Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 1475

Quo vadis, AUTOSAR?

Christine Jakobs[1], Peter Tröger[1]

Abstract: AUTOSAR is the de-facto standard for developing automotive software. It is utilized by leading car manufacturers and suppliers as a common platform for software portability and interoperability. Future car applications such as autonomous driving and Internet service usage change the static deployment model on single ECUs towards the classic idea of a dynamic distributed system. This paper identifies properties of AUTOSAR which do not match to such upcoming scenarios. We discuss how established patterns from distributed software development may contribute to an improved version of AUTOSAR that is more suited for current and future applications.

Keywords: middleware; AUTOSAR; component model; standardization; container; software architecture

1 Introduction

Embedded software plays an important role for the functionality of modern cars. It allows the realization of comfort functionalities such as driving assistance or navigation, but is also important for critical parts such as injection control or braking support. In recent years, the automotive industry developed a constantly rising demand for software-backed features, often because these functionalities are the main distinguishing factor in competition with other vendors.

In 2003, the automotive industry established the *AUTomotive Open System ARchitecture (AUTOSAR)* as unified development platform and middleware for embedded connected software. It continues the tradition of OSEK/VDX, a standardization effort for vehicle software that originated in the 90s in Germany and France.

AUTOSAR supports the development of embedded distributed software that is portable across heterogeneous automotive hardware platforms and can be verified in terms of reliability and safety. Such middleware goals are not new, they are on the agenda since the first occurrence of the term in the 1960s. Several industrial domains successfully managed to invent solutions that deal with standardized software entities and their execution environment. This lead to well-known solutions such as CORBA which is widely used in the aviation industry due to its combination of rigorous standardization, backward compatibility and support for different programming languages.

[1] Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Straße der Nationen 62, 09111 Chemnitz, Deutschland [christine.jakobs|peter.troeger]@informatik.tu-chemnitz.de

doi:10.18420/in2017_147

Given the fact that middleware has a long history in software engineering, it seems surprising that the automotive industry follows a comparatively young and very isolated approach. Other middleware stacks beside AUTOSAR offer a rich set of development tools, such as Eclipse, Visual Studio, and NetBeans, and support modern paradigms such as dependency injection, component repositories, or service orientation. The automotive world, in contrast, maintains an isolated universe of middleware tools, libraries and programming rules. Studies have shown that new upcoming challenges such as network security, over-the-air software update and payed vehicle feature activation get more and more important [Mc16] but with AUTOSAR, they must be tackled without re-using existing solutions. This raises the question if such a self-contained ecosystem is really the best-possible approach in terms of flexibility, cost effectiveness, security, reliability and timeliness. In the following sections, we discuss this issue with respect to the AUTOSAR 4.2 specification[2], focusing on two main questions:

- Are there unique capabilities in AUTOSAR that are hard to realize with other middleware frameworks?
- Is AUTOSAR prepared for the upcoming challenges of autonomous connected cars?

We begin with the scope and implementation requirements of the AUTOSAR standard in Section 2. In Section 3, we discuss the system model assumptions given in the AUTOSAR specification. Section 4 targets the software layers defined by AUTOSAR and Section 5 the component model. In each section, we derive some recommendation how to proceed with AUTOSAR as a platform for vehicle software development.

2 Scope and Requirements in AUTOSAR

The creation of a specification such as AUTOSAR demands significant collaborative efforts in an environment where competition is the default. A group of industry vendors and research institutions tries to find a lowest common denominator for some technology, protocol or API. This can only work if they have a clearly defined scope of their work, so that the resulting common denominators are agreeable by all participants.

One example from another domain is the *CORBA Component Model (CCM)* standard by the *Object Management Group (OMG)*. It starts the specification text by describing the scope of the document [Gr12b]. Goal settings such as "a programming model for constructing component implementations" are referred to by later parts of the specification.

AUTOSAR specifies a "standardized and open software architecture for automotive electronic control units (ECUs)"[AU14] as its primary goal. When going further through the basic information the idea is also to specify and "describe application interfaces and a methodology". This includes the following aspects [AUe, AUg, AUa]:

[2] The specification can be obtained from `http://www.autosar.org`

Architecture: AUTOSAR defines a layered architecture, composed of the *application layer* build from software components, the *run-time environment (RTE)* as the communication middleware and the *basic software* layer with a standardized API.

Interfaces: AUTOSAR defines an interface description syntax and a set of standard functions that must be implemented by a compliant execution environment.

