Connected In-Car Multimedia: Qualities Affecting Composability of Dynamic Functionality

A.Knirsch[1,2], J.Wietzke[2], R.Moore[2] and P.S.Dowland[1]

[1]Centre for Security, Communications and Network Research,
Plymouth University, Plymouth, United Kingdom
[2]ICM Labs, Faculty of Computer Science,
University of Applied Sciences Darmstadt, Darmstadt, Germany
e-mail: andreas.knirsch@h-da.de

Abstract

In-Car Multimedia systems have become a fundamental part of a car's human-machine interface. Recent developments within the domain of mobile consumer electronics create demands for flexible functionality by use of after-market applications, also within the automotive sector. Further, current systems already provide connectivity to enable dynamic data using cellular access networks. Future systems will provide comparable features not only for data, but also for functionality. The capabilities for modification or enhancement of functionality using network connectivity requires a thorough consideration of certain software qualities, also with respect to operation within a safety critical environment and provisioning of an adequate user experience. This paper characterises relevant software qualities with a strong focus on composability. The objective is to provide a base for building a modular system that appears as a homogenous whole, while providing sufficient dependability. An architecture is proposed to illustrate the applicability of those qualities to a particular software system.

Keywords

Embedded systems, automotive software engineering, infotainment, composability

1. Introduction

In recent years the significance of automotive information and entertainment (aka. infotainment) systems has grown rapidly. They represent the central information interface between the car and its occupants and already affect a prospective customer's purchase decision. They combine an increasing number of software-based functionalities of different importance and purpose, developed independently by multiple suppliers and integrated onto a shared hardware (HW) platform (aka. the 'head-unit'). The aim is to provide guidance and assistance, while enabling the driver to configure and control automotive functions and offer a rich variety of entertainment functionalities. This evolution has led to large-scale complex software systems of >20 million lines of code (MLOC), decomposed into software components to tackle the complexity and integrated into the head-unit to achieve cost efficiency in production. The head-unit features various interfaces to other systems and the occupants of the car, and their mobile or storage devices. Figure 1 depicts some common components and interfaces of next generation systems.

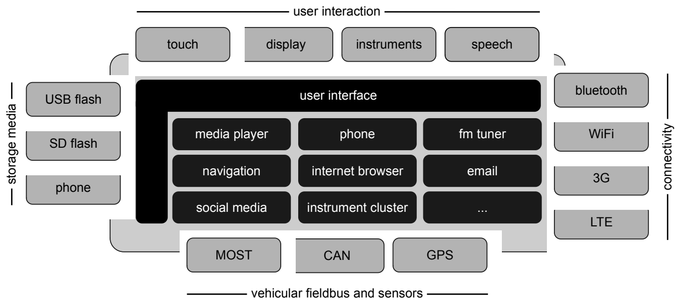

Figure 1: Exemplary components and interfaces.

Whereas the scope of past systems were limited to In-Vehicle Infotainment (IVI), current developments led to systems where the distinction of safety critical software functionalities, like e.g. the instrument cluster, becomes blurred due to integration. The instrument cluster provides functionality (e.g. indicators for gear, exterior light, speed control) that is classified safety critical using level ASIL B (Automotive Safety Integrity Level) following ISO 26262 “Road vehicles – Functional safety”. Currently there are already systems available that display IVI content within a fully digital instrument cluster. Those systems are realised using several distinct platforms (aka. electronic control units (ECU)) interconnected by in-vehicle bus systems (e.g. CAN, MOST) and direct links to achieve an efficient video transport (e.g. LVDS).

It is the intention of the car manufacturers to reduce the number of ECUs, while utilizing available multi-core HW architectures (Monot et al., 2012). A decreasing number of dedicated HW units increase the integration density at the software level for a single HW platform while using a multi-source code base. In result the head-unit evolves into a mixed critical system (MCS), combining distinct levels of assurance against failure (Burns and Davis, 2013). Nevertheless, safety critical and non-safety critical functions may be still separated at the software level to prevent unwanted mutual interference. The objective is to maintain the necessary reliability of vehicular software, with a failure rate of about one part per million in a year (Mössinger, 2010).

