2015

Improved Composability of Software Components through Parallel Hardware Platforms for In-Car Multimedia Systems

Knirsch, Andreas

http://hdl.handle.net/10026.1/3511

http://dx.doi.org/10.24382/3513

Plymouth University

Improved Composability of Software Components through Parallel Hardware Platforms for In-Car Multimedia Systems

Andreas Knirsch

January 2015

Copyright Statement

Improved Composability of Software Components through Parallel Hardware Platforms for In-Car Multimedia Systems

by Andreas Knirsch

A thesis submitted to Plymouth University in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

January 2015

Abstract

Improved Composability of Software Components through Parallel Hardware Platforms for In-Car Multimedia Systems

Andreas Knirsch

Recent years have witnessed a significant change to vehicular user interfaces (UI). This is the result of increased functionality, triggered by the continuous proliferation of vehicular software and computer systems. The UI represents the integration point that must fulfil particular requirements for usability despite the increased functionality. A concurrent present trend is the substitution of federated systems with integrated architectures. The steadily rising number of interacting functional components and the increasing integration density implies a growing complexity that has an effect on system development. This evolution raises demands for concepts that aid the composition of such complex and interactive embedded software systems, operated within safety critical environments.

This thesis explores the requirements related to composability of software components, based on the example of In-Car Multimedia (ICM). This thesis proposes a novel software architecture that provides an integration path for next-generation ICM. The investigation begins with an examination of characteristics, existing frameworks and applied practice regarding the development and composition of ICM systems. To this end, constructive aspects are identified as potential means for improving composability of independently developed software components that differ in criticality, temporal and computational characteristics.

This research examines the feasibility of partitioning software components by exploitation of parallel hardware architectures. Experimental evaluations demonstrate the applicability of encapsulated scheduling domains. These are achieved through the utilisation of multiple technologies that complement each other and provide different levels of containment, while featuring efficient communication to preserve adequate interoperability. In spite of allocating dedicated computational resources to software components, certain resources are still shared and require concurrent access. Particular attention has been paid to management of concurrent access to shared resources to consider the software components' individual criticality and derived priority. A software based resource arbiter is specified and evaluated to improve the system's determinism. Within the context of automotive interactive systems, the UI is of vital importance, as it must conceal inherent complexity to minimise driver distraction. Therefore, the architecture is enhanced with a UI compositing infrastructure to facilitate implementation of a homogenous and comprehensive look and feel despite the segregation of functionality.

The core elements of the novel architecture are validated both individually and in combination through a proof-of-concept prototype. The proposed integral architecture supports the development and in particular the integration of mixed-critical and interactive systems.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgement

The research project was made possible by the facility offered by the In-Car Multimedia Labs (ICM Labs) at the Faculty of Computer Science, Hochschule Darmstadt, DE, in co-operation with the Centre for Security, Communication and Network Research, Plymouth University, UK. I wish to thank both organisations for their support.

This work would not have been possible without the help and support of my Director of Studies, Prof Dr Joachim Wietzke. Thank you for your constant support, your patience and your indispensable advice, for establishing a superb research environment with the ICM Labs, assistance in publishing papers and presenting at international conferences. His experience and professionalism has been invaluable. I owe much of my success to his guidance.

Thanks must also go to my other supervisors, Prof Dr Ronald Moore and Dr Paul Dowland, who provided invaluable help and guidance throughout my studies and spent a significant amount of time proof reading research papers and finally this thesis.

Thanks also goes to my fellow researchers at ICM Labs. I made new friends here and I thank you for your support, the interesting discussions and fun we had. They are: Mrs Bettina Kurz-Kalweit, Mr Markus Glaab, Mr Pierre Schnarz, Mr Sergio Vergata, Mr Clemens Fischer and Mr Tobias Holstein.

Moreover, I would like to thank Mr Tim Bienias, Mr David Diefenbach, Mr Stephan Fries, Mr Andreas Theis and Mr Bastian Weißbach for their valuable inputs and support on the prototype implementation.

Finally, I wish to thank my family for their patience, ‘distractions’ and support throughout the past years: this thesis is dedicated to you.

Andreas Knirsch

Author's Declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Graduate Committee.

Relevant seminars and conferences were regularly attended at which work was often presented and several papers were published in the course of this research project.

Word count of main body of thesis: 58593

Signed . . . .

Date . . . .

1 Introduction and overview

"Computer Science is the study and management of complexity" (E. W. Dijkstra)

Not too many years ago, driving a car was very different from what it is today. This is the result of a change in various aspects that affect what driving a car encompasses, including increased engine power, improved roads, increased traffic density and other features. However, this is also caused by the introduction of computer systems into vehicles with their effects on active and passive safety, fuel economy, comfort, information and the occupants' entertainment. Although mechanical engineering evolved as well, it did not change as fundamentally within the last decades. Advances in the electrical/electronic (E/E) system have initiated changes within the entire automotive industry. The first automotive microcontroller was introduced in 1977 for the engine spark timing of the GM Oldsmobile Tornado (Charette, 2009). Today, the microcontroller constitute a significant part of both the automotive development and the user experience (UX), while simultaneously affecting all vehicular components. Electronics and software are key innovation factors within the automotive domain and have significant impact on the customer's purchasing decision, although it is possible that the user does not even take notice of the amount of software-controlled functionality (Manfred Broy et al., 2007; Pretschner et al., 2007; Manfred Broy et al., 2011b; Huth and Spahr, 2011; Schneiderman, 2013). This implies that the advances in E/E and mechanical engineering are jointly responsible for the economic success of car manufacturers. Hence, all players of the automotive industry are forced to keep up with the pace of innovation in order to improve or maintain their market position.

A distinctive example for the evolution, tangible even for occupants who are not very interested in the technology behind it, is the In-Vehicle Infotainment (IVI) system and its impact on both the dashboard and rear seat[1]. Infotainment combines information

[1]Nowadays, the rear seats are equipped with dedicated entertainment facilities, also referred to as rear seat entertainment (RSE).

and entertainment into an integrated presentation with the intention of increasing the occupants' comfort (i.e. entertainment during long rides). Two decades ago, IVI systems were non-existent with entertainment provided by an optional FM radio and a cassette or CD player, built into the vehicle's centre-console using standardised connectors and slot dimensions (ISO 7736:84, 1984), enabling the vendor-independent integration of aftermarket devices. The user interface (ECU) are also integrated to provide a UI to adjust the air conditioning or seat position. Whereas formerly, the car radio may have distracted the driver from steering the car safely through traffic, today's systems' functional complexity might lead to the assumption that driving the car distracts from operating the IVI system. However, we have already become used to those systems in a similar manner to the use of mobile phones. Our cognitive and capacitive abilities adapted to the steady improvement with every new generation of automobiles. Moreover, we have become dependent on those systems. This may relate to a dependence on the built-in route navigation or due to having to use hands-free telephony due to legal regulations, or because we desire to make use of our complete audio library synchronized with other devices, as is already state-of-the-art for mobile consumer electronics (CE). This process of socialisation is not reversible (Göschel, 2012).

