Automating Future (Function-) Updates

Mischa Möstl, Johannes Schlatow, Rolf Ernst, Institute of Computer and Network Engineering (IDA), TU Braunschweig

Marcus Nolte, **Markus Maurer**, Institute of Control Engineering (IfR), TU Braunschweig

Abstract

Recent trends in the automotive domain show a fast development towards automated driving functions with growing complexity that comes along with increasing computing power of automotive platforms. Due to the long lifetime of automobiles compared to the innovation cycles of functionalities, in future, new functionality will have to be added by software updates. However, performing such updates on a safety-critical system requires a methodologically safe approach. In this paper, we outline how this is tackled by the DFG research unit "Controlling Concurrent Change" that formalizes the integration part of a V-model like development process aiming at in-field integration. As a result, the platform is equipped with a self-protection mechanism against harmful updates and self-awareness capabilities w.r.t. environmental changes.

---

1. Introduction

The automotive domain is currently driven by the development of automated driving functions. The deployment of new and more sophisticated functions therefore becomes a competitive factor. As these often only require a software change, it is obviously desirable to deploy these in form of (automated) in-field function updates. Moreover, as we can observe a trend towards central high-performance platforms, such as *NVIDIA DRIVE PX* [8] or *Audi zFAS* [1], in future, new functionality can easily be added by a software update rather than requiring another ECU. However, the increasing complexity of upcoming automotive software systems also raises the likelihood that such an update influences the safe operation of the vehicle. Automatically managing software updates in safety-critical systems therefore requires a methodologically safe approach, which is subject of the research unit *Controlling Concurrent Change (CCC)*.

Automotive E/E systems are typically designed following the V-model process in which all requirements are specified on system level and subsequently refined to sub-systems, modules, and individual functions. After implementation of software and hardware (at the

bottom tip of the V), this process is followed by corresponding integration and tests throughout the design levels in reverse order (right branch of the V).

The goal of the CCC project is the automation of this integration process while leaving the design of individual functions in the lab. The automation ultimately allows changes and updates to be performed in field without the need of extensive (lab-based) testing. For this purpose, contracting mechanisms are researched in order to apply formal methods that can provide guarantees about critical system properties.

In the remainder of this paper, we first elaborate on our architectural approach and mechanisms that enable the automated management of function updates. After that, we describe how we implemented and applied this in our automotive use cases before we discuss how our approach leads to self-protection and self-awareness capabilities of embedded systems.

2. Architectural Approach

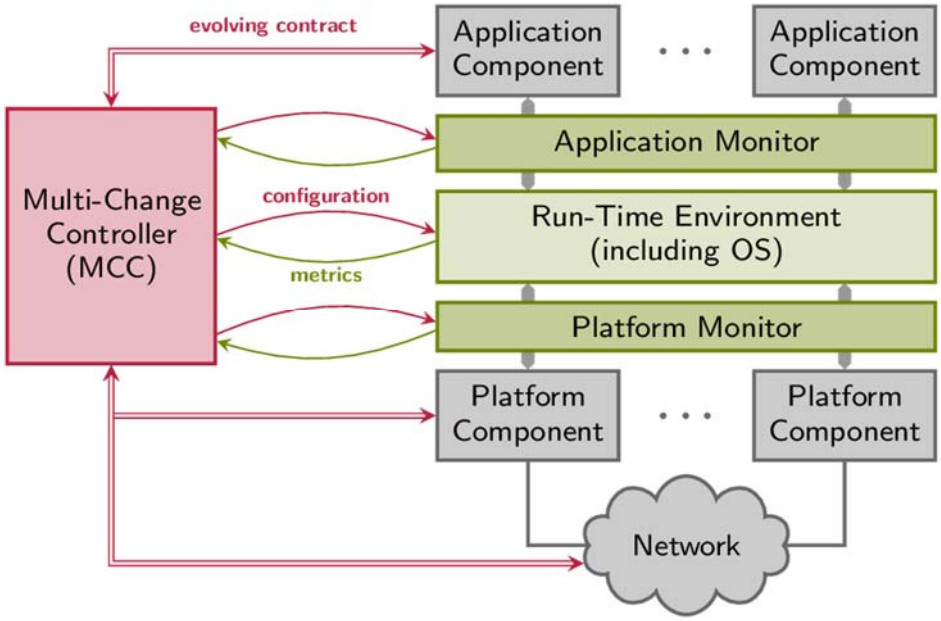

Figure 1: CCC architecture comprising a model domain (red), an execution domain (green) as well as changing application/platform components (gray).