Future IVI is anticipated to go beyond just information and entertainment due to the fusion with other existing and upcoming functionality to build an integral UI for the occupants. In the following the broader term In-Car Multimedia (ICM) is used to differentiate from IVI. Despite this heterogeneity at the software (SW) level, the user interface has to provide all functionality in a comprehensive and uniform way, blended into the car-manufacturer’s usage concept. To achieve an adequate and purpose-oriented user experience (UX), both the allocation and the presentation of the content has to respect the car’s operating state, the user’s preferences, and interaction with the system, while considering a multi-display environment. Additionally, the software system has to meet specific temporal requirements while being deployed to a resource constrained embedded HW platform. Moreover, the system contains safety critical components and is operating within a safety relevant environment and therefore has to provide sufficient dependability, or more simply: it must work as intended.

The impact of these challenges is amplified by the demands and needs for dynamic content and functionality, available through wireless access networks (i.e. 3G, LTE). This enables ICM systems to dynamically update both data (e.g. geographical maps, traffic information) and functionality (e.g. ‘apps’) as provided for consumer electronics (CE) using ‘app-stores/-markets’. Mössinger (2010) formulates this as follows:

“The next software revolution in vehicles is imminent as multimedia and consumer electronics enter the automotive world. Vehicles will be connected to the Internet and to all kinds of nomadic and home-based devices as new sources for automotive software, such as open source, emerge.”

Such SW deployment provides new opportunities for after-market solutions and maintenance, to reflect the relatively long product life cycle of automotive systems in comparison to CE. This also implies a new dimension of customisation by providing the ‘user’ freedom to choose what to integrate on the system. The consequence is an increased independency from and between suppliers of SW functionality. This raises additional issues regarding composability and the integration onto a common HW platform while maintaining a deterministic predefined temporal behaviour (i.e. dynamic aspects), to reflect the different degrees of importance of the integrated SW components.

In the following, we name and analyse relevant qualities and their relationships that have impact on the composability of dependable modular systems - using ICM systems connected to infrastructure-based wireless access networks as an illustrative example. Further, we match those qualities to a SW architecture for demonstrating the applicability onto practical systems.

2. Related Work

Burns and Davis (2013) provide a comprehensive review on MCS. They classify the prevention of interference between tasks from different components as primary concern with the implementation of MCS. Further, they name AUTOSAR as software standard of the European automotive industry for addressing mixed criticality issues. As foundation to standardisation they address research questions on how to reconcile the conflicting requirements of partitioning for assurance and sharing for efficient resource usage as fundamental:

“[...] how, in a disciplined way, to reconcile the conflicting requirements of partitioning for (safety) assurance and sharing for efficient resource usage.”

AUTOSAR provides mature means for partitioning and hence prevention of interference between different components (Mössinger, 2010). Hence it supports the shift from the “one function per ECU” paradigm to more centralized architecture designs (Monot et al., 2012). But it is rather static and does not provide the necessary capabilities to integrate software components (i.e. dynamic content) originated from the CE domain. However, an AUTOSAR operating system (OS) may complement a system architecture to contain safety critical components, which is hosting multiple parallel containers with less critical software to support the demands for dynamic data and functionality.

Chung and do Prado Leite (2009) provide a comprehensive overview on the treatment of non-functional requirements (NFR). They claim the concept of quality is fundamental to software engineering. Both functional and non-functional characteristics must be taken into consideration during development, because functionality is not useful or usable without provisioning the necessary non-functional characteristics or quality attributes. They provide definitions to clarify terms related to NFR, describe differences to non-SW systems, and reason about the lop-sided emphasis in functionality:

> "*However, partly due to the short history behind software engineering, partly due to the demand on quickly having running systems fulfilling the basic necessity, and also partly due to the 'soft' nature of non-functional things, most of the attention [...] has been centred on notations and techniques for defining and providing the functions [...].*"

Attiogbé et al. (2006) propose the verification of composability using a formal model. They define such a model based on components' characteristics. In detail those are an *identifier*, a *state*, and an *interface* made of services that realise the interaction with the environment (i.e. other components). They define composability by considering the links between the components' services and their behavioural compatibility. The focus is basically limited on the system's functionality. Although the *interface* may contain details on characteristics, they omit an explicit modelling of NFRs.