Moreover, the evolution is on-going still: the connectivity provisioned by IVI systems using wireless access networks enables new functionality, affecting the capabilities of next-generation advanced driver assistance systems (ADAS), as proposed by Bolle (2011). He details this with improvements and functionalities such as intelligent speed adaptation and cruise control by incorporating 'real-time' traffic information and up-to-date map data, with respect to passenger safety and fuel economy.

The evolution of IVI systems correlates with their increasing functional size. The time necessary for the development has decreased in accordance with the shrinking time span before one generation of an IVI system is being replaced with the following one. Additionally, the functionality is interdependent, which leads to a significant growth of complexity. This is countered by the use of modularisation into distinct hard- and software components, following the paradigm of separation of concerns. Components are multi-sourced, developed and maintained separately following a high division of labour. This requires well-defined functional interfaces, following the parallel development of the mechanical components (i.e. assembly units).

Manfred Broy et al. (2007) estimates that the Original Equipment Manufacturer (OEM; i.e. car manufacturer) in the automotive domain creates only 25% of the value. Their focus is set on the vehicle's engine, integration of subsystems and marketing the brand. The result is (a) an increasing number of automotive control units (i.e. ECUs) for modularization at the hardware level to achieve physical separation and (b) a challenging integration at the software level. The former (a) has a negative effect on the per-unit costs, energy efficiency and weight, and hence car manufacturers are interested in decreasing the number of ECUs (Monot et al., 2012). The latter (b) has gained significance due to the fact that a functional description of inter-component interfaces is not sufficient:

software components integrated in a common hardware platform share available hardware resources with a potential negative effect on the temporal behaviour due to unintended adverse interference. Currently applied approaches for integrating multi-sourced software components with different temporal characteristics into a common hardware platform do not effectively cope with the steadily increasing functionalities while providing a means to ensure adequate compatibility in combination with the necessary interoperability. This is detailed in Chapter 3.

Compatibility is a quality related to software components to characterise their coexistence without adverse mutual interference. Further, interoperability complements compatibility and implies the ability of software components to work constructively with one another. The combination of compatibility and interoperability is subsumed to the more abstract quality composability, which is introduced in this research programme as a foundation for the successful development and integration of future IVI systems. It describes the coexistence of different components related to the absence of negative reciprocal effects while facilitating functional interdependencies. Hence, composability constitutes a key aspect especially within the context of multi-sourced software systems with a rising number of heterogeneous functionalities. Following Peter G Neumann (2004) for the context of trustworthy systems, designing a system architecture for composability can have significant payoffs in simplifying system integration, maintenance and operations.

Although this issue applies to a wide field of different domains even beyond automotive systems, the domain of IVI systems has special characteristics. More than a fifth of the amount of software within a current car is related to IVI (Charette, 2009). In parallel IVI utilises the largest share of computational power available within the vehicle. Further, it combines different temporal requirements and relies both on time- and event-triggered computation. Nevertheless, it has to cope with harsh climatic conditions such as all other automotive systems and is operated within a safety-relevant environment, although current IVI systems are not classified as 'safety critical'. The latter aspect has led to a very static system design to enable predictable behaviour and improve test coverage. However, future systems will enable a dynamic deployment on user demand and during operation of software components to those systems, as is common practice for consumer electronics (CE) (Mössinger, 2010; Bolle, 2011). This is a fundamental change for those originally very static systems. It will have significant impact on the predictability of the systems' behaviour and cause a serious decrease of test coverage due to the exponential growth of possible permutations of installed components. A significant amount of IVI functionality (e.g. as related to entertainment) may not be classified as safety critical and would not require a rigorous (re)-verification process that is compliant with extensive certification standards. Nevertheless, compared to desktop applications or CE devices, it is still being employed within a safety-critical environment. Integration to a common platform or connectivity to a shared bus may cause adverse interference that would include non-critical functionality but also potentially affect critical functionality.

Aside from the functional growth of IVI, the scope of past and current systems was and is limited to information and entertainment with the focus on the occupants' comfort.

Current developments lead to systems where the distinction of safety-critical software functionalities, such as the instrument cluster, become blurred due to integration onto common hardware platforms. The instrument cluster provides functionality (e.g., indicators for gear, exterior light, speed control) that is classified as safety-critical using level ASIL B (Automotive Safety Integrity Level) to comply with the standard for functional safety of road vehicles (ISO 26262, 2011). Today, systems are already available that display IVI content within a fully digital instrument cluster (e.g. route navigation information in the direct vicinity of the speed gauge). Those systems are implemented using several distinct ECUs interconnected by in-vehicle bus systems (e.g., CAN, MOST) and direct links to achieve an efficient video transport (e.g. LVDS). However, it is the declared intention of car manufacturers to reduce the number of ECUs while utilising available multicore (MC) hardware architectures (Monot et al., 2012). This implies the integration of mixed criticality components while giving up the traditional 'major architectural element' that gives confidence in correct operation: physical separation (Knight, 2002). The consequence of a decreasing number of dedicated hardware units is an increasing integration density at the software level for a single hardware platform while using a multi-source code base. In result, 'IVI' evolves into a mixed-criticality system (MCS), combining distinct levels of assurance against failure (Burns and R. 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).

Due to progress which has already begun as concerns the shifting of the focus of IVI systems, the term IVI is set too narrow and does not reflect the denominated system very well. Future IVI is anticipated to go beyond mere information and entertainment due to the fusion with other existing and upcoming functionality to build an integral UI for the occupants. In the following chapters, the broader term In-Car Multimedia (ICM) is used to differentiate from IVI.

This sections has outlined the problem domain using several relevant aspects to justify the relevance of this work. The applied software engineering approaches have to address those aspects, especially the integration of heterogeneous software components onto a single computing platform. Again, for this context composability of components represent a key for the successful development of future ICM.

Based on these foundations, the next sections describe more concisely the challenges related to this research to narrow down the focus and make its aims more tangible.

1.1 Focus of this research

The independently developed software components of next-generation ICM have to meet specific temporal requirements while being deployed and integrated onto a shared embedded hardware platform. Despite the substantiated heterogeneity at the software level, the UI of those systems has to provide all functionality in a comprehensive and uniform way, blended into the OEM's usage concept. The UI provides multiple distinct modes for interaction that are usable simultaneously (a.k.a. multi-modal interaction). To achieve an adequate and purpose-oriented 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 multiple I/O facilities. Moreover, the system contains safety-critical components and operates within a safety-relevant environment and therefore has to provide sufficient dependability, or put more simply: it must work as intended. This relates to the components' functionalities despite concurrent use of the hardware resources as well as to the integration at the UI level, characterised with compatibility and interoperability – or subsumed composability.

