Component-Based Modeling and Integration of Automotive Application Architectures

Konstantin Schorp and Stephan Sommer
fortiss GmbH, Guerickestraße 25, D-80805 München, Germany, {schorp, sommer}@fortiss.org

***Abstract*—The introduction of new propulsion technologies such as electric or hybrid drives imposes fundamental changes to the overall structure of the vehicle's electric and electronic system architecture. It also increases the need for cross-domain functionality, such as centralized energy management or the orchestration of mechanical braking and electric energy recuperation during deceleration. This leads to new challenges with respect to architecture development as interconnections between features are introduced that are not yet fully understood. The vehicle's system architecture evolves from towards a distributed multi-functional control system. Component oriented, model based approaches with multiple viewpoints have already proven being suitable in other domains to manage the dependencies between functionality by decomposing a system into a network of functional entities encapsulated in components.**

In this paper, we present a domain-specific component model to describe functional interdependencies as well as non-functional requirements needed to enable safe integration of software components in a centralized automotive ICT architecture. The model enables the composition of high-level functions and the definition of compatibility constraints. The approach is then applied to unveil feature interaction in a component architecture. This forms the foundation of a sound development and integration process for heavily interconnected functions. It also enables online product validation mechanisms to ensure functional integrity and safety as well as meeting of deployment constraints and timing requirements.

I. Introduction

Over the past 30 years, information and communication technology (ICT) has made possible significant innovations in automotive construction from the anti-lock braking system in 1978 to electronic stability control in 1995 and emergency brake assist in 2010. Accordingly, ICT, and especially its software, has expanded significantly from about 100 lines of code (LOC) in the 1970s to more than 100 million LOC [1]. The introduction of new propulsion technologies such as electric or hybrid drives imposes fundamental changes, not only to the architecture of the powertrain, but also to the overall structure of the vehicle's electric and electronic system architecture. This also changes the functional characteristics of the vehicle. Limitations in battery capacity lead to the introduction of cross-domain functionality such as centralized energy management, or brake-blending which coordinates energy recuperation and friction braking.

Additionally, the mechanical complexity of the powertrain is massively reduced. The substitution of combustion engines and their related aggregates with new electric engine concepts is expected to have a positive impact on vehicle longevity. The number of components prone to mechanical and thermal wear is reduced and therefore, one of the main reasons for terminal vehicle failure is minimized. This introduces the need to apply upgrades to the vehicle software or even install new functionality while the vehicle is already in the field. To maintain an up to date driving experience and to adapt the vehicle to the driver's changing requirements, the manufacturer might want to provide software updates or feature upgrades as additional services.

The increase in size and complexity is also driven by the increasing number of software-based features which emerge from single stand-alone applications to a tightly interacting distributed system. The high complexity and the distribution of features over many different electronic control units (ECUs) and bus systems leads to high cost for developing and testing functionality. This complexity is additionally increased by the introduction of technology like X-by-wire driven by the demands to realize reliable highly and fully automated driving functionality.

One approach to tackle the complexity of the resulting system architecture is to partition applications and their mutual dependencies into different functional domains as implemented by AUTOSAR [2]. However, this partitioning is only sufficient as long as these domains only rarely interact and as long as the number of functions inside a domain stays manageable. Having in mind applications like driver assistance systems up to autonomous driving or generic brake blending for all engine/brake combinations in the product line, access to many functions located in different domains is necessary and so the approach to partition these functions into domains is not sufficient anymore [3]. In contrast, by partitioning the system into different (network) segments, additional cross domain gateways are required which again leads to an increase in complexity by the introduction of obscured dependencies and additional devices. This would also add new single points of failure.

A second approach to reduce the complexity of systems development is to introduce a suitable model-based development methodology as already done in many domains. Established modeling and development paradigms, e.g. SPES XT [4] are based on the concept of multiple viewpoints on a system. The most prominent viewpoints are the logical and the technical architecture viewpoint.