Component-based software engineering (CBSE) has been a research area for many years. Hence, there are already a number component models with different aims and targeted for different domains available. A component model essentially consists of rules defining the construction of the components and their assembly. Crnković et al. (2011) provides a comprehensive overview on many of those and proposes a classification framework to support differentiation based on different dimensions to factor in the different aspects of the development process using an expressive formal model. They specify a component by a set of 'component properties', which covers both functional and non-functional properties (referred to as 'extra-functional properties' (EFP)): a component consists of a functional interface providing or using services, and a set of non-functional properties. Bindings define the connections between interfaces, whereas bindings are distinguished into connections between components and platform (i.e. those which enable component integration by use of an adequate abstraction layer) and into connections between components (i.e. those which enable component interaction by use of interoperable functional interfaces). Further, they provide information on specification, management, and composition of NFRs. Those ideas and their terminology are adopted for the concepts provided below.

3. Software Qualities related to Composable Systems

Modular systems are rendered by use of more or less distinct and heterogeneous components. To form an integrated whole those components have to be composable. This feature is expressed by the quality 'composability', which has to be reflected by the system's requirements to define targeted characteristics. The requirements also have to consider other non-functional qualities, referred to as non-functional requirements (NFR). While functional requirements reflect the purpose of the SW system (Chung and do Prado Leite, 2009), NFRs express the SW system's characteristics and attributes to make it useful and usable under stated conditions.

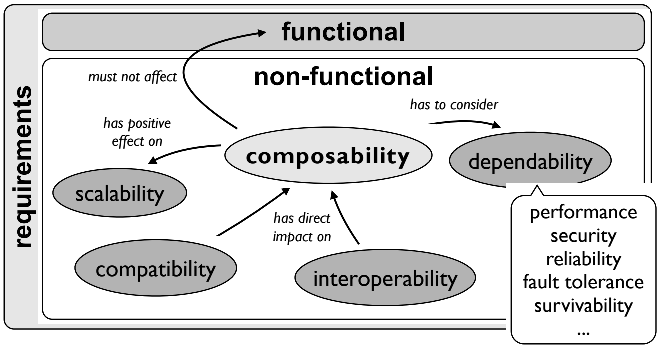

Figure 2: Qualities related to composability.

3.1. Composability

Composability is a complex quality. To make this term more tangible and also support the classification of whether a system meets a certain degree of composability, it is decomposed into the most significant and related qualities and characteristics, which are detailed in the following sections. Unfortunately, most classification schemes for NFRs are inconsistent with each other (Chung and do Prado Leite, 2009) and do not sufficiently recognize potential interactions between requirements. Hence, in the following sections the focus is kept on composability.

3.2. Compatibility and Interoperability

Composability mainly depends on compatibility and interoperability. Following the definition of Neumann (2004), compatibility implies the possible coexistence of different components (or entities) without adverse side effects. Interoperability addresses the ability of those different components to work constructively with one another. Both compatibility and interoperability are constructive aspects of SW engineering and therefore have to be considered already during the system design phase. This also implies that a system that lacks compatibility and/or interoperability might not be able to be refactored for improvement of those qualities without significant efforts. This statement is also supported by Chung and do Prado Leite (2009), who propose that NFRs play a critical role for architectural design.

3.3. Scalability

Further, composability and its derived qualities compatibility and interoperability have an effect on the scalability of a modular system. This is mainly related to the extendibility regarding the number of composed components. Also the reusability of the system's components increases with improved composability and therefore may have positive effects on the efficiency of the development process due to the possible reuse of already existing or legacy components.

3.4. Dependability

With regards to the operation within a safety relevant environment and containing safety critical functionality, the ICM system's dependability takes on a special role, which has to be considered throughout system development, maintenance and deployment of dynamic functionality. This includes, but is not limited to, the system's security, reliability, fault tolerance, survivability and performance.

4. Applied Development of ICM Systems

Based upon the information gathered through involvement in several multi-national development projects of ICM systems at different Original Equipment Manufacturers (OEMs), it can be observed that the industry does not apply a comprehensive approach to achieving composable systems. The system qualities and characteristics described above are not addressed adequately (if they are addressed at all) during the constructive design and development phase. In reality, the composing of components is mainly seen from a functional viewpoint, covering the components' interfaces with respect to the functional interdependencies. This is necessary but not sufficient to achieve composable systems. The following section describes some of the most critical issues that have been observed.