The availability of parallelism with MC hardware architectures provides an unexpected means to establish a software architecture that addresses such systems' challenges, including conflicting temporal requirements and mixed criticality.

This builds both the motivation and a general foundation for this research. The particular objectives are outlined in the following section in order to narrow the focus of this work even more, including relevant limitations.

1.2 Aims and objectives

The aim of this research is to propose an architecture for next-generation ICM systems which is capable of fulfilling demands for a growing number of multi-sourced and mixed-criticality software components and ensuring adequate composability[2]. The intent is to describe the elements for a software framework that incorporates this architecture to foster applicability for software development and integration process for ICM. Within this context the focus is set on increasing the determinism regarding the behaviour at runtime while improving the predictability of the integration and considering an integral UI to facilitate a positive UX.

The main objective of this research is to explore, propose and evaluate adequate architectural features that enhance the components' composability (Peter G. Neumann, 2006; Kopetz and Obermaisser, 2002). In order to achieve this, this project is divided into five distinct stages:

1. To investigate relevant characteristics and requirements for the ICM system and reason the need for architecture that fosters the integration and parallel operation of heterogeneous functionality that varies by means of criticality, temporal requirements, and demands for software updates.

[2]A detailed discussion on ‚composability' is provided in Section 2.3.1.

2. To create a comprehensive review of approaches regarding composing software components and the applicability of component isolation technologies for achieving compatibility to ICM systems.
3. To investigate the need for necessary communication channels between software components and shared resources and a shared UI respectively with regard to component interoperability.
4. To propose a novel software framework to support the integration of pristinely incompatible components without the need for modification.
5. To evaluate the applicability of the proposed concepts and derived software framework's features by use of a prototype implementation.

1.3 Method overview

This research employs empirical research methods for achieving its objectives. Basically, new ideas are developed following a review of literature and case studies of recent industry development projects (with the latter having had considerable influence on this research). Following Segal (2003) and Segal et al. (2005), the evidence from field studies of applied software engineering practice is essential in order to understand current practice. Those new ideas are transferred into architectural concepts and design principles for next-generation ICM systems to verify their feasibility. Therefore, a 'proof-by-construction' approach is employed, with building prototypes applied to particular cases. This allows deriving respective perspectives to engineer such software intensive systems. Thereafter, a 'pilot case' (a.k.a. demonstrator) characterises the value and appropriateness of the ideas by applying them to a representative case. The objective here is to demonstrate their 'in-context-value', again following a 'proof-of-construction' approach. Formal models are used to abstract the essentials of the proposed concepts and principles and provide alternative perspectives throughout this work.

Such an experimental approach is appropriate due to the problem requiring a complex software solution, as categorized with 'Experimental Computer Science' according to Dodig-Crnković (2002). The approach here is to identify concepts that facilitate solutions for compositing next-generation ICM systems and then evaluate the solutions through construction of prototype systems.

The selection of methods for this research basically matches with the findings of Höfer and Tichy (2007). They classified empirical research published in a software engineering journal into eight methods. 'Experiment' and 'Case Study' are the two preferred research methods which comprise a share of 66.2% based upon 133 reviewed articles.

1.4 Thesis outline

Chapter 2 begins by reviewing the software development methods applied to current ICM systems, establishing the importance of composability of software components. By presenting both significant characteristics and relevant requirements and their interdependencies respectively, a demand for a new design and implementation approach is identified. The chapter concludes by highlighting the need for a flexible architecture that supports the deterministic development of valid compositions, predictable both on functional and temporal behaviour.

Having established the need for an advanced architecture, Chapter 3 discusses particular aspects of software compositions in more detail. Therefore, approaches applied within recent exemplary development projects are presented, accompanied with these projects' characteristics and an interpretation of the results. The latter provides valuable information for future projects and the architecture for next-generation systems. The chapter concludes with a comprehensive literature review to date that backs up the interpretation of the projects and addresses derived essential architectural features for future systems.

Chapter 4 introduces a number of applied technologies for partitioning software components. Comparing their characteristics and mapping those to the heterogeneous requirements, an exemplary system design is composed. It mixes technologies in order to utilize their different advantages. The chapter then proceeds to describe how these technologies can be combined for an applied framework to positively affect the software development and integration process. The chapter concludes with a recapitulation of the advantages and disadvantages with the focus set to limitations on compatibility and interoperability.

Chapter 5 then reviews the issue regarding the arbitration of shared resources to mitigate limitations on compatibility. It proceeds with the presentation of an access arbiter that enables the prioritisation of concurrent access of shared resources. Based on a number of experimental results, both the effect and the costs are illustrated to demonstrate its applicability. The chapter finishes with a summary covering the applicability and limitations of shared resource arbitration.

Chapter 6 details the implications of a shared UI for partitioned software components with the focus set on interoperability. After describing relevant requirements, a novel compositing infrastructure for ICM systems is proposed that enables the implementation of a flexible visualisation of different graphical elements without the need to interpret the actual content. Its applicability is illustrated using a prototype implementation which is built upon the previously-described architectural features for partitioning heterogeneous software components. The chapter concludes with a critical discussion on the compositing infrastructures' benefits and limitations.

Chapter 7 assembles and evaluates the proposed architectural features for partitioning, resource arbitration and UI compositing. The chapter describes the integration of these into a comprehensive framework. Then an exemplar system design in combination with simulated use-case scenarios is described. The chapter finishes by discussing both the feature's applicability in coexistence and their impact to the system behaviour using the results of the simulation.

Finally, Chapter 8 presents the main conclusions from the research, highlighting its achievements and limitations. Future research and development for this project are also suggested in this chapter.

2

Review of software development for ICM

This chapter aims for a clear definition of the problem domain to provide a well-defined background and understanding of this research and for the following chapters. Hence, it builds the foundation of this work by introducing ICM systems, their characteristics, applied development approaches for such systems and supporting frameworks. In other words, this chapter describes what ICM is about and what the herein addressed challenges are.

2.1 Terminology and introduction to ICM

Initially the term ICM is examined to clarify its meaning, especially because it is not yet a widely-used term for the named systems in both academia and industry. Nevertheless, there is reason for this naming, discussed in the following.

Multimedia is a rather broad and overused term with different meanings depending on the context of use. Steinmetz and Nahrstedt (2004) provide the following definition:

> *"From the user's perspective, multimedia means that information can be represented in the form of audio signals or moving pictures."*

For this research, the term multimedia is used as proposed by Vaughan, who gives emphasis to the integral aspects of multimedia systems (Vaughan, 2011):

> *"Multimedia is [...] a woven combination of digitally manipulated text, photographs, graphic art, sound, animation, and video elements. When you allow an end user [...] to control what and when the elements are delivered, it is called interactive multimedia."*

An ICM system is a software intensive system that consists of components that provide information and entertainment within an automotive context. In regard to the statement of Vaughan, the 'end user' is any vehicle occupant who interacts with the multimedia system by use of knobs, levers, and buttons dispersed over the dashboard and the steering wheel, as well as touch panels, touch-displays and speech recognition systems. However, this enumeration is not extensive. The UI and hence the provided modalities for interaction may differ with the make and model of the vehicle. Nevertheless, the provided functionality is usable from different seats in the vehicle. Derived from the definition of Vaughan, ICM systems provide interactive multimedia, implemented by use of applications that handle user events.

