IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

IEEE VEHICULAR TECHNOLOGY SOCIETY SECTION

Received March 14, 2022, accepted May 10, 2022, date of publication May 23, 2022, date of current version June 6, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3176879

Lifecycle Management of Automotive Safety-Critical Over the Air Updates: A Systems Approach

HOUSSEM GUISSOUMA[1], (Member, IEEE), CARL PHILIPP HOHL[2], FABIAN LESNIAK[1], MARC SCHINDEWOLF[1], JÜRGEN BECKER[1], (Senior Member, IEEE), AND ERIC SAX[1]

[1]Institute of Information Processing Technologies, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
[2]FZI, 76131 Karlsruhe, Germany

Corresponding author: Houssem Guissouma (houssem.guissouma@kit.edu)

This work was supported by the German Federal Ministry of Education and Research through the German Federal Ministry of Education and Research (BMBF) in the Project Step-Up!CPS (Förderkennzeichen) under Grant 01IS18080D.

ABSTRACT With the increasing importance of Over The Air (OTA) updates in the automotive field, maintaining safety standards becomes more challenging as frequent incremental changes of embedded software are regularly integrated into a wide range of vehicle variants. This necessitates new processes and methodologies with a holistic view on the *backend*, where the updates are developed and released, and the *frontend* (vehicle), to which the updates are deployed. In this paper, we introduce an approach, including a process and a methodology, for continuous contract-based design, validation and deployment of modular updates for variant-rich automotive systems. The approach considers the vehicle as part of its connected environment enclosing a backend and concentrates on safety-critical applications. In addition, we present the UPDateable Automotive Test dEmonstratoR (UPDATER), which is a mock-up for modern Electric/Electronic architectures including a backend and a frontend part. It serves as a prototype for developing, deploying and monitoring automotive OTA updates. In a case study based on UPDATER, we apply the approach to three exemplary updates of a variable Advanced Driver Assistance System (ADAS). We show how the updates development and management may be achieved in an efficient and agile way.

INDEX TERMS OTA updates, contract-based design, variant and configuration management, safety-critical systems, middleware, DevOps, advanced driver assistance systems, monitoring.

I. INTRODUCTION

The automotive industry is experiencing substantial changes through the fast growing digitization of its functions to achieve more safety, automation, comfort and energy efficiency. The number of electronic control units (ECUs) has increased from about 20 to over 150 in the last 20 years [1], [2], and the amount of Line of Code (LoC) to 100 million [2]. This results in a growing level of complexity in terms of number of elements in the architecture and their interconnections. For example, there are about three million functions in the entire source code of the Volvo company according to a recent study [3]. These functions are called at about 30 million different places in the source code. In addition, there are about 45.000 signals in a modern Mercedes-Benz premium car [4]. Maintaining safety requirements is becoming more crucial than ever with the continuous increase of the level of vehicles autonomy on the market (see Figure 1). To keep these extensive software parts up-to-date as well as safe and secure, software updates are developed in shortening life cycles and deployed to the vehicles being in use in the field [6]. Up to now, most of these updates have been conducted in workshops during regular visits or as part of recall campaigns to fix safety-related bugs [6]. However, in recent years, Original Equipment Manufacturers (OEMs) started to deploy them Over The Air (OTA) instead of manual installation by a professional technician in the workshop. These are referred to as OTA updates. They result in a better satisfaction of customers since they save the time and effort required for the trip to the workshop [7]. Furthermore, they offer several

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Quan.

advantages for the OEM, who may deploy the updates as soon as they are available and save the costs needed to manage the installation process with the workshops [7].

To handle the changes described above, the processes and methods established for automotive development so far, mainly based on the Waterfall and V-model, need to adapt towards more agility [8]. This involves using new agile process models such as Scrum for software development [9], as well as a stronger link between the development and the operational phase. This stronger link resulted in models such as DevOps, where small incremental changes are regularly built and tested in an automated way before being deployed to the system in operation. However, this involves in most cases only very specific unit tests, which might not be sufficient for automotive functions with high safety standards.

To achieve an agile development and deployment of OTA updates, special care must be taken to deal with various challenges. On the one hand, maintaining the security of the target vehicle is a fundamental requirement for conducting OTA updates. Remote updating adds new entry points which may be used by potential hackers to tamper with the vehicle's software. It also removes the trained technician in the workshop from the update process [7]. On the other hand, the safety of the system must be maintained and proven after each new software release in order to get a valid update certification for safety-critical functions. The UNECE R156 regulation,[1] which will take effect in the EU starting from 2023, requires a new type approval each time an update has an influence on a safety criteria of the system or one of its subsystems [10]. Furthermore, the high variability of the automotive systems arising from the wide range of product configuration possibilities needs to be efficiently managed [6]. In other words, each developed update must be verified and validated for compatibility and safety for each affected variant, and the evolution of the variants must be continuously tracked and modeled. Although the focus of this paper lies on the safety and variability aspects, security issues and methods will be shortly covered.

Contribution: We introduce a reference systems approach for the lifecycle management of updates for safety-critical functions (see Figure 2). It is based on a novel methodology relying on formal specification of modular updates in a contract-based design environment together with incremental verification using delta-based modeling. The application of the methodology is shown within the phases of an update lifecycle management process including the phases of *pre-deployment*, *deployment* and *post-deployment*. We show how the introduced methodology may be applied for modern electric/electronic architectures (E/E architectures) by introducing the generic prototype UPDateable Automotive Test dEmonstratoR (UPDATER). Based on UPDATER, the development of three different updates is shown and evaluated.

[1] "Proposal for a New UN Regulation on Uniform Provisions Concerning the Approval of Vehicles with Regards to Software Update and Software Updates Management System."

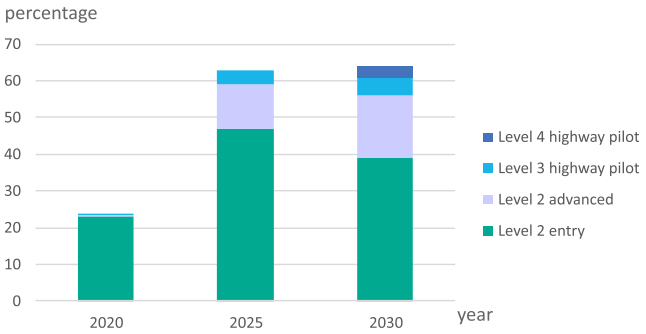


FIGURE 1. Expected evolution of portions of automated driving functions (SAE levels) of the total vehicle sales according to [5].

The rest of the paper is structured as follows: In section II, the state of the art of automotive E/E architectures, release and configuration management, security and relevant standards of OTA updates as well as contract-based design is presented. Section III gives an overview on related works. Thereafter, we introduce in section IV our process for update lifecycle management and highlight our methodology for incremental update design and verification for variant-rich vehicle fleets. Then, in section V, we introduce the environment required for developing, deploying and monitoring modular updates as described in section IV. This includes parts of the *backend*, where the incremental design and verification take place, and the *frontend* which represents the updatable E/E architecture of the vehicle. Furthermore, we describe the concept of the frontend middleware required to receive, check, install and monitor the updates. The implementation of this backend-frontend environment is shown by presenting our prototype UPDATER in section VI. Section VII shows the application of our process-oriented methodology by investigating three updates of an Advanced Driver Assistance System (ADAS) consisting of a safety-enhanced Adaptive Cruise Control (ACC). The presented approach is then evaluated in section VIII based on the results of the case study. Finally, we discuss some threats to validity (section IX) and conclude the paper in section X.

II. STATE OF THE ART

A. AUTOMOTIVE ELECTRIC/ELECTRONIC ARCHITECTURES

Jiang defines the vehicle E/E architecture as "the fundamental organization of vehicle electrical and electronic components, including ECUs, sensors, actuators, wiring, power distribution, onboard and wireless communication etc., to realize the desired function and performance goals, with emphasis on the interactions and interdependencies among the components and with the environment." [11]

Driven by the recent megatrends in the automotive industry, such as automated driving and connectivity, the E/E architecture is undergoing a transformation toward a software-defined car (cf. section I). To deal with

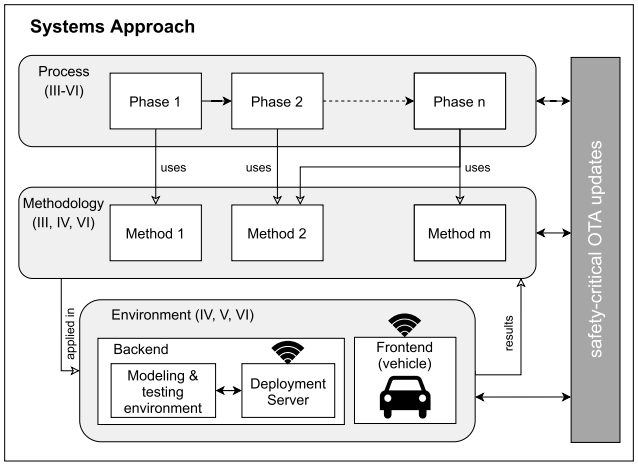


FIGURE 2. Structure of the suggested systems approach for managing OTA updates.

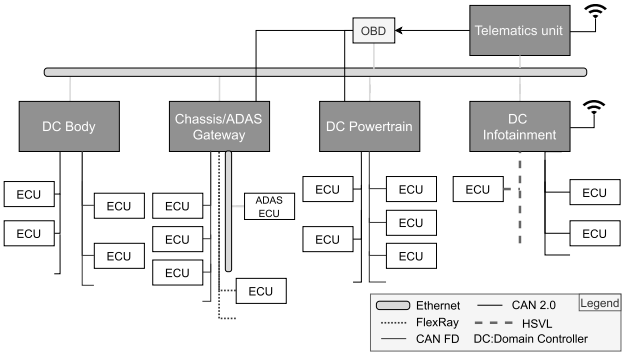


FIGURE 3. Schematic of a domain-centralized E/E architecture of a modern vehicle, according to [12].

these increasing electronic and software portions, the E/E architecture has evolved from a distributed to a domain-centralized architecture. Figure 3 shows how a domain centralized architecture of a modern car looks like. Here, the individual domains are separated and connected though domain controllers via an Ethernet backbone. This reduces the total amount of wiring harness since inter domain-communication is realized centrally over one single Ethernet backbone with high bandwidth. The future trend of next generation E/E architectures is moving toward a zonal architecture. Here, the individual ECUs will no longer be grouped by domain, but by physical zone of the vehicle [13]. This will lead to a further optimization of the wiring harness by profiting from the physical proximity of hardware platforms and integrating a central vehicle computing module between the zones.

The *Adaptive Platform* of the AUTomotive Open System ARchitecture (AUTOSAR) is currently accepted as the de facto standard for the development of automotive ECUs. Its basic idea is to extend the former *AUTOSAR Classic Platform* to a more dynamic architecture by enabling service-oriented communication between software components.[2] The Adaptive Platform is based on the AUTOSAR Runtime for Adaptive Applications (ARA). The latter is composed of application programming interfaces (APIs) provided by functional clusters and is associated to a (virtual) machine. The APIs are either linked to the *Adaptive Platform Foundation*, which provides essential features, such as execution management and logging, or to the *Adaptive Platform Services* providing amongst others services for diagnostics, updates and configuration. [14]

B. RELEASE AND CONFIGURATION MANAGEMENT

A release is the collection of one or more, new or changed, configuration items deployed into the live environment of a system [15]. In the automotive industry, releases are implemented on a regular basis, usually every three months during development, and every six months after start of production (SOP) [6]. Release management describes all activities related to the definition, development, publishing and distribution of the releases.

Configuration management is a more general term. It is defined as the planning of a product, a process or a document before, during and beyond its life cycle [16]. In the software engineering field, configuration management sums up all activities needed to manage the software parts of a product along its life cycle [17]. These activities may be classified into the four categories: configuration identification (or baseline), status control, change control and status account [18]. The elements constituting the baseline of the product are called *configuration items*.

System configurations are usually developed and maintained using so-called codeline diagrams (see Figure 4). In this diagram, introduced new features are marked by rectangles and release states by a circle with a version number inside, e.g., version 1.0 and 2.0. Codeline diagrams allow also for parallel development by defining different *branches*. The branches are opened for a specific purpose, for example fixing a bug, and may then be merged again with the main branch. The composition of different codelines to build the product defines the baseline.

Version management has the goal of managing and tracking the versions of software components. In addition, it ensures that developers working on multiple versions in parallel do not overwrite each other's changes [19]. Codeline diagrams are also used for version management such as by the widespread tool Git which is based on the notion of repositories [20]. In the automotive field, version numbers are usually given according to the semantic versioning[3] scheme [21]. To enhance the consistency and compatibility of software components, the concept of hierarchical versioning building on the existing hierarchy of ECUs within the E/E architecture is proposed in [4].

[2]In AUTOSAR adaptive, software components are realized as *services* according to the service-oriented architecture (SOA) principle.

[3]structures the version ID as *MAJOR.MINOR.PATCH*.