4.1. Temporal Behaviour

During the operation of the system the computational requirements and hence the computational load varies for different components, depending on the current system state, user interaction, or external events (i.e. triggered by automotive systems and sensors or through network communication). This may result in high-load (or peak-load) situations, where the system behaviour is not defined due to shared use of both computational and non-computational resources (e.g. input/output devices). Further, the system behaviour is difficult to test, due to various potential permutations of load distributions regarding the actual state of the components and depending on the actual integrated components. The latter gains significance for dynamic functionality (i.e. on user request), because neither the constellation of integrated components nor their potential mutual interferences are foreseeable. This may result in sporadic temporal interferences between components, violating the components' compatibility. Components that dynamically adjust the priorities of their executing threads and hence bias the scheduling without knowledge of other components' current state or their importance related to their semantics with a view on the overall system amplify such effects. Even more adverse is the circumstance that usually temporal behaviour isn't defined on the granularity of components. Latency requirements and performance characteristics serve as examples here. Hence, a violation of the required temporal behaviour often does not appear before integration of all components. With dynamic functionality for connected head-units this can cause unpredictable behaviour during the product-lifecycle, affecting the compatibility.

4.2. Memory Footprint

Similar to the computational resources, the platform's memory capabilities are also limited whereas the actual demands vary during system operation. While the footprint of a component can be estimated based on static analysis of the source code (if available), it is unlikely that the platform will provide as much memory as the overall system may theoretically demand. For dynamic functionality it is impossible to anticipate how much memory the platform has to provide to the components. The available memory may not be sufficient to cover all functionalities. A solution is provided by use of over-commit (aka. over-booking) (Urgaonkar et al., 2002) which also affects deterministic temporal behaviour and hence the component's compatibility.

4.3. User Interface

The automotive user interface (UI) provides multiple input and output facilities to enable a multi-modal interaction with various software components in parallel use. Many of those provide a graphical front end using the car's multi-display environment and concurrently utilise available graphical processing units (GPU) for HW acceleration. Due to a lack of the availability of multi-GPU platforms, this implies a bottleneck with potential adverse temporal effects for parallel computed components that rely on graphical output. Furthermore, as each of the components provide only a portion of the graphical user interface (GUI), the independent artefacts need to be blended to provide the car's occupants a consistent and uniform look and usage concept. The compositing of UI artefacts is related to both the components' compatibility and interoperability. A valid solution may be to implement the whole UI within a distinct component that relies on distinct "functional" components (this is basically the current approach). However, such an approach inhibits the modification and extension of those "functional" components without adaptation of the UI component. Hence this centralised UI approach is not applicable for dynamic functionality.

4.4. Tools and Techniques

Unfortunately, based on the experience gathered in industry projects, the applied business project management tools or techniques do not provide significant assistance. They may help to mitigate effects by adding transparency and traceability to the development process but also obfuscate the root cause: Insufficient addressing of composability throughout the constructive phases of the system development. This also applies to use of coding standards. Although they may help to improve maintainability and reliability by defining how the code must be structured and which language features should and should not be used, and hence represent an important building block for complex systems, composability lies beyond such nonsoftware-architectural implementation rules. Nevertheless, coding standards can be used for automated checking of the components’ sources statically for compliance with clear results to support the reliability of the system under development (Holzman, 2013), positively affecting the system’s dependability.

The increasing system complexity and the heterogeneous functionality makes the consideration of composability throughout the system development overdue. This gets even more emphasised with the perspective on integration of dynamic functionality while considering an adequate degree of dependability.

5. Proposed Architectural Features

Based upon the problems detailed above, several suggestions can be made to aim for the goal of a more deterministic component integration and prevention of adverse component interactions. The intention is to assemble building blocks that bridge the gap between the conceptual approaches to an applicable solution (or at least significant improvements) for ICM systems that integrate dynamic functionality. However, a single building block may improve the system but also introduce some drawbacks. Hence it is recommended to utilise all suggestions in combination to both: benefit from the improvements while mitigating individual negative effects. The following proposed architectural features were evaluated using a prototype implementation, based upon OpenICM (Knirsch et al., 2012a).

