Software Engineering for Automotive Systems: A Roadmap

Alexander Pretschner, Manfred Broy, Ingolf H. Krüger, Thomas Stauner

Alexander Pretschner is a senior researcher at ETH Zürich, Switzerland, currently concentrating on model-based testing and distributed usage control. He holds master's degrees in computer science from RWTH Aachen and the University of Kansas as well as a Ph.D. degree in computer science from Technische Universität München. Alexander has organized several workshops in the field of software engineering for automotive systems.

Manfred Broy is a professor at the Department of Informatics of Technische Universität München, 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. 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 interests is the development of a modeling theory for software and systems engineering.

Ingolf H. Krüger holds a Ph.D. from Technische Universität München, Germany, and an M.A. from the University of Texas at Austin. He is an Assistant Professor i.R. in the Computer Science and Engineering Department of UCSD's Jacobs School of Engineering, leading the "Service-Oriented Software and Systems Engineering Laboratory"; he also directs the "Software & Systems Architecture & Integration" functional area within the California Institute for Telecommunications and Information Technology (Calit2). Dr. Krüger is a member of the UCSD Divisional Council of Calit2. Krüger's major research interests are service-oriented software & systems engineering for distributed, reactive systems, software architectures, description techniques, verification&validation, and development processes. Together with Manfred Broy he has organized the Automotive Software Workshops 2004 and 2006 in San Diego, CA.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Thomas Stauner is a software specialist with BMW AG, Germany. He studied computer science at Technische Universität München and the University of Edinburgh. He obtained his PhD in computer science from Technische Universität München. After leading a team at BMW Car IT on specification and verification of automotive systems, he is currently responsible for compatibility management of automotive electronic control units at BMW.

Software Engineering for Automotive Systems: A Roadmap

Alexander Pretschner[1], Manfred Broy[2], Ingolf H. Krüger[3], Thomas Stauner[4]
[1]*Information Security, ETH Zürich, Switzerland*
[2]*Software&Systems Engineering, TU München, Germany*
[3]*CSE Department, UCSD, La Jolla, CA, USA*
[4]*BMW Group, München, Germany*
pretscha@inf.ethz.ch, broy@in.tum.de, ikrueger@ucsd.edu, thomas.stauner@bmw.de

Abstract

The first pieces of software were introduced into cars in 1976. By 2010, premium class vehicles are expected to contain one gigabyte of on-board software. We present research challenges in the domain of automotive software engineering.

1. Introduction

The amount of software in modern cars is increasing at a breathtaking pace. The current BMW 7 series, for instance, implements about 270 functions that a user interacts with, deployed over up to 67 embedded platforms. Altogether, the software amounts to about 65 megabytes of binary code. The next generation of upper class vehicles, hitting the market around the year 2010, is expected to run one gigabyte of software (on board, excluding data on DVD for navigation etc.). This is comparable to what a typical desktop workstation runs today. Reasons for this tremendous increase include the demand for new functionality on the one hand, and the availability of powerful and cheap hardware on the other hand. Furthermore, electronics in cars help reduce gas consumption as well as increase performance, comfort and safety as today's figures of increasing traffic with decreasing serious accidents show. Finally, software enables OEMs (the "car makers") and suppliers to tailor systems to particular customers' needs. In other words, software can help differentiate between cars. At least in principle, software also allows expensive hardware to be reused across different cars.

Automotive software is economically relevant. The worldwide value creation in automotive electric/electronics (including software) amounts to an estimated € 127 billion in 2002 and an expected € 316 billion in 2015 according to a Mercer study [DK04]. Software makes up an estimated 40 percent of this value creation by 2010 [HKK04].

The considerable and increasing complexity of automotive software systems and their huge economic relevance give rise to various organizational, engineering, and research challenges. In the first part of this paper, we describe the unique blend of characteristics that define automotive software systems. Reflecting on the market, technology, economics, and organization of labor, we describe software-related consequences of

- the heterogeneous nature of automotive software (embedded and infotainment systems) and the consequences for systems integration, tools, processes and engineering skills;
- the huge number of communicating processors and the resulting need for tailored middleware;
- the necessity to handle large numbers of variants and configurations;
- decisions grounded in a unit-based cost model;
- the interplay of long life and short development cycles; and
- specific requirements related to reliability, safety, and security.

As key research challenges, we identify the integration of heterogeneous subsystems from different sources as well as their evolution and maintenance, and reuse.

Division of labor in the automotive world has traditionally been organized in a highly vertical manner, resulting in distributed and concurrent engineering processes. The corresponding need for clear interface descriptions together with a tradition of model-based development of continuous systems explains, at least in part, the relative success of model-based approaches to developing automotive software systems. In the second part of the paper, we hence explore potential benefits of a seamless model-based development process that caters to the characteristics of automotive software in terms of requirements engineering and management, architecture development, and testing activities.

Overview. Section 2 describes the characteristics of the domain of automotive software and their consequences. Section 3 uses this analysis to identify research challenges. More specifically, we present one

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

particular facet of model-based development as a possible solution in Section 4. Our presentation of the fundamental models itself leads to many more relevant areas of future research. Section 5 summarizes our findings and concludes. A more detailed discussion of the domain characteristics can be found in [BKP+07].

2. Domain Profile

We use this section to present five salient features of the automotive software domain as it presents itself today. These features are the heterogeneous nature of the software involved, the organization of labor, the distributed nature of automotive software, the huge number of variants and configurations, and the predominance of unit-based cost models. In Section 3, we will use this analysis to highlight particularly relevant areas of research.

2.1 Heterogeneity of Software

2.1.1 Description

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 non-functional requirements:

- Multimedia, telematics, and human-machine interface (HMI) software: typically soft real-time software, which also has to interface with off-board IT, dominated by discrete-event/data processing;
- Body/comfort software: typically soft real-time, discrete-event processing dominates over control programs;
- Software for safety electronics: hard real-time, discrete event-based, strict safety requirements;
- Power train and chassis control software: hard real-time, control algorithms dominate over discrete-event processing, strict availability requirements; and
- Infrastructure software: soft and hard real-time, event-based software for management of the IT systems in the vehicle, such as software for diagnostics and software updates.

In terms of non-functional requirements, reliability and safety are concerns for all functions relevant to driving, from engine control and passenger safety functions to forthcoming X-by-wire [XBW98] functions where mechanical transmission is replaced by electrical signals, such as steer-by-wire (steering signals are electronically transmitted to the wheels) or brake-by-wire (braking signals are electronically transmitted to the brakes). In addition, with the development of infotainment functionality, the car is becoming an information hub where functions of cell phones, Laptops and PDAs are interconnected using wired and wireless network technologies (UMTS, Bluetooth, WiFi) via and with car information systems. Increasingly, the on-board electronics systems establish communication links beyond car boundaries, enabling various applications relating to, among others, sharing of infotainment content, remote vehicle analysis, software updates, global positioning and emergency services, inter-vehicle communication for crash prevention and automatic convoy forming.