The established human machine interface (HMI) provided by a car can be reduced to a steering wheel, pedals, a gear stick and a set of switches and knobs for lights, indicators and horn. But in recent years, it has evolved to an highly interactive information and control interface, providing the ability to use satellite-supported route navigation, to listen to Internet radio, receive emails, control automotive body functions, air conditioning, seat positioning, and many more, by use of ICM. The significance of automotive information and entertainment (conflated to the artificial term 'infotainment') has grown rapidly. Beside the driver, passengers are able to use those facilities through individual interfaces connected to the same software system, even from the rear seats. Nowadays, value creation within automotive domain is primarily dependant on software, with effects both on increased costs and complexity (Ebert and Jones, 2009). A manufacturer's competitive advantage relies more and more on compelling functionalities provided to the passengers (Manfred Broy, 2006). They represent the central information interface between the car and its occupants and already affect a prospective customer's purchase decision. Therefore, 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 platform: 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. They actually already implement a great share of the vehicular HMI, although not directly interacting with the car's driving dynamics, but communicating with other ECUs by use of different vehicular fieldbus networks. This evolution has led to large-scale complex software systems of >25 million lines of code (MLOC) (Charette, 2009; Smethurst, 2010), decomposed into software components to tackle the complexity and integrated into the head-unit to achieve cost efficiency in production. The components can be regarded as architectural subsystems that form a 'coalition of systems' that inherit the characteristics and challenges of large-scale complex IT-systems, as discussed by Sommerville et al. (2012). They conclude that a traditional centralized engineering perspective is no longer adequate:

> *"Current software engineering is simply not good enough. We need to think differently to address the urgent need for new engineering approaches to help construct large-scale complex coalitions of systems we can trust."*

The head-unit acts as central computing instance of an ICM system (Wietzke and Tran, 2005, p8). Thus, the head-unit is implicitly meant when addressing the hardware of ICM systems due to its central role within the system. It is basically an automotive embedded system that provides entertaining facilities, context-based information, and control over automotive body functions through an integral and comprehensive user interface. According to Smethurst (2010), the head-unit is the most complex component by software volume within a car. It contains approximately 70% of the total code in an automobile to provide the logical interface between the car and its users (i.e. consumer space). Moreover, he emphasis relevance of ICM and the head-unit for OEMs (Original Equipment Manufacturer) and their economic success respectively:

> *"The environment created in the car is a key differentiator for the OEMs compared to aftermarket solutions, and it's not a domain that the OEMs can afford or intend to surrender."*

The head-unit features various interfaces to other systems and the occupants of the car, their mobile gadgets and storage devices. Figure 2.1 depicts some common components and interfaces of next-generation systems.

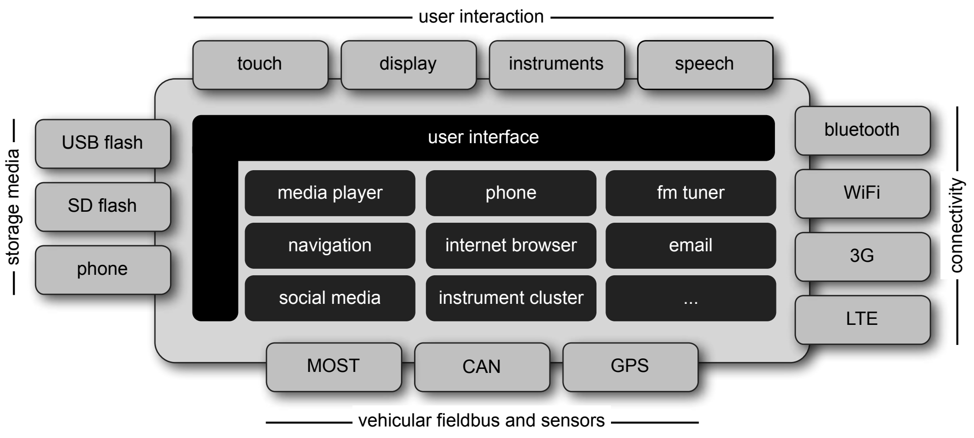

Figure 2.1: Exemplary ICM components and interfaces

In the past, the scope of such systems was limited to non-critical functionality. Current developments led to systems that incorporate safety-critical functions, such as, for instance, the instrument cluster that contains functionality classified ASIL B following (ISO 26262, 2011). Such systems provide functionality beyond information and entertainment: thus they became essential for the safe operation of the car. The resulting mixed criticality is a fundamental architectural driver and hence is discussed in more depth in the following section.

As mentioned, there do exist several terms for ICM that are already widely accepted, such as 'automotive infotainment' or 'in-vehicle infotainment' (IVI). These have been coined by the marketing divisions of different manufacturers, rather than defining a difference in functionality and handling. However, this puts emphasis on the need to differentiate from ‘legacy’ IVI and to reflect the growing area of responsibility parallel to the shift from straight supply of contents to interactive multimedia. This backs the decision to make use of the term ICM for this research to cover the next-generation’s wider range of functional features beyond ‘information’ and ‘entertainment’ while emphasising user interaction. Nevertheless, IVI and ICM might be used interchangeably as long as the described implications caused by the recent developments are considered.

2.2 Characteristics of ICM systems

For providing interactive multimedia, various disciplines within the domain of computer science have to be considered, driven by the combination of different but interdependent functionalities at the design level and subsequently applications and components at the implementation level.

A hierarchy of functions, applications and components support the management of the system’s complexity by the principle of ‘divide and conquer’. Based on the heterogeneity of those components, the application layer of ICM systems refers to different specialised fields of computer science. A selection of those is given by the enumeration below:

- Software engineering
- Multiprocessing
- Computer graphics and visualisation
- Human-computer interaction
- Natural language processing
- Telecommunication
- Media encoding and rendering
- Databases

ICM systems might appear to the user as a slightly enhanced CE device comparable to current Internet tablets or mobile phones both in computational performance and application diversity, but offered by automobile manufacturers for a premium price. In the following the characteristics and differences to other (mobile) embedded devices and their development respectively are discussed. Intent is to reason why ICM systems are ‘special’. Therefore, viewpoints of the automotive domain and CE domain are presented alternately.