Methodology: AUTOSAR specifies a common workflow for the software development process.

Orthogonal to these pillars, there are three main objectives frequently mentioned in the specification [AUe]: *non-functional guarantees*, *portability* and *interoperability*.

The support for non-functional guarantees is crucial in safety-critical systems and therefore mandatory in technologies such as AUTOSAR.

Portability is the capability of transferring software to new hardware. In contrast to many frameworks outside the automotive field, AUTOSAR does not seem to aim for binary portability that works without recompilation. This is obviously fine for the closed world of a traditional vehicle, where software only changes infrequently after the delivery to the customer.

Interoperability exists as third general requirement in AUTOSAR, but does not seem to be the major driving factor in the standardization. It is achieved by cross-checking all potential communication of AUTOSAR components at configuration / build time. This again implies a closed system that has no dynamic external communication. All interaction partners must be completely aware of each other. AUTOSAR therefore chooses best-possible predictability over best-possible interoperability, while most communication middleware stacks decide for the exact opposite. This is reasonable for safety critical systems with no dynamics at run-time but may be a larger problem in future open systems.

With respect to these objectives, AUTOSAR specifies rules for a high-level development process, low-level abstractions, portability mechanisms, communication patterns, non-functional properties and the programming language. It also includes the definition of a functional architecture with blueprints for software layers and hardware abstractions, implementation details such as XML formats, C header files, variable names, C90 restrictions, development process details and tool functionalities. The result is a complex mixture of high and low level aspects, which makes the specification look more like an implementation documentation than an adoptable standard. This way of describing things stands in contrast to many other successful middleware specifications. Standardization documents for CORBA or ECMA.NET focus on one layer/aspect at a time and references to other standards for everything that is not covered. This creates a collection of small and focused specifications, where each of them solves one problem at a time. One special example is the JavaEE specification set, which mainly combines other Java standards into an overall and complete definition of component model, communication middleware and standardized execution environment.

Standards in other domains for protocols, data formats or software interfaces typically specify the minimum, optimum and maximum amount of functionality that must be implemented in order be compliant to them. One example is CORBA, where "The minimum required for a CORBA-compliant system is adherence to the specifications in this standard and one mapping. Each additional language mapping is a separate, optional compliance point" [Gr12a]. The definition of such compliance commonly relies on the RFC 2119 specification [Br97], which defines keywords such as SHALL and MAY to indicate a requirement level.

Even though the AUTOSAR standard also refers to RFC2119 [AUe], there is no real prioritization of requirements. This leads to conflicting statements in different parts, for example regarding OSEK legacy support vs. standardized API usage. Given the sometimes contrary AUTOSAR goals on different levels, it remains often unclear which properties, services, functionalities and non-functional attributes are mandatory for AUTOSAR compliance. The descriptions are scattered over multiple documents [AUf, AUd, AUe] and are described with a widely differing level of granularity. Some of them even refer to the standardization process itself and not to the intended results of AUTOSAR implementations e.g. "AUTOSAR shall support the work-share in large development projects via well-defined exchange formats" [AUd].

The lack of sharp implementation priorities allows the (extreme) conclusion that either all requirements from all documents must be fulfilled to be AUTOSAR-compliant, or that nothing is mandatory. The first case would be impossible to implement in reality, while the latter would mean that every software ever written is AUTOSAR compliant. This example shows that the AUTOSAR specifications leave unnecessary large gaps for subjective interpretation, just by sub-optimal structuring and formulation, a problem we define as *model uncertainty* [Sm]. Personal feedback from automotive developers underlines that this problem exists in reality and leads to significant resources being invested for figuring out if and how some software parts are AUTOSAR compliant or not.

A good way to overcome these problems would be to split the specification in different parts. This would lead to better decoupling and easier adaption of the API. An extension of the standard by introducing profiles to create a clear structure of the requirements would allow to focus on important parts per single document. Also the consequent usage of the referred RFC2119 would lead to a clear prioritization of that parts which have to be implemented for compliance.

3 System Model

In distributed systems, one fundamental assumption is the independence of nodes. Software can be independently installed, updated and may also fail independently. This eases the realization of fault tolerance and extensibility, but makes the overall system less predictable in its behavior.

AUTOSAR is intended for monolithic software systems that span over multiple ECUs connected by a real-time communication bus. Software layers exist only in source code and dissolve into a single binary per ECU node after compilation (see Figure 1). This leads to the fact that configuration has to take place before runtime [AUb], which is one of the few irrefutable AUTOSAR development principles shining through in all documents.