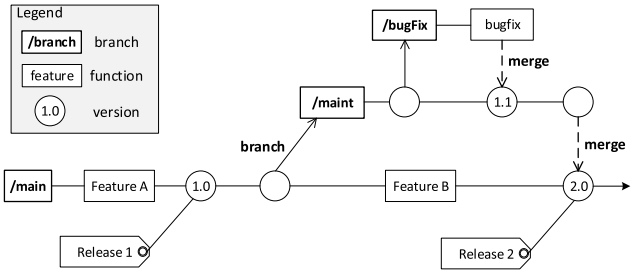


FIGURE 4. Examplary codeline diagram.

In addition to managing the evolution of systems in time (versions), one must take account of the existing variability in space (variants). This engineering field is known as variant management. Variants are parallel existing configurations of the system. To manage the components of variable systems and achieve efficient reuse strategies, the approach of product line engineering, or in the software domain, software product lines, is the method of choice [22]. Here, feature modeling is a widespread method to model the variant space of the product line. Features describe the variability points in a product line as well as their inter-dependencies [23]. Besides that, the mapping between features and implementation artifacts, such as ECUs or software components, is important for building system variants and maintaining traceability [22].

C. SECURITY OF AUTOMOTIVE OTA UPDATES

Security is an essential property for vehicles equipped with the technology of OTA updates because of the new risks it adds to the classical E/E architectures [6], [24]–[26]. Opening the vehicle to the outside by means of external over-the-air communication increases the risk of cyber attacks leading to possible intrusion into the internal vehicle network, e.g., Man-in-The-Middle attacks. Further, by adding the capability of remote updating of ECU software or firmware,[4] threats of installing malicious software arise, e.g., changing a firmware that disables the brakes or unlocks the door in unexpected situations [24]. Therefore, security mechanisms need to be integrated as a central part of the process for lifecycle management of OTA updates. These need to protect all kinds of interchanged data from potential malicious third parties on both frontend and backend sides. The data comprises software code, messages between server and client as well as further configuration and diagnostics information.

As described by Nilsson *et al.* in [24], the properties required to secure OTA updates are: data integrity, authentication and confidentiality. Kim *et al.* [26] differentiate between data and service integrity. Data integrity means that no change to the update data (software packages and configuration files) was introduced by an unauthorized party. This may be achieved by checking the validity of the data using a hash function [26]. Integrity checks are usually combined with data authentication based on digital signatures. Further, service integrity denotes that the service process, i.e., update access, download and installation, hasn't been modified by any malware [26]. This may be guaranteed by emergent blockchain methods including a data and network layer [27]. The second property, authentication, verifies whether the received data origins from the claimed sender. By using digital signatures added to the transmitted data, the receiver is able to check the authenticity of the data and at the same time its integrity [24]. Confidentiality represents the secrecy of the transmitted data, which shall not be revealed to unauthorized parties. Symmetric key encryption is an established method to guarantee this property [24]. For server-client communication on the Internet, the Hypertext Transfer Protocol Secure (HTTPS) protocol is a widely used standard based on certificates which provides all three properties of integrity, authentication and confidentiality [28]. Since most of the security methods are based on pre-programmed keys, and seen the high number of vehicles in OEM fleets, an efficient key management should be integrated on the backend side of an update management system. Also, the keys must be updated on a regular basis to reach an acceptable privacy level of users [25].

D. STANDARDS, NORMS AND FRAMEWORKS FOR AUTOMOTIVE UPDATES

When considering updating safety-critical systems like cars, several standards and norms need to be taken into consideration. The work of the UNECE Working Group in WP29 [29] proposes methods for addressing security concerns during over-the-air updates. It describes threats posed to safety-critical functions that manufacturers need to deal with during and after the update process. This resulted in the R156 specifications which will become standardized in the upcoming few years and need to be considered while developing, deploying, installing and operating updateable software in vehicles. Similarly, the Society of Automotive Engineers (SAE) provides guidance for developing systems in a secure way by introducing a framework to implement cybersecurity practices [30]. This is extended by the ISO/SAE 21434 which proposes security requirements applicable to processes and systems. It aims to give a guidance in managing and dealing with security risks during concept, production and maintenance of automotive systems [31]. In order to ensure that systems are safe as well as secure, ISO 26262 needs to be taken into account. It provides a number of methods to evaluate functional safety within a vehicle and ways to deal with potential hazards. Development according to ISO 26262 has been an industry practice for some years and will continue to be so alongside, especially for highly automated vehicles. The standard ISO/PAS 21448 introduces a concept called Safety of the Intended Functionality (SOTIF) that improves upon known concepts of ISO 26262 by integrating the notion of operational design domain.

AUTOSAR (cf. section II-A) provides recommendations, requirements and technical specifications for updating the

[4]Firmware is a special category of software for low-level control tasks.

vehicle's firmware and software over-the-air. The document Nr. 945 of the release R19-11 [32] gives a detailed description of a Firmware Over The Air (FOTA) process and the required technical parts within the E/E architecture. These are a *FOTA Target ECU* which receives the software and forwards it to the low level memory stack instance (*flashing*), the *FOTA Master ECU* which caches all new ECU software artifacts, and a *backend server* for updates delivery. In addition to the specification of the three FOTA parts, the document specifies the rollback process[5] and the communication protocol between the modules. Furthermore, the AUTOSAR adaptive community works on standardizing a so-called Update and Configuration Management (UCM) module [33]. This module is a service within the middleware of a service-oriented architecture responsible for updating, installing and removing software with special consideration of safety and security aspects [33]. It uses the *Crypto Interface* of the Adaptive Platform to verify the package integrity and authenticity and to decrypt the update packages and communicates through the standardized *ara::com* interface.

In addition to the standards described above, different frameworks for automotive OTA updates have been introduced in industry and research. One of them is UPTANE [34]: an open source framework for the data security and configurability of automotive software updates. It distinguishes between two types of ECUs: primary ECUs, which communicate directly over-the-air with the server and perform complete security checks, and secondary ECUs, which check the metadata of the primary ECUs as a second level of verification. In UPTANE, there is an *image* and a *director* repository on the server side. The image repository contains all versions of the ECU software components of the OEM and the corresponding necessary metadata for their authentication. The director repository is responsible for identifying the software images that are required for an ECU network of a vehicle configuration. Another framework is built by the consortium of automotive companies eSync Alliance [35]. Its goal is the development and establishment of a uniform and standardized data pipeline for OTA updates. For that, the consortium developed a software development kit (SDK) for a fast realization of OTA updating and diagnosis. The central part of the eSync architecture is an *orchestrator module* communicating with a server and delivering the updates to distributed *eSync Agents* running on each updatable ECU.

E. CONTRACT-BASED DESIGN

The idea of using contracts to improve software development has first been introduced in the 1970s in the context of the formal verification theory by the IBM Laboratory Vienna. Later, towards the end of the 1980s, Bertrand Meyer proposed the concept of "Design by Contract" for sequential programming [36], which is the foundation of the object-oriented programming language Eiffel. Meyer defines contracts as a set of *pre-* and *post-conditions* that must be met before entering and after leaving one method. The preconditions represent the specification of the environment of the method, whereas the postconditions specify its guaranteed behavior under the specified environment. In addition, contracts may include static statements called *invariants*, which must hold at every state of the system. The basic idea of contracts is similar to distributed automotive development processes where an OEM must agree with its suppliers (tier-1, tier-2) on the subsystem or component to be delivered [37]. By formally specifying the interfaces of these components and designing them in a modular way, contract-based design facilitates the verification and validation of the developed systems. Besides, contracts are a useful way to maintain requirements traceability and to enable formal proofs for their fulfillment, which is crucial for safety-critical systems.

In the 2010s, contract-based design has been gaining more attention for application in the fields of embedded and Cyber Physical Systems (CPSs). This resulted in the introduction of the theory of contracts by Benveniste *et al.* in [38]. In this theory, a contract is defined intuitively as a pair of assumptions and guarantees:

$$\mathcal{C} = (A, G) \text{ of } \{\text{Assumptions, Guarantees}\}, \tag{1}$$

where A are properties formally describing under which context the design is assumed to operate, and G are properties describing its obligations [38], i.e. requirements assigned to its behavior and output. Contracts are intended to be intentionally abstract and they must distinguish the responsibilities of a component from those of its environment [38]. A more formal definition of a contract $\mathcal{C}$ is given by equation 2, where $\mathcal{M}_{\mathcal{C}}$ is the set of implementations[6] of $\mathcal{C}$ and $\mathcal{E}_{\mathcal{C}}$ the set of its legal environments [38].

$$\mathcal{C} = (\mathcal{M}_{\mathcal{C}}, \mathcal{E}_{\mathcal{C}}) \tag{2}$$

Contracts may be directly derived from requirements associated to the system components by different formalisms. These include automata modeling, temporal logic constructs (e.g., for safety requirements), probabilistic constraints (e.g., for reliability requirements) and linear/nonlinear constraints on real numbers [39].

There are different mathematical properties helping to reason about a system architecture based on contracts and their associated component interfaces. On the one hand, several contracts may be used to construct a composite contract. The composition of two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$, denoted as $\mathcal{C}_1 \otimes \mathcal{C}_2$, is defined by the following A/G pair [38]:

$$\begin{cases} A = (A_1 \cap A_2) \cup \neg(G_1 \cap G_2) \\ G = (G_1 \cap G_2) \end{cases} \tag{3}$$

On the other hand, to specify the vertical contracts associations within the system's hierarchy, the *refinement* ($\preceq$) relation is defined. A contract $\mathcal{C}'$ refines $\preceq$ another contract $\mathcal{C}$ if and only if 1) any implementation of $\mathcal{C}'$ is an

[5]rollback: switching back to the old software or firmware version in case of a detected error during installation.

[6]One component M implements one contract $\mathcal{C}$ if it fulfills its A/G pair.

implementation of $\mathcal{C}$, and any environment of $\mathcal{C}$ is an environment of $\mathcal{C}'$.

Contracts do not only enable formal verification, such as using model checking methods, but also are an efficient way to monitor safety specification at runtime, in simulation as well as in the real-world system [40].

In this paper, we formalize requirements into A/G contracts using temporal logic constructs, which were first introduced by Pnueli in 1977 to reason about propositions in time [39]. For this purpose, the two well-established formal languages Linear Temporal Logic (LTL) [41] and Signal Temporal Logi (STL) [42] are accepted as suitable specification means. LTL combines so-called *atomic propositions AP* using different logical operations, such as negation ($\neg$) or disjunction ($\vee$), and two basic temporal modalities, *next* ($\bigcirc$) and until ($\mathcal{U}$). STL is an extension of LTL to cover timing properties of dense-time real signals and signals with continuous dynamics.

III. RELATED WORKS

In [43], Wang *et al.* introduce a concept for integrating model checking with unified system modeling in SysML for the purpose of system safety analysis and verification. For that, they specify transformation rules to convert semi-formal SysML models (BDDs, STMs) into formal models which may be verified by symbolic model checkers, here the NuSMV tool. Based on the results of the model verification, the functional model of the system is incrementally refined before processing to the implementation. The application of this approach is shown for a case study of integrated modular avionics. In the same context, Xie *et al.* propose in [44] a compositional verification concept of SysML models of safety-critical CPSs. They use Assume/Guarantee contracts instead of generic safety properties to enable for modular refinement verification and translate a safety profile into a Fault Tree Analysis (FTA) to conduct the safety analysis. The modeling environment relies on a transformation of the SysML models into OCRA[7] models for conducting the virtual integration. However, no incremental verification of small changes to the system architecture, which are necessary in update cases, is considered in the two mentioned works.

In the context of automated driving, a method for runtime verification of STL specifications in the CARLA simulator is presented in [45]. It uses the RTAMT library, also employed in our work, to monitor requirements for an experimental ACC system at simulation runtime. This enable early verification of different driving scenarios as well as systematic design space exploration, e.g., through property falsification. The monitor calculates a robustness metric for the fulfillment of the specification. It is used to optimize the PID controller of the ACC in a design-space exploration approach. Similar to this work, Watanabe *et al.* [40] rely on runtime verification of STL specifications for safety verification during the design of intelligent vehicles. They use the tool Breach to preform a case study for the integration of two ADAS features: Cooperation Pile-up Mitigation System (CPMS) and False-Start Prevention System (FPS). The specifications are formulated as Assume/Guarantee contracts. The authors explain the required checks of assumptions and guarantees at the design and runtime phase which must be managed by OEMs. Both of the described works focus on runtime monitoring of contract specifications for automated driving. However, none of them considers the challenge of dealing with the high number of system variants. Also, they do not show how the validation of the contracts can be guaranteed at the system level by combining the monitoring results at component level.

Finally, Ayres *et al.* introduce in [46] an approach for deployment of automotive ECU software updates based on lightweight virtualization known as containers. This leads to more efficiency in download site and times through layer sharing. Although we don't explicitly use containers in this work, but *update packages* to be deployed as virtual machines, we regard containers as an efficient alternative technology. This is especially a suitable approach for smaller ECUs with limited resources, as well as for scalable deployment strategies using the Kubernetes system [47], [48].

IV. PROCESS AND METHODOLOGY FOR LIFECYCLE MANAGEMENT OF UPDATES

A. PROCESS OVERVIEW

1) REQUIREMENTS FOR PROCESS DEFINITION

To maintain an automotive system using updates, there need to be mechanisms in place that enable and facilitate their development, deployment and operation. A lifecycle model for update development must therefore incorporate phases and steps for the incremental specification, implementation and distribution of software components.