2.2.1 Classification

An ICM system is classified as embedded system, which according to Marwedel (2011) is defined as an information processing system embedded into an enclosing product. For ICM, the ‘enclosing product’ is the car that contains a set of up to 80 distinct but interconnected embedded systems (i.e. ECUs), including ICM (Manfred Broy et al., 2007;

Charette, 2009; Monot et al., 2012). Those embedded systems consist of a hardware platform and a software system. This link between the software and a physical system is further supported by E. A. Lee (2006) who defines embedded software as software integrated with physical processes, while the technical problem is managing time and concurrency in computational systems. This perception coincides with the one of Ebert and Salecker (2009), who also defined embedded systems as microcontroller-based systems built into technical equipment, while the software in those systems is defined as embedded software. Furthermore, they state that the end-user usually only recognizes the set of function provided by the overall system and not the embedded software. This applies particularly to ICM. It further implies embedded systems only exist as part of a larger system, such as technical processes that pose external constraints that have to be addressed. This led to cyber-physical systems (CPS), which are according to E. A. Lee (2007) integrations of computation and physical processes. He further remarks on considerable challenges in regard to CPS, because the physical system components issue safety and reliability requirements different from those in general-purpose computing and the CE domain due to their inherent criticality.

Definition *Criticality*

Criticality is a designation of the level of assurance against failure needed for a system component (Burns and R. Davis, 2013).

Furthermore, ICM systems are automotive software systems due to their ‘enclosing product’. Manfred Broy et al. (2011a) define the characteristics for such systems as follows:

- Multiple, conflicting and error-prone requirements
- Real-time properties
- Stringent and high-volume communication requirements
- Multi-functionality with complex dependencies between functions
- Heterogeneity of the application domains

Although these are mainly focused on automotive control software, they are also applicable to ICM, admittedly with limitations regarding (hard) real-time requirements. Those do only apply for a subset of ICM functionality, particularly for components that communicate with other automotive control units, or for real-time audio processing and to support adequate responsiveness to user interactions. However, the given characteristics are not unique to automotive software, but the combination amplifies challenges of software engineering, particularly with regard to design, analysing and debugging such systems.

2.2.2 Environment

ICM systems have to cope with a harsh operational environment due to their ‘enclosing product’ when compared to other software-intensive multimedia systems. This includes shock resistance, usability within divergent climates (e.g., wide temperature and humidity ranges and varying atmospheric pressure), and system lifetimes of 15 years and longer after end of production. While those mainly affect the hardware part of the systems, there are further issues also to be considered by the software design. This includes as well low voltage situations during start of the vehicle’s motor, fast system availability (standby is not a solution due to tough bias-current requirements), and problematic system maintenance. Within this context the term ‘maintenance’ includes not only repairs but also all modifications of the software system during system life. This includes the update, removal and enhancement of the provided software functionality and software components respectively. However, the long system lifetimes demand applicable concepts to update the software and thus improve the provided functionality and to address new demands.

2.2.3 Architecture

From the user’s viewpoint, the value of an ICM system mainly depends on its functional features and the way they are presented. The functionality is implemented in software by use of several application components. These appear as a uniform system to the user by use of one integral interface. Neither the composition of functionality of individual components nor their physical structure (e.g., partitioning into separated ECUs vs. integration onto a common hardware platform) is recognized by the user. This makes ICM systems highly integrated software systems, composed of heterogeneous and interdependent components. A massive over-commitment of hardware resources emphasizes this. In the past, a distinct ECU was equipped with hardware resources to explicitly match the requirements of the software (and vice versa). Following this approach, sufficient computational power and memory is available to fulfil the specific use-case assigned to that subsystem. Although current systems benefit from increasing computational power and memory capacities, a hardware platform may not provide the resources necessary to run all application components in parallel. At a certain threshold (i.e., at high system load), the system’s capacity is exhausted and the software components’ demands may not adequately be fulfilled anymore, causing degradation, for example, in performance, availability and responsiveness. Such under-provisioning of the platform’s resources by intention implies increased flexibility during integration, as there are no hard limits regarding how much functionality is deployed to the target system. It is widely applied within embedded computing for the CE and desktop computing. But for safety-relevant automotive systems, it obviates determinism regarding system behaviour in high system load situations as long as no adequate measures apply that reflect the software components’ mixed criticality.

Definition *Mixed Criticality*

A system with mixed criticality has multiple distinct levels of assurance against failure needed for different system components (Burns and R. Davis, 2013).

Furthermore, an ICM system incorporates multiple roles. It forms a converging point for the functionality of multiple domains of computing, signal processing, and communications. This leads to systems that integrate components reflecting a heterogeneous set of Models of Computation (MOC) (Martin, 2002).

Inadequately structured complex systems cannot be effectively maintained. Moreover and in relation to the demanded systems reliability because of the inherent criticality (cf. Section 2.2.5), it cannot be expected that verification methods would cope with such systems (Kreiker et al., 2011a; Kreiker et al., 2011b). This puts emphasis on the architecture and applied design principles. The most significant, as already mentioned, is the use of software components. It is a fundamental engineering principle - even beyond software engineering (SE) - to decompose larger systems into smaller parts (top-down) and to compose smaller parts into larger systems (bottom-up). Component-based software engineering (CBSE) has been a research area for many years and represents a specific field within the domain of SE. According to Crnković et al. (2011b), it addresses topics such as composability, predictability of functional and non-functional properties, modelling of component based systems, reusability, deployment, software architecture, and middleware. Furthermore, they denote the idea of keeping component development separate from system development, a basic concept of CBSE. This in particular enables multi-source software development to improve efficiency and can help to shorten the development period. A fundamental principle is composability that considers both functional and non-functional properties (Crnković et al., 2011b), with the goal of supporting compatibility and interoperability (Peter G Neumann, 2004).

Definition *Composability*

Composability combines coexistence of different components without adverse side effects with the ability to work constructively with one another.

Adequate composability positively affects the predictability of success and efforts necessary for systems integration. A detailed discussion of composability is provided in Section 2.3.1.

The concepts and principles of CBSE fit very well to the demands that arise with next-generation ICM systems which must incorporate pre-existing components (i.e., legacy or COTS), build new components to address new requirements as reusable entities, and evolve by replacing components. This aspect is discussed in more detail in Section 2.2.4.

More recently within the mobile CE market segment, dynamic architectures have proved their applicability (Pope and Muller, 2011). Functionality can be updated, substituted, removed or added by the end user or during regular system service. A particular component can be composed into the system during operation on user demand, or respectively an already-incorporated component can be removed. Although the latter may only apply for a subset of the initial installed components. In both cases, the resulting system must still provide all 'services' as intended and specified.

Certain aspects regarding system architecture are taken up in Section 2.4, including current trends within automotive domain (e.g., AUTOSAR and GENIVI).

