

[32] L. Fanucci, A. Giambastiani, F. Iozzi, C. Marino, and A. Rocchi, "Platform based design for automotive sensor conditioning," in *Proc. Conf. Design, Automation and Test in Europe*, vol. 3, 2005, pp. 186–191.

[33] M. Eckrich, J. Schäuffele, and W. Baumgartner, "New steering system—BMW on the road to success with ASCET-SD, ES1000 and INCA," *RealTimes*, vol. 1, pp. 20–21, 2001.

[34] K. El Eman, J. N. Drouin, and W. Melo, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. Los Alamitos, CA: IEEE Computer Society Press, 1997, p. 450.

[35] B. Emaus, "Hitchhiker's guide to the automotive embedded software universe," in *Proc. Keynote Presentation at SEAS'05 Workshop*. [Online]. Available: http://www.inf.ethz.ch/personal/pretscha/events/seas05/bruce_emaus_keynote_050521.pdf

[36] Esterel Technologies Inc. [Online]. Available: http://www.esterel-technologies.com

[37] European Commission/Directorate General Energy and Transport. (2006, Jul.). *Road safety evolution in EU*. [Online]. Available: http://ec.europa.eu/transport/roadsafety/road_safety_observatory/doc/historical_evol.pdf

[38] A. Ferrari, G. Gaviani, G. Gentile, M. Stefano, L. Romagnoli, and M. Beine, "Automatic code generation and platform based design methodology: An engine management system design case study," presented at the Proc. SAE World Congr., 2005, paper 2005-01-1360.

[39] A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, S. Rittmann, and D. Wild, "Concretization and formalization of requirements for automotive embedded software systems development," in *Proc. 10th Australian Workshop Requirements Eng.*, K. Cox, J. L. Cybulski *et al.*, Eds., Melbourne, Australia, 2005, pp. 60–65.

[40] L. Fletcher, L. Petersson, and A. Zelinsky, "Road scene monotony detection in a fatigue management driver assistance system," in *Proc. IEEE Intell. Vehicles Symp.*, 2005, pp. 484–489.

[41] U. Freund, H. Lönn, E. Silva, J. Migge, M. Weber, M.-O. Reiser, M. von der Beeck, B. Godard, and D. Bugnot, "The EAST-ADL—A joint effort of the European automotive industry to structure distributed automotive embedded control software," in *Proc. 2nd Eur. Congr. Embedded Real Time Soft.*, Toulouse, France, 2004.

[42] J. Fröberg, K. Sandström, C. Norström, H. Hansson, J. Axelsson, and B. Villing, "Correlating business needs and network architectures in automotive applications—A comparative case study," in *Proc. 5th IFAC Int. Conf. Fieldbus Systems and Their Applications (FET)*, Aveiro, Portugal, 2003, pp. 219–228.

[43] J. Fröberg, K. Sandström, and C. Norström, "Business situation reflected in automotive electronic architectures: Analysis of four commercial cases," in *Proc. 2nd Int.*

Workshop Softw. Eng. Automotive Syst., St. Louis, MO, May 2005, pp. 1–6.

[44] E. Geisberger, "Requirements Engineering eingebetteter Systeme—ein interdisziplinärer Modellierungsansatz," Dissertation TU München, München, Germany, 2005.

[45] P. Giusto, A. Ferrari, L. Lavagno, J.-Y. Brunel, E. Fourgeau, and A. Sangiovanni-Vincentelli, "Automotive virtual integration platforms: Why's, what's, and how's," in *Proc. IEEE Intl. Conf. Comput. Design: VLSI in Comput. Processors*, 2002, pp. 370–378.

[46] K. Grimm, "Software technology in an automotive company: Major challenges," in *Proc. 25th Int. Conf. Softw. Eng*, 2003, pp. 498–503.

[47] Hansen, "Software-defined features," *The Hansen Report on Automotive Electronics. A Business and Technology Newsletter*, vol. 18, no. 4, May 2005.

[48] ——, "Software process standards gaining influence," *The Hansen Report on Automotive Electronics. A Business and Technology Newsletter*, vol. 18, no. 6, Jul./Aug. 2005.

[49] K. Hänninen, J. Mäki-Turja, and M. Nolin, "Present and future requirements in developing industrial embedded real-time systems—Interviews with designers in the vehicle domain," presented at the 13th Annu. IEEE Int. Conf. Workshop Eng. Comput. Based Syst., Potsdam, Germany, 2006.

[50] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of software in distributed embedded automotive systems," in *Proc. EMSOFT*, 2004, pp. 203–210.

[51] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[52] G. Halmans and K. Pohl, "Communicating the variability of a software-product family to customers," *Softw. Syst. Modeling*, vol. 2, no. 1, pp. 15–36, Mar. 1, 2003.