A logical architecture describes the behavioral aspects of a system, its structure and the communication channels between logical components that encapsulate behavior. A technical architecture focuses on the technical entities, such as computing

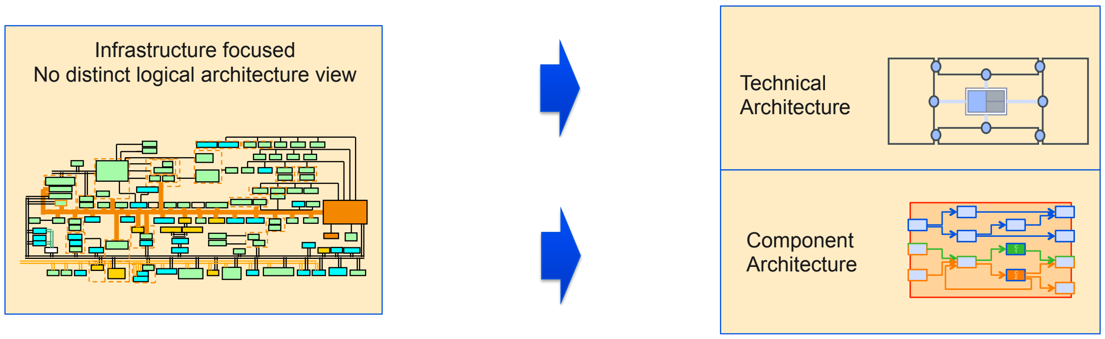

Fig. 1. Distinction between technical and logical view on the architecture

nodes or communication interconnects over a network. This distinction is made because the decomposition of the system into parts is driven by different objectives depending on the viewpoint. A decomposition based on the logical view might not be suitable from a technical perspective and vice versa.

A. *State of the Art*

While there is already a distinction of logical and technical aspects established in current automotive development methodologies, they usually only cover the properties of an isolated component, not taking into account the global structure of the architecture.

Across a vehicle's overall E/E-architecture, the logical architecture representing the partitioning of functionality into software modules as well as the interconnection of these software modules is highly influenced by the technical architecture (physical network layout and available ECUs). This is driven by the well established binding of software, hardware/ECUs and the relationship of vehicle manufacturer and the variety of different suppliers. Based on the tight coupling between ECUs and functionality, the well established separation of functionality by the AUTOSAR domains is perfectly suitable for traditional systems as long as there are only few exceptions to that paradigm. Due to the addition of more advanced software functions, the cross domain communication evolves from an exception to becoming the rule. This evolution successively neutralizes the advantages of domain separation and turns them into a major drawback for new developments in the sense of complexity and maintainability.

B. *Component and Feature driven Development*

Today, developers focus on the technical infrastructure and information flow between aggregates while developing and integrating the vehicle system. This inhibits the application of state of the art methods to describe and engineer a proper logical architecture. Such methods are crucial to efficiently develop future vehicles with the described characteristics: Distributed multi-functional modular control systems.

In order to allow efficient development and a short time to market, a transition from the infrastructure driven development paradigm towards a feature and component driven development methodology needs to be performed. This includes a transition from a federated system architecture towards an integrated approach. We leverage the advantages of component based engineering according to [5] to develop a logical component architecture view. It is extended by a functional view carrying acceptance criteria for quality attributes of the applications realized by the components and showing dependencies between features.

C. *Contributions*

The goal of our activities is to decouple the technical and logical architecture of the system as depicted in Figure 1. They are currently aligned and limit modularity as well as flexibility. At the platform level, a first step is to provide a generic runtime environment like the one proposed in the RACE-Project [6] as a foundation for an integrated system architecture. It also includes an ARINC 653 [7] inspired automotive execution environment. This software stack provides a safety and security aware, homogeneous execution environment that can safely and securely be extended after an initial deployment. The fundamental structural differences in this newly developed technical platform enable the introduction of a distinct logical software architecture paradigm and a suitable development methodology.

Hence, the second contribution, which is the main contribution of this paper, is a component oriented model-based development methodology and process that supports a descriptive approach based on modeling functional entities (components), their dependencies as well as their functional and extra-functional properties. Integration is performed by describing the required properties of vehicle features and mapping those requirements onto the properties of the component network. This enables continuous evaluation in early design stages which avoids problems like unknown feature interaction [8]. Based on that, we can perform functional integrity verification as well as the validation of required properties like Safety Integrity Levels (SIL) [9] and timing constraints.