2.2.4 Development

The component-based architecture effects the development of ICM systems. The development process of such systems is usually highly parallelised through division of labour with involved parties from different locations, organizations, or domains. Among others, this is attributed to encounter the automotive domain's substantial time-to-market pressure. ICM systems are too complex to be designed from scratch for each new product or generation due to productivity reasons (Manfred Broy et al., 2011b). Thus, ICM systems are both functional and architectural assemblies of legacy, COTS and new components. The complexity issued through composing of different sets of functional requirements demands profound design and integration knowledge, more than is required for single-component development groups (Martin, 2002).

The segmentation into distinct components has been an approved approach within the domain of automotive mechanical engineering for decades, following a strict strategy of 'divide and conquer' (Manfred Broy, 2006). Components are specified using functional requirements, implemented and produced by a supplier, and assembled on a clocked assembly line at the OEM. Therefore, the components have to fulfil the predefined interfaces on which all related parties have agreed upon. Or put more simply, they have to fit together.

For software engineering, the assembly line is the integration of the components during the development process. However, for software an essential limitation must be considered: although the components' interfaces might be well defined at the functional level, this must not necessarily apply to non-functional requirements (NFR). As this might be adequate for mechanical components, software shares common resources, such as field bus communication lines, computational power, and memory storages. This may affect the temporal behaviour during high system load situations. Moreover, even if such behaviour is defined, it is challenging to implement. The reason for this is the potential 'non-deterministic nature' of interactions among components within complex systems of systems – at least for the developers who focus on only a subset of the overall system. Although operating systems do provide means to counter those problems by use of scheduling policies and task priorities, the independent and parallel development of those systems' components decrease their applicability. Hence they are not sufficient to ensure the prevention of negative inter-component interferences and to achieve valid compositions. The situation is exacerbated by the fast-growing system complexity to address new demands and utilize new evolvements including, for instance, wireless ad-hoc networks, web-based services and ADAS (Bolle, 2011). However, the automotive industry is a highly competitive domain. Hence costs for design and production must be kept low. This implies it is vital for the different stakeholders to cope with the complexity (Simonot-Lion, 2009).

Additionally, there is a significant economic pressure which affects both the development as well as the maintenance and warranty for ICM systems. Although the economic pressure is not unique for the automotive market domain, both the need for the timely coordination with the chain of production for the whole vehicle and the difficult maintenance situation with respect to product recalls requires a different level of professionalism to improve predictability related to the development of reliable systems. This becomes even more weighty when considering the user's perception of ICM systems. As part of the automobile, an ICM system is recognized as part of the OEM (i.e., the car manufacturer), and not of the Tier-1 OEM (i.e., supplier to the OEM providing the ICM system). Therefore, a problem with the ICM is addressed to the OEM, even if the OEM just mounts it to the dashboard as a distinct process step along the production line of the car. This implies that problems occurring in the systems which have been caused by items provided by a supplier affect the reputation of the OEM, which may in turn affect the economic success. Di Natale (2008) provides a brief summary regarding the structure of the automotive supply chain.

Further, the development process has to be coordinated with all other parts of the vehicle due to a synchronised assembly at the OEM's production line. Therefore, a development project has to consider the start of production (SoP), which forms the deadline where all development has to be completed. With SoP, the maintenance period begins and continues basically until the end of production (EoP), as illustrated in Figure 2.2.

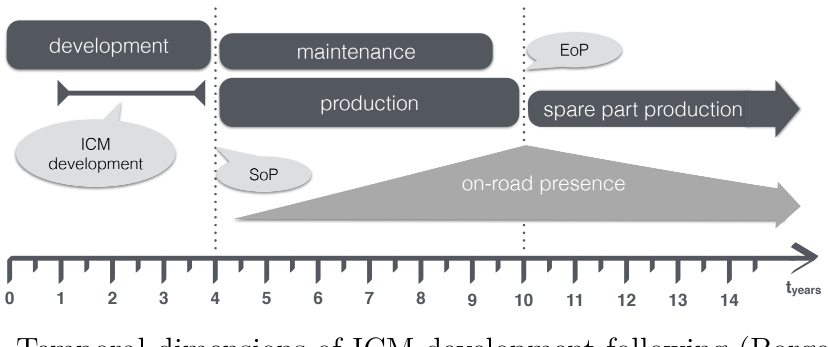

Figure 2.2: Temporal dimensions of ICM development following (Borgeest, 2014).

2.2.5 Criticality

The criticality of a system or a system's component is related to the impact to its environment. It correlates with the 'idea of safety' and has an effect on required system qualities like reliability, dependability and trustworthiness. Due to the related potential affects to a person's health or damage of objects, the criticality of ICM is emphasised and discussed in-depth in the following. But, before detailing the relation to safety and reliability, the use of the terms safety, reliability, failure, dependability, and trustworthiness are defined.

Safety is the state of being safe from consequences of events that are considered non-desirable. This implies a protection from any events that could cause health problems or economic losses, or at least the ability to insure that particular hazards can be contained within a defined level of risk (ICAO, 2013, p2-1). A more colloquial definition to the point from an unnamed safety officer is provided by Stroustrup (2009, p887):

> *"'Unsafe' means 'somebody may die.'"*

Reliability is defined as the probability that a system or component satisfies specified behavioural requirements over time and under given conditions (i.e., does not fail) (Leveson, 1995). A failure is defined as the inability of a component or system to perform its intended function (i.e., caused by a system error). Intended function is defined with respect to the component's behavioural requirements (Leveson, 2011, p8). A common cause of failures in the field is related to 'implicit environmental assumptions'[1] within the software (Sha, 2009).

Dependability is defined as the ability of a (computer) system to deliver service that can justifiably be trusted, while the service delivered is its behaviour as it is perceived by its users (i.e., humans or other systems) (Avižienis et al., 2001). Following Avižienis et al. (2004), it is an integrated concept that encompasses system attributes including reliability. Other attributes are availability, safety, integrity and maintainability, whereas the weight of each attribute may vary related to the field of application, type of system, and accepted dependence. Adopting the view of Peter G Neumann (2004), the concept of dependability is essentially indistinguishable from what is termed trustworthiness. Avižienis et al. (2004) systematically decompose the concept of dependability into threats, attributes and means. They illustrate this using a tree, depicted in Figure 2.3. Especially the attributes related to dependable systems are relevant for defining requirements for a particular system by putting different emphasis on different attributes. Therefore it might be necessary to further decompose those to address relevant drivers for system architecture and design to achieve a verifiable system.

Nevertheless, the system's criticality is the most fundamental characteristic to justify the need for a particular degree of dependability. The criticality of a system might be

[1] i.e., not part of any specification; not necessarily obvious to all involved in development or maintenance

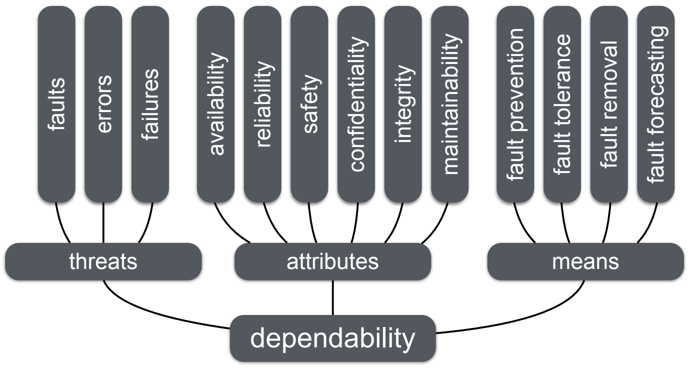

Figure 2.3: Dependability tree following (Avižienis et al., 2004)

described and justified most illustratively by its negative effects. This is done in the following, based upon quantitative figures from available reports related to relevant research to establish the link to ICM.