The development must start with well formulated requirements that can be interpreted and used in system verification and validation. These requirements must be based on the initial development phase (before SOP) and continuously updated through, e.g., analyzing collected diagnostics data about the vehicles' functionality and performance. Another reason for changing or extending the requirements after SOP may be legal regulations set by a governing body. Third is the introduction of new functionality driven by evolving customer needs. The resulting set of requirements must lead to a potentially updated system design that is capable of meeting the safety standards. The latter is then incrementally transformed into a specification for an implementation, which can be virtually integrated into the existing system environment. If there is confidence that the system design fulfills the set of requirements, it can be implemented and individual components can be verified and validated using various techniques such as Software-in-the-Loop (SiL) or Hardware-in-the-Loop (HiL). After validation, the updated software components need to be stored and prepared for deployment. During deployment,

[7]tool for checking the refinement of temporal contracts.

security aspects with regards to the data transmission need to be considered (cf. section II-C). Once transferred to the system, the software components must be installed and taken into operation without disrupting the systems integrity.

2) PROCESS PHASES

We propose a process that covers all the requirements stated in section IV-A1, and, at the same time, provides flexibility to designers and developers. It is split into three distinct phases: the *pre-deployment* phase, which includes planning, modeling, implementation, verification and validation of an update, the *deployment* phase, in which the update packages are securely transferred to the target vehicle and the *post-deployment* phase, in which the update is running on the vehicle and being continuously monitored to ensure its safe operation. The advantage of this order is twofold. On the one hand, the three phases can be viewed in their own way and their work products serve as handovers between them. On the other hand, this structure allows for one phase to be altered and tailored to a specific domain or context. The process is further divided into the following six steps:

- Design
- Virtual Integration
- Implementation and Build
- Verification
- Deployment
- Runtime and Monitoring

These steps and the respective phases that they are executed in are shown in Figure 5.

In the pre-deployment phase, all the steps are incremental, which means that only changed or affected components within the system architecture need to be re-designed and re-verified. These steps are also iterative, meaning that going back to the previous phase is allowed whenever it is required to (see dashed upward arrows in Figure 5). Furthermore, the first two steps (design and virtual integration) are conditional. In other words, one or both may be left out if the required update does not affect them. Although the process model in the pre-deployment phase is similar to the Waterfall model, its included methodology may be applied in different other models.

A detailed description of the methodology used in the three phases is given in the following sections.

B. PRE-DEPLOYMENT

We consider update lifecycles to be continuous, meaning that a new update cycle is started whenever a trigger for an update decision is detected or introduced. *Update Triggers* can be of many types, but the most common ones addressed in this paper are: regulatory changes, marketing demands or feature additions, error corrections and performance improvements. These triggers are partly based on collected feedback data from the post-deployment phase (cf. upward arrow in Figure 5).

The type of trigger has an influence on the necessary development steps during the pre-deployment phase. According

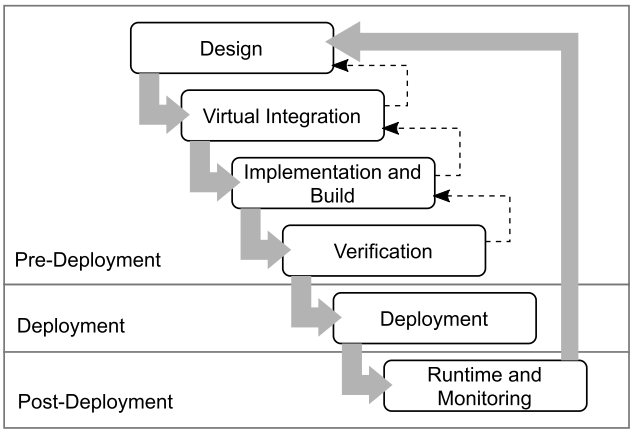


FIGURE 5. Process overview.

to ISO/IEC/IEEE 14764 [49], there are four main categories for software maintenance after delivery: "corrective" to fix discovered problems, "adaptive" to keep the product usable in changing environments, "perfective" to provide enhancements for users and "preventive" to prevent errors before they occur in live system. In the automotive field, we may regard preventive software changes as a special kind of corrective ones, with the only difference that they are triggered by predictive maintenance analysis or by collected monitoring reports in "shadow mode"[8] tests. After tailoring the ISO/IEC/IEEE 14764 categories to the automotive context, we define the following three types of updates:

- **Corrective update:** corrects bug(s) encountered during operation. It doesn't require any requirements change,[9] since it is only a correction of the system in order to comply with existing specifications.
- **Adaptive update:** adapts software to keep it usable in changed environments, if possible, within the same requirement framework. An example is adjusting the interfaces of a software component after changing a hardware module or an operating system. The environment may be external such as a new legal regulation.
- **Perfective update:** serves the purpose of enhancing the system's functionality and performance to improve the user's experience and respond to its new demands. Thus, this update type adds functionality to the system resulting in additional requirements.

To apply contract-based design, it is necessary to divide the system architecture into a set of *components*. This is known as functional decomposition as described in [50]. The first abstraction consists of one top-level or system component which is refined by at least two subcomponents. Each of them may be further refined until reaching the lowest granularity

[8] a system running passively in the background of an automated driving function during usage.

[9] We assume, following the ISO/IEC/IEEE 14764, that the initial set of requirements is correct and complete with regards to the safety-related properties of interest.

TABLE 1. Types of updates and their impact on the component under update.

Update type	Corrective	Adaptive	Perfective
Trigger	bug, e.g., detected by contract violation	changed environment	new features
Implementation change	yes	yes	yes
Contract change	no	may have	yes
Interface change	no	may have	may have
Monitor change	no	may have	yes
Example	output value outside of safe range	adapt to a new sensor's interface	adding an object detection feature

level which we define as *module*'s level. So, a module is the smallest updatable component which will be released as a software package and may be deployed to the system's frontend. The update design starts at one of the granularity levels by specifying the change of one specific component, which we call component under update (CUU). Table 1 shows a comparison of the three considered update types with respect to their triggers and their respective needed modular changes to the affected CUU. As we are relying on contract-based design as an enabler for modularity and hierarchy, we differentiate between changes of interfaces, of contracts and of implementation. The latter is common to all types of updates, since every update must result in a change of the software code. In order to achieve an incremental model-based development for all three kinds of updates, we rely on the concept of delta-based design as described in [51]. This concept models a system's update $\mathbf{U}$ as a matrix of so-called time deltas Δ:

$$\mathbf{U} = \begin{bmatrix} \Delta_{1,1} & \Delta_{1,2} & \cdots & \Delta_{1,L} \\ \Delta_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \Delta_{N,1} & \cdots & \cdots & \Delta_{N,L} \end{bmatrix}, \quad \text{with}$$
$$\Delta = \begin{pmatrix} \Delta I \\ \Delta \mathcal{C} \\ \Delta Imp \end{pmatrix} \tag{4}$$

where L is the number of granularity levels in the system architecture and N the maximum number of components within one granularity level. As expressed by equation 4, a delta is a vector of three delta transformations applying to one component M. The are ΔI for adding or removing interfaces (inputs/outputs), $\Delta \mathcal{C}$ for contract changes and ΔImp for implementation changes (see example in Figure 6).

To deal with the high variability of the maintained vehicle configurations and increase the reuse of components,

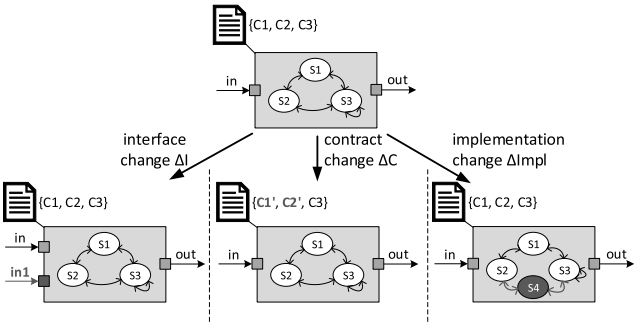


FIGURE 6. Delta categories as unit of change for modular updates.

OEMs usually adopt a product line approach. Here, the product variants are developed within an overarching basis platform by identifying the common and variable parts of the requirements, architecture and implementation. Based on the update trigger, which is translated by, e.g., the product manager(s) into a change request, the affected CUU as well as the affected system variants within the product line need to be identified. For this, we use a feature tree and a variable architecture model, called 150% model; both will be introduced in section V-A. This step is called, according to [23], *impact analysis*. It results in a reduced variant model, named *variant-representative model*. This model represents a reduced set of the system architecture, as the system of interest for a specific update is only a part of the whole vehicle system. The resulting deltas are then verified virtually in an incremental way (see *virtual integration* in Figure 5). This includes the verification of the consistency and refinement relations of the contracts from the smallest updatable modules up to the system component.

After completing the implementation and build process of the modules under updates (MUUs),[10] these are verified with regards to the validated contract specifications. This is achieved by dedicated monitoring programs, which are able to read the contract specifications and compare them with the relevant inputs and outputs of the MUUs in the test environment. The monitoring may be executed online (during simulation or hardware-tests) or offline by checking collected execution traces. The monitors may need to be re-generated each time the interfaces or contracts of the updated modules are changed. We propose to conduct most of the monitoring-based verification virtually within flexible digital twin models. However, and depending on the quality of the used digital twin models, at least parts of the tests still need to be conducted on hardware-in-the-loop setups and physical vehicles for final update validation.

C. DEPLOYMENT

Deployment is defined according to Dearle in [52] as a post-production activity between the acquisition and

[10]One CUU leads, after finishing the update's implementation, to a set of n updated software modules called MUUs.

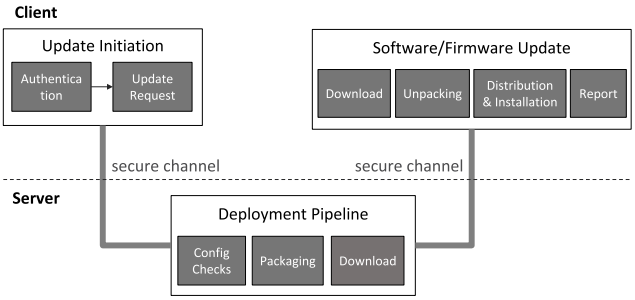


FIGURE 7. Overview of the deployment process phase.

execution of software. It is conducted by a *software deployer* which is in our OTA updates environment an *update client* installed on the frontend and communicating with a remote server on the backend side. An overview of the deployment phase in our process is depicted in Figure 7. It is divided into the following sequential stages:

- **Update initiation:** From the server perspective, the deployment of updates is initiated by the client in a "pull" approach. A "push" approach from the server to the client is also feasible, but requires modification of the deployment workflow as well as the database infrastructure. After successful authentication using certificates, the client sends a request to the server to check for updates. For this, it sends the relevant configuration data of the vehicle in a pre-defined format.
- **Deployment pipeline:** The configuration data, which defines the vehicle's fingerprint, is received by the server over-the-air through a securely encrypted communication channel. This fingerprint of the requesting vehicle is then compared to the configuration data of the released software/firmware modules saved in the release database. Here, especially the configuration constraints for variant compatibility validated in the pre-deployment phase must be checked. If compatible new module versions are found, they are packaged (including encryption) into a so-called *deployment package*. After that, the download job begins.
- **Software/firmware update:** The download is executed by the client on the frontend side. After finishing it, the updates are first decrypted and unpacked, then distributed to the corresponding ECUs of the vehicle. The affected modules are then substituted with the new versions by the update service of the ECU middleware. Finally, the status of the installation is reported back to the server through the update client of the vehicle. The communication with the server uses the same bi-directional secure channel both for download and reporting.

D. POST-DEPLOYMENT

After completing the deployment, the updates transition into the last step of the lifecycle process, the post-deployment phase. As the update is now in-place and running, it is subject to runtime monitoring: The behavior of the updated software components is continuously verified against their contract specification, which has been installed as part of the deployment package. Monitoring is performed by dedicated modules that are able to detect violations of the applications' contracts. Once such a violation is detected, the gathered information is returned to the developer using an encrypted over-the-air channel. This closes the loop of the development process, triggering a new update cycle. In addition to reporting, further actions can be performed optionally: Depending on the criticality of the violation, it can be necessary to perform a rollback to a last known-good version or transition into a state of degraded functionality.

V. ENVIRONMENT FOR LIFECYCLE MANAGEMENT OF UPDATES

To realize the process and methods described in the previous section, an appropriate environment for update lifecycle management is necessary. This involves different design and testing artifacts in the backend, as well as specific components and services of the E/E architecture in the frontend. The systems approach is defined as the combination of this environment with the process (cf. Figure 2).

A. BACKEND

1) MODELING ENVIRONMENT

As described in section IV, model-based design is used to facilitate the incremental design, verification and validation of automotive updates. Besides, to optimize the level of reuse during the development of highly configurable automotive systems, the approach of product line engineering is adopted [53]. By integrating contract-based specifications within the product line, the safety assessment of the system may be achieved in a more efficient way by increasing the reuse of components, as shown by Nesic in [54]. An overview of our suggested modeling environment is represented in Figure 8.