Having both of these contributions as a foundation, the switch from a technically dominated system view towards an engineering approach based on multiple views similar to [4] can be performed and the advantages of component oriented architecture engineering can be leveraged.

The remainder of this paper is structured as follows: In Section II, the engineering methodology and its building blocks are elaborated. In Section III, we apply the methodology to discover feature interactions in the system under analysis. Related work is discussed in Section IV and the paper is concluded in Section V with a summary of the contributions and an outlook for future work.

II. Architecture Engineering

The transition from a federated static architecture towards a centralized dynamically extensible architecture implies a paradigm shift in the automotive ICT architecture development process. The first step towards an adapted process is a suitable modeling and integration methodology for vehicle components and features. The main architectural features of such ICT architecture as proposed in the introduction are:

Virtualization and segregation of resources: allowing the deployment of multiple software components of mixed criticality on a single computing unit. It is enabled by the generic RTE that provides time and space partitioning with guaranteed computing resource and time budget.

Deterministic execution: based on the virtualization features. This is accomplished by time-triggered execution using a globally synchronous clock. Unlike traditional asynchronous event-triggered systems, causality and deterministic behavior are independent from the current deployment and schedule.

Transparent communication: decouples information flow from the actual deployment of a software component. Regardless whether two components are deployed on the same or on different computing units, software interfaces and communication behavior are identical.

Distinct technical architecture: describes solely technical and implementation related properties of sensors, actuators and generic computing units and how these resources are physically interconnected.

These features are provided by the platform developed in the RACE-Project [6]. To leverage the features of the new platform approach, we need to introduce fundamental changes to the development process, stepping away from its focus on infrastructure and the isolated development of functions. It has to take into account the abstraction of software components from their former dedicated ECUs and has to ensure that there is no implicit hidden data flow between those components that might be dependent on a specific deployment. In upcoming section II-A, the developed meta models for features and components are introduced. In addition, we explain how a connection between the two different views on a system can be established and how the development process can benefit from that.

A. Models for Features and Components

Initially, a method to describe the objectives of a vehicle system is needed, modeling its (customer visible) functional features, which come with certain quality attributes such as SIL requirements or reaction times. In addition, the logical software components have to be modeled. This is necessary as the components contain the behavior and interfaces that realize the described features. The component model includes a description of component interdependencies modeled as data-centric publish subscribe (DCPS) ports, adapted from DDS (Data Distribution Service) [10]. Feature as well as component models are parts of the overall system model, the system's *manifest*. This manifest provides information about the components, the communication requirements and ports as well as details about the components themselves by providing, e.g. worst-case execution times. This description forms the foundation for a consistent, system-wide representation of all models and enables the discovery of connections between components and features. While the feature model holds requirements and constraints, the component model holds characteristics of actual architecture elements.

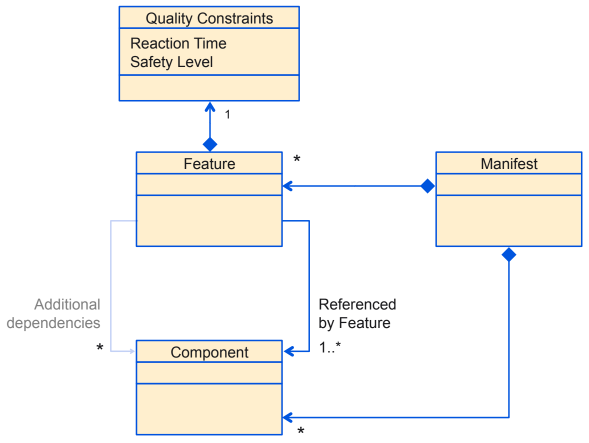

Fig. 2. Feature Meta Model

1) Feature Meta Model: System features contain criteria the component architecture has to comply with in order to fully realize a feature. A feature in our sense is not an actual implementation, but a container for quality requirements that the network of logical components that realize the feature have to fulfill. As described in Figure 2, generic quality constraints are defined regarding for instance overall reaction time or SIL. A reaction time constraint denotes the allowed time delay for the system to react on a defined output port upon the reception of an excitation of a specified input port. With respect to SIL, the ISO 26262 [9] standard defines a set of safety integrity levels for vehicle features. As certification is performed at the feature level, a minimum SIL to be realized by the component architecture is defined as an attribute of the feature. The standard also includes a set of rules about how safety integrity levels of a single component in combination with its adjacent components affect the overall feature SIL.

