A Safety Aware Run-Time Environment for Adaptive Automotive Control Systems

Jelena Frtunikj*
Vladimir Rupanov*
Alexander Camek*
Christian Buckl*
*Cyber-Physical Systems
fortiss GmbH
Guerickestrasse 25
80805 München, Germany
surname.lastname@fortiss.org

Alois Knoll‡
‡Fakultät für Informatik
Technische Universität München
Boltzmannstrasse 3
85748 Garching bei München, Germany
knoll@in.tum.de

***Abstract*—As current studies show, in the last couple of years software functionality of modern cars has increased dramatically. This growth will gradually increase the system complexity, because the new functionality is more and more interconnected. To cope with this process, it is necessary to change the current electrics and electronics (E/E) architecture. An adequate run-time environment (RTE) is the heart of such a new E/E architecture and orchestrates the interaction and communication between the components in such systems. Due to high safety requirements of modern driver assistance, the RTE must also provide built-in safety features. This paper analyses these requirements and derives a set of software modules of an RTE that enforce the safety critical behavior of the entire system. The suggested software architecture can act as a blueprint for other run-time environments that deal with similar requirements. The proposed concept has been integrated in the RTE, that is developed in the RACE project[1].**

***Keywords*-embedded systems, automotive systems, safety-critical systems, middleware, components and reusability, safety, fault tolerance**

I. Introduction

Over the past 30 years, software and electronics have made significant innovations in automotive construction possible: from the anti-lock braking system in 1978 to electronic stability control in 1995 and to the emergency brake assist in 2010. Accordingly, software has expanded significantly, from about 100 lines of code (LOC) in the 1970s to as much as ten million LOC [5].

For a recent study [3], 240 experts worldwide have been interviewed to discover how the automotive electrics and electronics (E/E) architecture will evolve in the next 20 years. The study identified several architectural changes of modern cars influenced by disruptive changes in society and environment. Currently people are getting older, but nevertheless want to stay active as in the past. In order to support these social changes, the new automotive architecture must support autonomous functionality, such as driving [6], [15], parking and new driver assistance systems. Due to high criticality of these features, the E/E architecture implementing them must provide built-in mechanisms to achieve fault-tolerance. Architectural solutions for both hardware and software, which support critical applications, already exist in the industry. Yet they lack the possibility of providing additional services or integrating new components after assembly of the car (through Plug&Play). This missing feature can be found in service oriented architectures (SOA). SOA has been studied in the IT community for years as an architectural pattern [2], and corresponding middleware technologies have been developed. Therefore, we believe that the automotive industry may benefit from these established solutions. Nevertheless, to fulfill the safety-requirements automotive RTE must provide functionality that allows for reliability, safety and security.

This paper describes the challenges (Section II) of the new automotive architecture and its related run-time environment. Section III describes the proposed RTE architecture with special attention to safety and reliability, and how these can be satisfied and integrated into a generic safety concept. In section IV, a summary of development process requirements that need to be taken into consideration when implementing such a RTE is given. The essential components of the proposed concept have been developed and integrated in a demonstrator, which is presented in section V. Section VI compares our approach to available solutions provided by industry and scientific community. Section VII provides the conclusion and summarizes future steps.

II. Challenge: Adaptive Automotive Run-Time Environment

As outlined in the introduction, the novel run-time environment must provide a flexible basis for executing platform-independent high-level automotive applications and support fail-operational behavior. A preliminary requirement to be

[1]Robust and Reliant Automotive Computing Environment for Future eCars, http://www.projekt-race.de/