Product line engineering separates the *problem space* from the *solution space*. On the one hand, the user goals and objectives, the usage contexts and the quality attributes of the system are modeled in the problem space [53]. A feature model (cf. section II-B) is used to model the variability in this space. Constraints, such as inclusion or exclusion, may be added to allow cross-tree relationships [55]. An example is "*feature 1.1 requires f 2.1.1*" as depicted in Figure 8. Additionally, a logical view of the 150% architecture model incorporating all components, their refinement relations and their contract specifications is included as part of the problem space. We differentiate between *parametric* and *concrete components*. *Parametric components* are an abstraction of one or more similar alternative components. By setting parameters of a specific configuration in the parametric component, the corresponding concrete component is derived. Mapping relations between the feature model and the components define the association between them. In this

way, it is easily possible to derive the logical architecture of any variant selection. Furthermore, we define a feature as *concrete* if and only if: it is a leaf of the feature tree or it is mapped to at least one concrete component.

The *solution space*, on the other hand, models and manages the variability of the concrete functional units, the operating environment and the domain technologies [53]. We extend the feature model of the problem space with further features defining the exact solution configuration, e.g., version of a deployable software module or a specific ECU type. Subsequently, similar to the feature composition concept in [55], each feature in the solution space is implemented by a distinct module or component, and the system variant is synthesized by composing feature modules. These features are mapped to implementation models, software module versions and ECU configurations within the technical view of the 150% architecture model. These parts of the technical architecture are implementations of the concrete components in the logical architecture, and subsequently must fulfill their contracts.

2) DIGITAL TWIN

To check the functional and non-functional requirements of the updated components, as well as their integration into the system architecture, we rely at the backend side on a *digital-twin*. It consists of simulation models (*virtual space*) of the vehicles in the field (*real space*) which are reconfigurable in order to allow for testing all existing system variants. This virtual testing is essential because not all hardware variants, i.e., sensors, actuators and ECUs, can be available in form of physical systems in the lab. Based on the digital twin, different scenarios and test cases may be run in a simulated environment while taking into account changing environmental conditions. In this way, the updated model or software may be checked with regards to the specified contracts within every relevant system variant. So the digital twin is the basis for conducting the steps of the process phases *Implementation and Build* as well as *Verification* (cf. Figure 5).

We propose the digital twin concept depicted in Figure 9. It is defined as the combination of a simulation setup, including dynamic models of the system under test and a virtual driving environment, with an oracle of scenarios and test case catalogs saved in a database. The dynamic models or binary code are directly retrieved from the 150% architecture model of the product line, which is associated to the feature model through mapping relations (cf. section V-A1). Through variant selection based on the feature model, the simulation model may be configured to build up a specific variant configuration. The architecture model is kept up-to-date by synchronizing it with the systems in use by feeding different kinds of data from the *real space*. The field data includes update feedback informing about the status of running update processes, monitoring reports describing eventual contract violations at runtime and other diagnostics data such as system health monitoring or context information. We propose to conduct the verification by means of monitoring formal system specifications, i.e., contracts. The monitoring may be conducted either online (at simulation time) or offline by analyzing logged simulation traces. Besides that, the behaviors of the components may be observed on different granularities (system, sub-system, component, module) and abstraction levels (Matlab/Simulink model, C/C++ code on host machine, virtualized ECU).

After testing the models used for implementation based on the digital twin, the software of the modules under updates can be built using continuous integration (CI)/Continuous Delivery (CD) pipelines which may be integrated in the version management system of the company. CI/CD is currently the state of the art for agile software development in most industrial areas. To allow easier deployment, the modules can be released as container applications.

3) DEPLOYMENT SERVER

The server is an abstraction of a possibly network of numerous server computers building a whole cloud infrastructure. It represents the interface between the three phases pre-deployment, deployment and post-deployment (cf. section IV-A). On the one side, it integrates the results of the implemented and validated updates, which are released and saved in the database of the server. On the other hand, it communicates with the update client to deploy new software versions over-the-air (cf. section IV-C). In addition, it collects all feedback data and processes them to support the continuous improvement of the vehicles being used in the field.

The *release database* is the central repository of released module versions. Those are ready to be deployed onto the vehicles both during production and throughout the after-sales phase in form of OTA updates. Figure 10 shows an entity-relationship diagram visualizing the main objects (dark gray boxes) of this database, their relations (light gray diamonds) and their associated data (white ellipses). The *deployment package* is assembled dynamically by the server after processing an update request from a specific frontend configuration. It may contain from one to n different *update packages*. Each update package characterizes one module and has a list of configuration constraints. This list is one of the results of the pre-deployment phase and defines the conditions to ensure variant compatibility of an update, e.g., that the vehicle must include a specific ECU with a constrained software version. The constituting parts of the update package are the binary code, information about the module configuration and the contracts specifying its behavior and safety properties. Both the update and deployment packages should be prepared in a compressed format to save memory and transmission time.

Another important task of the deployment server is the security verification and management. This is essential to maintain the properties of data integrity, authenticity, confidentiality (cf. section II-C).

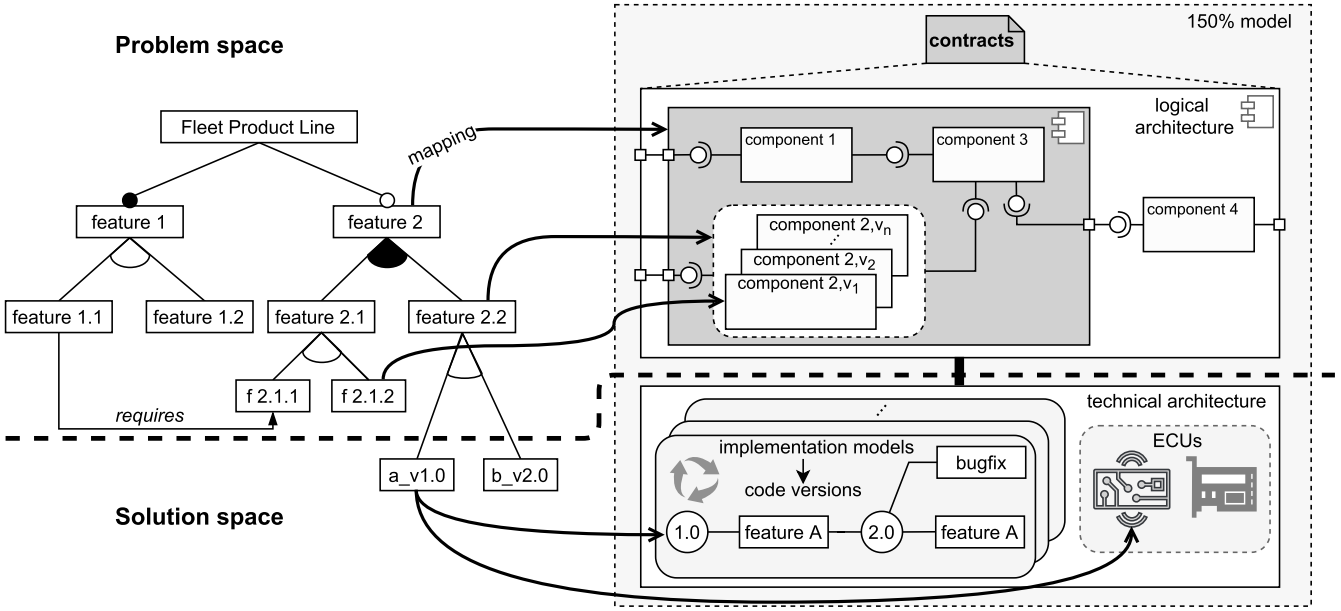


FIGURE 8. Overview of the modeling environment covering the problem and solution space for incremental update design and verification.

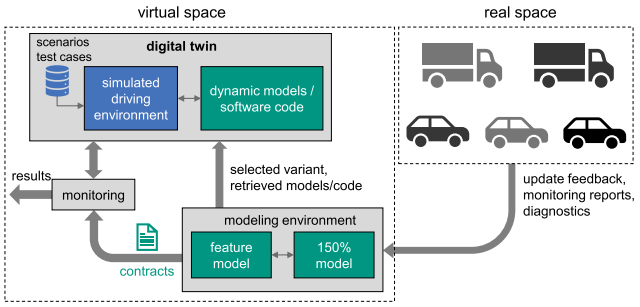


FIGURE 9. Structure of the digital twin, elements of the modeling environment are colored in green and additional digital twin elements are depicted in blue.

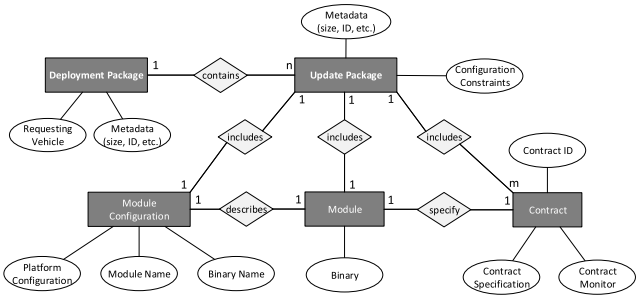


FIGURE 10. Entity-relationship diagram (Chen) of the release database.

B. FRONTEND

1) E/E ARCHITECTURE

As described in section II-A, the E/E architecture includes all hardware components (electric and electronic) of the vehicle with the numerous software components and services running on them. A flexible and modular E/E architecture is one of the central cornerstones for the realization of the update process. To perform an update as specified by the deployment process of Figure 7, the vehicle must include an update client that communicates with the deployment server on the backend. This client should be installed on a central hardware component which has direct access to wireless communication channels, e.g., using WLAN or 4G/5G. In the E/E architecture example presented in section II-A (see Figure 3), this could be the telematics unit if it has enough processing resources or one of the domain controllers/gateways. Another important requirement for this unit is that it must have an extended memory storage for locally saving the deployment package before distributing the software to the target ECUs. Furthermore, all updatable ECUs must have a dedicated update service which orchestrates the internal update process. To allow for a modular exchange of software components and to enable flexible runtime monitoring, a suitable middleware needs to be installed on each updatable ECU.

2) MIDDLEWARE FOR MODULAR UPDATES

The middleware provides functionality for running applications on the hardware platform as well as features necessary for lifecycle management of OTA updates within the vehicle. It is important to support software functions of mixed criticality, e.g. running safety-critical driving applications alongside with infotainment functions, while keeping cross-application effects to a minimum. Therefore we propose a hypervisor as central component of the middleware architecture. It allows for fine-grained isolation between guest domains[11] and yields extensive control over the execution schedule [56].

[11] Software of any kind running under control of the hypervisor.

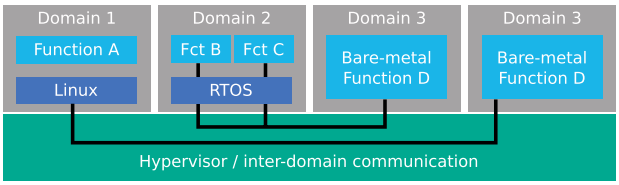


FIGURE 11. Generic middleware concept showing multiple functions connected via inter-domain communication.

To support modular updates, we leverage the ability of splitting large applications across several virtualized domains on a hypervisor-based platform. This results in multiple advantages:

- Resource constraints (e.g. CPU cores, memory) can be defined for each domain.
- Isolation between unrelated applications is enforced.
- Communication between guests can be supervised.
- Guests can be started, stopped and replaced dynamically during runtime.

These additional control mechanisms lead to improved safety and security of the overall system while also enhancing modularity and observability of each deployed software function [57]. These advantages are not free, as the hypervisor causes some overhead in terms of processing time and additional management tasks. The high processing power of modern Systems on Chips (SoCs) combined with suitable resource management and scheduling can however compensate for this loss of performance. This paper demonstrates how to fulfill the demanding requirements for updatable automotive functions using a hypervisor-based platform.

While the update verification and checks can be executed within an unprivileged domain, the central update procedure is carried out on a privileged hypervisor domain (see Figure 11). The update procedure primarily consists of replacing a domain with the updated version. In addition, the updated domain can be run in parallel to the old version while its behavior is being verified. This allows to realize a ''shadow mode'' testing or a hot-standby configuration with the ability to quickly return to the previous version in case of errors.

Some requirements are imposed on applications and the update process. Applications need to be partitioned by the developer in advance, resulting in the individual units to be updated, which are defined as MUUs (see section IV-B). To allow individual domains to interact with low latency and high performance, a fast and reliable inter-domain communication mechanism is required. A possible implementation of such a mechanism is described in section VI-B introducing the UPDATER prototype. The communication mechanism is extended by monitoring capabilities realized by a monitoring component or service installed within the middleware. The latter is able to verify the status of software components according to their contract specification.

3) CONTRACTS MONITORING

As introduced in section II-E, contracts can be used to formally describe valid working conditions of a component. While some contracts can be checked entirely before runtime, such as compatibility to installed hardware and software components, others, such as precise timing and resource performance, can only be verified during runtime. This task of *contracts monitoring* is carried out by a service called *monitor*. The monitor is generated as part of the development process (IV-B) according to the contract specification. Another possibility is to realize a generic monitor which automatically reads syntactically unified contract files on the frontend side. As contract specification language, we use STL which extends LTL with real-time constraints (cf. section II-E). When set up in parallel to the respective software module, the monitor observes its input and output data and continuously checks whether this data matches the contract specification. This operation has to be transparent to the function, i.e. the application (module) may be able to detect if it is being monitored, but its execution shall not be influenced when it is in the ''monitored'' state.