In 2010 a total of 92,492 people died of road traffic injuries within the European Region[2], whereas 50% were car occupants. For every person dying, 23 were admitted to hospital and 112 attended to an emergency room (Mitis and Sehti, 2013). These figures prove that cars are safety-relevant. Hence, automotive-embedded systems are operated within a safety-relevant environment. This applies without reference to their individual purpose. Such systems can be classified as safety-critical based on their impact to the safety of the vehicle's passengers (i.e., with direct impact on the driving dynamics such as braking and stability systems). Electronic automotive systems have to fulfil tight safety requirements (Di Natale, 2008). This basically also includes ICM. Admittedly, those systems are not the cause for all of these injuries and loss of life, and ICM has no direct impact on the safety of the vehicle's occupants. In reality, the 'human factor' is seen as a very important aspect for the cause of most accidents. A car represents a safety-critical actuator device controlled by an error-prone operator (i.e., the driver) (Sha, 2009). Hence, the driver still has an important role in regard to traffic safety of the driver, passengers, other vehicles' occupants and others. This puts emphasis on the interface to the driver, including ICM systems.

ICM systems can and will be operated while the driver operates the car through the traffic which can be a demanding task by its own. Although this increases the comfort for the driver, it also causes driver distraction, which can affect safety. As of the relation to the safety, driver distraction indirectly is also related to the concept of criticality, because a level of assurance against failure has to be considered at least for parts of the system.

Further, distraction is increased due to inadequate usability provided by the system. The latter may cause (1) unnecessary cognitive load for the user (which has a limited cognitive capacity) or an increased (2) visual and (3) manual distraction, following the three sources of driver distraction according to Strayer et al. (2011).

[2]European Region as defined by the World Health Organization (WHO)

Horberry et al. (2006) showed that performing an additional in-vehicle task (e.g., tuning the radio or conducting a conversation) may have a negative impact on driving performance. For the interaction with the entertainment system, they showed the affect on measures like maintaining speed as well as preparedness for hazards which might compromise the safety of occupants and other road users. That means an ICM system is regarded only as secondary[3]/tertiary task for the driver as its not directly related to the driving task, following the classification of Tönnis et al. (2006). Any distraction from the primary task should be minimised. Pfleging et al. (2014) provide a comprehensive overview on the current field of research related to vehicular human-machine interaction to reduce driver distraction, including multimodal, implicit and context-sensitive user-input.

Despite these improvements regarding usability and usage concepts of ICM systems, the provided reliability also has an impact on the actual focus of the driver. This means unexpected system behaviour or failures may also cause the drivers to withdraw their attention from processing information necessary to operate the car safely, take their eyes from the road, their hands from the steering wheel, and combinations of these sources of driver distraction. One could possibly imagine the distraction caused due to, for example, the media player's audio volume increasing to maximum (or even mute; without the possibility of restoring it to a normal state), the climate control to maximum heating or fan to maximum speed, delayed or incomprehensible route guidance, a sporadic display blackout, a hands-free phone system with incomprehensible audio communication. This list could be easily expanded in accordance with the increased amount of provided functionality.

Additionally, the number of systems that visualise content and require user interaction has increased dramatically, affecting the driver's cognitive load as well as increasing visual and manual interaction. This may increase the risk of driver distraction and make the achievement of an adequate usability and reliability even more important.

The impact of distraction is further supported by the research of the U. S. National Highway Traffic Safety Administration (NHTSA). They report for 2010 that a share of 17% of all police-reported crashes involved some type of driver distraction. Of those 899,000 crashes, the distraction was caused by a device or control integral to the vehicle in 26,000 crashes, which means 3% of the distraction-related police-reported crashes (NHTSA, 2012). A further research NHTSA states 10% of reported fatal crashes were distraction-affected, causing the death of 3,331 people in 2011 in the United States (NHTSA, 2013). This does not imply ICM systems are necessarily violating the safety of the vehicles' occupants. However, it does put emphasis on the safety relevance of vehicular systems that are not directly related to driving dynamics.

The intent of the OEMs is to reduce the number of ECUs (Navet et al., 2010; Monot et al., 2012). This implies the integration of former distinct systems onto common hardware

[3]With respect to the potential integration of the instrument cluster into the ICM function set to support maintaining of the vehicle's speed.

platform. Hence, components with different levels of criticality are combined. This applies to ICM, e.g., for the integration with the instrument cluster that is classified as safety critical, or for the role as data provider and network gateway for next-generation ADAS functions (Bolle, 2011). That emphasises the demands for reliable ICM systems with consistent and easy to use HMIs, compatible to the remaining car controls.

However, it is often assumed safety correlates with reliability; following the rule 'increased reliability implies increased safety.' This might be based on the perception that 'accidents' or safety related incidents will not occur as long as a system or component does not fail. However, according to Leveson (2011, p7), safety and reliability are different properties, meaning one does not imply or require the other. She provides good examples for both (a) safe but unreliable and (b) reliable but unsafe systems. An obviously safe but unreliable system (a) is designed to fail into a safe state. This means that although one or more components fail, the system's robustness mitigates those (isolated) failures to prevent an accident. However, this does not imply it does not matter whether a particular component may fail or not. But in such a case, the effect is deterministic. Further, within complex systems an accident may also result from interactions among components, each satisfying their individual requirements. Although a system's components are reliable when examined individually the system might be unsafe due to the component's interactions (b).

Such component interaction accidents are gaining significance as the system's complexity increases. The derived logical complexity is seen as a major driver for software defects (Sha, 2009). According to Perrow (2011), this can be related to the reduced intellectual manageability of design, the insufficient ability to plan, understand, and anticipate interactions among components. He claims accidents[4] caused through system characteristics like interactive complexity and tight coupling can be classified as 'normal accidents' or system accidents. This means with the given system, characteristics multiply and unexpected failures are inevitable (i.e. 'normal'). This view can be mapped to software systems that assemble multiple interacting and multi-sourced components. Within this domain, interaction can be divided into planned and inherent interaction. Planned interaction is basically the functional interface that implements the communication between components necessary to fulfil the requirements and derived from the system design. Inherent interaction covers mutual interference caused, e.g., through integration onto a common hardware platform that implies a concurrent use of resources or sharing of a communication media. The latter are not necessarily obvious and hence are not easy to understand and thus support the anticipation of possible effects that may violate the required system behaviour.