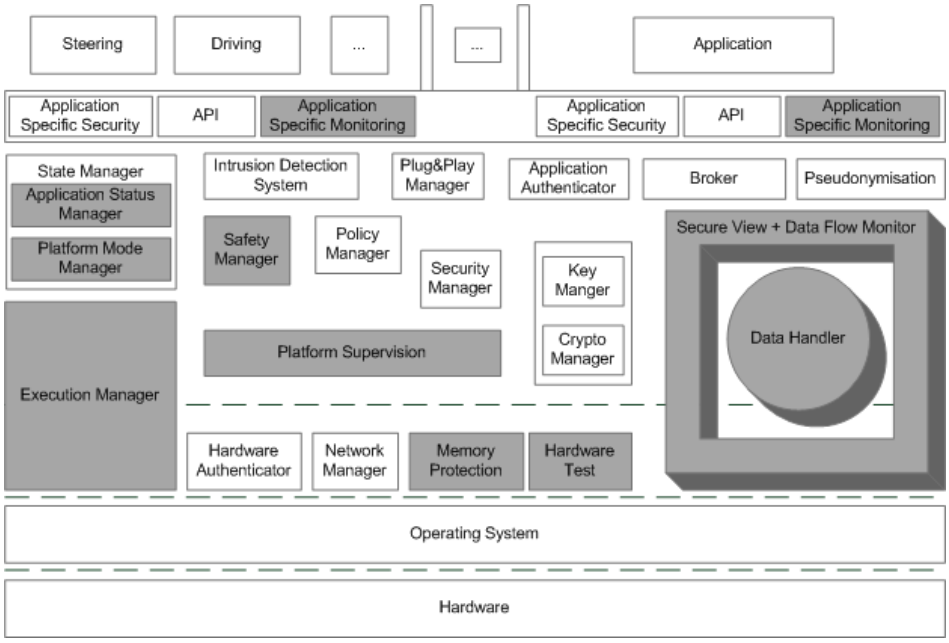

Figure 1. Middleware architecture

able to satisfy these challenges is a hardware architecture providing redundancy and reliability both regarding communication, processing and power supply. The run-time environment must support the execution of components up to the highest safety class ASIL D. Such architecture is suggested in [16]. In this architecture all system components are interconnected by a network system in such a way, that a single-point failure can be tolerated without noticeable degradation. For the remainder of this paper, we assume such hardware architecture as the underlying basis for the discussed run-time system.

Besides mechanisms provided by common run-time systems such as real-time deterministic scheduling and data exchange services, the RTE must provide a set of safety-related mechanisms. These include health monitoring and diagnosis. The architecture should benefit from *early error detection* and a unified error handling. Redundant execution of functions shall be supported by the RTE without any special treatment at application level. A centralized diagnosis unit should allow for *identification of errors* based on multiple possibly disjoint symptoms. *Reconfiguration* services should be available to support graceful degradation and application migration scenarios.

Besides safety requirements, another important aspect pointed out in the study [3] was adaptivity. Via Plug&Play an owner should be able to personalize his car, stay up-to-date by adding new hardware and software components, or upgrade old components with newer software. To support Plug&Play, the traditional message-oriented design should be replaced by a data-centric approach: instead of specifying sender-receiver relationships, the component developers have to specify the component interfaces by a standardized data model. Based on this data model, the RTE shall establish data paths between components.

This late system expansion with components, which influence the safety of cars, must comply with pre-defined quality specifications. To provide flexibility in safety mechanism instantiation and configuration, separation of these mechanisms from application code is required. The main advantage of the data-centric approach is that data redundancy can be exploited to achieve fault-tolerance and improve error detection. If, for example, several sensors provide information about the velocity of the car, detection of an erroneous sensor can be simplified, and the failure of this sensor can be tolerated by using the redundant sensor information.

III. Run-Time Environment Components

The proposed RTE architecture is depicted in Figure 1 and the components that ensure the safety requirements are described below. The RTE data model exploits the data-centric approach to decouple applications from the infrastructure components, as specified by [2]. In this approach, data is not directly delivered from sender to receiver. Instead a publisher announces the available data and a subscriber registers to these publications (Publish-Subscribe, [4]). This approach provides the first step towards enabling Plug&Play. A generic implementation of Publish-Subscribe mechanism requires route calculation at runtime. However, dynamic data routing makes it hard to provide real-time guarantees. To address this issue, the Plug&Play procedure is logically sep-

arated into two phases. At (re-)configuration time (*"Plug" phase*) data paths are (re-)calculated. After (re-)configuration has been completed (*"Play" phase*), static data routes are used, so an inefficient route lookup is avoided.