During normal operation, the monitor does not detect any contract violations. Otherwise, if a violation is found, the monitor creates an alert. As a consequence of this alert, information about the violated contract is communicated to the developer by the update client. If necessary, the alert can also be picked up by other services in the frontend, e.g., to trigger a rollback mechanism (section IV-D).

VI. UPDATER-PROTOTYPE IMPLEMENTATION

In the following, we describe the hardware components, tools and technologies used to implement the backend and frontend parts of our environment for lifecycle management of OTA updates; both constituting the UPDATER prototype. Based on this implementation, we analyze later in section VII the case study of the ADAS system.

A. BACKEND IMPLEMENTATION

As described in section IV-B, the variant management is an important part of the lifecycle management process of safety-critical automotive functions since the compatibility of the changed software modules must be guaranteed for all affected system variants. That is why we use a product line model to optimise the reuse of components (cf. section V-A1).

We implement the feature model covering both problem and solution space in the Eclipse-based tool FeatureIDE which supports all phases of feature-oriented software development [58]. In this tool, we are able to graphically model the features of the product line, to maintain their dependencies and to export specific or all variant configurations in XML format. To build the logical architecture of the 150% model (see Figure 8) and enrich it with the contract specifications of each component, we use standard architecture modeling according to SysML, including the following diagram types: Requirement diagram, Block Definition Diagram (BDD) and Internal Block Diagram (IBD). The virtual integration is done by checking the consistency and refinement relations of contracts with the help of the OCRA tool [59]. This

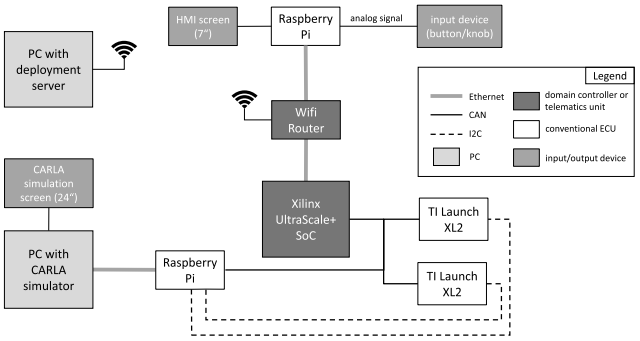


FIGURE 12. E/E architecture of the UPDATER.

tool uses the semi-formal, LTL-based language Othello to write the contracts, and checks their validity within the system architecture based on the model checking tool NuSMV. The SysML logical architecture is associated to a textual architecture description in Othello, called OCRA System Specification (OSS). By means of this textual description, we realize the concept of parametric components as introduced in section IV-B. The mappings between the feature model and the logical architecture are achieved by using the feature names in FeatureIDE as parameters within the parametric component models. Finally, we automate the incremental verification process for updates by implementing Python scripts to identify the consistency and refinement relations to be checked based on delta transformation functions.

In the solution space, the implementation models are available in the tool Simulink of Mathworks, and associated to the components in the logical architecture by mapping them to the product line features. The variability of the Simulink dynamic blocks is managed by the Matlab Variant Manager [60]. By using the embedded coder of Simulink, software code in C programming language is generated for each of the update modules. Then, small modifications are needed to integrate this code into the service-oriented architecture of the middleware. Through a CI/CD pipeline, binary code for each module is then generated.

The released software modules are then saved within the storage of the deployment server. The latter is implemented as a web server in JavaScript including a MySQL database structured as introduced in Figure 10. Certificates are added both on client and server side to allow an HTTPS-based communication.

B. FRONTEND IMPLEMENTATION

The UPDATER E/E architecture consists of multiple ECUs and is attached to a simulated driving environment to provide stimuli for the virtual sensors in the system (see Figure 12). The control loop consists of the following parts:

Main control unit: The SoC Xilinx ZCU102 is used as the main hardware platform on which the middleware, including the update client, is implemented and all computationally intensive application modules run. So, it may be regarded as one of the central domain controller within a modern car. It is connected with every device used in the UPDATER E/E architecture and is able to communicate remotely with the deployment server through a Wifi router.

TI Launch XL2: To fulfill the requirements of safety-critical real-time applications, two TI Launch XL2 RM57L are acting as brake and motor ECU and build the actuation interface to the simulated driving environment. They run with ARM Cortex-R computing cores. They are connected via a Controller Area Network (CAN) bus to the *main control unit*. The outputs of the TI Launch XL2 are converted into simulator compatible signals using an interpreter running on a Raspberry Pi, which builds a gateway to the simulation environment.

Simulated driving environment: Since the UPDATER does not represent a real car, a simulation environment is needed. For this, CARLA, an open-source, high-performance autonomous driving simulator [61], is used. This is the same simulator used in the digital twin during the verification phase (cf. Figure 9). It is running on a conventional PC with extended graphics processing unit (GPU) resources. The PC is connected to the gateway Raspberry Pi via Ethernet. For this purpose, we establish a TCP socket communication between the Raspberry Pi and the simulation environment, and a CAN/I2C communication between the Raspberry Pi and the rest of the demonstrator. Via this connection, simulated sensor signals may be transmitted to the main control unit. The throttle and brake values provided by the *TI Launch XL2* are used for the longitudinal control of the simulated car.

Infotainment system: To visualize the current vehicle status, a second Raspberry Pi is connected to the main control unit via Ethernet (see Figure 12). It receives the current ego vehicle speed, the engine speed (*rpm*) and the information whether a lead car has been detected or not and displays them on a Human Machine Interface (HMI) screen. This function is called instrument cluster in the automotive field. Furthermore, the same Raspberry Pi is used to enable interaction with the user by setting the parameters of ADAS functions or checking for OTA updates. If a new compatible update is available, it can be installed. Updates are checked and installed through the middleware service *update client* implemented in Python. In addition, it allows to display the current software versions of the embedded modules. We summarize all the described features under the category infotainment which motivates the name of the system.

With this setup in place, different ADAS functions for longitudinal control may be deployed, monitored and updated by applying the process and methodology described in section IV. By extending the interface to the simulation environment with a steering signal, other ADAS functions, including lateral control, may be investigated.

We implemented the middleware based on the Xen hypervisor, allowing to run multiple operating systems in parallel [62]. Supported guests include Linux-based systems, real-time operating systems (RTOS) and bare-metal

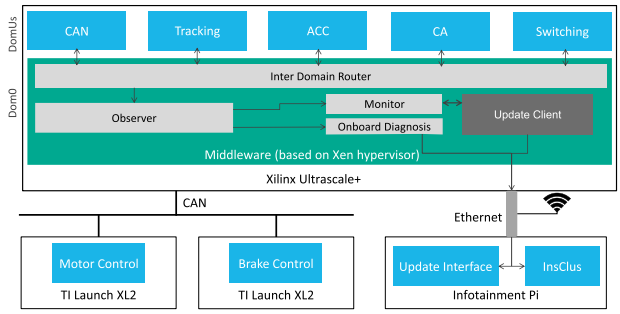


FIGURE 13. Overview of application modules and middleware services of UPDATER.

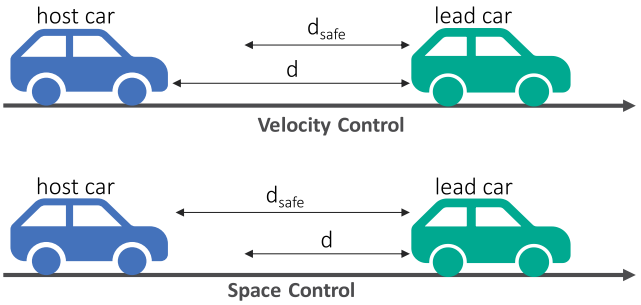


FIGURE 14. How does an ACC system work.

applications. We propose to use Yocto [63] to build a custom Linux distribution for both privileged and unprivileged guest domains.

Several services are provided by the middleware system (see Figure 13). The update client and monitoring are the most important services for the update life cycle. Following the principle of least privilege, services with critical influence on memory and platform configuration, such as the update client, are only run on the privileged domain (called *Dom0* on Xen platforms) if such privileges are required. Other services, as the monitoring system, may run in an unprivileged domain to improve security. For simplicity within this proof of concept implementation, we also install the monitoring service within Dom0 (see Figure 13).

VII. CASE STUDY

A. SYSTEM UNDER UPDATE

The system that is being updated is an extended ACC system.[12] It is based on the speed and distance information of a radar sensor installed on the front of an ego car. There are two different modes in which the standard ACC can operate: *cruise* and *follow*. In the first mode, there is no lead car in the lane, or the distance is greater than the desired safe distance ($d > d_{safe}$). d_{safe} is calculated based on the time headway parameter h given by the driver and the current ego velocity v according to the following equation:

$$d_{safe} = h * v + d_{default}, \tag{5}$$

where $d_{default}$ is the standstill default spacing, usually set to 10 m by regulations. In the cruise mode, the system regulates to the desired speed set by the driver, so the control goal is: $v = v_{set}$ (cf. velocity control in Figure 14). In the follow mode, a leading car in the lane is driving at the same speed or slower than the set desired speed. In this case, the ACC regulates to the desired safe distance and the control goal is: $d = d_{safe}$ (cf. Space Control in Figure 14). Further, we extend the ACC with a Collision Avoidance (CA) feature which is realized by an Autonomous Emergency Brake (AEB) as specified in [64]. This adds a third mode to the system (*safety_critical*) which is triggered when the distance to the lead car is below a critical threshold.

In our ACC system, only the longitudinal control is realized by the UPDATER frontend. This means that the lateral control is out of scope of the system of interest and is realized within the simulated driving environment. For the implementation of the described ACC system, we use a standard control loop based on a PID controller for each of the modes cruise and follow, and a state machine to realise the CA feature, i.e., the safety-critical mode.

B. FUNCTIONAL DECOMPOSITION

The logical architecture realizing the system under update described in section VII-A is derived based on the functional decomposition principle (cf. section IV-B). The components definition starts at the top-level system and ends at the level of modules (cf. section IV-B). Each component $M_{j,i}$ within the architecture is specified by a list of contracts constraining its safe behavior in different environments or operation modes. Formally, this means that the conjunction of the contracts of each component must hold at each time.

Since we aim at updating only the ACC function, our variant representative model is not the whole vehicle, but only the ACC network consisting of the relevant perception, planning and control loop. This network is modeled for the basic variant by the component *ACC_Net* at granularity level 1 (see the SysML BDD in Figure 15). This component takes as input the radar data (*radar_points*, *radar_t*), the curvature of the road *curvature*, the inputs from the driver (set velocity *v_set*, time headway *h* and activation signal *acc_activate*) as well as the current longitudinal velocity of the car *v_long*. After doing all required calculations and control tasks, the system gives two output signals to control the throttle and brake of the vehicle (*throttle_cmd*, *brake_cmd*).

The *ACC_Net* component is specified by a unique contract $\mathcal{C}_{sr}$,[13] which is modeled by an associated SysML constraint block (cf. Figure 15). The exact dependency between the inputs and outputs at this high level of abstraction is not exactly known. For this reason, the contract formulates only that each time radar points are fed in and the ACC is activated, a reaction in form of an active brake or throttle must take place

[12] *extended*: stands for the extension with a safety mode allowing emergency brake in critical situations.

[13] *sr* is an abbreviation of "system reaction".

TABLE 2. Contract $\mathcal{C}_{sr}$ of the top-level (system) component of the basic variant in semi-formal LTL language (OCRA tool).

$\mathcal{C}_{sr}$	A	**always** $(radar$ **and** $(radar\_t \geq 0))$
	G	**always** $((v_{long} \geq 5)$ **and** $(acc\_activate = ture))$ **implies in the future within** $[0, 20]$ $(0 \leq throttle\_cmd \leq 1)$ **or** $(0 \leq brake\_cmd \leq 1))$

TABLE 3. Components in the architecture of the extended ACC system in its basic variant.

L_1	L_2	L_3	L_4
ACC_Net	ACCwSP	SP1	ACC
-	MC1	ACCwCA	CA1
-	BC1	-	Switching1

MC=Motor control; BC=Brake control; SP=Sensor processing; CA=Collision avoidance

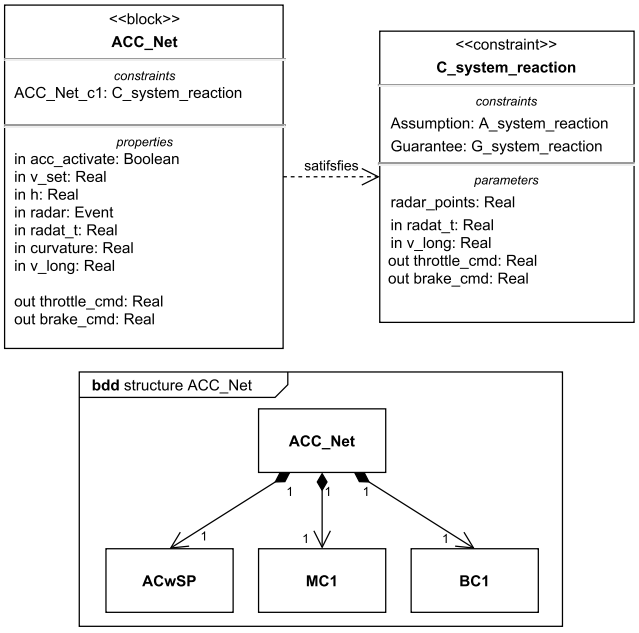


FIGURE 15. Block definition diagram of the top-level system of the basic variant including its contract specification.

in the future after a certain delay time. Formally, this contract is written in a semi-formal version of Linear Temporal Logic in Table 2.