Features are realized by components, but components are not exclusively dedicated to a specific feature. Therefore, the association between feature and component in the model is non-exclusive. This enables modular reuse of components and the composition of features from components of different sources. In order to be successfully mapped onto the component architecture, a feature has to explicitly reference at least one component. The model can be enriched by adding additional, implicit dependencies that emerge through data flow dependencies in the component architecture. Those additional dependencies are not static, but depend on the set of features to be included in the vehicle product. Because of that, the dependency discovery has to be performed for every feature configuration.

2) Component Meta Model: The main purpose of the component model is to describe dependencies between components. Those dependencies need to be defined explicitly and in an unambiguous manner. The modeled data dependencies are not referring to a specific instantiation of data source or sink. Instead, a rich type-system is defined in a domain-specific data dictionary to allow for data-centric definition of data produced (published) or consumed (subscribed) by a component. The data dictionary contains definitions of semantic data types. A semantic data type, called *topic*, contains a technical data type, e.g. in form of a data structure definition, and a set of attributes that add a specific meaning to the type. Attributes can for instance define a physical unit, the position of the sensor that uses the data type or the measurement accuracy. To support a distributed yet centrally managed development process, this data dictionary is managed by a central administrator. This ensures compatibility at the technical and semantical level.

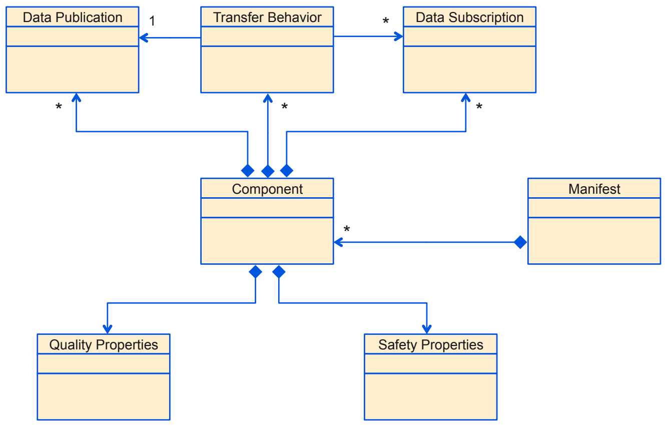

Fig. 3. Component Meta Model

As Figure 3 shows, the model covers not only external dependencies described using data publications and subscriptions, but also internal transfer behavior between component inputs and outputs. For each outgoing port, there exists a causality relation that defines which input ports have an effect on this specific output. This was introduced for two reasons: First, internal processing delay is different for each individual relation between input and output. Second, without this information we have to assume full internal interconnection leading to a number of $n \times m$ for n inputs and m outputs. Causality relations can significantly reduce the complexity of the subsequent discovery and analysis. We deliberately designed the model without support for further decomposition into logical subcomponents as we focus on deployable software units and integration-related properties in this view.

Apart from data flow related information, properties regarding the implementation of the component include local timing behavior such as execution frequency and safety related properties such as the component-specific SIL.

Together with the network of components that is generated by "wiring" compatible input and output ports, the properties of the resulting logical component architecture can be mapped onto the modeled features. This step is explained in the upcoming section.

B. Feature Dependency Discovery

Assuming that all data dependencies can be non-ambiguously satisfied, the result is a network of connected components or a component architecture. At this point, solely relying on that architecture, we cannot determine if the system will act as desired if deployed onto the technical architecture. The information in the feature models define quality and safety requirements the architecture has to fulfill, so if these features can be mapped onto the component architecture, the conformance to these criteria can be verified.

The first step is to evaluate the relationship between the component architecture elements and the features. Apart from the components that are explicitly required by the feature, there are usually several components in the architecture the feature also (implicitly) relies on. Those components act as direct or indirect data sources or sinks for explicitly required components. As the component architecture is a graph network, we can employ algorithms such as depth first search (DFS) [11]:

Vertices: do not equal components, but rather their ports. This enables us to also take relations within the component into account.