To represent various information of the car, e.g., velocity or exterior temperatures, different semantic data types (so called topics) are defined. These publications are used for the communication among software components over hardware boundaries.

In the following, we describe in detail the most relevant components of the suggested RTE.

A. Data Handler (DH)

Central component of the RTE is the signal database or *Data Handler*, which stores system data, corresponding safety and quality attributes, as well as information concerning the data flow. All communication between applications is performed via the DH. Therefore, separation of the different application components can be reached and validity of data flow can be checked.

Furthermore, the data handler can restrict the access to the different data elements and therefore enforce safety and security rules. Data elements, which are instances of topics, can only be deleted or created in the *"Plug" phase*. Components are not allowed to delete or add new data elements to the DH. The operations for reading data from and writing data to the DH are atomic and so are not interruptible by other processes, thus these operations are performed exactly once. In case of an operation failure, the operation will not be executed, or the transaction will be reverted, so that no inconsistency will remain in the storage.

It will be even possible to log every operation, such as read, write, and recognition of safety problems for future audits. These records could contain information about date and time of the event, the identified subject, and the outcome of the event. In addition, one could add a mechanism to dump the content of the DHs periodically or at shut down to a persistent storage, e.g., hard disk, to allow error recovery. This dump can also be checked for integrity and consistency at specified time periods.

B. Data Flow Monitor (DFM)

Data Flow Monitor is an important requirement to critical application as defined by [10]. The DFM provides mechanisms for plausibility and consistency checks on data, such as a validation of data against a pre-defined range of expected values. Only valid data is forwarded to applications. DFM also hides data redundancies from the applications by merging data from different sources, which represent same information. Beyond semantic checks, checking of data correctness can be performed by applying data hardening strategies in a transparent way, such as duplication with encoding.

C. Scheduling and Execution of Functions

Safety-critical systems and their corresponding fault-detection mechanisms require deterministic execution of calculations [10]. The *Execution Manager* (EM) implements time-triggered execution of applications and RTE components in a cyclic manner with a pre-defined minor cycle length, e.g., 10 ms. A major scheduling cycle containing multiple minor cycles allows scheduling functions with longer periods. A schedule set consists of multiple static scheduling tables. Typically one scheduling table corresponds to a single platform mode. Activation of a specific schedule is triggered by a mode change.

An application component can have at least one alternative implementation specified, which enables configuration of a recovery block. To free resources for more critical applications in case of graceful degradation, passivation of application components shall be supported.

The schedule configuration can be modified by privileged RTE components at runtime to enable adaptive behavior (including Plug&Play). Such changes are performed on a schedule copy and applied in a transactional manner, which guarantees a consistent schedule set. If reconfiguration cannot be performed in at least one schedule of the required set, the changes are not feasible, and the transaction is rolled back. The switch from an old to a new configuration is performed at the end of a major cycle to eliminate any unwanted side effects.

D. Health Monitoring (HM)

The health monitoring concept is built around various monitor components, tests, and plausibility checks, which report their status through publication of indication messages. These messages are instances of a special topic type named *"MonitoringIndication"* and are processed by the *Platform Supervision* service. The overall structure of health monitoring subsystem is depicted in Figure 2. Below, first we describe in detail the different types of health monitoring components and after that we give an overview of the Platform Supervision concept whose main task is the processing of the results produced by the HM.

The concept proposes the following health monitoring components:

An *Application-specific monitor* is a specific supervisor developed for a concrete software unit independently based on function specification. Typical examples are plausibility checks developed by an independent team. It has read access to input data and internal state of the function and performs checks to ensure state consistency and transition correctness. If a function is specified as a state machine, generation of concurrent state checker logic is possible, as given in [18], [13]. Otherwise, a set of constraints on state and inputs of the function needs to be specified by a domain expert or is generated from assertions in the function code, as specified by [9]. One specific class of such monitors are control flow