5.1. Component Containment (CON)

Following the classification of Crnković et al. (2011) an exogenous management of extra-functional properties (i.e. NFRs) using containers to encapsulate the components is suggested. While the components concentrate on functional aspects, the containers take care of the NFRs by preventing unwanted interference. This obviates any modification to the components for system integration and effectively implements a separation of concerns. The functional binding between the components is independent of the management of the NFRs. This corresponds to the concept of execution domains (ED), whereas such containers for managing temporal NFRs are implemented using CPU affinity techniques in multi-core environments, virtualisation techniques, or both in combination (Vergata et al., 2011). For a stricter encapsulation Schnarz et al. (2014) describe an asymmetric multiprocessing (AMP) approach in combination with a multi-OS environment. Whereas EDs provide containment within a given OS, approaches based upon a multi-OS environment support containment using an OS domain (OSD), and vOSD for virtualisation respectively. Containment domain (CD) is used as generic term for ED, OSD and vOSD. Figure 3 illustrates containment using distinct CDs, whereas the execution platform depends on the actual implementation of the CD (i.e. ED, OSD, or vOSD).

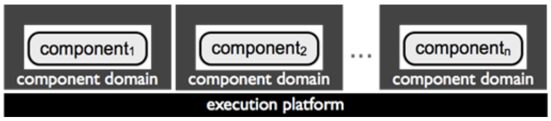

Figure 3: Using CDs for component containment.

An assembly of components sharing a single CD is refereed to as 'composite component'. Although the NFRs of a composite component are derived from the individual components, the characteristics of the composition are not. The composition is a set of components that interact together and hence also interfere with each other affecting compatibility. The possibly resulting adverse behaviour depends e.g. on the current system state, user interaction, component interaction and system load. This leads to a non-deterministic behaviour that may violate super-ordinated NFRs, and potentially affect functional requirements. However, clustering components based on certain characteristics (i.e. similarities) like criticalness, component provider, or semantics may provide adequate means to limit the propagation of adverse interaction. Such containment for dynamic functionality introduced after-market can be realised by use of a CD.

Figure 4 illustrates an exemplary ICM design using different types of CDs. Those are arranged hierarchically to demonstrate the flexibility in assembling mixed approaches for component containment. This means a particular CD may contain one or more other CDs. The depicted system consists of three distinct OSDs for very strict isolation, relying on an AMP based approach: AMP_1 contains instrument cluster components (i.e. classified ASIL B), AMP_2 contains the infotainment subsystem, whereas AMP_3 realises a OSD for an Android OS that provides capabilities to add, update and run dynamic functionality (i.e. 'APPs'). The HW platform provides four CPU cores, with two of them assigned to AMP_2. The latter host two EDs, while one ED contains a vOSD for creating a sandboxed Linux environment and providing four virtual cores. The vOSD then again contains two further EDs, utilizing the vcores.

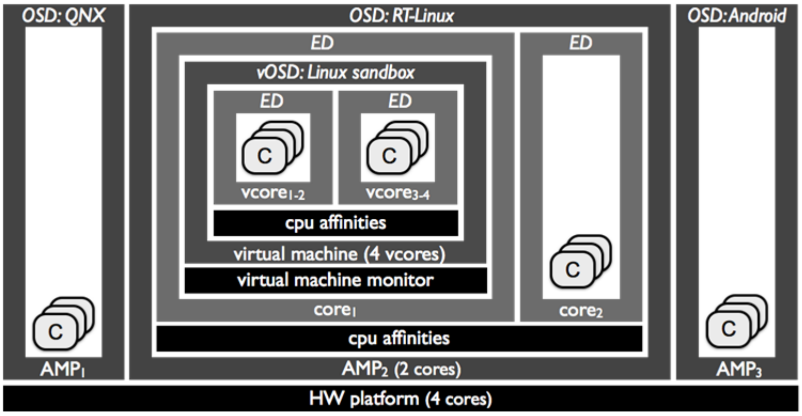

Figure 4: Exemplary composition of different containment techniques.