2.1.2 Consequences

From the integrated software and systems engineering perspective, the five clusters suggest a need for development skills from various disciplines.

- The different SW/HW systems consist of a high number of separated functions and processes that exchange information over several communication links. These systems exhibit all the details and complexities of distributed computer networks—computer science and computer engineering skills are required to successfully build automotive systems.
- These systems are connected to sensors and actuators that read physical values and operate physical processes, respectively, in real time. The SW/HW systems have to respond to these inputs in a classical control-theoretic manner—knowledge of control theory is required.
- For some functions, such as power train control software, an understanding of mechanical engineering is mandatory.

Historically, the methods as well as the models of control theory have been quite different from the models of information processing. In control theory, tools such as Matlab/Simulink [Mat06,WM95] are used to model differential equations. In contrast, engineers in business information processing are increasingly eager to use models of data and behavior as expressed in the Unified Modeling Language (UML), and other kinds of discrete data flow or state machine models. In automotive development, these separate engineering cultures have to be united to reflect all relevant aspects of the different engineering domains.

The heterogeneity of the domain and the lack of a widely accepted, let alone standardized, set of models and development approaches also explains the large number of different tools in use in automotive software development today. Because the tools are developed by different vendors, there is no comprehensive system model, and as a consequence, the tools are usually not integrated. There exist attempts to creating 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 [PT06, KSL+03, AKMP05], but in most

cases, full production support has not yet been achieved.

2.2 Distribution of Labor

2.2.1 Description

As mentioned above, the car industry has traditionally been organized in a highly vertical manner. Mechanical engineers worked hard for over a century to render the various sub-systems 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 product value is created by the OEMs who tend to concentrate on the engine, integration, design, and marketing of the brand.

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 portion of the engineering and production activities were, and still are, outsourced. This also facilitates optimization of cost and risk distribution. 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, as the example of the central locking system in §2.3 will demonstrate.

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 will often strongly constrain the technologies that may be used by a supplier.)

2.2.2 Consequences

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.

A negative side effect of the distribution of labor is that complex distributed processes need to be coordinated. In particular, development is geographically distributed and communication gets more complicated. Clear interfaces as well as liabilities need to be defined. The large number of parties involved in itself is a reason for unstable requirements, with frequent changes and revisions of the requirements during the development process as a consequence. Furthermore, the OEM often only has black-box specifications of the subsystems to be integrated, and it is difficult or impossible for the OEM to localize errors and to modify parts of the subsystems.

2.3 Distribution of Software

2.3.1 Description

As mentioned in the introduction, today's premium-class vehicles are equipped with as many as 67 processors that implement roughly 270 *user functions* that a user interacts with (one of the reasons for the huge number of processors being the organization of labor as described in §2.2). These functions are composed of as many as 2500 "atomic" *software functions*. These functions address many different issues including classical driving tasks but also other features in comfort and infotainment and many more. These functions do not stand alone, but exhibit a high dependency on each other. A telling example for the increased interaction among previously unrelated sub-systems is the central locking system (CLS) as found in most modern vehicles [KNP04]. 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 HMI functions, such as signaling the locking and unlocking using the car's interior and exterior lighting system. Many of these functions are realized in sub-systems that are distributed according to the major mechanical breakdown of the vehicle into engine, drive-train, body and comfort systems. In some vehicles this seemingly simple functionality is physically distributed over up to 18 electronic control units (ECUs). Reinforced by a trend to combine on-board with off-board IT, the car has turned from an assembled device into an integrated system.

2.3.2 Consequences

With the huge amount of software-based functions, phenomena like unintentional feature interaction [Zav93] have become issues. Feature interaction is the technical term created in the telecommunication domain for intentional or unintentional dependencies between individual features. Feature interactions are visible at the user level as specific dependencies between distinct functions.

A second consequence of the highly distributed nature of automotive software systems is the intricacy of an appropriate technical infrastructure, including operating systems and middleware. There are five bus systems (even more for some luxury vehicles) that serve as communication platform for ECUs. There are real time operating systems; and a lot of system-specific technical infrastructure on which the applications are

based. One challenge this infrastructure has to face is the significant amount of multiplexing at the bus level to efficiently support communication among the dozens of ECUs for thousands of software-enabled tasks in parallel. Among others, one consequence of the high degree of multiplexing is that the transmission time of messages exhibit jitter so that systems appear to be nondeterministic. In many cases, timing deadlines cannot be guaranteed. Similar problems arise in the individual ECUs where tasks and schedulers manage (virtual) parallelism. We can thus find all the issues 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.

Due to the unpredictable load, time guarantees often cannot be provided. In many ways the communication buses serve as a global "memory" or database, capturing information about the current state of the vehicle and its electronics components. Therefore, a lot of interesting potentials for improvement, such as a car-wide data model and management system, or the introduction of drive-by-wire systems have not been realized so far. Time-synchronous bus systems like TT-CAN [ISO01] or FlexRay [MHB+01] are current attempts at solving these problems.

2.4 Variants and Configurations

2.4.1 Description

The need for differentiation in the mass market motivates the desire for customized items. 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 2^{80} variants to be ordered and produced for a car. In a similar vein, the authors of [BBC+01] calculate for a simplified power train control application 3,488 possible component realizations by instantiating different algorithms and their variants.

A different kind of variability is a consequence of the development process. A car model is usually produced for seven to eight years. The customer expectation of a long lifetime is reflected in the OEM's duty to offer service and spare parts for at least 15 years after the purchase of a vehicle (compare this to an estimated lifecycle of, say, 4 years for an average workstation program, with several hot fixes during this period). The life cycle of hardware components such as CPUs or DSPs is much smaller, say less than 5 years. Some of them will no longer be produced and have to be replaced by newer types. Already after the first three years of production, 25 percent of the ECUs in the car typically have to be replaced by newer ECUs due to discontinuation of an ECU's specific technology.

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. This means that, over time, there are various versions for each piece of software in a car. When defective ECUs are 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.

2.4.2 Consequences

Market demands, short innovation and long life cycles lead to a huge number of variants and configurations. Updating or replacing software in cars is a challenge. New versions of software are brought in when exchanging entire ECUs, or during maintenance by "flashing" techniques for replacing the software of an ECU. In this context, it is estimated that today more than fifty percent 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, or for 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 defective hardware but in ill-designed or incompatible software.

When exchanging entire ECUs or updating the respective software, one has to be sure that new software versions correctly interoperate with the remainder of the vehicle software (compatibility [BBD+06]). Because of the substantial scattering of functionality in cars today, this is difficult, and a lot of the problems we see today in the field are indeed compatibility problems. Obviously, an elaborate design and test methodology is required for this.

Furthermore, because of the long lifecycles of cars, long-term maintenance processes must be organized. Today, an OEM's vehicle fleet is predominantly maintained by vehicle dealers following prescribed semi-automated procedures. With its increasing amount of software, the vehicle more and more inherits the characteristics of a complex IT system. There is a difference with the desktop software market, however: We deem it likely that future maintenance of on-board software will continue not to be delegated to the users. Among other things, this is a consequence of the OEM's desire to create an overall brand "experience" to which the customer can relate as a "package".