The **static monolith** concept of AUTOSAR is a relevant pain point for the software development process itself. Although AUTOSAR has all the necessary building blocks for decoupled and decentralized development, such as a component model and abstract interface descriptions, it always ends up in demanding whole system compilation, configuration and optimization for testing. Decoupled ECU operation, cross-ECU fail-over mechanisms, dynamic load balancing, dynamic addition of components, or partial updates for some of the ECUs violate the whole system validation assumption and therefore become complicated to be realized in practice.

The obvious motivation for the static monolith approach is the wish for predictable behavior in terms of non-functional properties, such as timeliness and reliability. When each and every component interaction is statically configured and described in machine readable formats, it becomes possible to analyze and plan timing, safety, security and other properties for the system as a whole. Real-time communication and resource usage can be heavily optimized at configuration time, so that no unnecessary hardware adds to the vehicle weight. This aspect is widely unsupported by mainstream middleware, despite the well-known TTA or RT-CORBA [Ob05, KB03], so the choice being made in AUTOSAR is reasonable from a historic perspective. But it becomes increasingly invalid for future cars. These systems are expected to support software update after delivery [Ta16], e.g. for activating features as pay-per-use or for getting newer functionality as over-the-air update.

Newer developments such as SOME/IP [AUj] show that the stack must be heavily tailored for such dynamics, which exist in other frameworks for decades. Adding naming services, encrypted communication, service discovery and late binding to classical AUTOSAR immediately violates the *everything is static at run-time* mentality of the core specification. Some companies such as Tesla therefore try to separate the static AUTOSAR system with safety-critical functionality from the dynamic parts, but the bridging points for exchange of data and commands still impose a risk. A popular example about what can go wrong is the Jeep hack from 2015, where security researchers where able to remotely control brakes and acceleration of a car [Sc15].

Although the vehicle industry has to achieve a predictable behavior, the current AUTOSAR software development process seems not to be future proof anymore. With the monolith approach, it is not possible to evolve towards dynamic software deployment which e.g. allows software updates during runtime. The attempt of adjusting the old mechanisms and models may work for the moment, but is clearly not future proof.

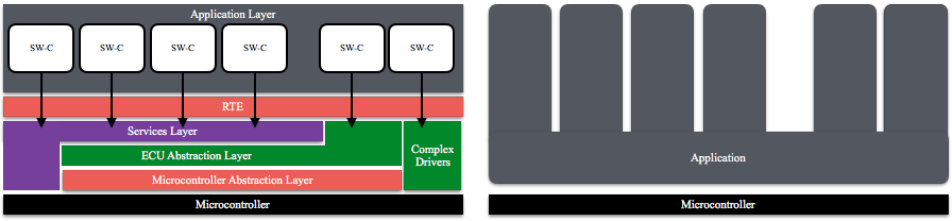

Fig. 1: AUTOSAR software layers before and after compilation.

4 Layers in AUTOSAR

Layering is a traditional concept for dealing with complexity in large software systems. The independent evolution of each layer is enabled by clear API barriers and the separation of responsibilities.

AUTOSAR defines architectural layers [AUc] that look similar to the ones known from classical operating system design (see Figure 1):

Microcontroller Abstraction Layer: Similar to the hardware abstraction layer (HAL) in an operating system (OS) kernel.
ECU Abstraction Layer: Similar to the I/O management parts in an OS kernel.
Services Layer: High-level functionality, which is comparable to an OS syscall interface.
Runtime Environment (RTE) layer: Portability layer and coordination of component communication, similar to standard middleware products.

AUTOSAR applications can use the Services Layer as their operational foundation for local functionality, but may also access the ECU abstraction layer directly [AUc]. Another option is the utilization of *complex drivers*, which basically allows the application to circumvent all abstractions and talk to the hardware directly.

Personal interviews with automotive developers showed us that most of them are satisfied with the layering concepts regarding the interfaces, software components and services. This indicates that the idea of the specification is very good. Feedback from industry also exposed that the full hardware abstraction of AUTOSAR (Memory, I/O system services, communication) is often not used, so AUTOSAR is reduced to a pure communication middleware.

When taking the AUTOSAR specification literally, complex drivers are an absolute exception for software which is so far not integrated in AUTOSAR. Specific requirements in the standard (RS_BRF_02280, RS_BRF_02288) underline this understanding of an exceptional situation[AUf]. Our industry feedback revealed that the opposite takes place: Complex drivers are a standard solution for local hardware access.