[53] G. R. Hellestrand, "The engineering of supersystems," *IEEE Comput.*, vol. 38, no. 1, pp. 103–105, Jan. 2005.

[54] ——, "Systems architecture: The empirical way—Abstract architecture to optimal systems," in *Proc. 5th ACM Int. Conf. Embedded Syst.*, Sep. 2005, pp. 147–158.

[55] T. A. Henzinger, C. M. Kirsch, and S. Matic, "Composable code generation for distributed Giotto," in *Proc. 2005 ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Syst.*, Chicago, IL, Jun. 2005, pp. 21–30.

[56] W. Hohmann, "Supporting model-based development with unambiguous specifications, formal verification and correct-by-construction," in *Proc. Embedded Softw. World Congr. Exhibition*, Detroit, MI, Mar. 2003.

[57] K. Huang, M. M. Trivedi, and T. Gandhi, "Driver's view and vehicle surround estimation using omnidirectional video stream," in *Proc. IEEE Intell. Vehicles Symp.*, Columbus, OH, Jun. 9–11, 2003.

[58] ISO, Road Vehicles—Interchange of Digital Information—Part 1: Controller Area Network Data Link Layer and Medium Access Control, Standard ISO/CD 11898-1, International Organization for Standardization.

[59] ISO, Road Vehicles—Controller Area Network (CAN)—Part 2: High Speed Medium Access, Standard ISO/CD 11898-2, International Organization for Standardization.

[60] ISO, Road Vehicles—Controller Area Network (CAN)—Part 4: Time Triggered Communication, Standard ISO/CD 11898-4, International Organization for Standardization, 2001.

[61] ITU-TS, Recommendation Z.120: Message Sequence Chart (MSC), Geneva Switzerland, 1999.

[62] A. Jameel, M. Stuempfle, D. Jiang, and A. Fuchs, "Web on wheels: Toward internet-enabled cars," *Computer*, vol. 31, no. 1, pp. 69–76, 1998.

[63] M. Jersak, K. Richter, R. Ernst, J. Braam, Z. Jiang, and F. Wolf, "Formal methods for integration of automotive software," in *Proc. Conf. Design, Automation and Test in Europe*, Mar. 2003, Designers' Forum—Vol. 2.

[64] R. K. Jurgen, *Automotive Electronics Handbook*. New York: McGraw-Hill, 1999.

[65] H. Kaaranen, S. Naghian, L. Laitinen, A. Ahtiainen, and V. Niemi, *UMTS Networks: Architecture, Mobility and Services*. New York: Wiley, 2001.

[66] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design of Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[67] H. Kopetz, "The time-triggered approach to real-time system design," in *Predictably Dependable Computing Systems*, ser. ESPRIT Basic Research Series. Berlin, Germany: Springer-Verlag, 1995.

[68] I. H. Krüger, R. Matthew, and M. Meisinger, "Efficient exploration of service-oriented architectures using aspects," in *Proc. Int. Conf. Softw. Eng.*, 2006, pp. 62–71.

[69] I. Krüger, E. Nelson, and K. V. Prasad, "Service-based software development for automotive applications," in *Proc. CONVERGENCE*, 2004, pp. 309–317.

[70] K. Lamberg, M. Beine, M. Eschmann, R. Otterbach, M. Conrad, and I. Fey, "Model-based testing of embedded automotive software using Mtest," presented at the Proc. of SAE World Congress 2004, Detroit, MI, 2004, SAE Tech. Paper Series 2004-01-1593.

[71] W. Längst, A. Lapp, K. Knorr, H.-P. Schneider, J. Schirmer, D. Kraft, and W. Kiencke, "CARTRONIC-UML models: Basis for partially automated risk analysis in early development phases," in *Proc. Workshop on Critical Systems Development With UML*, 2002, pp. 3–18.

[72] G. Leen, D. Heffernan, and A. Dunne, "Digital networks in the automotive vehicle," *Comput. Control Eng. J.*, vol. 10, no. 6, pp. 257–266, Dec. 1999.

[73] G. Leen and D. Heffernan, "Expanding automotive electronic systems," *IEEE Comput.*, vol. 35, no. 1, pp. 88–93, Jan. 2002.

[74] ——, "TTCAN: A new time-triggered controller area network," *Microprocessors and Microsystems*, vol. 26, no. 2, pp. 77–94, Mar. 17, 2002.

[75] H. Lönn, T. Saxena, M. Nolin, and M. Törngren, "FAR EAST: Modeling an automotive software architecture using the EAST ADL," in *Proc. Workshop Softw. Eng. Automotive Syst.*, pp. 43–50, 2004.