Similar concepts for data and non-computational resource accesses are already in use and approved for CE devices (e.g. Android Application Sandbox, Apple App Sandbox), while the focus is not set on multiple in-parallel user-operated applications (or components). Also other operating system specific solutions like 'adaptive partitioning' for QNX focussing on the platforms computational resources and the even more elaborate 'cgroups' on Linux feature the implementation of a containing model. For a portable implementation the use of a generic system interface or the abstraction within a domain specific software framework might be advantageous.

However, they may aid the partitioning within a particular component, but inadequately separate components of different safety criticalness. For the latter only multi-OS based containment might provide the required rigid partitioning.

5.2. Component Communication (COM)

Containment isolates the components, but they need to be able to interact with each other; they are interdependent. Basically, the connections in between are realised through functional interfaces, which enables component composition (also referred to as binding). Those interfaces provide the services of the respective components, i.e. implementing actions that both the provider and the consumer of the interface understand. Hence the interfaces realise interoperability.

ICM Systems are highly interactive, communicating with users and other in-vehicle systems. Hence, they rely on an event-based system based on event-triggers. However, some system components have to fulfil strict temporal requirements and therefore implement time-triggered behaviour of real-time systems. Nevertheless the communication between the components is event-based, which affects both the interfaces of the components and the communication channels. The latter have to be implemented efficiently to reflect the required qualities like performance and responsiveness of the system and the limited available HW resources. This leads to the application of shared memory (SHM) communication, which provides flexibility and adequate throughput. Events are processed using a central dispatching service to relay messages from sender to addressee using synchronised queues (Knirsch et al., 2012a). More complex communication is realised using synchronised data structures (aka. ‘data containers’) within SHM. This implements a loose coupling of components, while fostering an efficient communication flow and functional interoperability.

5.3. Management of Shared Resources (SHR)

Compatibility means coexistence without adverse side effects. Components are integrated into the head-unit and therefore share common resources. Although the next generation multi-core HW provides more computational resources to the system, components have to compete for other shared resources (SHR). Even worse, the access to SHR was implicitly managed through the system’s task scheduler and applied thread priorities and scheduling strategies on single-core HW. This is not the case for in-parallel computed components on multi-core systems. The result is a non-deterministic behaviour due to unmanaged access to SHR and the related latencies. Important (i.e. high priority) components have to wait for unimportant. This affects the compatibility of the components, independent of an applied concept for component containment (e.g. the above described). A management layer as described with the Shared Resource Arbiter (SHARB) in (Knirsch et al., 2012b) is able to make the temporal behaviour related to the access to SHR more deterministic. Hence, resource access management has positive effect on the compatibility.

5.4. Composite User Interface (CUI)

The user interface (UI) has to address the required flexibility for future systems as outlined in 4.3. It constitutes a SHR with special characteristics: multiple components may use the UI in parallel. For the graphical part, several components may render a subset of the visualised frontend, to be blended and mixed on multiple displays (e.g. centre console, instrument cluster, rear mirrors, head-up display). Usually only one single HW graphic accelerator is available to support an appealing presentation of information and entertainment content. The containment of components creates requirements for a specific communication for UI (i.e. streaming of video and audio). To prevent adverse interference while maintaining an efficient communication, a SHM based compositing architecture provides an adequate solution. Notwithstanding a partitioning of components using vOSDs (CDs based on virtualisation techniques), subsets of the UI rendered by different components can be composited while utilising multiple HW graphic accelerators (Knirsch et al., 2013). This facilitates the compatibility due to the opportunity to build a homogeneous UI while partitioning the components into CDs.

5.5. Software Framework (SWF)

The features proposed here leverage composability by affecting derived qualities. It is not recommended to apply a single feature only due to negative or not sufficient effects. A SW framework is able to combine those to simplify their application. Additionally, such a framework is able to cover additional constructive aspects that may have positive impact on composability. In accordance to (Neumann, 2004) this includes modularity and encapsulation (i.e. containment), clean hierarchical and vertical abstraction, separation of policy and mechanism, object orientation and strong typing. Table 1 maps those aspects with the proposed architectural features. Hence, a framework considering and effectively addressing the constructive aspects by use of those features leads to improved composability.

constructive aspects	5.1 CON	5.2 COM	5.3 SHR	5.4 CUI	5.5 SWF
modularity and encapsulation	■	■			
clean hierarchical and vertical abstraction	■	■			
separation of policy and mechanism			■	■	
object orientation					■
strong typing					■

Table 1: Constructive aspects mapped to architectural features.