In order to equip a platform with safe change capabilities, we take an architectural approach (as shown in Figure 1) that comprises two segregated domains: the *model domain* and the *execution domain*. Both domains implement mechanisms that contribute to our overall goal. The execution domain is deployed on several (networked) platform components and provides the run-time environment (RTE) including the operating system (OS), which are both required for hosting several applications/functions. This domain not only executes the actual

applications, but also includes monitoring capabilities that a) act as protection layers against harmful behavior and b) extract metrics that enable planning of future changes and optimization. By this, the execution domain provides a solid foundation for the application of formal methods within our model domain, which implements the contract-based deployment and integration of software updates. It is therefore important to close the gap between the model domain, which must make assumptions about the actual behavior, and the execution domain, which should enforce these assumptions or at least detect harmful deviations.

In the following we detail both domains and sketch the resulting development flow.

2.1 Model Domain

The model domain is an essential part of the CCC architecture as it plans and controls the integration process by ensuring that any prospective change passes all necessary acceptance and conformance tests. This domain is implemented by the *Multi-Change Controller (MCC)*.
This approach requires sufficient *analysis methods* to test and evaluate certain properties as well as sufficient *input data* to perform such an analysis. We collect the input data and properties to be evaluated in so-called component contracts which specify these in form of assumptions a component makes towards the platform and run-time environment (RTE) as well as guarantees it can provide given that the assumptions are fulfilled (see *Development Flow & Contracting Language*). More specifically, we take a multi-viewpoint approach to contracting as presented in [5], which allows specifying and verifying different concerns in separation.

Contracts allow combining several components with compatible contracts such that a desired functionality is achieved. Together with sufficient formal analysis methods, it is guaranteed that the composed system satisfies all contracted requirements if all tests are passed. In particular, this process is suitable for uncovering unwanted side effects when multiple applications share a platform, which is especially crucial for applications with heterogeneous safety requirements. For a certain application a guarantee in the form of a best-effort fulfillment of requirements can be satisfactory (while not necessarily being safe), while other applications demand hard bounds and freedom from interference as required by safety standards [6].

Any update is modeled in form of a change request, e.g. contract modification or new functionality accompanied by a contract, on which the MCC is invoked. It then generates possible configuration candidates that undergo several admission tests to check whether all functional and non-functional requirements are satisfied. This can be e.g. a latency constraint that can be verified by compositional performance analysis of the composed system [11].

Once a configuration candidate was admitted, it is applied by the MCC to the execution domain. This may also require a reconfiguration of monitoring mechanisms to guarantee the adherence of the components' implementation to their contracts where necessary, which is explained in the following section.

2.2 Execution Domain

As mentioned above, the execution domain plays an important role in our contract-based approach. This is mainly due to the fact that (hard) guarantees and assumption formulated in the model domain must hold (at any time) during the actual execution of the system. I.e. the model domain relies on the properties that can be given or be enforced by the execution domain.

In order to apply the contract-based design we therefore resort to microkernel-based systems, which already provide a solid foundation for a safe, reliable, and secure platform sharing. More specifically, our execution domain is based on the Genode OS Framework that provides a component-based run-time environment in which components are executed in isolated address spaces and using explicit (service-oriented) communication interfaces. This framework can further be deployed on top of several microkernels with different properties. Its strong isolation of application components reduces mutual interference (e.g. by fault propagation) and allows for fine-grained access control while still providing a high degree of flexibility required for in-field changes. The resulting RTE is a trade-off between very flexible monolithic kernels and restrictive statically-configured systems. Between two updates, the execution domain therefore operates without any participation of the model domain and resembles a statically-configured system. Only when a change request was admitted by the model domain, a (partial) reconfiguration of the execution domain is performed in order to apply the change.

Nevertheless, not every property is guaranteed by design of the RTE. In order to still provide those guarantees, certain parameters of the system can be run-time monitored by application and platform monitors. Those monitors may supervise the execution time of a component or police the access to network resources [4] for instance. By augmenting the system with run-time monitoring, it can be asserted that the execution domain behaves as expected by the model domain. In addition to the enforcement of parameters (e.g. for critical functions), monitoring can also be used for providing feedback to the model domain so that the MCC can refine the models to the actual execution behavior (e.g. for non-critical functions).