[76] S. Matic and T. A. Henzinger, "Trading end-to-end latency for composability," in *Proc. 26th IEEE Int. Real-Time Syst. Symp.*, Dec. 5–8, 2005, pp. 99–110.

[77] R. E. McDermott, R. J. Mikulak, and M. R. Beauregard, *The Basics of FMEA*. New York: Quality Resources, 1996.

[78] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[79] A. Möller, M. Åkerholm, J. Fröberg, and M. Nolin, "Industrial grading of quality requirements for automotive software component technologies," in *Proc. Embedded Real-Time Systems Implementation Workshop in Conjunction With the 26th IEEE Int. Real-Time Syst. Symp.*, Miami, Dec. 2005. [Online]. Available: http://www.cs.york.ac.uk/rtslab/demos/ertsi/final_papers/anders_moller.pdf

[80] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lormann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann, "FlexRay—The communication system for advanced automotive control systems," SAE, Doc. No. SAE 2001-01-0676, 2001.

[81] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts, "Constraint-based design-space exploration and model synthesis," in *Proc. EMSOFT*, 2003, pp. 290–305.

[82] Object Management Group OMG, CORBA: The common object request broker, architecture and specification, Framingham, 1995.

[83] Object Management Group OMG, UML 2.0 Superstructure Specification, 2003, OMG Adopted Specification ptc/03-08-02.

[84] D. Parnas, "On the criteria to be used to decompose systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, 1972.

[85] F. Pettersson, M. Ivarsson, and P. Öhman, "Automotive use case standard for embedded systems," in *Proc. 2nd Int. Workshop Softw. Eng. Automotive Syst.*, 2005, pp. 1–6.

[86] K. Pohl and A. Reuys, "Considering variabilities during component selection in product family development," in *Proc. 4th Workshop on Product Family Eng.*, 2002, pp. 22–37.

[87] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and IST automation," in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, 2005, pp. 392–401.

[88] J. Romberg, "Synthesis of distributed systems from synchronous dataflow programs," Ph.D. dissertation, Fakultät für Informatik, Technische Universität, München, Germany, 2006.

[89] S. Rittmann, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, and D. Wild, "Integrating service specifications on different levels of abstraction," in *Proc. IEEE Int. Workshop Service-Oriented Syst. Eng.*, 2005, pp. 63–70.

[90] C. Salzmann and T. Stauner, "Automotive software engineering," in *Languages for System Specification: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specification From FDL'03*, pp. 333–347, 2004.

[91] S. Sims, R. Cleaveland, K. Butts, and S. Ranville, "Automated validation of software models," in *Proc. 16th IEEE Int. Conf. Automated Softw. Eng.*, 2001, p. 91.

[92] J. Snell, D. Tidwell, and P. Kulchenko, *Programming Web Services With SOAP*. Sebastopol, CA: O'Reilly, 2002.

[93] Software Engineering Institute, Capability Maturity Model Integration, CMMI, Version 1.1, Carnegie Mellon Univ., Pittsburgh, PA, 2003.

[94] D. H. Stamatis, *Failure Mode and Effectanalysis: FMEA From Theory to Execution*. Milwaukee, WI: ASQC Quality Press, 1995.

[95] J. Staschulat, R. Ernst, A. Schulze, and F. Wolf, "Context sensitive performance analysis of automotive applications," in *Proc. Conf. Design, Automation and Test in Europe*, Mar. 2005, vol. 3, pp. 165–170.

[96] T. Stauner, "Compatibility testing of automotive system components," in *Proc. 5th Int. Conf. SW Testing (ICSTEST)*, Düsseldorf, 2004.

[97] I. Stürmer, M. Conrad, I. Fey, and H. Dörr, "Experiences with model and autocode reviews in model-based software development," in *Proc. SEAS' Workshop*, 2006, pp. 45–52.

[98] Sun Microsystems. (2006). Java 2 Platform, Enterprise Edition (J2EE). [Online]. Available: http://java.sun.com/javaee/index.jsp

[99] Y. Suzuki, T. Fujii, and M. Tanimoto, "Parking assistance using multi-camera infrastructure," in *Proc. IEEE Intell. Vehicles Symp.*, 2005, pp. 106–110.

[100] J. Thate, L. Kendrick, and S. Nadarajah, "Caterpillar: Automatic code generation," presented at the Proc. SAE World Congress, 2004, paper 2004-01-0894, publication SP-1822: Electronic Engine Controls.

[101] S. Thiel and A. Hein, "Modelling and using product line variability in automotive systems," *IEEE Software*, vol. 19, no. 4, pp. 66–72, Jul./Aug. 2002.