ACC_Net is composed out of the three components *ACCwSP*, *MC1* and *BC1* (see Figure 15). The specification of the other components throughout all four granularity levels is done in the same way. This results in the component hierarchy listed in Table 3.

C. VARIANTS AND CONFIGURATIONS

To build a product line, we define exemplary features of the ACC system for the following two variability categories: 1) sensors/actuators and 2) functional scope of ECU software.

Different models of sensors and actuators, e.g., different types of radar systems or variable vehicle dynamics, should

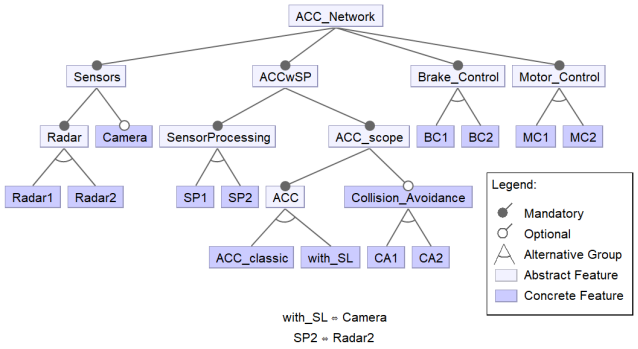


FIGURE 16. Feature model of the considered product line in the investigated ACC use case.

be made configurable in the modeling environment. Here, we choose, as an example, two alternative realizations of the radar sensor (*Radar1* and *Radar2*). These lead directly to two associated alternatives of the Sensor Processing (SP) of the radar input data, identified by the features *SP1* and *SP2* in the feature tree model (see Figure 16). In addition, we cover the possibility of adding extra sensors to the basic ACC configuration such as using the input of a camera to upgrade to a Predictive Adaptive Cruise Control (pACC) variant. For simplification, we assume that a signal *speed_limit* is provided by a corresponding component *SL*, and add a feature *with_SL* to the *ACC* feature (see Figure 16).

For the ECU software, we consider the configuration of the ECU network of UPDATER with different software versions. This applies to each software module in the direct environment of the *ACC* component, namely the brake ECU, the motor ECU, and the switching logic,[14] as well as for the ACC itself which may exist with a safety-related extension with a Collision Avoidance (CA) component and/or as a pACC variant as described above.

After adding constraints, the above described feature tree leads to a total of 48 system variants. The concrete features of the product line are mapped to the components in the 150% architecture model listed in Table 4. To enable the modeling of deltas in the alternative components (cf. Figure 8), a parametric component model is used. This means that similar components, such as *ACC_Net* and *ACC_Net_SL* for the system variant with the speed limit feature, are modeled within one parameterizable component. The parameterization is done based on mapping rules to the features.

D. UPDATES UNDER DEVELOPMENT

We specify different updates under developments (UUDs) for the considered case study system. Each update is an example for one of the three update types: corrective, adaptive and perfective. In the following, we introduce the functional description of these UUD.

[14]interface component between the high level (ACC, CA) and low-level (throttle, brake) control.

TABLE 4. Components in the ACC 150% model, bold: basic variant.

L_1	L_2	L_3	L_4
ACC_Net	**ACCwSP**	**SP1**	**ACC**
ACC_Net_SL	**MC1**	**ACCwCA**	**CA1**
-	**BC1**	SL	**Switching1**
-	ACCwSP_SL	SP2	ACC_SL
-	MC2	ACConly	CA2
-	BC2	ACCwCA_SL	Switching2
-	-	ACConly_SL	-

SL=Speed limit; MC=Motor control; BC=Brake control; SP=Sensor processing; CA=Collision avoidance

- **Corrective update U_c:** In certain situations, the vehicle brakes with an acceleration $a < a_{min}$ (negative value) or accelerates with an acceleration $a > a_{max}$ (positive value) leading to a contract violation. The reason for this are too large output values of ACC controller. The update corrects this behavior by changing the control algorithm of the ACC.
- **Adaptive update U_a:** We assume in this update the fictitious scenario of introducing a new regulation for the permitted highway speed in a specific country or region. This would lead to limiting the highest speed of the ACC to 120 km h^{-1}. So this update "adapts" the ACC system to this new regulation by limiting the system to this speed threshold even when the driver is sending a higher set velocity.
- **Perfective update U_p:** Before the update, the car may switch quite often between throttle and break in the follow mode when the relative velocity to the lead car is near zero. This results in an uncomfortable driving behavior and shall be avoided by this perfective update.

E. INCREMENTAL DESIGN VERIFICATION

The incremental verification is done both in time (evolution within the basic variant) and in space (evolution within the rest of variants in the product line). In the following, we show for each of the three UUDs described in section VII-D which deltas and verification steps are needed for the formally specified modular components. We start from the configuration of the basic variant as described by the functional decomposition in section VII-B, denoted by the version 1.0.

1) U_c: VERSION 1.0 → VERSION 1.1

The first update U_c is a corrective one and is triggered by a contract violation detected during the operation phase based on the monitoring system. We assume that this violation is detected at granularity level $l = 4$ for the ACC component in both its variations: *ACC* and *ACC_SL*. Based on the feature model in FeatureIDE, we may, by selecting the corresponding features, easily identify that this update is required for all 48 variants in the product line.

Since the contracts stay the same, no new virtual integration is needed and the consistency and refinement relations of the contracts are still valid. We assume that the error is triggered due to the ACC controller, realizing the *ACC* component, calculating too small ($a < a_{min}$) or too high acceleration values ($a > a_{max}$) in certain situations of the *follow* mode. To correct this behavior, we incrementally add a simple saturation block in the Simulink implementation model. This change is verified subsequently by means of the digital twin simulation by monitoring the contracts of the ACC component in a scenario reproducing the errors detected at runtime.

2) U_a: VERSION 1.1 → VERSION 2.0

This update requires a change of the contracts of the ACC component, respectively of the alternative concrete component *ACC_SL*. Both components are modeled on a common basis by using the parametric component *ACC_param*. Since the change targets the curise mode of the system, it is applied to the contract $\mathcal{C}_{ACC\_cruise}$. This is expressed by the following time delta:

$$\Delta_{1,4} = \begin{pmatrix} 0 \\ change\_C(\mathcal{C}_{ACC\_cruise}) \\ \Delta Imp \end{pmatrix} \tag{6}$$

Consequently, the conjuncted contract of all three modes $\mathcal{C}_{ACC\_conjuncted}$ is also automatically updated. For the incremental verification of this delta, the refinement relations of this contract at higher granularity levels are verified using the tool OCRA. The verification starts for the basic variant by checking the following two refinement relations directly affected by the change:

$$\mathcal{C}_{\mathbf{ACC\_cruise}} \otimes \mathcal{C}_{CA\_fb1} \otimes \mathcal{C}_{switching} \preceq \mathcal{C}_{syscontract\_cruise\_CA}$$
$$\mathcal{C}_{\mathbf{ACC\_conjuncted}} \otimes \mathcal{C}_{CA\_fb1} \otimes \mathcal{C}_{switching} \preceq \mathcal{C}_{syscontract\_sc}$$

Since the refinement check is passed at granularity level three. The rest of the system hierarchies do not need to be checked. After validating the basic variant, the rest of the model configurations in the variant representative model are verified iteratively. Hereby, the space deltas between each of the variants and the basic variant as well as the already validated refinement relations are taken into account. For the variants with the feature *with_SL* (speed limit extension), the same initial delta is applied to the alternative concrete component *ACC_SL*. Finally, after conducting the virtual integration of the changed contracts for all variants, the update results in the following update matrix describing the validated deltas for the whole product line:

$$\mathbf{U_a} = \begin{bmatrix} 0 & 0 & 0 & \Delta_{1,4} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta_{4,4} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{7}$$

TABLE 5. Changed contract C_{sa} of the switching component; changed parts are in red and added ones in green.

C_{sa}	A	true
	G	always $((CA\_status \neq 0)$ **implies** $(a\_ref = CA\_decel$ **and** $switch = 2))$ **and** always $((CA\_status = 0$ **and** $a\_driver \neq 0)$ **implies** $(a\_ref = a\_driver$ **and** $switch = 4))$ **and** always $((CA\_status = 0$ **and** $a\_driver = 0$ **and** $a\_acc > 0)$ **implies** $((a\_ref = a\_acc$ **and** $switch = 0)$ **or** $(a\_ref = 0$ **and** $switch = 3)))$ **and** always $((CA\_status = 0$ **and** $a\_driver = 0$ **and** $a\_acc < -0.5)$ **implies** $(a\_ref = a\_acc$ **and** $switch = 1))$ **and** always $((CA\_status = 0$ **and** $a\_driver = 0$ **and** $a\_acc < 0$ **and** $a\_acc >= -0.5)$ **implies** $(a\_ref = a\_acc$ **and** $switch = 1)$ **or** $(a\_ref = 0$ **and** $switch = 3))$ **and** always $((CA\_status = 0$ **and** $a\_driver = 0$ **and** $a\_acc = 0)$ **implies** $(a\_ref = 0$ **and** $switch = 3))$

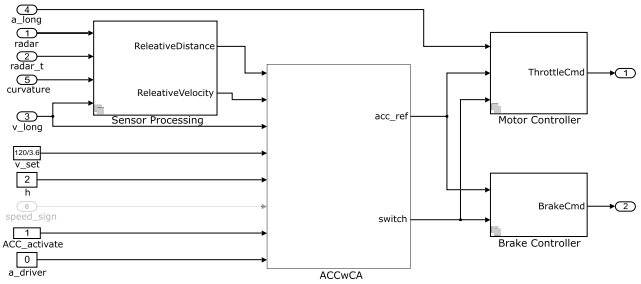


FIGURE 17. Simulink model of the extended ACC system; the *ACCwCA* component is further decomposed into the components *ACC*, *CA* and *Switching*.

3) U_p: VERSION 2.0 → VERSION 2.1

Although there are different possibilities to realize the update U_p, we decide to implement it by extending the behavior of the switching component. Similarly to the adaptive update, this needs a change of the contract of components *Switching*1 and *Switching*2, both modeled by the parametric component *Switching_param*. The delta applied to the contract of C_{sa}[15] of component *Switching*1 is represented in Table 5. This delta is incrementally verified by checking the affected refinement relations in the upper level granularity for the different involved modes of the system. The verification starts by checking the basic variant and then the rest of the variants in the product line. Since the changed contracts still refine the old upper level contracts, no changes are applied to the neighboring or parent components and the update is accepted. The update matrix looks consequently similar to the one of equation 7 with the difference that the deltas are associated to the third and sixth components in the fourth column.

[15] *sa*: abbreviation for *switching_all*.

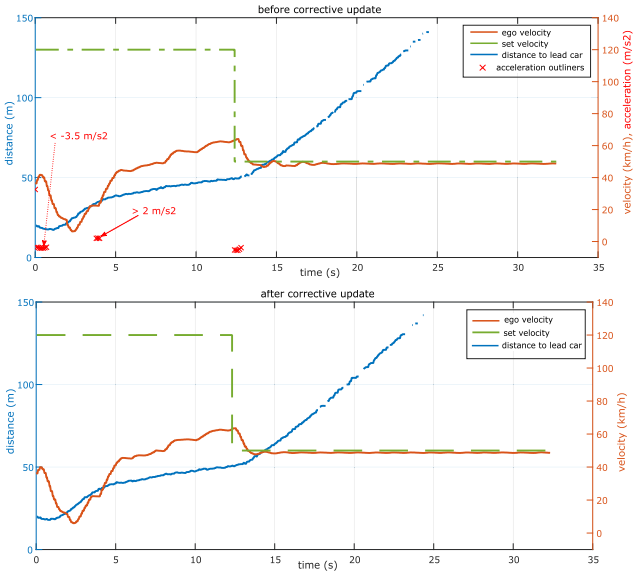


FIGURE 18. Simulation results before and after the corrective update.

F. DIGITAL TWIN TESTING

We implemented a digital twin test system in order to perform incremental variant-aware tests during the verification phase of our process. The digital twin is built by combining the updated implementation models in Matlab/Simulink with simulated driving scenarios in CARLA as described in section V-A2. The Simulink model is depicted in Figure 17. The contract verification is realized based on the RTAMT tool for online monitoring of STL specifications [65].