2.5 Unit-Based Cost Model

2.5.1 Description

The automotive industry operates in a highly competitive mass market with strong cost pressure. Here the rules of business of scale prove to be crucial: how

many units of a product are sold? Depending on the market segments targeted by an OEM, competition occurs over product price, product quality, product image and differentiating product features. Competition by differentiation requires innovation and a strong brand profile. Competition over price requires permanent optimization.

Traditionally, the cost per unit produced has played 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 € 1 for 20 such components (including 1 processor each) in each car would then lead to an overall cost reduction of € 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.

2.5.2 Consequences

As a consequence, engineers concentrate on reducing the amount of required memory and computation power. Code is then written and directly optimized for specific individual processors. Such optimization requires that the software be very closely tuned towards the processors' characteristics. Trying to squeeze the code into as little memory as possible requires a further set of code optimizations. As a consequence, it becomes difficult to port the code to another processor. Thus, keeping pace with processor life cycles (§2.4.1) 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. The negative results are as follows. It is very difficult to add any functionality to the system later on, and it is very difficult to change parts of the code or to fix defects. The code is more complex than necessary, for instance, in terms of strong coupling between modules. Changing the code becomes very difficult, and reusing this code in future car models or on other processors is almost impossible. Finally, some defects in the code may be a result of the optimization itself and finding defects may become even more difficult, since now application logic issues are obscured by optimization.

In sum, exclusively thinking in terms of unit-based costs with the associated need for optimizations makes the software complex and difficult to handle. 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 recognize that unit-based costs are important for software-based functions, as the above numbers show. However, they are but one factor.

3. Research Challenges in Software Engineering

The domain characteristics of §2 directly translate into many fascinating areas of research in software engineering (we omit application-specific challenges such as crash prevention, advanced energy management, driver assistance systems, further X-by-wire technologies, HMI-related challenges, personalization, etc. here). At the bottom line, they all relate to quality and cost, reflected by the need for integration, evolution, and reuse:

1. Languages, models, and techniques for requirements engineering that support the structured specification of multi-functional systems and their mutual dependencies;
2. Languages, models, and techniques for requirements engineering that cater to heterogeneous systems and the engineers that build them, to the interplay between OEMs and suppliers, and to the huge number of variants and configurations;
3. Platform and HW/SW designs and design methodologies at different levels of abstraction that address the heterogeneity of the systems involved as well as the compatibility problem;
4. Middleware at different levels of abstraction that enables the communication between heterogeneous subsystems;
5. Comprehensive cost models that take into account development, maintenance and opportunity cost (for failure of enabling reuse, for instance);
6. System models that enable the semantics-preserving integration of different tools;
7. Design and coding practices that lead to portable and reusable code;
8. Security of the communication within a car as well as between cars and several forms of off-board IT;
9. Reliability estimates, timing predictability, measurements, and assurance;
10. Techniques and error models for quality assurance—particularly relevant in a domain with huge numbers of deployed entities and very limited control over them—that, in particular, address the integration problem as well as the huge numbers of variants and configurations;
11. Integration of on-board and off-board IT;
12. Approaches to error diagnosis and recovery; and
13. Approaches to reuse for the benefit of reducing complexity and cost.

We will use Section 4 to address items 1, 2, 3, and 6 collectively in some detail under the headlight of

"model-based development". In the remainder of this section, we briefly summarize research challenges for some of the other areas.

3.1 Middleware: Communication Services

A classical approach to handling complexity is separation of concerns. The application logic, for instance, should obviously 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. With each of the five busses (§2.3.2) having its own characteristics and protocols, the definition of a respective adequate middleware, of course, comes as a significant challenge.

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 CORBA and web services). Flexibility at runtime is still dominated by the need for flexibility at design time in the automotive domain, because the runtime layout of automotive software is still mostly static. The following issues need to be considered for automotive middleware:

- *Resource optimization:* due to the unit-based cost structure (§2.5), 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 (§2.1).
- *Optimizability to hardware but also transferability from one hardware platform to another:* due to the lifecycle gap (§2.4) 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 (§2.4), it must be possible to upgrade a system during its lifetime and extend it with new features.

In contrast to other middleware approaches there is no single instance in the system that handles the communication dynamically (like an ORB in CORBA), but the middleware layer is generated statically for this special configuration of the system. This can lead to lean, highly optimized middleware portions 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 achievable by increasing the expressive power of automotive modeling approaches towards model-based system specifications supporting code generation of middleware components in an optimized and validated way. The AUTOSAR [Aut06] 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.

3.2 Safety and Security

The life-criticality of many *avionics* systems has led to reliabilities of 10^9 hours mean time between failures. This high reliability is, on one hand, due to the use of very sophisticated error tolerance methods and redundancy techniques, and on the other hand due to sophisticated ways of error modeling (like Failure Mode and Effect Analysis (FMEA), which is also heavily applied in the automotive industries, but rather not at the level of software). Furthermore, driven by government mandates, avionics companies invest heavily into quality management, including rigorous code inspection techniques throughout the development process. For many equally life-critical systems in the automotive domain, the respective numbers are not even known (but the requirements are admittedly different). Research into measuring and improving reliability is, hence, required.

Personalization and the related privacy and security issues are becoming increasingly important, notwithstanding usability issues. The management of intellectual property, including digital rights management, is particularly challenging in a distributed development process as described in §2.2. From a liability perspective, unauthorized "tuning" of code must be prohibited or at least be detectable in hindsight. Liability also is an important issue in ad-hoc distributed safety-critical applications such as crash prevention that relies on communication between cars. Today's secure communication protocols need to be extended for real-time applications.

There are clear benefits to hardening the automotive infrastructure against the intrusion of unauthorized services and components. The more the vehicle becomes connected to the cyber-infrastructure the more

susceptible it becomes to attacks carried out via this cyber-infrastructure. Management of security is, therefore, a necessity both on-and off-board. As an example for on-board authentication requirements, consider the importance of identifying which of the myriad of functions and ECUs is responsible for a failure: if the functions, as they become active and communicate, authenticate themselves, the system could identify the presence of unauthorized components, or could determine which of the authorized components malfunctioned. These topics are under research also in their more traditional home grounds of internet-enabled business information systems; their inherently cross-cutting nature makes them particularly challenging in automotive architectures with a high degree of scattered functionality.

3.3 Error Diagnosis and Recovery

Failure management, from a systems engineering perspective, is a further area that requires increased attention. Because of its role as the system integrator, the OEM is uniquely positioned to deal with failures at the composite system level—as compared to the typically localized failure management at the component level prevalent today. This requires, however, comprehensive logical *and* technical domain models of failures, failure effects, failure detectors, mitigators and mitigation strategies that influence the choice of both logical architectures and their mapping to technical architectures (§4.2).