2.3 Development Flow & Contracting Language

With our architectural approach, we aim at the following development flow. Applications are developed using model-based design tools (e.g. MATLAB/Simulink as explained in “Use-cases”). These tools already describe particular aspects of the contracts (e.g. interfaces, data dependencies) but need to be augmented with properties from other viewpoints (e.g. timing requirements) depending on the application domain and function. This can either happen by manual annotation (within the design tool) or by automated post-processing. For instance worst-case execution times can be added to the contracts by using state-of-the-art analysis tools. Up to this point, this process resembles the left branch of the V-model enriched with contracting information.

The component contracts are specified in our contracting language [5]. Along with the pre-compiled component binaries, they are loaded onto the platform prior to the change request. The contracts are stored by the MCC within a *contract repository*. After that, the automated integration process (i.e. the right branch of the V-model) can be triggered by a change request. When processing this request, the MCC queries the available components and their contracts from the contract repository by using a parser for the contracting language. The MCC then invokes the so-called analysis engines that implement the above mentioned admission and conformance tests. The different analysis engines may use this parser as well in order to acquire additional information from the contracts.

3 Use Cases

In the CCC project, the Institute of Control Engineering contributes two experimental vehicles that act as demonstrators for the developed runtime environment: A 1:5 model vehicle (**M**odul**a**r **X**-by-Wire, MAX) and a full-scale x-by-wire vehicle (MOBILE). Both vehicles currently act as platforms for research in E/E-systems and vehicle dynamics. MOBILE features four close-to-wheel electric drives ($4 \times 100\mathrm{kW}$, as well as individually steerable wheels, and electro-mechanic brakes [3]. A similar actuator topology has been implemented for MAX. Both vehicles feature a FlexRay communication backbone for inter-ECU-communication and additional CAN bus interfaces, which are for instance used for communication with sensors and actuators in the full scale vehicle.

The ECUs which are responsible for vehicle control are programmed in a customized MATLAB/Simulink tool chain. Combined with detailed vehicle-dynamics models, the tool chain serves as a means to establish a rapid-prototyping process for vehicle control algorithms. The goal here is to develop algorithms from an idea over simulation and small-scale verification in MAX to full-scale validation in MOBILE (cf. Figure 2).

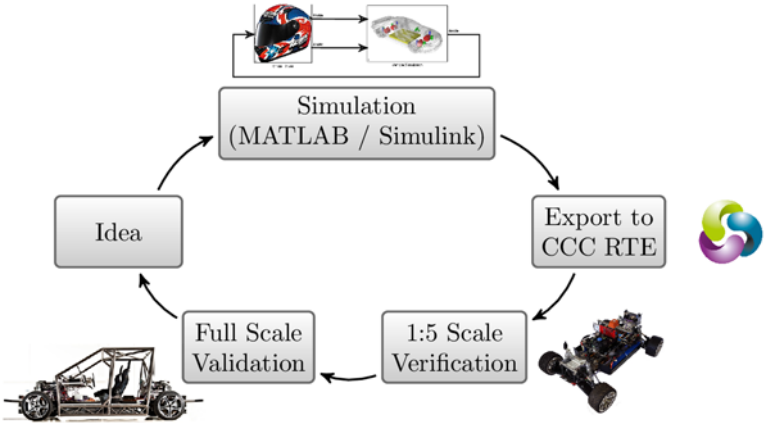

Figure 2: Customized MATLAB/Simulink tool chain with MAX and MOBILE.

For the CCC project, this tool chain has been extended to support the export of Simulink subsystems as software components runnable in the CCC framework (see *Control Layer*). Apart from the mere export, possibilities have been created to apply metrics from object oriented programming (i.a. cohesion, Law of Demeter), to Simulink models in order to support the developer in the creation of reusable components and well-defined interfaces.

In the scope of the CCC project, the applications implemented on the research vehicles can be divided into *control applications* (referred to as “Control Layer” in the next sections) and applications which implement *automated driving functions* based on environment perception algorithms (referred to as “Rich Application Layer” in the next sections). For the operation of the vehicle, specific ECUs running the CCC middleware are connected to a central gateway controller of the vehicle and can be switched off, if this controller detects unexpected deviations from the behavior of the proven-in-use legacy controllers.

3.1 Control Layer

As the development of vehicle-dynamics control algorithms is a major purpose of the research vehicle MOBILE, a set of control applications has been chosen to demonstrate the applicability of the CCC approach in the automotive domain. For this purpose, a cruise-control, as well as a force-feedback and a stability-control application for the x-by-wire vehicle have been selected. These control applications are designed to be updatable using the MCC.