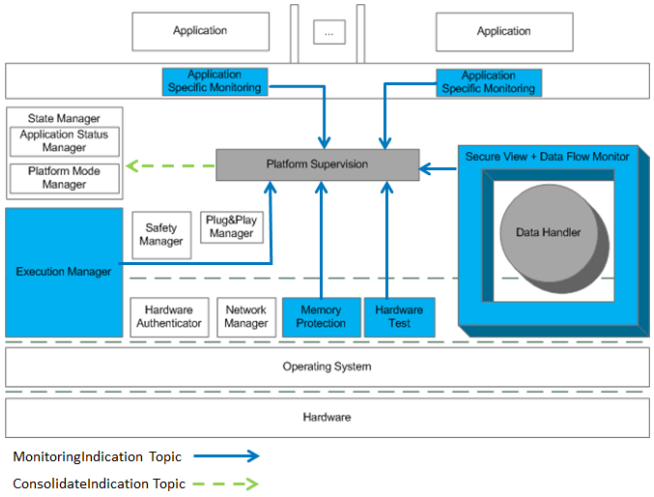

Figure 2. Example structure of Health Monitoring

monitors, which can compare series of checkpoints during module execution with allowed set of control flow paths.

Monitoring of application data flow is provided by DFM and mostly deals with data exchanged over the network. Design diversity is supported by configuration of components, which perform correlation or comparison over function outputs. In this way diverse implementations of the same function run in parallel, and the outputs are fused into a single one.

Application-independent protection mechanisms perform local protection of data and control flow, such as transparent monitoring of memory blocks with a hash function, replication of memory blocks or repeated execution of functions. Memory and time partitioning belong also to this category. Along with that to support external monitoring facilities, as required by [10], corresponding components can be instantiated within RTE (e.g., heartbeat monitoring or interface to external watchdog).

Global tests include, for instance, periodic CPU tests, validation of ROM/Flash checksums, memory tests, and hardware-assisted built-in self-test for arbitrary hardware components.

Internal consistency checks are executed on RTE component boundaries and within RTE components. Similar to *application-specific monitors* these checks can be generated from state-machine descriptions or assertions in code. Assertions are usually inserted at design time to allow detection of inconsistent states within RTE components. In production code the generated checks are used to detect systematic latent faults in software.

E. Platform Supervision (PS)

The *Platform Supervision* service handles hardware (random) and software (systematic) faults, as well as illegal access in combination with the *Security Manager*. Its main functions are: reception of *MonitoringIndication* messages from different components, aggregation and inference of these indications to produce *ConsolidatedIndications* (Figure 2). The latter set is then synchronized with other nodes in order to achieve a consistent view on the system state as a whole. *ConsolidatedIndications* are then updated and made available to the state management components. Systematic errors in specific functions are signalled to the *Application State Manager* service, which performs necessary recovery of the function. Detected faults currently classified as random hardware faults or RTE systematic faults are passed to the *Platform Mode Manager* and also accumulated locally to achieve statistically sound diagnosis of intermittent faults. The process of fault detection, consolidation and reaction and the corresponding mapping to RTE components is depicted in Figure 3.

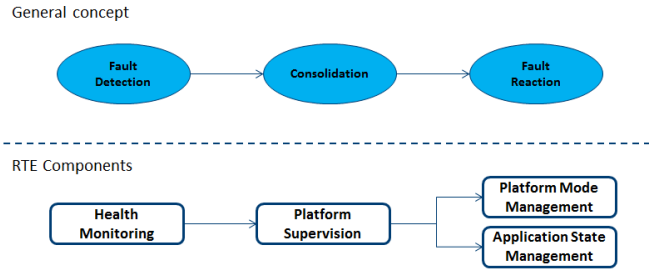

Figure 3. Separation of fault detection from fault handling

F. Platform Mode Manager (PMM)

PMM monitors the presence and triggers reconfiguration of computing nodes in the network. It supports various *platform modes*, such as self-test, startup, integration of new nodes into the core, and self-isolation. PMM controls those platform modes and it triggers changes of the schedule. The new mode is calculated in each cycle based on the current platform mode, ConsolidatedIndications published by PS and the reconfiguration events.