One reason could be that developers need to solve problems quickly and simply do not find

the *right* function for their problem in the high-level API. Their problem granularity level is so different, that the AUTOSAR basic services do not pay off for them. This problem is not specific to AUTOSAR and can be denoted as the *API Dilemma*.

In the specific case of AUTOSAR, this may be simply related to age: API definitions such as POSIX needed decades to mature, while the AUTOSAR standardization body is constantly evolving, expanding and changing different parts of the middleware. The RTE change history [AUi] shows that very clearly.

A contemporary separation of the standard into long term support and innovative versions would show clearly for which parts backward compatibility is essential at any time, like in POSIX. This would help the developer to replace ad-hoc manual solutions with long term code that relies on stable interfaces. The profiling ideas would further reduce the learning curve for developers.

5 Interfaces and Components

An *interface* or *application programming interface (API)* is a description of a set of possible operations that a caller may request [TB01]. It acts as written contract that is fulfilled by the component implementing the interface. Interfaces allow interaction of independently developed software and encourage separation of concerns. They can be syntactically described in terms of a programming language (e.g. Java, C#) or in terms of a dedicated language (e.g. IDL, WSDL, protocol automata).

When working with interfaces there may be vertical as well as horizontal interface dependencies, since components either rely on remote components (horizontal) or on interfaces from layers below (vertical). In an ideal world of AUTOSAR applications, an OEM (= car manufacturer) would define the interfaces for which the supplier (such as Bosch) delivers the implementation.

AUTOSAR defines specialized entities such as AUTOSAR Application Interface, AUTOSAR interface, standardized AUTOSAR interface, software component interface, interface, and private interface [AUb]. The difference lies in the use case and the developer of the interface: There are interfaces standardized by AUTOSAR and interfaces used for horizontal or vertical decoupling.

One specific example is the software component interface, which is defined as an XML document [AUh]. It relies on classical port semantics and therefore supports different communication paradigms. As already discussed before, the standard leaves open which of these paradigms are mandatory to implement [MB11]. Furthermore, the mapping to C language is not obvious. The XML type system was never invented for function signatures, which is a fact that was already known in the SOAP/WSDL world when modern AUTOSAR versions were developed [Vi08]. The XML definition also makes it hard to formulate interface descriptions as human developer, which is one of the reasons why classical CORBA IDL or

its modern variations [Go17] look more like a programming than a description language. Some papers, as a solution, discuss how to generate software component descriptions out of annotated C header files [Mu09] but we think it would be easier to adopt modern interface description strategies in the automotive world.

Interfaces are implemented by components, which are binary units of independent production, acquisition, and deployment that interact to form a functioning system [Sz02]. They are intended for multiple use, can be independently developed and deployed and are composable with other components. A component using another components functionality is commonly described as *client*.

AUTOSAR has the central concept of a *software component*. They interact with other components in the vehicle over communication buses and through well-specified interfaces. The glossary for AUTOSAR [AUb] again defines several variations of the general component concept: application, software component (SW-C), application software component, atomic software component, basic software module and functional unit. The difference lies in the intended use case. An *atomic software component*, for example, is *a non-composed software component that may be not relocatable* [AUb], which makes the whole application immediately non-portable.

A component model normally describes the mapping between interface description, code, configuration and rules for packaging. This leads to standardized units of deployment, which can be installed and instantiated independent from each other on nodes in the system. The most prominent examples from the non-automotive world are the *CORBA Component Model (CCM)* [Gr12b] and the various JavaEE packaging formats such as EJB [EJ09] and WAR [CY03].

Although AUTOSAR has a concept of abstract interface and configuration descriptions, it does not define a binary packaging format. This is not a surprise, since run-time deployment never was a target in the standardization. This makes AUTOSAR components look more like modules or packages in classical programming, which is fine for a static setup. They fit to the intended use case, but are again not ready for future applications with dynamic loading and unloading scenarios. In such cases, the resource and life-cycle management must be covered by lower layers in the stack, which typically leads to a container run-time environment. These containers need a true package format that enables binary portability of the software component, which is currently not given in AUTOSAR.

6 Containers

A *container* is an execution environment that manages connectivity, life-cycle events and resources for instantiated components. Examples from standard middleware products are the Java EE application server, the CORBA ORB, the Ruby interpreter or the Windows Service Host for DCOM/COM+ components.