Edges: are data flow connections between ports. These can either be connections between components based on publish/subscribe or connections within components in the form of causality relations modeled as transfer behavior.

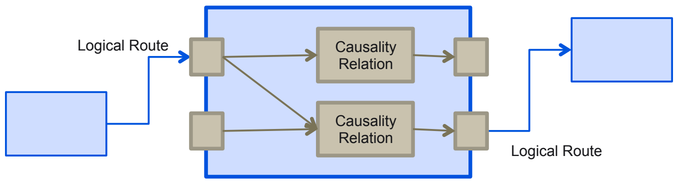

Fig. 4. Logical routes between and causality relations within components

The combination of logical routes between components and causality relations within a component, as depicted in Figure 4, enables the discovery of only the relevant components that have actual data dependencies. Simply relying on channels between components would lead to many false positives as not all outputs usually rely on all input ports.

Figure 5 shows an arbitrary component architecture where a feature explicitly references component `C1`. In order to discover all components that might be relevant for the feature, we have to traverse the connected components and follow their inner causality relations. This has to be performed in both directions: Backwards from the input ports of `C1` toward the system boundary and forwards toward the systems outputs. As displayed in the figure, discovered data dependencies are shown by highlighted arrows, the corresponding components that are now known to be relevant for the realization of the feature are highlighted with thick borders. The algorithm is also capable of discovering cycles, primary sinks and sources. A primary sink is defined as a system output, modeled by a specific transfer behavior containing no component output port. Similarly, a primary source originates from a component output port that has a corresponding transfer behavior modeled. Deployment related issues such as mapping of logical primary sources/sinks and their corresponding technical entities in the form of sensors and actuators are part of the engineering methodology, but are beyond the scope of this paper.

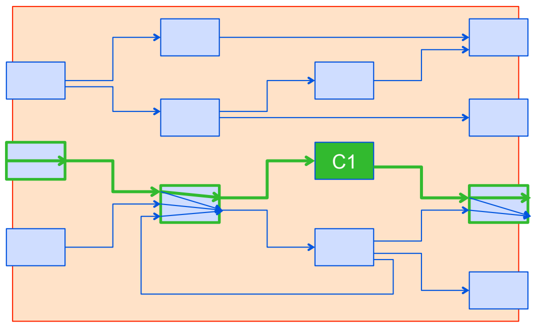

Fig. 5. Feature Mapping

C. Architecture Validation

Having discovered all implicit component dependencies of the feature, we are able to perform a variety of acceptance tests relying on the requirements modeled at the feature level and the properties modeled on the component level. Basic SIL validation forces all involved components to reach at least the minimum SIL of the feature. However, extended SIL analysis can take into account additional properties such as component redundancy or voters to allow the integration of components with lower SIL.

Timing constraints are crucial for the reactive behavior of the system, especially for control applications. This constraint is specifically defined for data flows of the feature that control the reactive process, usually between sensors and actuators. Set values and parameters do not have to meet the same strict constraints as the control loop. Timing constraints and phase synchrony are analyzed based on individual data flows, taking into account the transfer delays between components and the processing delay within a component. The deterministic time-triggered model of execution allows timing calculations based on the global clock without taking into account a specific schedule or deployment. The transparent communication feature of the platform assures deployment independent communication behavior, therefore we can compute the overall delay independent from a concrete technical architecture deployment.

Another application of this approach as a method for architecture validation is presented in section III. Here, the approach is used to discover feature interaction from components shared between features.

III. Discovery of Feature Interaction

The discovery of implicit component dependencies is performed for each feature that is modeled. How this can be used to validate the fulfillment of safety or timing requirements was introduced in the preceding section. This validation is performed by evaluating each feature isolated from other features. A huge issue in automotive architectures though, as stated in [8] is the discovery of unintentional interactions between features.

Not every interference is problematic. Two features sharing a primary source, usually a sensor, can be a good example for successful reuse of components that reduces cost and development time allowing a more efficient architecture and deployment. Problematic interferences arise if a feature is behaving differently in presence of another feature [12]. To ensure the safety and integrity of the vehicle, each revealed feature interaction has to be examined manually, especially discovered interferences such as in the example.