The achieved simulation results of the digital-twin environment before and after implementing the corrective update U_c are depicted in Figure 18. The simulation is conducted for a scenario starting by following a lead car, with $v_{set} = 120\,\mathrm{km\,h^{-1}}$, and then switching to the *cruise* mode by setting v_{set} to the lower speed of $50\,\mathrm{km\,h^{-1}}$. Although almost no visible differences in the course of the ego velocity can be observed between the two figures, the ACC controller in its version 1.0 is generating acceleration commands outside of the contractually accepted range $[-3.5\,\mathrm{m\,s^{-2}}, 2\,\mathrm{m\,s^{-2}}]$. This is corrected after changing the parameters of the ACC controller adequately.

The adaptive update is implemented in Simulink by limiting the input signal *v_set* of the *ACC* block to the maximum value of $120/3.6 = 33.33\,\mathrm{m\,s^{-1}}$. The updated behavior is tested in the simulation environment by running a scenario where the ego vehicle is in the *cruise* mode and setting the driver speed command to higher values than $120\,\mathrm{km\,h^{-1}}$. The correct behavior could be observed and validated by the fulfillment of the updated module's contract.

The perfective update is implemented by extending the switching module by a block assessing the acceleration command of the ACC with regards to its previous value. If the last acceleration is positive and the new command has a small negative value, the *Switching* module just deactivates

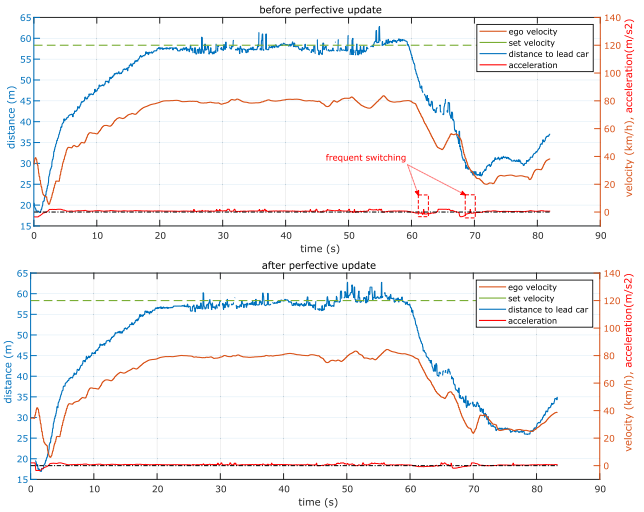


FIGURE 19. Simulation results before and after the perfective update for a driving scenario with a lead car.

the throttle instead of applying the brake. In this way, frequent switches between brake and throttle are avoided (see Figure 19).

VIII. EVALUATION

A. EVALUATION OF VARIANT MANAGEMENT

The proposed approach for variant management based on a feature model and an associated 150% architecture model allows for a high level of reuse of components and a clear traceability of requirements and/or contracts. For the presented simplified example of this case study, we needed a total of 21 concrete components and 23 features to model and manage the product line. By introducing the concept of parametric components, the number of maintained components is further reduced to only nine. For modeling the safety specifications, these components are associated to a total of 13 variable contracts, where each component is specified by only one "strong" contract and/or multiple mode-dependent contracts. After applying the deltas to instantiate specific variants, 22 concrete contracts are derived. The mapping between features and components is achieved by using the features themselves as parameters for configuring the parametric components.

Additionally, by using the Variant Manager of Simulink, we achieved a consistent mapping between the deployable concrete components, called modules, and the implementation models, which are subsequently realized as software services.

B. EVALUATION OF INCREMENTAL VERIFICATION

By using the commonalities of components and contracts in a product line approach, the effort for integration testing is reduced considerably. Checking the refinement for all contracts of the ACC case study in a brute force approach requires an average of 46.81 s per single variant and a total of 37 *min* for all considered variants. By using the concept of parametric reusable components, only the deltas between the variants are tested. For the presented adaptive update, only 10 refinement checks with a duration of 12.81 s were needed to validate the updated contracts. Similarly for the perfective update, 16 refinement checks with a duration of 17.14 s were necessary (see Table 6). For the corrective update, no refinement checks are required at all since the contracts remain the same.

TABLE 6. Results of the incremental contract-based integration for the presented three updates.

Update	$\Delta\mathcal{C}$	# (ref checks)	T (ref checks)
U_c	0	0	0
U_a	2	10	12.81 s
U_p	2	16	17.14 s

TABLE 7. Measurement of duration of deployment steps for an update of the ACC module (based on 10 measurement repetitions).

Step ID	Deployment step	Mean duration (s)
1	get current configuration	0.117
2	authentication and availability check	0.592
3	download	0.577
4	unpack and install	0.215
5	system restart	29.127
6	configuration and startup	3.674

C. EVALUATION OF DEPLOYMENT SYSTEM

The deployment system is evaluated by conducting different updates developed according to the presented approach. Table 7 shows the time required for the deployment of an update of the *ACC* module. The deployment package (zip file) has a size of 173 kB. The measurements show that, for only one small update package, the time duration of the first four steps is short (less than one second per step). However, by increasing size of deployment packages and the number of updated modules, the duration of steps 3 and 4 increases. Since the current implementation of the middleware requires a reboot of the system, the longest deployment time is spent for this task (cf. step 5 in Table 7). Subsequently, re-starting the guest domains with their potentially updated software takes about 3.6 s. However, the duration of steps 5 and 6 is not dependent on the size and number of update packages. Nevertheless, in this period the vehicle cannot be used.

D. EVALUATION OF MONITORING SYSTEM

As mentioned in section VII-F, we use the RTAMT monitor [65] by installing it on dom0 to check the contracts during system runtime. This library calculates a robustness metric to assess the fulfillment of contracts expressed as STL specification. Positive values mean that the contracts are fulfilled and negative ones that there is some contract violation.

TABLE 8. Summary of pros and cons/challenges of the presented methods and approaches.

Method/Approach	Pros	Cons/Challenges
formalizing requirements into contracts	end-to-end use for verification, formal proofs	increased effort for contract derivation, expert knowledge of contract language
update types	type-dependent update process, compliance with ISO/IEC/IEEE 14764	Some updates are hard to classify (mixture of different types).
contract-based design	easier integration, multi-level updates (multiple hierarchies)	applied only to critical safety properties
150% architecture model	high reuse degree, efficient verification through integrated contracts	High number of models need to be continuously maintained and kept consistent.
delta-based design	fine-granular changes, enabler for incremental analyses	ΔImp depends on modeling and programming method and technology.
incremental virtual integration	considers only representative variants, covers both variants and versions	requires up-to-date and complete variant models, needed manual changes in case of failed tests
digital twin	flexible and re-configurable, parallel testing of variants	high-fidelity simulation models necessary
"pull" deployment	secure OTA communication, updates deployed shortly after being released, updates of single modules or subsystems possible	security of in-vehicle network, maintenance of large databases
runtime monitoring	using same contracts as in pre-deployment phase, guarantees safe behavior at usage time	additional consumption of resources
hypervisor-based middleware	isolation of mixed-criticality applications, supervised communication of guest domains	overhead in processing time

In order to automatically monitor the updated components, the contracts, deployed as JSON file, are read by a Python script which initializes then the RTAMT monitor. For our simplified E/E architecture, we didn't notice any limiting constraints on the platform resources due to the monitoring overhead. For further enhancement, due to e.g., resource limitations, the Python implementation should be translated into a compiled language implementation, such as C/C++.

IX. THREATS TO VALIDITY

Despite the validation of the methodology introduced in this paper for the described ADAS case study, some limitations may constitute potential threats to validity in future OTA updates management systems.

First, the investigated case study is rather limited in its complexity in terms of the considered variant space (only 48 system variants) and it does not represent real-world fleet data. Also the considered function of the extended ACC has only autonomy level 1-2 according to the SAE classification [66], whereas SAE 3+ functions require for their realization significantly larger amount of sensor and software components. However, despite the simplified example, we may assume that the presented framework is representative of larger functional networks with higher automation levels since it includes the three main functional parts: perception, planning and control. Still, a validation for a more realistic product line needs to be conducted.

Furthermore, we did not focus on security aspects in UPDATER and limited its scope to the use of a secure communication channel using Transport Layer Security (TLS)/HTTPS. So, neither the integration of an efficient key management system for the fleet, nor security mechanisms within the internal vehicular network, e.g. using hardware security modules, have been considered.

Another threat to validity is the set of used Assume/Guarantee contracts. Since the contracts focus on safety properties, they are not complete, and applying our approach guarantees only the fulfillment of the specified safety requirements. Subsequently, classical unit, integration and regression tests are still needed after each update to ensure that the rest of the non-safety-relevant requirements is not affected. In theory, it is possible to "contractify" all functional and non-functional requirements of the system, however, this requires a very high effort for generating and maintaining the contracts throughout the lifecycle of the product lines. Novel methods based on artificial intelligence could support in making this vision a reality by automatically generating the contracts from existing validated implementations.

X. CONCLUSION

This paper outlined a systems approach for the design, implementation, verification and deployment of over-the-air updates of vehicles. The approach is centered on a process designed in accordance with known standards and which can be tailored to the needs of different manufacturers. It provides mechanisms for dealing with incremental and variant-aware verification, early integration, cyber security risks and monitoring during operation. The different steps of the process allow efficient development of software that can be deployed securely over-the-air to a safety-critical automotive system. The concept of contract-based design was implemented to facilitate the handling of integration

issues while maintaining the system integrity and safety. This paper also proposes ways to test updated components in such a way that minimal effort needs to be taken in order to ensure the safety of the system under consideration as well as its variants. The prerequisites for the approach rely on a well maintained system model, the use of contracts and incremental verification of updates. We evaluated our approach using the demonstrator ''UPDATER'' and showed that is in fact applicable to real world systems. A summary of the pros and cons, or rather challenges, of the introduced methods and approaches is given in Table 8. Our work may serve as a guideline for automotive companies planning to introduce OTA updates for safety-critical functions. It can also support the development of future standards for safety-critical automotive updates.

In future work, we will evaluate the applicability of the approach for different kinds of update triggers as well as the influence of update priorities (critical/urgent, non-critical) on the process. Additionally, the scalability of the approach and the UPDATER prototype needs to be studied by considering a larger product line with more variants. Finally, although the presented middleware is based on service-oriented communication between the deployment modules, a thorough study of the process applicability for microservices and event-driven architectures should be conducted.

ACKNOWLEDGMENT

The authors thank Nadir Khan for his continuous support with regards to the security aspects of UPDATER as well as the student Yao Xiao for his strong engagement during the implementation and testing phase.

REFERENCES

[1] P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha, *Autonomous Vehicles: State of the Art, Future Trends, and Challenges*. Cham, Switzerland: Springer, 2019, pp. 347–367.

[2] M. Staron, *Automotive Software Architectures*. Cham, Switzerland: Springer, 2017.

[3] V. Antinyan, ''Revealing the complexity of automotive software,'' in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, P. Devanbu, M. Cohen, and T. Zimmermann, Eds. New York, NY, USA: ACM, Nov. 2020, pp. 1525–1528.

[4] A. Vetter and E. Sax, ''Hierarchical versioning to increase compatibility in signal-oriented vehicle networks,'' in *Proc. 27th Int. Conf. Syst. Eng. (ICSEng)*, H. Selvaraj, G. Chmaj, and D. Zydek, Eds. Cham, Switzerland: Springer, 2021, pp. 435–444.

[5] *Private Autonomous Vehicles: The Other Side of the Robo-Taxi Story*. Accessed: Jan. 18, 2022. [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/private-autonomous-vehicles-the-other-side-of-the-robo-taxi-story

[6] H. Guissouma, H. Klare, E. Sax, and E. Burger, ''An empirical study on the current and future challenges of automotive software release and configuration management,'' in *Proc. 44th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2018, pp. 298–305.

[7] T. Chowdhury, E. Lesiuta, K. Rikley, C.-W. Lin, E. Kang, B. Kim, S. Shiraishi, M. Lawford, and A. Wassyng, ''Safe and secure automotive over-the-air updates,'' in *Developments in Language Theory* (Lecture Notes in Computer Science), vol. 11088, M. Hoshi and S. Seki, Eds. Cham, Switzerland: Springer, 2018, pp. 172–187.

[8] M. Traub, H.-J. Vogel, E. Sax, T. Streichert, and J. Harri, ''Digitalization in automotive and industrial systems,'' in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1203–1204.

[9] K. Schwaber and J. Sutherland, ''The scrum guide,'' *Scrum Alliance*, vol. 21, no. 1, pp. 1–38, Jul. 2011.

[10] T. Brandt and T. Tamisier, ''The future connected car-safely developed thanks to UNECE WP. 29?'' in *Proc. 21st Internationales Stuttgarter Symp.*, M. Bargende, H.-C. Reuss, and A. Wagner, Eds. Wiesbaden, Germany: Springer, 2021, pp. 461–473.

[11] S. Jiang, ''Vehicle E/E architecture and its adaptation to new technical trends,'' SAE Tech. Paper 2019-01-0862, 2019.

[12] D. Scheer, O. Glodd, H. Günther, Y. Duhr, and A. Schmid, ''STAR3—Eine neue generation der E/E-architektur,'' *Sonderprojekte ATZ/MTZ*, vol. 25, no. S1, pp. 72–79, Dec. 2020, doi: 10.1007/s41491-020-0056-5.