6. Conclusions and the Future

In the past dynamic content for ICM systems was limited to data. Next generation systems will provide capabilities to install and update functionality during the whole product life cycle (i.e. after-market). At the same time, safety critical applications are integrated onto the same platform, constituting systems of mixed criticality. This puts emphasis on non-functional qualities, in particular on dependability and composability, affected by parallel usage of shared resources (e.g. GPU, I/O, etc.) on multi-core HW. This work is intended to provide guidance for the design of ICM systems and related SW frameworks. Therefore qualities related to composability, their interplay and effects were characterised.

Further, a set of architectural features needed to improve composability, while also considering ICM system's safety requirements and demands for appealing UIs have been proposed. In summary, the combination of certain constructive aspects by use of those features leverages the system's SW components' composability. This provides support for the integration of dynamic functionality and hence prepares ICM systems for future demands while ensuring a deterministic behaviour.

7. References

Attiogbé, C., André, P. and Ardourel, G. (2006), "Checking Component Composability", Software Composition, LNCS, Vol. 4089, Springer, pp. 18-33.

Burns, A. and Davis, R. (2013), "Mixed Criticality Systems - A Review", 3rd Ed., Department of Computer Science, University of York, 2013.

Chung, L. and do Prado Leite, J.C.S. (2009), "On Non-Functional Requirements in Software Engineering", Conceptual Modeling, LNCS, Vol. 5600, Springer, pp. 363-379.

Crnković, I., Sentilles, S., Vulgarakis, A. and Chaudron, M.R.V. (2011), "A Classification Framework for Software Component Models", IEEE Transactions on Software Engineering, Vol. 37, No. 5, pp. 593–615.

Holzmann, G.J. (2013), "Landing a Spacecraft on Mars", IEEE Software, Vol. 30, No. 2, pp. 83–86.

Knirsch, A., Vergata, S. and Wietzke, J. (2012a), "Strukturierung von Multimediasystemen für Fahrzeuge", Echtzeit 2012, Informatik Aktuell, Springer, pp. 69-78.

Knirsch, A., Schnarz, P. and Wietzke, J. (2012b), "Prioritized Access Arbitration to Shared Resources on Integrated Software Systems in Multicore Environments," 3rd IEEE International Conference on Networked Embedded Systems for Every Application, pp. 1–8.

Knirsch, A., Theis, A., Wietzke, J. and Moore, R. (2013), "Compositing User Interfaces in Partitioned In-Vehicle Infotainment", Mensch & Computer 2013, Oldenbourg, pp. 63–70.

Monot, A., Navet, N., Bavoux, B. and Simonot-Lion, F. (2012) "Multi-source software on multicore automotive ECUs - Combining runnable sequencing with task scheduling," IEEE Transactions on Industrial Electronics, Vol. 59, No. 10, pp. 3934–3942.

Mössinger, J. (2010), "Software in Automotive Systems", IEEE Software, Vol. 27, No. 2, pp. 92–94.

Neumann, P.G. (2004), "Principled Assuredly Trustworthy Composable Architectures", Computer Science Laboratory, SRI International, Menlo Park, CA, USA.

Schnarz, P., Wietzke, J. and Stengel, I. (2014), "Towards Attacks on Restricted Memory Areas through Co-Processors in Embedded Multi-OS Environments via Malicious Firmware Injection", 1st Workshop on Cryptography and Security in Computing Systems, Vienna.

Urgaonkar, B., Shenoy, P. and Roscoe, T. (2002), “Resource Overbooking and Application Profiling in Shared Hosting Platforms,” SIGOPS Oper. Syst. Rev., Vol. 36, No. SI, pp. 239–254.

Vergata, S., Knirsch, A. and Wietzke, J. (2011), “Integration zukünftiger In-Car-Multimedia-systeme unter Verwendung von Virtualisierung und Multi-Core-Plattformen”, Echtzeit 2011, Informatik Aktuell, Springer, pp. 21–28.