Components need to follow specific rules in their packaging, interfaces and behavior for being maintainable by a container. They describe their resource demands in a *deployment descriptor*, instead of allocating and releasing resources by themselves. At run-time, a separation of responsibilities now becomes possible: The container is managing execution resources by the help of the operating system, and the component focuses on the core functionality of the software feature. This makes components truly portable across different hardware environments and use cases.

AUTOSAR has no visible container concept. Although basic software layer and RTE layer offer a standardized API for components, they do not take over the complete resource management or the life-cycle of AUTOSAR components. Software defects in AUTOSAR components, such as exhaustive resource consumption or non-cooperative communication behavior, can easily impact the whole ECU without any reasonable way to mitigate such problems. This makes testing and validation extremely critical in classical automotive software development and skips runtime protection completely.

A container concept would serve as logical continuation of the interface, software component and RTE ideas in AUTOSAR. It would offer a layer of defense that utilizes operating system and hardware capabilities for managing components during their lifetime. If the binary format of the component implementation is interpreted code, such as with Java or script languages, then the components easily become movable and deployable on heterogeneous hardware. Portability mechanisms as OSGi [OS14] show that this even works under tight resource constraints.

We can only speculate why the AUTOSAR inventors omitted such a concept so far. One reason might be the demand for best-possible timeliness by removing layers at run-time. This is understandable when the strict cost constraints for vehicle ECU hardware are the main decision factor. On the other hand, it is undeniable that such a container concept would open new possibilities for dynamic code loading, isolation and component migration, without breaking the existing programming model. It would allow to make dynamically use of underutilized resources at run-time, instead of having completely unused computation and memory resources that only exist for safety margin purposes.

Some research projects already added a reconfiguration layer or software component activation/deactivation concepts on top of classical AUTOSAR [Ze13]. This is a good starting point, but we see more potential in giving AUTOSAR a real container concept, similar to middleware stacks such as JavaEE and CCM. This would allow to separate resource and life-cycle management strategies from the component binary, and eases the realization of backward compatibility by designing dedicated container implementations for legacy APIs.

The combination of a clear container concept on top of a real-time operating system like QNX would support the needed dynamics for future applications. Legacy support as well

as a gradual transition from classic AUTOSAR ECUs to the new architecture could be supported by operating system drivers which do support the RTE wire protocol translation.

It remains the task of the automotive industry to decide if a minimal increase in resource consumption and round-trip times, as being produced by a container indirection, can be tolerated for getting advanced isolation and flexibility for code at run-time.

7 Conclusion

The AUTOSAR specification is a success story. One example is Volvo, with approximately 20 Million ECU's based on AUTOSAR in seven car models. By 2020, this is expected to have tripled [Ul16]. However, when comparing AUTOSAR with the state of the art in research and other industries, it seems to be time for the standard to catch up again for remaining future proof. We identified the following major points of potential improvement for future AUTOSAR versions:

- Classic AUTOSAR aims for component source code that is portable from one hardware ECU platform to another. This demand seems to be the underlying origin for most design decisions that are given today in the standard. Binary portability, which needs a packaging format and robust life-cycle support, is not covered so far, but will become a relevant aspect for future applications.
- The current specifications lack a strong notion of compliance. This makes AUTOSAR tool chains (code generators, validators, configuration validators) not comparable in their quality or completeness with respect to the standard. The resulting effect is, that tools only become compatible to themselves, which ultimately pushes OEMs into a tool vendor lock-in situation they wanted to avoid in the first place.
- The AUTOSAR standard itself is a very complex document and seems to make gradual improvement extremely difficult. One part of the problem seems to be the wild mixture of implementation strategies, architectural principles and data formats in the documents. A closer look on the structured meta model approach of the OMG may help to find better ways for describing the core ideas of the AUTOSAR specification.

One interesting trend with respect to these suggestions is *Adaptive AUTOSAR*, which aims for a service-oriented communication stack that is still interoperable with classic AUTOSAR components. Although *Adaptive AUTOSAR*[AU17] is officially released, information sources about this attempt remain scarce. It seems like the enforcement of isolation, e.g. by disallowing a complex driver pass-through, is getting more attention now. We are curious to see if in the end, the problems raised in this article are solved by *Adaptive AUTOSAR*.

References

[AUa] AUTOSAR: , Application Interfaces User Guide. Document ID 442 (AUTOSAR_EXP_AIUserGuide), AUTOSAR Release 4.2.2.

[AUb] AUTOSAR: , Glossary. Document ID 055 (AUTOSAR_TR_Glossary), AUTOSAR Release 4.2.2.