Today the amount of error diagnosis and error recovery in cars is rather lightweight. In the CPUs some error logging takes place, but there is no consideration nor logging of errors at the level of the network and the functional distribution; there is no comprehensive error diagnosis and no systematic error recovery beyond individual CPUs (note that as of today, this appears appropriate because systems are essentially designed in a way that will lead to a safe state, even if bus communication crashes). One result of inadequate error management is the maintenance problem mentioned in §2.4.1, resulting in the replacement of many non-defective ECUs. Failure logging to the end of better error diagnosis for maintenance then emerges as a relevant research problem.

There are some fail-safe and graceful degradation techniques found in cars today, but a systematic and comprehensive error treatment is missing. With the upcoming multi-core controllers for embedded applications, an interesting area for research is how this can be exploited also for redundancy/recovery strategies. In the long run, comprehensive error models in cars seem desirable, and so does 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.

3.4 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 (of the systems that exist in both generations) differs mostly not more than 10%, while much more than 10% of the software is re-written. The short hardware lifecycles (§2.4) may require frequent re-implementations. Nevertheless, today the process of software reuse is not systematically planned between OEMs and suppliers, as required, say, for software product lines ([CN01]; see the comment below). From the OEM point of view, reuse rather occurs on the level of whole ECUs than on the level of software, and the reuse objectives of OEMs and suppliers may be in conflict with each other. Reuse is arguably one of the most challenging problems, clearly transcending the automotive domain, and we are not aware of convincing solutions for the general problem. However, some reuse problems are of an accidental rather than an essential nature. For instance, too strong an optimization of the software towards the hardware (§2.5) can make reuse in the form of porting it to new hardware impossible or very expensive. With the increasing importance of software, we deem it a mere question of time until it becomes more economical to use more generous hardware structures and to stay away from low-level code optimization.

Reuse comes in different forms. Reuse at the level of single code modules has proven to be utterly difficult. 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. Well-designed libraries and frameworks form a promising avenue of research.

Reuse is also facilitated by standardized middleware [Aut06] that allows for coordinated and standardized interfaces. At the level of requirements engineering, there definitely is a need for further research into product lines (with the common argument that product lines cater to anticipated changes only). 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 (as reflected by suppliers being seemingly more open to product line approaches than OEMs). Research into reuse must of course include studies of the cost effectiveness, and hence be 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.

3.5 Cost Models

In the development of software intensive systems, many aspects of costs are involved, including development cost, maintenance cost, different forms of opportunity cost, and reputation-related costs for the OEM's brand. So far the comprehensive cost situation is not understood in sufficient detail. What is quite clear is that the costs for the electronic devices both for the development and for the production are rising (§1). But it is not so clear how development cost is distributed between software and hardware costs. Because current cost models usually relate to the cost per unit, software is considered an *integral* part of the development process and not explicitly calculated in the contracts between the supplier and the OEM despite its continuous rise (there is an estimation that, per year, about five percent of the costs "migrate" from hardware to software).

The importance of intellectual property (IP) issues seems to exceed that of hardware developments. The IP for a large piece of software is remarkable. The next generation of premium cars will exhibit hundreds of millions of lines of code. If the overall costs for such an amount of code are calculated according to the classical development costs, the value of the software costs of a premium car amounts to somewhere between three hundred and eight hundred million €. Owning the software and being able to reuse it is an important factor in the cost models.

In sum, the exponential increase of software does not justify the use of restricted unit-based cost-models alone. The research challenge consists of understanding processes and products and defining more appropriate, comprehensive cost models. The automotive industry needs decision and cost models that take into account rising development costs, maintenance cost, software-related project risk and time-to-market. It is likely that such models require a more transparent cooperation between suppliers and OEMs

4. Model-Based Development

In this section, we describe some further research challenges. We will cast facets of a possible solution to the abovementioned problems into the general ideas of model-based development, by taking into account the automotive idiosyncrasies. In this paper, model-based development means working with artifacts representing domain and design knowledge at different levels of abstraction, throughout the development process, and possibly also at runtime. Many crucial software-related facets of a system are then represented by the following artifacts.

- *Requirements models that address multi-functionality and feature interactions* embrace all requirements-related issues, dealing with the direct behavior of embedded software-based functions from the users' point of view. These include the driver, passengers, maintenance staff and other persons dealing with the car. Use cases and related behavior specifications are one part of the requirements models.
- The *logical architecture* is a breakdown of the functionality into interacting logical components. It represents the functional decomposition of a system into functional components, as well as the behaviors of these components at the logical level. The functional components provide the functionalities described in the requirements model.
- The *technical architecture* defines the deployment architecture, i.e. all the hardware units, the basic software (operating system and middleware) on them and their connections: controllers, communication devices, actuators and sensor, as well as a mapping (the "deployment function") from the logical architecture (its structures and behaviors), to this deployment architecture. This includes the definition of source code modules, the platform, and the representation of the application software in terms of tasks, based on the chosen platform, as well as the mapping of these tasks to ECUs and their schedules.

Each of these models must of course be connected to non-functional requirements, including safety reliability, maintainability, portability, performance, etc. The architectures and the implementation of the system then have to ensure these requirements.

In the remainder of this section, we discuss requirements models (§4.1), the logical and technical architectures (§4.2), and the role of detailed behavior models (§4.3). We will argue that a clear separation of concerns together with a comprehensive understanding of the domain-specific issues (§2) and their cross-cutting aspects is likely to be the main benefits of this approach. We consider code generation, in particular in the domain of discrete systems, to be but one of the benefits of a model-based approach. Furthermore, we deem the availability of comprehensive product models

the exception, rather than the norm. This is a consequence of the complexity of the systems and the nature of the distributed development process (§2.2).

With different levels of abstraction, seamlessness and traceability are, of course, major concerns and belong to the fundamental research challenges: how can, conceptually, models at different levels of abstraction be related to one another [BBJ+05], and how can the respective tools be integrated? The different levels of abstraction discussed in this section support requirements tracing for functional requirements (of course, requirements tracing should also include the link between requirements, their origin—e.g., requirements from marketing—and design decisions). The flow-down is as follows. The relationship between the function hierarchy—a part of the requirements model—and the logical architecture indicates which components of the logical architecture are contributing to (i.e., collaboratively realizing) the respective function. The logical components are later represented by specific software. The deployment onto the technical architecture determines which software runs on which hardware and which logical communication channels are implemented by which bus systems. In sum, by also taking into account the quality models, elements of the function hierarchy can be traced to the hardware level, which greatly facilitates requirements verification, maintenance, and evolution after the start of production.

4.1 Model-Based Requirements Engineering

There is a general agreement that requirements engineering for embedded systems is a key discipline in the automotive domain—a discipline that is not sufficiently mastered today [WW03]. Reasons include the following.