The *cruise control* serves as a very basic example with soft real-time requirements, because of the rather inert lateral dynamics of the research vehicle. It is implemented in three different variants. Besides a basic variant, two extensions feature a speed-limit detection and a vehicle-to-vehicle component, both imposing external limits on the controlled velocity of the vehicle. If active, the vehicle’s velocity is controlled to the minimum of the received limit and the desired velocity of the driver.

The *force-feedback application* is responsible for providing haptic feedback at the steering wheel about the road surface to the driver. It serves as an example for a more time-critical application, because of the higher dynamics of the feedback actuator at the steering wheel. Updates of the application are limited to parameter updates so that the desired strength of the feedback can be made configurable.

The *stability-control application* keeps the vehicle controllable for the driver, performing yaw-rate and slip control. As it needs to be able to control the vehicle at the limits of handling, the controller is sensitive to varying cycle times. The application is intended to provide updatable control strategies, ranging from yaw-rate control by differential braking to force-allocation strategies utilizing all available actuators in the vehicle.

One challenge in the project scope is to derive suitable contracts in order to support the automated integration of components in control applications. Typically, in the control-engineering domain, requirements are formulated as functional requirements in terms of control- quality metrics (e.g. in terms of the control response). These functional requirements must then be transformed to non-functional timing requirements for the integration in embedded systems. Recent research [7,12] has shown, that often there is little common ground between the domains of control and real-time engineering. On the one hand, this can result in over-engineered systems, because of overly-strict formulated timing requirements in the control-engineering domain, as detailed information about the robustness of the system against varying sample times is not always available. On the other hand, this missing common ground can lead to expensive reiterations of the design process if the integrated system cannot meet the required control-quality constraints.

In order to facilitate communication between both domains additional extensions to our MATLAB/Simulink tool chain have been created. Similar to an approach presented in [7], Simulink subsystems can be activated asynchronously in order to evaluate the influence of timing behavior on control performance during simulation. In this way, a design-space exploration or sensitivity analysis for a control system can be performed in order to identify components which are particularly sensitive e.g. to input data delay. The obtained timing requirements at the component's interfaces can then be annotated in the Simulink subsystems. The addition of annotations for control-quality metrics to a component interface also enables monitoring of the system performance during runtime (see *Self-awareness for applications*).

3.2 Rich Application Layer

With regard to automated driving functions, a use case in the form of automated obstacle avoidance will be implemented in the experimental vehicle MOBILE. This is basically a further

extension of the stability-control use case (cf. *Control Layer*). In general, stability-control systems only steer the vehicle into the direction given by driver input at the steering wheel. However, as the driver does not always perform safe steering maneuvers, particularly in critical driving situations such as fast obstacle avoidance, this extended stability control will follow a safe pre-planned trajectory instead of following a potentially unsafe path steered by the driver. For this trajectory-following stability control, data is acquired from multiple environment sensors in the form of three lidar scanners, a radar sensor and a camera. This data is then used to create a map of the static environment, which provides the basis for a model-based trajectory planning utilizing all actuators (particularly all-wheel steering) for maximal maneuverability.

Compared to the control applications which consist of only a few different components, system complexity increases when many software components for environment perception are added to the system. Also monitoring of software components in a sensor-data processing chain is far more challenging than monitoring the performance of control algorithms, as the metrics needed for monitoring are not that easy to formulate. Thus one challenge in the project scope is to gather suitable performance metrics for tasks which are executed by the automated vehicle.

4 Discussion

With the overall architectural approach and use cases in mind, we can now elaborate on the benefits of the approach. In particularly, we focus on the monitoring which is an important corner stone by enabling enforcement and providing feedback.

4.1 Monitoring as Self-Protection Mechanism

The architectural approach in CCC, with monitoring as a joining link between execution and model domain, allows the MCC to gather performance data of the currently executing configuration. This enables the MCC to perform self-reassessment of the current configuration based on the available models and contracting information as well as the data gathered by the monitoring mechanisms. The CCC platform is therefore capable of self-organization, i.e. finding a system configuration based on the current state plus incorporating an update or reorganizing components in a more optimal configuration. Moreover, as the main goal, it is capable of protecting itself against change requests that would jeopardize the correct execution of functions on its platform.