[13] M. Maul, G. Becker, and U. Bernhard, ''Service-oriented EE zone architecture key elements for new market segments,'' *ATZelektronik Worldwide*, vol. 13, no. 1, pp. 36–41, Feb. 2018.

[14] *Explanation of Adaptive Platform Design*, document Release R19-11, AUTOSAR, Nov. 2019. Accessed: Feb. 18, 2022. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_EXP_PlatformDesign.pdf

[15] G. Schuh, S. Aleksic, and S. Rudolf, ''Module-based release management for technical changes,'' in *Progress in Systems Engineering*, H. Selvaraj, D. Zydek, and G. Chmaj, Eds. Cham, Switzerland: Springer, 2015, pp. 293–298.

[16] J. M. Quigley and K. L. Robertson, *Configuration Management: Theory and Application for Engineers, Managers, and Practitioners*. Boca Raton, FL, USA: CRC Press, 2019.

[17] S. P. Berczuk, S. Berczuk, and B. Appleton, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Reading, MA, USA: Addison-Wesley, 2003.

[18] B.-J. Madauss, *Dokumentations-und Konfigurationsmanagement im Projekt*. Berlin, Germany: Springer, 2020, pp. 457–484.

[19] I. Sommerville, *Software Engineering*, vol. 137035152, 9th ed. London, U.K.: Pearson, 2011.

[20] D. Spinellis, ''Git,'' *IEEE Softw.*, vol. 29, no. 3, pp. 100–101, Apr. 2012.

[21] M. Bellanger and E. Marmounier, ''Service oriented architecture: Impacts and challenges of an architecture paradigm change,'' in *Proc. 10th Eur. Congr. Embedded Real Time Softw. Syst. (ERTS)*, Toulouse, France, Jan. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02457046/

[22] H. Holdschick, ''Challenges in the evolution of model-based software product lines in the automotive domain,'' in *Proc. 4th Int. Workshop Feature-Oriented Softw. Develop. (FOSD)*, New York, NY, USA, 2012, pp. 70–73, doi: 10.1145/2377816.2377826.

[23] H. Guissouma, C. P. Hohl, H. Stoll, and E. Sax, ''Variability-aware process extension for updating cyber physical systems over the air,'' in *Proc. 9th Medit. Conf. Embedded Comput. (MECO)*, Jun. 2020, pp. 1–8.

[24] D. K. Nilsson, L. Sun, and T. Nakajima, ''A framework for self-verification of firmware updates over the air in vehicle ECUs,'' in *Proc. IEEE Globecom Workshops*, Nov. 2008, pp. 1–5.

[25] S. Halder, A. Ghosal, and M. Conti, ''Secure over-the-air software updates in connected vehicles: A survey,'' *Comput. Netw.*, vol. 178, Sep. 2020, Art. no. 107343. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128619314963

[26] G. Kim and I. Y. Jung, ''Integrity assurance of OTA software update in smart vehicles,'' *Int. J. Smart Sens. Intell. Syst.*, vol. 12, no. 1, pp. 1–8, 2019.

[27] W. Gao, W. G. Hatcher, and W. Yu, ''A survey of blockchain: Techniques, applications, and challenges,'' in *Proc. 27th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2018, pp. 1–11.

[28] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, ''Analysis of the HTTPS certificate ecosystem,'' in *Proc. Conf. Internet Meas. Conf.*, New York, NY, USA, Oct. 2013, pp. 291–304, doi: 10.1145/2504730.2504755.

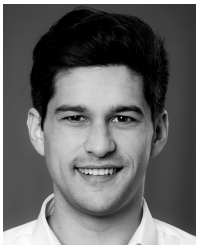
[29] WP UNECE. (2018). *GRVA, 'Draft Recommendation on Cyber Security of the Task Force on Cyber Security and Over-the-Air Issues of Unece WP. 29 GRVA'*. [Online]. Available: https://unece.org/fileadmin/DAM/trans/doc/2018/wp29grva/GRVA-01-18.pdf

[30] Vehicle Cybersecurity Systems Engineering Committee, ''Cybersecurity guidebook for cyber-physical vehicle systems,'' SAE Int. J3061, 2016.

[31] International Standard Electrical, Electronic Components, and General System Aspects, *Road Vehicles—Cybersecurity Engineering*, Standard ISO/SAE 21434:2021, International Organization for Standardization, 2021.

[32] *Explanation of Firmware Over-the-Air*, document Release R19-11, AUTOSAR, Nov. 2019. Accessed: Feb. 18, 2022. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_EXP_FirmwareOverTheAir.pdf
[33] *Specification of Update and Configuration Management*, document Release R19-11, AUTOSAR, Nov. 2019. Accessed: Feb. 18, 2022. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_SWS_UpdateAndConfigManagement.pdf
[34] T. K. Kuppusamy, L. A. DeLong, and J. Cappos, "Uptane: Security and customizability of software updates for vehicles," *IEEE Veh. Technol. Mag.*, vol. 13, no. 1, pp. 66–73, Mar. 2018.
[35] eSync. *A Multi-Company Initiative for OTA Updates and Diagnostics*. Accessed: Jan. 23, 2022. [Online]. Available: https://esyncalliance.org/
[36] B. Meyer, "Applying 'design by contract,"' *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.
[37] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, Jan. 2012.
[38] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, *Contracts for System Design*. Boston, MA, USA: Now, 2012.
[39] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.
[40] K. Watanabe, E. Kang, C.-W. Lin, and S. Shiraishi, "Runtime monitoring for safety of intelligent vehicles," in *Proc. 55th Annu. Design Autom. Conf.*, New York, NY, USA, Jun. 2018, pp. 1–6, doi: 10.1145/3195970.3199856.
[41] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annu. Symp. Found. Comput. Sci. (SFCS)*, Sep. 1977, pp. 46–57.
[42] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Formal Modeling and Analysis of Timed Systems*, K. Chatterjee and T. A. Henzinger, Eds. Berlin, Germany: Springer, 2010, pp. 92–106.
[43] H. Wang, D. Zhong, T. Zhao, and F. Ren, "Integrating model checking with SysML in complex system safety analysis," *IEEE Access*, vol. 7, pp. 16561–16571, 2019.
[44] J. Xie, W. Tan, Z. Yang, S. Li, L. Xing, and Z. Huang, "SysML-based compositional verification and safety analysis for safety-critical cyber-physical systems," *Connection Sci.*, vol. 34, no. 1, pp. 911–941, 2021, doi: 10.1080/09540091.2021.2017853.
[45] E. Zapridou, E. Bartocci, and P. Katsaros, "Runtime verification of autonomous driving systems in Carla," in *Runtime Verification*, J. Deshmukh and D. Ničković, Eds. Cham, Switzerland: Springer, 2020, pp. 172–183.
[46] N. Ayres, L. Deka, and D. Paluszczyszyn, "Continuous automotive software updates through container image layers," *Electronics*, vol. 10, no. 6, p. 739, Mar. 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/6/739
[47] Y. Wang and Q. Bao, "Adapting a container infrastructure for autonomous vehicle development," in *Proc. 10th Annu. Comput. Commun. Workshop Conf.*, 2020, pp. 182–187.
[48] H. F. Stoll, "Die (re-)konfigurierbare fahrzeugarchitektur," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany, 2021.
[49] *Software Engineering—Software Life Cycle Processes—Maintenance*, Standard ISO/IEC/IEEE 14764:2022, 2022, vol. 14764.
[50] A. M. Phillips, "Functional decomposition in a vehicle control system," in *Proc. Amer. Control Conf.*, vol. 5, 2002, pp. 3713–3718.
[51] H. Guissouma, M. Schindewolf, and E. Sax, "ICARUS–incremental design and verification of software updates in safety-critical product lines," in *Proc. 47th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Sep. 2021, pp. 371–378.
[52] A. Dearle, "Software deployment, past, present and future," in *Proc. Future Softw. Eng. (FOSE)*, 2007, pp. 269–284.
[53] K. C. Kang and H. Lee, "Variability modeling," in *Systems and Software Variability Management*, vol. 5, R. Capilla, J. Bosch, and K.-C. Kang, Eds. Berlin, Germany: Springer, 2013, pp. 25–42.
[54] D. Nešić, M. Nyberg, and B. Gallina, "Constructing product-line safety cases from contract-based specifications," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, C.-C. Hung and G. A. Papadopoulos, Eds. New York, NY, USA, Apr. 2019, pp. 2022–2031.
[55] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proc. 6th Int. Conf. Generative Program. Compon. Eng. (GPCE)*, New York, NY, USA, 2007, pp. 95–104, doi: 10.1145/1289971.1289989.
[56] G. Heiser, "The role of virtualization in embedded systems," in *Proc. 1st Workshop Isolation Integr. Embedded Syst. (IIES)*, New York, NY, USA, 2008, pp. 11–16, doi: 10.1145/1435458.1435461.
[57] O. Sander, T. Sandmann, V. V. Duy, S. Bahr, F. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lubbers, J. Hairbucher, A. Richter, C. Herber, and A. Herkersdorf, "Hardware virtualization support for shared resources in mixed-criticality multicore systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2014, pp. 1–6.
[58] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An extensible framework for feature-oriented software development," *Sci. Comput. Programm.*, vol. 79, pp. 70–85, Jan. 2014.
[59] A. Cimatti, M. Dorigatti, and S. Tonetta, "OCRA: A tool for checking the refinement of temporal contracts," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2013, pp. 702–705.
[60] MathWorks. *Variant Manager Overview*. Accessed: Feb. 10, 2022. [Online]. Available: https://www.mathworks.com/help/simulink/gui/variant-manager-interface.html
[61] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proc. 1st Annu. Conf. Robot Learn.*, vol. 78, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., Nov. 2017, pp. 1–16. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a.html
[62] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 19th ACM Symp. Operating Syst. Princ. (SOSP)*, New York, NY, USA, 2003, pp. 164–177, doi: 10.1145/945445.945462.
[63] O. Salvador and D. Angolini, *Embedded Linux Development With Yocto Project*. Birmingham, U.K.: Packt, 2014.
[64] UN-WP.29/GRVA. (Feb. 2019). *Agreed Proposal Based on ECE/Trans/WP.29/Grva/2019/5*. [Online]. Available: https://unece.org/fileadmin/DAM/trans/doc/2019/wp29grva/GRVA-02-39c1e.pdf
[65] D. Ničković and T. Yamaguchi, "RTAMT: Online robustness monitors from STL," in *Automated Technology for Verification and Analysis*, D. V. Hung and O. Sokolsky, Eds. Cham, Switzerland: Springer, 2020, pp. 564–571.
[66] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for on-Road Motor Vehicles*, Oceania-Regional Anti-Doping Organizations Committee, Suva, Fiji, Jun. 2018.
[67] T. Strathmann, G. Hake, H. Guissouma, C. P. Hohl, Y. Bebawy, S. V. Maelen, and A. Koerner, "Project overview for step-up! CPS–process, methods and technologies for updating safety-critical cyber-physical systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 1326–1329.

HOUSSEM GUISSOUMA (Member, IEEE) received the bachelor's and master's degrees in electrical engineering and information technology from the Karlsruhe Institute of Technology (KIT), Karlsruhe, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree in automotive electronics systems engineering with the Institute for Information Processing Technologies. His research interest includes new design and verification methods for over the air (OTA) software updates in the automotive field.



CARL PHILIPP HOHL received the degree in electrical engineering from the Karlsruhe Institute of Technology. He is currently pursuing the Ph.D. degree with the FZI Forschungszentrum Informatik. He has participated in the Formula Student Competition. His research interests include processes, methods, and tools working with automotive E/E-architectures and similar systems.

FABIAN LESNIAK received the bachelor's and master's degrees in electrical engineering from the Karlsruhe Institute of Technology (KIT), in 2015 and 2018, respectively, where he is currently pursuing the Ph.D. degree with the Institute for Information Processing Technologies in the field of mixed-critical embedded systems and dynamically reconfigurable hardware.

MARC SCHINDEWOLF received the bachelor's and master's degrees in electrical engineering and information technology from the Karlsruhe Institute of Technology, in 2016 and 2019, respectively, where he is currently pursuing the Ph.D. degree with the Institute for Information Processing Technologies in the area of automotive E/E architectures.

JÜRGEN BECKER (Senior Member, IEEE) received the diploma and Ph.D. (Dr.-Ing.) degrees from Technical University Kaiserslautern, Germany. Since 2001, he has been a Full Professor of embedded electronic systems and the Head of the Institute for Information Processing Technologies (ITIV), Karlsruhe Institute of Technology (KIT). He has authored more than 400 peer-reviewed papers and he is active in various committees of international conferences. His research interests include hardware/software systems-on-chip (SoC), cyber-physical systems (CPS), heterogeneous multicore architectures, reconfigurable computing, embedded systems in automotive, industry 4.0, avionics incl. HPC, and AI integration.

ERIC SAX is currently the Head of the Institute for Information Processing Technology (itiv.kit.edu), Karlsruhe Institute of Technology. A tight link to industry derives from the fact that he was responsible for E/E at Daimler Buses, from 2009 to 2014, and before he was the Head of test engineering at the MBtech Group. His research interests include processes, methods, and tools in systems engineering, data driven, and service-oriented architectures supported by the idea of machine learning.