- Many new innovative functions in cars today are based on embedded software systems. There is no experience so far with these functions and the best way to engineer the human-machine interactions with them. The process of deciding on the optimal realization of functions, the interaction between functions themselves, and the interaction between users and functions is a difficult and error-prone learning process. Models and prototypes can provide initial solutions to these problems.
- The systems are multi-functional, and a huge number (§2.3.1) of functions is offered to the user. These functions exhibit complex interactions, are mutually dependent and give rise to intended and unwanted feature interactions.
- The suppliers realize a lot of the functionality (§2.2). Therefore the overall ideas of functions have to be fixed by the OEMs and then documented in a way such that the supplier can implement them.
- Over the development process requirements occur in strongly varying levels of detail. In the beginning requirements are often very abstract, e.g. based on benchmarking with competitors. However, in the same process phase requirements to reuse some ECUs from other products may already be fixed. The need to integrate these ECUs not only strongly restricts the set of possible solutions, it also adds a large set of very detailed requirements resulting from the ECUs to be integrated.
- Requirements specifications must deal with a large number of vehicle variants (§2.4; for instance, “2 doors”, “4 doors”), and in particular variants resulting from different combinations of auxiliary equipment.
- Besides the functional requirements there is a large number of non-functional requirements concerning cost, time-to-market for innovations, safety, security, reliability, maintainability etc.
- Often, there are further constraining platform-specific requirements (“this ECU has to be reused”), pulling deployment specifics already into the levels of requirements engineering and logical architecture design.

In general, requirements are originally expressed in natural language. It has turned out that rigorously imposing structure on the text is most useful. A first step from text to models are taxonomies that can be computed from text by natural language processing techniques [Kof05]. They are used as a basis for the abovementioned requirements models (that of course reflect a lot of structuring activities and which are hence richer than mere feature trees [BLP04]).

The system’s (intended) functionality is modeled by a function hierarchy that collects all software-based functions. These will be implemented by functional entities defined at the level of the logical architecture. Requirements are associated with the elements of this hierarchy. Its nodes relate to one another in an “is-subfunction” relation. The dependencies between the functions must also be described. Each function is modeled in isolation. A function may, for instance, define an interactive service that is described by a state machine. System behavior in the presence of failures needs to be specified as well. Based on the function hierarchy, detailed behavior patterns can be provided for the individual functions. This model of the functionality comes along with models of the required quality aspects of the system.

These models are a necessary starting point for mapping requirements to elements of the logical and

technical architectures (ideally, the latter mapping is indirect via elements of the logical architecture); the quality models are used to assess and optimize architecture decisions. Therefore, for a systematic model-based requirements definition, all requirements have, sooner or later, to be formulated in terms of the structured view on the architectures.

Research Challenges. This approach to requirements engineering entails many research challenges. The definition of the tracing structures and their effective tool support is, so far, an unsolved problem.

As indicated above, requirements exist at various levels of detail. They range from marketing-driven requests ("the car has to have the following comfort-functions") to very detailed platform specifications, which may limit the design space for both logical and technical architectures. One research challenge is, therefore, to elucidate a comprehensive requirements model that brings out these levels of abstraction and optimizes the resulting solution space for logical and technical architecture.

A systematic way to structure the requirements after capturing them, to make them precise, and to validate them is still a challenge. First of all, a reference model is needed that defines all the artifacts that are to be considered as results of requirements engineering and requirements dependencies (see [GBB+06]).

So far the models offered for requirements engineering are limited. We need structured hierarchies of all the software-based functions in a car that reflect all their mutual dependencies (specified feature interaction). Good ways to model the functional hierarchies and their dependencies that support automatic analysis are a challenge for research. In addition, the functional behavior of the individual functions has to be modeled.

The systematic step from the requirements to the design phase, taking into account both functional *and* quality requirements, is largely unsolved. Eliciting domain-specific design and analysis patterns can be a promising research direction to tackle this problem.

4.2 Logical and Technical Architectures

As outlined above, the complexity of automotive systems rivals that of other ultra large scale systems, including avionics, command and control, and internet-wide business intelligence systems. In fact, automotive systems combine many of the requirements challenges we see elsewhere only in isolation. Historically, there has been a tight coupling between automotive software functions and the physical processes they manage, and thus with dedicated, networked ECUs. This tight coupling has contributed significantly to the fragmentation of the automotive platform into its current state. This, in turn, has led to a strong entanglement between the logical architecture or function network, and its deployment on a concrete, technical architecture. This entanglement gives rise to a scattering of functionality (§2.3.1). One of the central challenges for next-generation automotive system development is, therefore, to disentangle logical and technical architectures. This will help unleash so far untapped potentials at

- reducing the number of ECUs required to deliver the desired functionality based on the ability to establish globally optimal mappings from functions to ECUs;
- enabling dynamic reallocation of computing and communication resources to effect globally optimal energy and QoS management, or to manage failures by means of an appropriate reconfiguration of the system;
- reducing the dependency on physical proximity for the provisioning of automotive functionality by introducing location transparency, say, for functions such as navigation;
- enabling conceptual reuse by allowing independent evolution of logical and technical architecture; and
- enabling faster modeling, design and test cycles, because the OEM can ultimately perform continuous integration of functionalities as they become available from suppliers – rather than having to wait until all functions of all ECUs are implemented towards the end of the overall system development cycle during system integration.

Modern approaches to software and systems architecture and integration recognize the importance of separating logical and technical architectures. Model-Driven Architecture, for instance, distinguishes between Platform Independent Models (PIMs) and Platform Specific Models (PSMs) to separate logical functionality from its mapping to a deployment model. Architecture standards, such as the Department of Defense Architecture Framework, distinguish *operational* from *systems* views to effect a similar disentanglement between logical and technical system aspects.

In essence, the models relevant for logical architecture focus on *capabilities* and their mapping to logical entities (sometimes called *operational nodes*). These capabilities realize the functions in the requirements model. The models relevant for technical architecture focus on deployment, i.e. the physical layout of the system including physical nodes and networking structures—and the mapping from the logical architecture to this layout. Consequently, many models, including structural and behavioral models, crosscut logical and technical architectures. Often, the technical architecture introduces additional constraints at, for instance, performance, safety, security and reliability that

influence the mapping from logical to technical architecture. The use of *reflective* models, i.e. models that are accessible to and can be modified by the runtime infrastructure, can establish a link between the logical and technical architecture; this can provide a means to adapt the mapping between logical and technical architecture according to resource constraints, or to overcome failures.

4.2.1 Logical Architecture

The focus of the logical architecture in general is the set of capabilities provided and requested by the overall system and its subsystems. This describes the (logical) implementation of the overall functionality by a network of logical entities (operational nodes including software components) and the necessary links between these logical entities. In addition, the logical architecture encompasses mapped use cases, the relevant data models, as well as QoS, bandwidth, (real-time) performance, security and other cross-cutting concerns to the degree they are relevant on the logical level. Data models, logical entities and behavior models are typically linked by means of data-flow models. Depending on the level of detail at which these models are available, the logical architecture can support early simulation, optimization, prototyping, verification & validation, including testing.