Figure 6 shows components `C1` and `C2` which are explicitly referenced by two different features. The remaining components belong to other features that might be developed by a different supplier, so elaborate specifications beyond the component models might not be available. Dependency discovery is performed for both features. Apart from the additional components that are now implicit parts of the respective feature, some components are used by both features. From the analysis of the transfer behavior within the components, we can also derive that these components are not only relevant for the integration of both features, but that there is actual interference between the features. A system sink, highlighted in the rightmost circle, is a termination point of the data flow of both features.

Mapping and assessing the revealed feature interactions on the component level allows the developer also to gain an overall view on the quality of the component architecture with respect to modularity and the impact of changes, e.g. the effect of adding or removing a feature from the vehicle product. For future Plug & Play capabilities, it is crucial to integrate feature interaction discovery and mitigation into the automatic configuration process.

IV. Related Work

Component models are widely used in many fields to describe dependencies between architecture elements and to allow for flexible adding and installing new components into

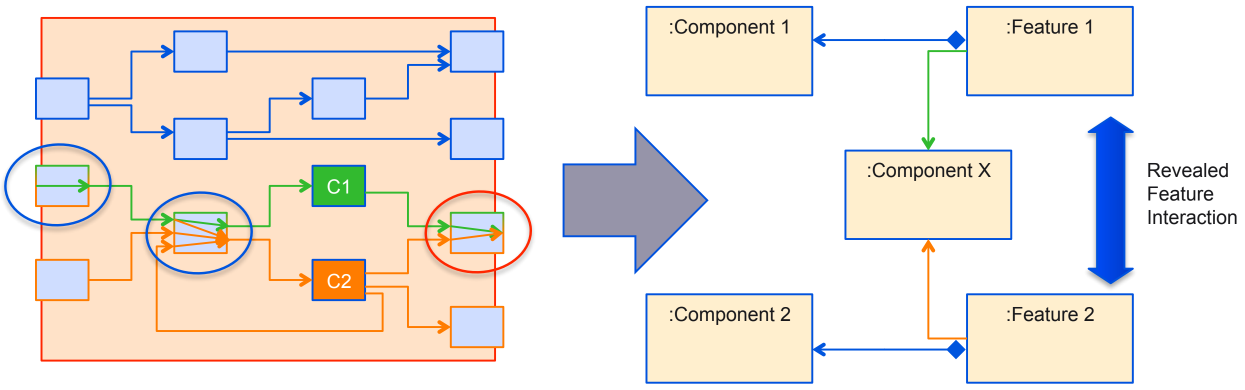

Fig. 6. Discovery of Feature Interaction

an existing architecture. In many common environments, such as a Linux-System or an IDE, there are no (safety-)critical non-functional requirements to be considered. As soon as, e.g. control applications are described, there are far more aspects to consider such as timing, safety and the complex deployment constraints introduced by a distributed system [5].

Bures [13] proposes a first approach for the automotive domain. UML-MARTE [14] provides an extensive description language for embedded systems modeling. It also provides a generic component model and methods to describe high-level functions (high-level application modeling). With respect to MARTE as a generic method, our approach is a domain-specific solution tailored to the automotive domain.

RUNES is a component based middleware ranging from small resource constraint sensor nodes up to high performance desktop PCs. It provides a run-time reconfigurable modularized system consisting of a middleware kernel and services. The middleware consists of two major parts. The foundation is a language-independent, component-based programming model that is sufficiently minimal to run on any of the devices typically found in networked embedded environments. On top of this foundation layer, the middleware functionality is implemented by different, self-contained modules providing the functionality. By composing these modules, the middleware can be individually assembled for each deployment [15], [16]. In comparison to our approach, RUNES is missing the notion of features implemented by an assembly of modules containing the application logic. As a consequence, feature interaction and feature driven system analysis is not possible.

The well established AUTOSAR standard [2] describes a platform which allows implementing future vehicle applications and minimizes the current barriers between functional domains. It will be possible to map functions and functional networks to different control nodes in the system, almost independently from the associated hardware. In contrast to AUTOSAR where no tool support for component / feature mapping and component architecture analysis is provided, our approach supports the user during development, validation and system assembly.