This is achieved in two steps; first of all the model-based admission checks need to be passed in order to release a new configuration in the execution domain. Passing all relevant admission checks is only possible if a contract supplies sufficient contracting information, such that the

model analysis guarantees a safe platform sharing. Safe in this sense refers to the fact that no assumptions of other (possibly higher critical) components is violated. Furthermore, the model-based analysis of an update is capable of delivering the parameters that need to be enforced on an update's components to guarantee non interference for other components. This augments the execution domain's monitoring capabilities since these can be configured with this data and set to enforce these parameters.

Nevertheless, self-protection is a challenging task as it requires that the enforcement mechanisms configured by the MCC need to respect the viewpoints' analysis engines and all other functional and non-functional requirements of components that are currently instantiated. The key aspect of the self-organization and protection mechanisms is that the MCC provides guidance and control for this process such that any design decision made during the automated integration process still undergoes the formal admission tests.

4.2 Monitoring for Self-Aware Applications

Not only the platform or RTE need to be aware of the current state of operation, but also the functions implemented in software components need to register if they are not performing according to their specification.

For the control-engineering domain, in practice, the design-space exploration or sensitivity analysis mentioned in *Control Layer* is probably only feasible for safety-critical systems, as each corner case must be covered. However, also a rough estimate of typical operating conditions enables the control engineer to formulate timing requirements for the tested operating conditions. For those systems where a detailed sensitivity analysis is not performed we propose a monitoring concept. By integrating a monitoring component based on an ideal system model, control performance under ideal timing conditions can be predicted. If the actual control performance of the system lies outside of predefined bounds, a misbehaving subsystem can be detected and countermeasures, e.g. by activating cold stand-by components and gradually passing over control to those redundant components, can be performed.

With regard to automated driving functions, particularly when targeting VDA levels 4 and 5 [2], monitoring for the applications needs to be extended. As the driver is considered to be out of the control loop for these levels of automation, automated vehicles need to be equipped with fallback strategies which either provide a completely fail-operational system or at least provide the possibility to maneuver the vehicle in case of failures with reduced and thus safe driving functionality [9].

For monitoring as well as for the determination of fallback strategies, we propose the concept of ability and skill graphs [10]. Skills for the system are implemented by engineers in the development phase and can consist of one or multiple software components. Abilities are runtime instantiations of skills, which can be assigned with an actual performance level in the running systems. Abilities and skills can be composed of lower-level abilities and skills as depicted in Figure 3 (blue: abilities, yellow: sensors, orange: actuators, green: function). The actual performance level of an ability can be attributed to an edge in this tree. By finding appropriate models for the propagation of performance changes from lower-level to higher-level abilities, a detailed and abstracted performance monitoring for the whole system can be performed. In the scope of the project, we aim at a formal description for ability and skill graphs, as well as means of contracting for abilities.