G. Application State Manager (ASM)

ASM functionality is based on *application states*, which are computed based on ConsolidatedIndications published by PS. Application state reflects the state of an application or an application cluster[2] and the quality of its execution on all nodes.

In case redundancy of applications is applied, ASM aggregates application states across all network nodes. ASM also performs preconfigured recovery actions (restart, migrate, or shutdown) triggered by application or application cluster state changes. Two recovery strategies for application clusters are supported: a recovery action that handles erroneous

[2]An application cluster is a group of applications that share software components and belong to the same fault containment region.

applications only, and recovery action that considers the entire cluster.

H. Safety Manager (SM)

Safety Manager is a specialized configuration component, which is triggered by the Plug&Play Manager[3] during instantiation of a new safety-critical application. Based on safety related information delivered with the component, the SM will execute the following steps: During the *"Plug" phase* SM is responsible for: (a) performing a lightweight run-time safety analysis in order to check early the possibility of adding a application to the local node, (b) checking the feasibility of extension of local HM services in order to support adding a application, (c) performing necessary final configuration of application-independent monitoring services, such as memory protection, before instantiating the application, (d) ensuring correct configuration of ASM and PMM to support fault-tolerant execution of the application, and (e) reaching consensus with other nodes ready to deploy the application.

When adding *safety-critical* function into a system, the concept proposes the following sequence:

1) Minimum required degree of redundancy Red_{min} is explicitly specified during installation of the application within safety information. If execution with Red_{min} allows reaching the required safety level $ASIL_{req}$, the SM accepts it and proceeds with the configuration. To check this, a lightweight qualitative and quantitative safety analysis is performed, and the result of the analysis is compared to the requirements for $ASIL_{req}$. Otherwise, a higher redundancy degree is selected.
2) The set of health monitoring services for the application is determined. For example, if the application is supplied with a function-specific application monitor, and its diagnostic coverage is already high, then there is no need in scheduling additional tests to guarantee consistency. Otherwise mechanisms are selected based on requirements to residual failure rate and available resources.
3) In a redundant execution scenario, additional network paths and DFM component instances need to be configured before completion.
4) Consensus must be met with other nodes ready to execute the component with the same degree of redundancy.
5) Reconfiguration is completed when all nodes ready to execute the component are synchronized in "ready to start" state. SM completes state manager configuration, and signals Plug&Play Manager that the function is ready to be started.

[3]The Plug&Play Manager is responsible for calculating new configurations if new components should be added. In case a valid configuration can be calculated, new components will be integrated into the system.

IV. Quality Requirements to the Development Process of the Run-time Environment

According to ISO 262626, the development process of a run-time environment as proposed above, shall consider techniques that satisfy the requirements on development of ASIL D systems. Even though the techniques described below are not the focus of this paper, their importance should be taken into consideration during development.

1) Software Quality Control: Some of the necessary techniques [10, part 6] that ensure high quality of the development process are:

- enforcement of low complexity through well defined software architecture;
- coding and modelling guidelines, including naming conventions;
- usage of language subsets (e.g., MISRA C [12]);
- usage of design principles, such as limited use of interrupts, scheduling properties, etc.;
- application of design for testability concept (not only at unit level but also on system level) e.g. fault-injection, back-to-back testing;
- utilisation of test coverage metrics that state the completeness of tests with metrics such as branch coverage and MC/DC (Modified Condition/Decision Coverage).

The architectural concept described above does not only allow application of the mentioned techniques, but also provides some benefits through, e.g., DH logging capabilities, which allow in-field monitoring.

2) Requirements to Tool Certification for Specified RTE: To support configuration of RTE and applications, configuration tools are required. Tool confidence level (TCL) 2 [10, part 8] should be reached by such tools. This means that tool development process should be evaluated, and an extended verification and validation workflow should be developed to check the output of the generator against expectations. An example technique is to generate testbenches for the configuration generated by the tool, using two separate and independent code generation paths or templates.