[AUc] AUTOSAR: , Layered Software Architecture, Document ID 053 (AUTOSAR_EXP_LayeredSoftwareArchitecture), AUTOSAR Release 4.2.2.

[AUd] AUTOSAR: , Main Requirements, Document ID 054 (AUTOSAR_RS_Main), AUTOSAR Release 4.2.1.

[AUe] AUTOSAR: , Project Objectives, Document ID 599 (AUTOSAR_RS_ProjectObjectives), AUTOSAR Release 4.2.1.

[AUf] AUTOSAR: , Requirements on AUTOSAR Features, Document ID 294, AUTOSAR Release 4.2.2.

[AUg] AUTOSAR: , Requirements on Methodology. Document ID 362 (AUTOSAR_RS_Methodology), AUTOSAR Release 4.2.2.

[AUh] AUTOSAR: , Software Component Template. Document ID 062 (AUTOSAR_TPS_SoftwareComponentTemplate), AUTOSAR Release 4.2.2.

[AUi] AUTOSAR: , Specification of RTE, Document ID 084 (AUTOSAR_SWS_RTE), AUTOSAR Release 4.2.2.

[AUj] AUTOSAR: , Specification of Service Discovery. Document ID 616 (AUTOSAR_SWS_ServiceDiscovery), AUTOSAR Release 4.2.2.

[AU14] AUTOSAR: Basic Information - Short Version. 2014.

[AU17] AUTOSAR: Adaptive Platform Release Overview, Document ID 782 (AUTOSAR_TR_AdaptivePlatformReleaseOverview), AUTOSAR AP Release 17-03. 2017.

[Br97] Bradner, Scott: Key words for use in RFCs to Indicate Requirement Levels (BCP-14). RFC 2119, Network Working Group, March 1997.

[CY03] Coward, Danny; Yoshida, Yutaka: , Java Servlet Specification, Version 2.4, 11 2003.

[EJ09] EJB 3.1 Expert Group: , JSR 318: Enterprise JavaBeans,Version 3.1, 11 2009.

[Go17] Google Developers: , Protocol Buffers Version 3 Language Specification, 05 2017.

[Gr12a] Group, Object Management: , Common Object Request Broker Architecture (CORBA), Version 4.0, Part 1: CORBA Interfaces, 2012.

[Gr12b] Group, Object Management: , Common Object Request Broker Architecture (CORBA), Version 4.0, Part 3: CORBA Component Model, 2012.

[KB03] Kopetz, Hermann; Bauer, Günther: The time-triggered architecture. Proceedings of the IEEE, 91(1):112–126, 2003.

[MB11] Melin, Jesper; Boström, Daniel: , Applying AUTOSAR in Practice: Available Development Tools and Migration Paths, 2011.

[Mc16] McKinsey & Company: Automotive revolution - perspective towards 2030. Advanced Industries, 2016.

[Mu09] Mutschler, Christopher: Automatic Generation of AUTOSAR Software Component Descriptions. 2009.

[Ob05] Object Management Group, Inc.: , Real-time CORBA Specification, Version 1.2, 01 2005.

[OS14] OSGi Alliance: , The OSGi Alliance OSGi Core, Release 6, 06 2014.

[Sc15] Schneider, David: Jeep Hacking 101. IEEE Spectrum, Aug 2015.

[Sm] Smets, Philippe: Imperfect Information: Imprecision and Uncertainty. In (Motro, Amihai; Smets, Philippe, eds): Uncertainty Management in Information Systems, pp. 225–254. Springer US.

[Sz02] Szyperski, Clemens: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[Ta16] Taub, Eric A.: Your Car's New Software Is Ready. Update Now? The New York Times, Sep 2016.

[TB01] Tari, Zahir; Bukhres, Omran: Fundamentals of Distributed Object Systems: The CORBA Perspective. Wiley-Interscience, 1st edition, 2001.

[Ul16] Ulf Alvebratt: , AUTOSAR 4 Clean-Cut Implementation. 9th AUTOSAR Open Conference, Gothenburg, Sweden, 09 2016.

[Vi08] Vinoski, Steve: Convenience Over Correctness. IEEE Internet Computing, 12(4):89 – 92, 2008.

[Ze13] Zeller, Marc; Prehofer, Christian; Krefft, Daniel; Weiss, Gereon: Towards Runtime Adaptation in AUTOSAR. SIGBED Rev., 10(4):17–20, December 2013.