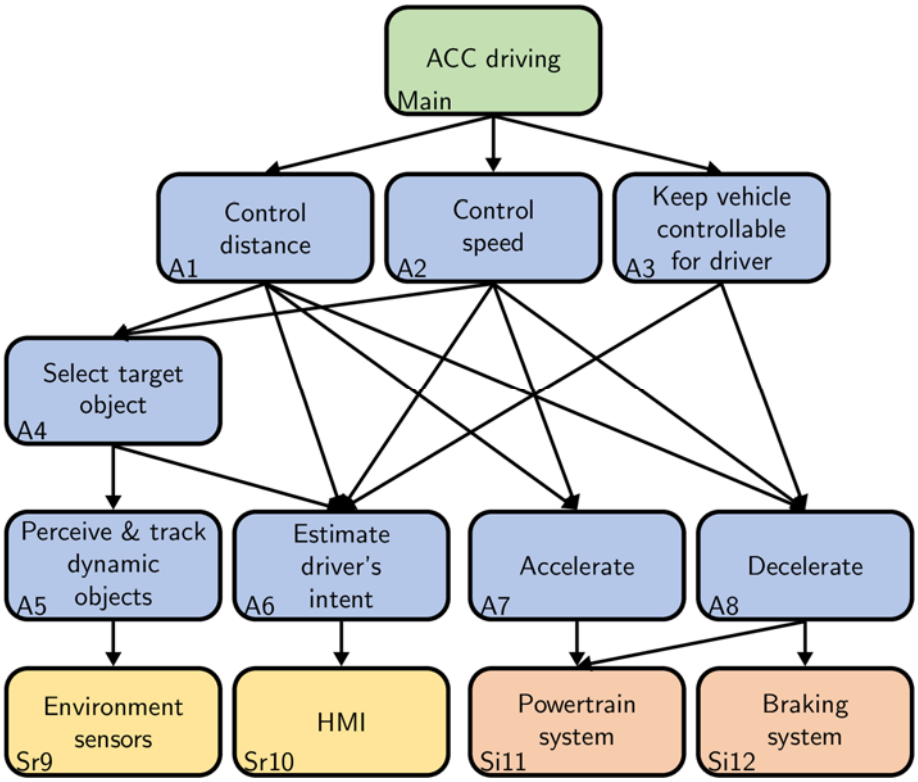

Figure 3: Exemplary simplified ability graph for an adaptive cruise control.

5 Conclusion

In this paper, we addressed the issue of automating function updates in automotive vehicles by taking a contract-based approach that formalizes the integration task in order to provide guarantees (functional or non-functional) for critical functions. We presented our architectural approach that builds a framework for this integration flow and minimizes the gap between model-based design methods and the actual implementation of software components. A key element for this is monitoring that not only serves as a contract-enforcement mechanism but

also provides feedback to the model domain to e.g. enable self-awareness. We further showed how we apply this concept to various automotive use cases that evolved in the scope of the CCC project.

As a result, our architecture equips a system – comprised of functions and platform – with capabilities that provide self-protection against harmful updates and self-awareness w.r.t. environmental changes while still being in full control about any changes to it. We believe that allowing changes and even evolution are essential features of future automotive systems. However, these must be built upon a safe foundation that avoids harmful decisions made for instance by a machine learning algorithm.

[1]: “Audi ZFAS.” 2016. Accessed September 13, 2016. http://www.audi.com/com/brand/en/vorsprung_durch_technik/content/2014/10/zentrales-fahrerassistenzsteuergeraet-zfas.html.

[2]: “Automatisierung - Von Fahrerassistenzsystemen Zum Automatisierten Fahren.” 2015. Berlin: VDA. https://www.vda.de/dam/vda/publications/2015/automatisierung.pdf.

[3]: Bergmiller, Peter. 2014. “Towards Functional Safety in Drive-by-Wire Vehicles.” Dissertation, Technische Universität Braunschweig.

[4]: Hamad, Mohammad, Johannes Schlatow, Vassilis Prevelakis, and Rolf Ernst. 2016. “A Communication Framework for Distributed Access Control in Microkernel-Based Systems.” In *12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT16)*, 11–16. Toulouse, France.

[5]: Holthusen, Sönke, Sophie Quinton, Ina Schaefer, Johannes Schlatow, and Martin Wegner. 2016. “Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates.” In *First International Workshop on Pre- and Post-Deployment Verification Techniques (PrePost)*. Reykjavik, Iceland.

[6]: *ISO 26262 - Road Vehicles – Functional Safety*. 2011. 2nd ed. International Organization for Standardization - ISO; International Organization for Standardization - ISO.

[7]: Lampke, Steffen, Simon Schliecker, Dirk Ziegenbein, and Arne Hamann. 2015. “Resource-Aware Control - Model-Based Co-Engineering of Control Algorithms and Real-Time Systems.” *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 8 (1). doi:10.4271/2015-01-0168.

[8]: “NVIDIA DRIVE PX.” 2016. Accessed September 13. http://www.nvidia.com/object/drive-px.html.

[9]: Reschka, Andreas, and Markus Maurer. 2015. “Conditions for a Safe State of Automated Road Vehicles.” *It - Information Technology* 57 (4).

[10]: Reschka, Andreas, Gerrit Bagschik, Simon Ulbrich, Marcus Nolte, and Markus Maurer. 2015. “Ability and Skill Graphs for System Modeling, Online Monitoring, and Decision Support for Vehicle Guidance Systems.” In *2015 IEEE Intelligent Vehicles Symposium (IV)*, 933–39. Seoul, Korea: IEEE.

[11]: Schlatow, Johannes, and Rolf Ernst. 2016. “Response-Time Analysis for Task Chains in Communicating Threads.” In *22nd IEEE Real-Time Embedded Technology and Applications Symposium (RTAS 2016)*. Vienna, Austria.

[12]: Schmidt, Karsten, Denny Marx, Kai Richter, Konrad Reif, Andreas Schulze, and Torsten Flämig. 2015. “On Timing Requirements and a Critical Gap Between Function Development and ECU Integration.” In *SAE Technical Paper*.