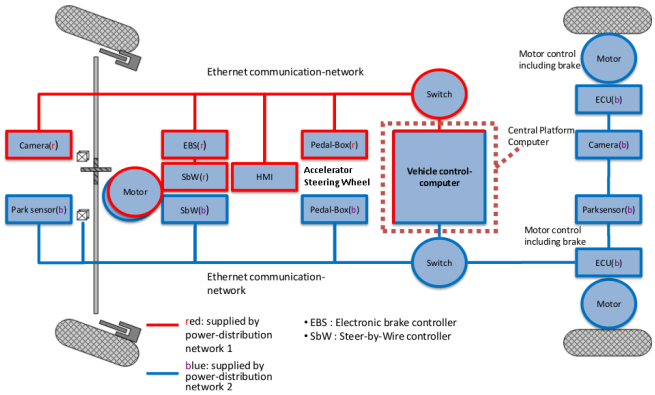

Figure 4. RACE system architecture

V. Experimental Evaluation

To evaluate our RTE architecture, the essential components were combined and tested firstly in a simulation environment and then we integrated them into one demonstrator.

The "Revolution" car prototype is set up in the context of RACE project. The essential features of the proposed runtime environment together with the operating system and drivers, run on all vehicle control computers of the central platform computer (Figure 4).

VI. Related Work

A. AUTOSAR

The AUTOSAR standard [1] describes a platform which allows implementing future vehicle applications and minimizes the current barriers between functional domains. AUTOSAR maps functions and functional networks to different control nodes in the system, almost independently from the associated hardware. Therefore, AUTOSAR introduced a RTE, which implements together with the operation system, AUTOSAR COM, and other Basic Software Modules the concept of Virtual Functional Bus (VFB). The VFB interface realizes the communication between AUTOSAR software components, thus the RTE encompasses both the variable elements of the system infrastructure as well as standardized services. The communication in AUTOSAR can be categorized by sender-receiver, message passing facility, or client-server, which provides function invocation. Communication is provided not only on a task basis (intra-task and inter-task) of the same partition, but also as a paradigm to exchange data between partitions and ECUs. Partitioning was introduced in version 4.1 of AUTOSAR [1] in order to enhance safety and support ISO26262 [10]. AUTOSAR applications and RTE are configured during design phase and then deployed on the ECU. Since AUTOSAR does not provide interfaces enabling Plug&Play, the whole system must be replaced when integrating a new functionality. This is not necessary in our system, so our approach can be seen as an extension to AUTOSAR.

In [11] an approach to bring safety mechanisms complementary to AUTOSAR is described. The goal of the approach is to limit influence to the target middleware and OS services, and to use certain platform interfaces as "sensors" for the defense software system, which in turn through "actuator" interfaces influences the behavior of the system. This allows integration of described strategy not only into AUTOSAR systems, but also implies high variability of the corresponding implementation. For embedded systems the overhead resulting from glue layer between the actual system and the defense system might be unacceptable.

B. SOA in the Automotive Domain

SOA systems provide Plug&Play functionality and enable integration of new services into existing systems. Different car manufactures started to integrate the SOA approach into their automotive systems.

One example is SOA for Diagnostics that grants access to the diagnostic information by a special crafted application through a wireless interconnection. Diagnostics helps manufacturers to find and understand faults in the system by accessing the Onboard-Diagnostics version 2 (OBD-II) port with a specific connector and software. Manufacturers improved the access to the diagnostics memory by wireless connection through the infotainment system. An example is given in [8]. This solution is only used to acquire diagnostic information and to install firmware updates for selected components. Furthermore, standard SOA approaches do not take into account safety requirements. In contrast, our approach offers the possibility to dynamically upgrade the whole system with new functionality, such that safety-critical functionality remains correct.

Another example is SOA in the Infotainment domain, as given in [7]. There, the main objective is the installation of applications, download of media streams, or usage of services outside the vehicle. Compared to our approach, infotainment systems and services are not safety-critical.

C. Adaptive Systems in Automotive Domain

There are numerous approaches to integrate different aspects of adaptivity in the real-time systems domain. However, most researchers from the adaptive systems community treat safety as one of many properties of the real-time systems, as noted by [14]. Many researchers try to adapt strategies of the high performance computing in the embedded domain, e.g., for storage [17]. This improves the reliability of the distributed data storage and allows certain self-healing behavior, but adaptivity in those approaches is limited by a specific application and node set.