In the automotive domain, the so-called *function network* (which is not the same as the function hierarchy of the requirements model) is often used as a key—sometimes the only—expression of the logical architecture. The function network is, in essence, a representation of the functionality to be provided by the vehicle together with links indicating (communication) dependencies among these functions. The modeled functions are then mapped to the HW/SW implementations as part of the technical architecture, considering the also captured real-time and bandwidth requirements.

Because the OEM to a large extent plays the role of system integrator, the logical architecture from the OEM's point of view will mainly stay at the level of an integration architecture. The detailed development of functions is often left to suppliers; consequently, the OEM will have only a black-box view on these functions, limiting opportunities for global optimization, and deep verification and validation. This places particular importance on the specification of interfaces at the logical level; in particular, the interfaces need to be rich in the sense that they need to convey not only structural information (such as function names and data types) but also behavioral information (§4.3).

Research Challenges. A first step towards accomplishing the desired disentanglement of logical from technical architecture aspects is to consistently think of the system and its subsystems in terms of capabilities rather than in terms of deployment components. A promising aid to that end is the notion of service. Often, service-oriented architectures consist of at least two distinct layers: one *domain layer*, which houses all domain objects and their associated logic; and one *service layer*, which acts as a façade to the underlying domain objects—in effect offering an interface that shields the domain objects from client software. Typically, services in this sense coordinate workflows among the domain objects; they may also call, and thus depend on, other services; and a respective service model that takes into account the specifics of automotive requirements needs to be defined. Enriching domain-specific architecture definition languages with the corresponding abstractions and notations to capture the cross-cutting, coordinating nature of services is also a rich topic of future research [KNP04, AKMP05].

Another research challenge is the management of the various levels of granularity and detail available in a systems of systems engineering project. Because of the complexity and size of automotive systems, having complete knowledge about all subsystems and the overall systems is an illusion. Hence, we need requirements and logical architecture models that can deal with the partiality of information. Again, the notion of service discussed above can be a valuable step into this direction. Services, defined via interaction patterns among roles, provide partial views onto the overall system, albeit in an end-to-end fashion. Composition and combination of services then leads to a composite view of the *relevant* parts of the overall system integration. Exploiting this partiality for tasks such as simulation, verification and validation holds significant promise in complexity management. Of course, to be viable, the service notion has to reflect the combined control- and event-driven behavior spectrum.

Ultimately, combining the aforementioned models into a notion of service- and component-interfaces that includes behavior descriptions and can be shared between OEMs and suppliers, is a long-term research goal at the logical architecture level. Solving this challenge would enable OEMs and suppliers to engage in meaningfully tool-supported exchange of interface models that can support the value-added development support we have alluded to, above.

4.2.2 Technical Architecture

The technical architecture identifies the ECUs, the basic software (operating system and middleware) on them, their interconnection via busses, as well as the partitioning of functions or SW components from the logical architecture onto ECUs. This, of course, requires the identification and definition of software components representing logical entities defined at the level of the logical architecture. The respective partitioning decisions need to be future-proof, because of

the business characteristic of long life-cycles (§2.4): changes in the partitioning are very likely to lead to incompatibilities to legacy systems. An ECU with a new partitioning will usually not work in an already produced vehicle. As backward compatibility is breached, a new branch in the configuration space is opened, or an obsolescence management strategy is needed for the old ECU variant.

The technical architecture also specifies how logical communication is mapped onto technical communication. For instance, the technical architecture specifies how a logical signal is mapped onto protocol data units of the bus system that is used for communication of the deployment components. Due to the specifics of real-time bus systems, this mapping often is a complex issue. For instance, for a signal with hard real-time requirements that will be sent via the time-synchronous Flexray [MHB+01] bus, the decision remains whether the signal should be transmitted in the static segment of bus communication or within the guaranteed dynamic segment.

In the context of upcoming time-synchronous bus systems such as Flexray, a further task in the definition of the technical architecture is to define a bus schedule. This schedule specifies what information is sent in which time slot, and it needs to be coordinated with the task schedules of the ECUs. A close coordination allows to minimize communication latency. On the other hand, it decreases the maintainability of the overall system. If coordination is very close, minor changes in the bus schedule might require changes to all task schedules of the ECUs that are connected to the bus. Currently, first tools for the generation of bus schedules start to be available [D06, P06].

Research Challenges. From the point of view of model based development, the integration of models for the technical architecture and models for bus traffic analysis is highly desirable. A key property of these models is that they abstract communication behavior of hardware, middleware and application software in a stochastic way, focusing on size, frequency of occurrence and timing of the data to be sent.

Furthermore, as already mentioned in §3.2, safety analysis can greatly benefit from model-based development, if further research identifies how those models can be integrated with models for FMECA (Failure Mode, Effects and Criticality Analysis), FTA (Fault Tree Analysis), and also for reliability analysis in general. Of course, the integration with the technical architecture alone does not suffice here. This is because additional information on which functions are affected in which way is needed, i.e. the link to the logical architecture. Similarly, models of the logical and technical architectures can be used as a basis for diagnosis models that help in the localization of faults. A central research question here also is how modeling can help in performing software diagnosis for shipped software, i.e. without the possibility to inject stimuli from outside the system to localize faults. This also requires models for the system environment (the *plant* in control theory terminology).

In vehicle networks that contain both time-asynchronous busses like CAN [Bos91] and time-synchronous busses like Flexray, a further question is which functions are deployed onto CAN-ECUs and which on Flexray-ECUs. While functions with hard-real time requirements obviously are good candidates for Flexray-ECUs, the question remains where the border with the asynchronous world is to be drawn. Usually there remains a lot of communication between the two worlds, but their interface is non-trivial. In particular for the case of signals with soft real-time requirements that are forwarded via a gateway from CAN to Flexray, these signals are not only delayed by the latency when accessing the CAN bus, but also by the latency time that is a consequence of waiting for the next appropriate Flexray time slot. This means that latency from CAN to Flexray either is rather high or that bandwidth is wasted on the synchronous bus. Semantics-preserving deployments of synchronous models on heterogeneous architectures [BCC+04, Rom06], including time-synchronous and asynchronous architectures [HS06], deserve further investigation.

Integration of models of the technical architecture in overall models for systems engineering is a further important topic. With respect to the technical architecture there is a close correlation to models for the flow of electrical energy and geometric models for the placement of wiring and ECUs. Furthermore, a link to cost models (§§ 2.5 and 3.5) is mandatory for partitioning decisions. For instance, with respect to cost, the partitioning is strongly affected by the business choice of which functions are part of every vehicle and which ones are optional. With such an integration, a detailed evaluation of architectural decisions becomes possible. For the integration with models from systems engineering, establishing of a tool chain is highly demanding since for all disciplines good, isolated tools exists, but we cannot assume that their semantics is identical in detail in the model elements they have in common.

For a seamless model-based development, there is a need for models of the technical architecture that comprise all the features offered by a platform to the SW components deployed on it. Therefore, research into a modeling paradigm is needed that supports a kind of (automotive-specific?) layer-based modeling, which makes all relevant platform aspects visible but abstracts from details. As suggested in §3.1 such a modeling of platform/middleware features must provide that some features are system wide, for instance, due to

a common middleware, while others are domain or platform/ECU specific.