[102] K. Tindell, H. Kopetz, F. Wolf, and R. Ernst, "Safe automotive software development," in *Proc. Design, Automation and Test in Eur. Conf. Exhibition, 2003*, Munich, Germany, Mar. 2003, pp. 616–621.

[103] S. Voget and M. Becker, "Establishing a software product line in an immature domain software product lines," in *Proc. 2nd Int. Conf. Software Product Lines*, San Diego, CA, Aug. 19–22, 2002, pp. 60–67.

[104] M. Weber and J. Weisbrod, "Requirements engineering in automotive development: Experiences and challenges," *IEEE Softw.*, 2003, vol. 20, pp. 16–24.

[105] R. Weeks and J. J. Moskwa, *Automotive Engine Modeling for Real-Time Control Using MATLAB/SIMULINK*, 1995, SAE Paper 950 417.

[106] D. Wild, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl, and S. Rittmann, An Architecture-Centric Approach Towards the Construction of Dependable Automotive Software, SAE, Tech. Paper Series 2006, 2006.

[107] C. Wilwert, N. Navet, Y.-Q. Song, and F. Simonot-Lion, "Design of automotive X-by-wire systems," in *The Industrial Communication Technology Handbook*, R. Zurawski, Ed. Boca Raton, FL: CRC Press, 2004.

[108] World Wide Web Consortium (W3C), Web services architecture, Working Group Note. [Online]. Available: http://www.w3.org/TR/ws-arch

[109] X-by-Wire Consortium, X-by-Wire—Safety Related Fault Tolerant Systems in Vehicles—Final Rep., Project BE95/1329, Contract BRPR-CT95-0032, 1998.

[110] P. Zave, "Feature interactions and formal specifications in telecommunications," *IEEE Computer*, vol. 26, no. 8, pp. 20–28, 1993.

[111] D. Ziegenbein, P. Braun, U. Freund, A. Bauer, J. Romberg, and B. Schätz, "AutoMoDe—Model-based development of automotive software," in *Proc. Conf. Design, Automation and Test in Europe*, Mar. 2005, vol. 3, pp. 171–177.

[112] H. Zeng, A. Davare, A. Sangiovanni-Vincentelli, S. Sonalkar, S. Kanajan, and C. Pinello, *Design Space Exploration of Automotive Platforms in Metropolis*, Society of Automotive Engineers Congress, Apr. 2006.

ABOUT THE AUTHORS

Manfred Broy is a Professor at the Department of Informatics, Technische Universität München, Garching, Germany. His research interests are software and systems engineering comprising both theoretical and practical aspects. He is leading a research group working in a number of industrial projects that apply mathematically based techniques to combine practical approaches to software engineering with mathematical rigor. There, the main topics are requirements engineering, ad hoc networks, software architectures, componentware, software development processes, and graphical description techniques. In his group, the CASE tool AutoFocus was developed. Today one of his main interest is the development of a modeling theory for software and systems engineering.

Ingolf H. Krüger received the M.A. degree from the University of Texas, Austin, in 1996 and the Ph.D. degree from the Technische Universität München, Garching, Germany, in 2000.

He is an Assistant Professor in Residence in the Department of Computer Science and Engineering, Jacobs School of Engineering, University of California at San Diego (UCSD), leading the Service-Oriented Software and Systems Engineering Laboratory. He also directs the Software and Systems Architecture and Integration functional area within the California Institute for Telecommunications and Information Technology (Calit2). He is a member of the UCSD Divisional Council of Calit2. His major research interests are service-oriented software and systems engineering for distributed, reactive systems, software architectures, description techniques, verification and validation, and development processes. The application domains to which he applies his research results span the entire range from networked embedded systems to Internet-wide business information architectures.

Dr. Krüger together with M. Broy organized the Automotive Software Workshops 2004 and 2006, San Diego, CA.

Alexander Pretschner received the M.S. degree in computer science from RWTH Aachen, Aachen, Germany, and the University of Kansas, Lawrence, and the Ph.D. degree in computer science from the Technische Universität München, Garching, Germany.

He works as a Senior Researcher at ETH Zürich, Zürich, Switzerland, concentrating on model-based testing and usage control. He has organized several workshops in the field of software engineering for automotive systems.

Christian Salzmann received the M.S. degree in computer science degree from Karlsruhe University, Karlsruhe, Germany, and the University of Massachusetts, Dartmouth, and the Ph.D. degree in software engineering from the Technische Universität München, Garching, Germany, in 2002.

Until 2006, he was affiliated with BMW Car IT, where he was heading the automotive infrastructure group. Since 2006, he has been with BMW AG, München, where he is responsible for the development of central gateways.

Dr. Salzmann is the organizer of several ICSE Workshops on automotive software engineering and a member of the Special Interest Group on automotive software of the German "Gesellschaft für Informatik."