A practice-oriented approach is presented in [14]. The authors share their views on safety certification perspectives for open adaptive systems, and present their approach, which is based on conditional safety certificates. Conditional certificates describe preprocessed technical safety requirements with a domain-specific language and transfer these requirements to the runtime safety model. Their approach is initially oriented at assisted adaptive living or car2car communication and similar loosely coupled systems. Our approach also uses runtime safety analyses before it instantiates new functions dynamically, but is focused on automotive control systems.

VII. Conclusion

This paper describes an architecture blueprint of a future adaptive automotive RTE. The essential challenges and expectations of the new automotive architecture and its related RTE have been summarized. To tackle these challenges a proposal for an RTE architecture is presented with special focus on safety and reliability. The concept has been validated by simulation and developed demonstrator.

Security as an additional requirement for such systems is also considered in the proposed RTE concept, as both safety and security can cooperatively use many mechanisms. The potential for optimization of the new E/E architecture to reach high cost efficiency will be explored in future. Compatibility with existing development processes and supply chains is also to be considered.

ACKNOWLEDGMENTS

The work presented in this paper is partially funded by the German Federal Ministry of Economics and Technology under grant no. 01ME12009 through the project RACE.

REFERENCES

[1] AUTOSAR Group. AUTomotive Open System ARchitecture (AUTOSAR) Release 4.1, 2013.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2 edition, Apr. 2003.

[3] M. Bernhard, C. Buckl, V. Döricht, M. Fehling, L. Fiege, H. von Grolman, N. Ivandic, C. Janelle, C. Klein, K.-J. Kuhn, C. Patzlaff, B. Riedl, B. Schätz, and C. Stanek. *The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future, Summary of results of the "eCar ICT System Architecture for Electromobility" research project sponsored by the Federal Ministry of Economics and Technology*. ForTISS GmbH, March 2011.

[4] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[5] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[6] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel. Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. *Int. J. Rob. Res.*, 29(5):485–501, Apr. 2010.

[7] M. Eichhorn, M. Pfannenstein, D. Muhra, and E. Steinbach. A soa-based middleware concept for in-vehicle service discovery and device integration. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 663–669, 2010.

[8] G. Gehlen, E. Weiss, and A. Quadt. Service oriented middleware for automotive applications and car maintenance. In *Proceedings of the 1nd Workshop on Wireless Vehicular Communications and Services for Breakdown Support and Car Maintenance*, pages 42–46, Nicosia, Cyprus, Apr 2005. RWTH Aachen University.

[9] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] International Organization for Standardization / Technical Committee 22 (ISO/TC 22). ISO/DIS 26262 - Road vehicles. Functional safety. Technical report, ISO, Geneva, Switzerland, November 2011.

[11] C. Lu, J.-C. Fabre, and M.-O. Killijian. Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach. In *ETFA*, pages 1–8. IEEE, 2009.

[12] MISRA (Motor Industry Software Reliability Association). MISRA C3: Guidelines for the Use of the C Language in Critical Systems, 2013.

[13] K. Mohanram and N. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 893–901, 2003.

[14] D. Schneider and M. Trapp. A safety engineering framework for open adaptive systems. In *Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on*, pages 89–98, 2011.

[15] G. Silberg and R. Wallace. Self-driving cars: The next revolution. Technical report, KPMG, 2012.

[16] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll. Race: A centralized platform computer based architecture for automotive applications. In *Vehicular Electronics Conference (VEC) and the International Electric Vehicle Conference (IEVC) (VEC/IEVC 2013)*. IEEE, October 2013.

[17] M. Zeller, S. Grosse, D. Eilers, and R. Knorr. Fail-safe data management in self-healing automotive systems. In *Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on*, pages 24–29, 2010.

[18] C. Zeng, N. Saxena, and E. McCluskey. Finite state machine synthesis with concurrent error detection. In *Test Conference, 1999. Proceedings. International*, pages 672–679, 1999.