4.3 Detailed Behavior Models

Finally, we will have a look at behavior models that specify functionality at a rather detailed level. This is needed for architecture specifications at both the logical and—in refined form—the technical levels. These models come in different flavors. *Existential* models focus on the main system runs in an exemplary manner. In the form of sequence diagrams, they are often used as specifications. *Universal* models, on the other hand, are detailed enough to permit the generation of production code [FGG05, BOJ04], of simulation code that is used for prototyping and hardware-in-the-loop simulations [Spi01], and of test cases [PPW+05].

Except for the generation of production code, all these activities require the development of environment models (with high costs and high potential for reuse). We ignore them here for brevity's sake, and focus on universal models of systems.

In accordance with the heterogeneous nature of automotive software (§2.1.1), the models are continuous, mixed discrete-continuous, and purely continuous (see [HS06] for a complementary perspective).

- Modeling *continuous systems*—more concretely, control algorithms—is common practice and has a long tradition in the automotive domain. In the automotive domain, the most prominent toolset for such models is Matlab Simulink [Mat06,WM95]. The language of *block diagrams* allows the engineer to graphically specify differential equations, with blocks representing operations such as multiplication, integration, or differentiation, and arrows between blocks representing data flow. Block diagrams can be seen as a graphical special-purpose programming language for control algorithms. At the control-theoretic, purely continuous, level there is a huge body of methods for the analysis of properties like robustness, stability, attraction, etc. [Sta04]. Because of the low level of detail, impressively efficient simulation and production code can be generated. This, of course, involves discretization and the respective fundamental problems with the transition from floating point to fixed point numbers.
- *Mixed discrete-continuous systems* exhibit different modes in which they operate continuously, and modes are switched in a non-continuous—i.e., discrete—manner [GKS00]. Approaches to specifying hybrid systems include hybrid automata, hybrid Petri nets, and equation-based approaches. In practice, extensions of the Matlab Simulink languages (Stateflow) are most commonly used.
- *Discrete systems,* as predominantly found in the infotainment domain (§2.1.1), are probably most familiar to software engineers. They are typically specified in one of a plethora of state machine variants. As of today, in contrast to continuous systems, their usage is not commonplace in the automotive industry. When speculating about the reasons, one might want to quote
  1. the lack of convincing tools with excellent production code generators that would allow for roundtrip engineering;
  2. a rather close proximity between genuine C++ code and state machines with C++ as action language on transitions—hierarchical state machines then act as structuring means only;
  3. the closely related problem of choosing appropriate abstraction levels (for continuous models, abstraction takes place by means of language constructs, not deliberate loss of information [PP05,PP04]);
  4. cost issues in cases where models are not used for the generation of production code but as specifications only: two artifacts, model and code, have to be maintained and synchronized;
  5. the necessity to add yet another language and yet another toolset to the existing tool chain for continuous systems;
  6. so far unfulfilled promises as far as the verification of such models is concerned; and
  7. education issues.

It is noteworthy that the goals of code generation and verification are somehow contradictory. The former requires a rather low level of abstraction as embodied by corresponding language constructs whereas the latter usually requires abstraction in the sense of an actual loss of information [PP04]. Furthermore, verification tasks by definition require knowledge of the properties to be verified, and these properties are often not known (these problems are of course not unique to the automotive domain).

The above objections are hard to overcome. Indeed, as far as the automotive domain is concerned, it seems very possible that the benefits of model-based development for discrete systems do not lie in the generation of code but rather in the definition of clear interfaces and the relationships between components (§§ 4.2.1 and 4.2.2). What does appear appealing in this context is the use of behavior models as black-box specifications, serving as communication interface between OEMs and suppliers (§2.2.1; [PP05, AKMP05]). In addition, these behavior models can be used to the end of generating tests [PPW+05], thus facilitating the task of verifying a system with respect to its specification.

Research Challenges. Challenges in the context of *continuous models* include even more efficient code

generators as well as a sufficiently precise "standard" semantics. Such a standard would of course come with the political problems that every standardization of semantics has to face, see the UML. On the other hand, it would allow assessments of the correctness of code generators—as of today, production and simulation code from one product do not necessarily exhibit identical behaviors (we mention TDL [PT06] as a notable exception), and neither does generated code from two different products [SC04]. Because the idiosyncrasies of different code generators are known, the current approach to handling this problem consists of avoiding "critical" constructs, which results in modeling guidelines. Because of the enormous state spaces of continuous systems, their analysis also remains a challenging task. Function blocks are equipped with a multitude of possible parameters in order to cater for different application domains such as automotive and avionics. As a consequence, it is very hard to keep track of all the relevant and irrelevant parameters; the models become unnecessarily complex. A possible solution, and thus a research challenge, is an automotive "profile" for Matlab/Simulink.

Because the classical proof methods from control theory are not applicable, the analysis of *mixed discrete-continuous* systems presents itself as a vast research problem, with only first steps in the understanding of proof methods [Sta04] and in terms of reachability analyses being taken today. The conceptual clarity of time-synchronous languages such as Esterel [BG92] and Lustre [HCR+91] is appealing and might turn out to impact the integration of discrete and discretized continuous systems. Furthermore, the combination of continuous and discrete subsystems into a joint, comprehensive domain model supporting early simulation and validation is missing so far. In particular, this will require a combination of *timed* behavior models of varying degrees of rigidity and *event-driven* behavior models. This combination will be a first step towards integrating time into a general programming model for embedded systems.

Fundamental research challenges in the context of using models both as specifications and source of test cases include the definition of domain- and purpose-specific abstraction levels, tools for push-button generation of tests, and the definition of domain-specific and domain-independent test case specifications. It is unclear if the effort, including synchronization, of maintaining both a model and a piece of code is justified by the resulting quality of systems and test cases.

Embracing all three classes of models, further open research problems include the question of how to derive detailed behavior models from more coarse-grained ones (§4.2), how to map this functionality to software components and possibly different ECUs, domain-specific modeling methodologies, code generators, roundtrip engineering, tool integration, and verification technology.

Finally, empirical investigations into cost effectiveness are needed to assess the benefits of behavior models when used for specification (where component-based engineering might turn out to be the better solution) and test case generation (where setting up better structured test processes might in itself solve a lot of problems). We realize that many of these research challenges are shared with other technical domains.

5. Conclusions and Outlook

Software engineering for automotive systems embraces almost all areas of computer science and computer engineering, and includes all software engineering activities. In this paper, we have characterized the domain of automotive software and highlighted some particularly important research problems, of course without any claims to completeness. Because of the broad scope of our subject it is not surprising that many problems exist in other domains as well. As a consequence, we have taken some care to identify problems that are specific to the automotive realm—which explains why we did not discuss important problems as diverse as continuously changing requirements, timing predictability, usability, portability, design and coding standards, etc. Essentially, the problems we have identified relate to evolution and integration. Today, integration is mainly enabled in an ex-post manner, by testing and changing where necessary. For evolution, this becomes increasingly complicated, a consequence of the huge number of variants and versions that require support. We have indicated how model-based approaches to systems development can help meet the challenges, and provided some particularly relevant research directions in the intersection of model-based development and automotive software systems.