V. Conclusion and Outlook

Motivated by new challenges in automotive ICT architecture development, we introduced a distinction between technical and logical architecture in the automotive domain and presented a domain-specific component model and defined compatibility constraints at the feature level as a foundation of a sound development and integration process for heavily interconnected functions. By mapping logical software components and vehicle features, we enable the verification of feature requirements against the properties of their realizing components. This enables continuous evaluation in early design stages which avoids problems like unknown feature interaction and allows for functional integrity verification as well as the validation of required properties such as Safety Integrity Levels.

The development of well designed logical architectures using discovery, evaluation and mitigation of feature interactions is a main subject of our future work. An important topic in this scope is also the evaluation and classification of interactions. Similarly to the logical component architecture, a feature dependency graph can be a useful tool to cover feature dependencies and feature interactions. A long term goal is the formalization of a functional architecture and the integration of logical, functional and technical view into a consistent methodology. Implementing these concepts in a tooling environment can help system architects as well as engineers to cope with the newly introduced architecture complexity. Apart from this analytical approach, we are investigating a constructive approach to optimize architecture engineering with respect to modularity and verifiability of feature requirements by introducing a reference software architecture for centralized automotive ICT systems.

Acknowledgments

This work is partially funded by the German Federal Ministry of Economics and Technology under grant no. 01ME12009 through the project RACE (http://www.projekt-race.de/).

References

[1] M. Broy, “Challenges in automotive software engineering,” *Proceeding of the 28th international conference on Software engineering - ICSE ’06*, p. 33, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1134285.1134292

[2] AUTOSAR Group, “AUTomotive Open System ARchitecture (AUTOSAR) Release 4.1,” AUTOSAR, 2013.

[3] Bernhard, M., Buckl, C., DO Richt, V., Fehling, M., Fiege, L., von Grolman, H., Ivandic, N., Janelle, C., Klein, C., Kuhn, K.-J., Patzlaff, C., Riedl, B., Schätz, B., Stanke, C, *The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future*. fortiss GmbH, March 2011.

[4] J. Thyssen, D. Ratiu, W. Schwitzer, A. Harhurin, and M. Feilkas, “A System for Seamless Abstraction Layers for Model-based Development of Embedded Software.”

[5] I. Crnkovic, “Component-based software engineering for embedded systems,” *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 712–713, 2005.

[6] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll, “Race: A centralized platform computer based architecture for automotive applications,” in *VEC/IEVC 2013*. IEEE, October 2013.

[7] Airlines Electronic Engineering Committee, *Avionics Application Software Standard Interface: ARINC Specification 653P1-2*. Aeronautical Radio, 2007.

[8] A. Vogelsang and S. Fuhrmann, “Why Feature Dependencies Challenge the Requirements Engineering of Automotive Systems: An Empirical Study,” *RE13*, pp. 267–272, 2013. [Online]. Available: http://www4.in.tum.de/ vogelsan/publications/RE13.pdf

[9] International Organization for Standardization, “ISO/DIS 26262 - Road Vehicles - Functional Safety,” November 2011.

[10] Object Management Group, “Data distribution service for real-time systems, version 1.2,” *Available from: www.omg.org*, 2007.

[11] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0201010

[12] C. Prehofer, “Feature interactions in statechart diagrams or graphical composition of components,” in *Second International Workshop on Aspect-Oriented Modeling with UML (¡¡ UML¿¿ 2002)*, 2002.

[13] T. Bures, J. Carlson, S. Sentilles, and A. Vulgarakis, “A Component Model Family for Vehicular Embedded Systems,” *2008 The Third International Conference on Software Engineering Advances*, pp. 437–444, Oct. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4668143

[14] OMG, “Modeling and analysis of real-time embedded systems, uml profile for marte v1.1,” http://www.omg.org/spec/MARTE.

[15] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis, “The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario,” *Pervasive Computing and Communications, IEEE International Conference on*, vol. 0, pp. 69–78, 2007.

[16] P. Costa, G. Coulson, C. Mascolo, G. P. Piccoand, and S. Zachariadis, “The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems,” in *Proc. of the 16th Annual IEEE Intl. Symposium on Personal Indoor and Mobile Radio Communications (PIMRC’05)*, 2005.