Acknowledgment. The first author would like to thank Manuel Hilty for comments on a draft version of this article. The third author was partially supported by the UC Discovery Grant and the Industry University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2).

6. References

[AKMP05] J. Ahluwalia, I. Krüger, M. Meisinger, W. Phillips: "Model-Based Run-Time Monitoring of End-to-End Deadlines". Proc. EMSOFT, 2005

[Aut06] AUTOSAR consortium: www.autosar.org, 2006

[BBC+01] K. Butts, D. Bostic, A. Chutinan, J. Cook, B. Milam, and Y. Wang, "Usage Scenarios for an Automated

Model Compiler". Proc. EMSOFT, LNCS 2211, pp. 66-79, 2001

[BBJ+05] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein: "AutoMoDe—Notations, Methods, and Tools for Model-Based Development of Automotive Software". Proc. SAE 2005 World Congress, Detroit, MI, April 2005. Society of Automotive Engineers

[BBD+06] M. Bechter, M. Blum, H. Dettmering and B. Stützel: "Compatibility models". Proc. 3rd Intl. Workshop on SW Engineering for Automotive Systems, pp. 5-12, 2006

[BCC+04] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, A. Sangiovanni-Vincentelli: "Heterogeneous Reactive Systems Modeling: Capturing Causality and the Correctness of Loosely Time-Triggered Architectures (LTTA)". Proc. EMSOFT, September 2004

[BG92] G. Berry, G. Gonthier: "The ESTEREL synchronous programming language: design, semantics, implementation". Sci. Comput. Program. 19, 2 (Nov. 1992), 87-152, 1992

[BKP+07] M. Broy, I. Krüger, A. Pretschner, C. Salzmann: "Engineering Automotive Software". To appear in Proceedings of the IEEE, 2007

[BLP04] S. Bühne, K. Lauenroth, K. Pohl, M. Weber: "Modeling Features for Multi-Criteria Product-Lines in the Automotive Industry". Proc. 1st Intl. Workshop on SW Engineering for Automotive Systems, pp. 9-16, 2004

[BOJ04] M. Beine, R. Otterbach, M. Jungmann: "Development of Safety-Critical Software Using Automatic Code Generation". Proc. SAE World Congress, 2004

[Bos01] Robert Bosch GmbH: "CAN Specification Version 2.0", 1991

[CN01] P. Clements, L. Northrop: "Software Product Lines—Practices and Patterns", Addison Wesley, 2001

[DK04] J. Dannenberg, C. Kleinhans: "The Coming Age of Collaboration in the Automotive Industry", Mercer Management Journal 18:88-94, 2004

[D06] Decomsys: "Designer Pro", http://www.decomsys.com/htm/frs/3_flexraydesign_pro.htm, 2006

[FGG05] A. Ferrari, G. Gaviani, G. Gentile, M. Stefano, L. Romagnoli, M. Beine: "Automatic Code Generation and Platform Based Design Methodology: An Engine Management System Design Case Study". Proc. SAE World Congress, paper 2005-01-1360, 2005

[GBB+06] E. Geisberger, M. Broy, B. Berenbach, J. Kazmeier, D. Paulish, A. Rudorfer: "Requirements EngineeringReference Model (REM)". Internal Report, Software & Systems Engineering, TU München and Siemens Corporate Research Princeton

[GKS00] R. Grosu, I. Krüger, T. Stauner: "Hybrid Sequence Charts". Proc. 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000), IEEE, 2000

[HCR+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud: "The synchronous data-flow programming language LUSTRE". Proceedings of the IEEE, 79(9):1305–1320, September 1991

[HKK04] B. Hardung, T. Kölzow, A. Krüger: "Reuse of Software in Distributed Embedded Automotive Systems". Proc. EMSOFT, 203-210, 2004

[HS06] T. Henzinger, J. Sifakis: "The Embedded Systems Design Challenge", 2006

[ISO01] ISO. Road Vehicles – Controller Area Network (CAN) – Part 4: Time Triggered Comunication. Standard ISO/CD 11898-4, International Organization for Standardization, 2001

[Kof05] L. Kof: "Text Analysis for Requirements Engineering". PhD thesis, TU München, 2005

[KNP04] I. Krüger, E. Nelson. K.V. Prasad: "Service-based Software Development for Automotive Applications". Proc. CONVERGENCE 2004, 2004

[KSL+03] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty: "Model-Integrated Development of Embedded Software". Proceedings of the IEEE, January, 2003

[Mat06] The MathWorks, Inc., www.mathworks.com, 2006

[MHB+01] R. Mores, G. Hay, R. Belschner et al.: "FlexRay – The Communication System for Advanced Automotive Control Systems". Doc. No. SAE 2001-01-0676, SAE, 2001

[P06] preeTEC: "TDL Tool Suite". http://www.preetec.com/, 2006

[PP04] W. Prenninger, A. Pretschner: "Abstractions for Model-Based Testing". ENTCS 116:59-71, 2004

[PP05] A. Pretschner, J. Philipps: "Methodological Issues in Model-Based Testing". Model Based Testing of Reactive Systems, LNCS 3472, pp. 281-291, 2005

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, T. Stauner: "One Evaluation of Model-Based Testing and ist Automation". Proc. 27th Intl. Conf. on Software Engineering, St. Louis, 2005, pp.392-401

[PT06] W. Pree, J. Templ: "Modeling with the Timing Definition Language TDL". Proc. Automotive Software Workshop, San Diego, 2006

[Rom06] J. Romberg: "Synthesis of distributed systems from synchronous dataflow programs". Dissertation, Technische Universität München, 2006.

[SC04] I. Stürmer, M. Conrad: "Code Generator Testing in Practice". Proc. GI Jahrestagung (2), pp. 33-37, 2004

[Spi01] B. Spitzer: "Modellbasierter Hardware-in-the Loop Test von eingebetteten elektronischen Systemen". PhD Dissertation, Univ. Karlsruhe, 2001

[Sta04] T. Stauner: "Properties of Hybrid Systems—A Computer Science Perspective". Formal Methods in System Design 24(3):223-259, 2004.

[WM95] R. Weeks, J.J. Moskwa, "Automotive Engine Modeling for Real-Time Control Using MATLAB/SIMULINK". SAE Paper 950417, 1995

[WW03] M. Weber and J. Weisbrod: "Requirements engineering in automotive development: Experiences and challenges". IEEE Software 20 (2003) 16-24

[XBW98] X-by-Wire Consortium, "X-by-wire – Safety related fault tolerant systems in vehicles – final report". Project BE95/1329, Contract BRPR-CT95-0032, 1998

[Zav93] P. Zave. "Feature Interactions and Formal Specifications in Telecommunications". IEEE Computer 26(8):20-28, 1993