Moreover, the already-mentioned problematic maintenance situation and the relatively long system lifetimes compared to CE demand reliable systems. The reliability must be preserved, as systems and their software gradually change and evolve to provide improved

[4]defines an accident as (at the minimum) an unintended and untoward event that involves damage to people, objects, or to both.

services and address new demands. This challenge is also addressed by Hoare and Milner (2005) as one of seven themes of the UK Computing Research Committee (UKCRC) that describe the grand challenges for computing research. Although this issue is independent of those systems' inherent complexity, the complexity may affect software maintenance. This is basically related to the fact that it is still human beings who specify, design, and implement software. These may have different skills, domain background, languages, perceptions, goals (i.e., due to different strategic aims of involved parties) and change over time. Those different characteristics lead to the causes for 'normal accidents' according to Perrow (2011) because of decreased qualities regarding the stakeholders in software engineering:

- Manageability of design
- Ability to plan (i.e., also as result of limited manageability)
- Understanding
- Anticipation of interactions among components

A common countermeasure in software engineering to increase understanding of the system (i.e., manageability) is abstraction of complexity by use of modelling languages. Following M. Broy (2005) none of the methods for modelling and specification adequately addresses the needs of automotive software. He claims they do not feature a comprehensive model of all relevant aspects for such system's architecture. Those include user functionality, logical components, hardware architecture, software architecture and deployment. Although, most of the methods for modelling cover those at least partly, this is still an open field of research in particular for mixed criticality and interactive ICM.

Further, modelling can reduce the details of information that is necessary to understand and anticipate all relevant interactions including the inherited and non-obvious. The result is an overall system design which inhibits limitations for enabling verification in regard to reliability. This issue demands an improved architecture and design to foster those characteristics. However, architecture and design with focus set only to the components' interaction may not improve the characteristics regarding manageability as well as ability to plan and understand for 'local' logical complexity (i.e., inside the components) which was already proposed by Randell (1975) four decades ago and is still valid:

> *"The variety of undetected errors which could have been made in the design of a nontrivial software component is essentially infinite. Due to the complexity of the component, the relationship between any such error and its effect at run time may be very obscure."*

This demands a holistic (or 'integrated') design and development process that covers both component complexity as well as the complexity caused by components' interaction.

Additionally, the criticality of the ICM components varies. This is believed to be due to their temporal behaviour (i.e., real-time requirements, time and event triggered tasks) and impact to the user (i.e., cluster instrument, route navigation) and other systems (i.e., ADAS that rely on geographical map data). This implies the need to integrate heterogeneous functionality that varies by means of safety criticality. Although the separation into distinct but interdependent components provides good means for structuring ICM system at the software level, this does not solve the thereby assembled MCS. Burns and R. Davis (2013) classify a MCS to have two or more distinct levels of criticality and is integrated onto a common hardware platform. They further claim the prevention of interference between tasks from different components as a primary concern with the implementation of such MCS.

Potential threats of liability costs are strong arguments for achieving systems that consider adequate and verifiable reliability (or dependability/trustworthiness respectively). In reality, reliability often conflicts with economic demands. According to Leveson (2011, p5 f.), this is one reason for changing regulatory and public views of safety:

> *"[...] responsibility for safety is shifting from the individual to government. Individuals no longer have the ability to control the risks around them and are demanding that government assume greater responsibility for ensuring public safety through laws and various forms of oversight and regulation as companies struggle to balance safety risks with pressure to satisfy time-to-market and budgetary pressures."*

The applicable standard for safety-related software within the automotive industry is defined by ISO 26262 (2011), an adaption of the international standard for functional safety (IEC 61508, 2010). It defines requirements for the development and the verification of automotive electric/electronic (E/E) systems. This is basically related to the processes and methods applied therefore.

According to Stirgwolt (2013), the application of ISO 26262 strongly affects the development behaviour by defining process requirements for a shift from quality management system (QMS) as defined by ISO/TS 16949 (2009) to a safety-oriented work culture. He further describes this as a shift from control-oriented production (i.e. 'make the product right') to 'developing the right product' by defining safety requirements using a top-down approach and providing quantitative product reliability (i.e., failure in time (FIT) methodology).

The goal of the verification is to formally prove its conformance with its specification (Kreiker et al., 2011b, p26). This is based upon the perception that functional safety forms an integral part of system development and hence must be integrated into the system-rendering process from the start (Lederer and Ebert, 2008). This includes traceability of all requirements to the implementation level to enable full coverage of safety goals and their fulfilment. Furthermore, all system functions must be examined with the goal of determining the risks that could result from potential system failures. These have to be classified regarding the exposure (i.e., probability of occurrence), severity and controllability (Keul et al., 2013; Keul and Brock, 2013). With this classification, the critical functions are categorized into one of four different levels according to ASIL, numbered from A (low criticality) to D (high criticality). This means in quantitative terms a failure probability rate of 10^{-8} per driving hour for an ASIL D categorized function. Such a mapping of a function's criticality to assurance levels for software development is a common approach within various domains, like aeronautics, automation, nuclear, railway, and space (Machrouh et al., 2012). The automotive ISO 26262 features a relatively fine-grained criticality allocation policy at system level. The scope on functions implies the probability of the need to integrate differently categorized functionality (regarding ASIL criticality-levels) onto a common hardware platform, or a mix of integrity levels at software level (Ledinot et al., 2012). The result is a 'mixed-ASIL' system that contains functionality of different safety relevance/criticality, which adopts challenges of MCS such as, for instance, sharing common resources or functional dependencies between functions of different ASIL categorization (i.e., an ASIL n function may depend on an ASIL m function only as long as $n \leq m$) (Heling et al., 2012). However, a categorization into safety levels based upon estimated probability, anticipated severity and assumed controllability contains several weaknesses. Despite these, a safety standard like ISO 26262 is still viewed amongst the best of the available engineering standards and practices in use (Verhulst and Sputh, 2013).

Although ICMs were usually not categorized for ASIL levels in the past, future systems will contain safety-critical components. Hence, ICM systems evolve to mixed-ASIL systems that will contain both non-ASIL relevant and ASIL categorized components, such as, for instance, signalling the status of headlights, direction indicator, rear-view mirror, engaged gear, or integrated ADAS functionality[5] (Bräuer, 2011; Mehr, 2014). This makes ICM relevant for software validation, in particular for ISO 26262. However, it is not feasible to validate all components (or functionality) of a state-of-the-art ICM system without violating usual economic limits and developmental schedules. In parallel, current ICM already provides data connection to the Internet and web-based services. This allows future systems to stream media (e.g., music), dynamically update data (e.g., geographical maps) and functionality (i.e., 'apps') (Bolle, 2011). Such capabilities may affect the reliability of integrated critical components, not least because of an increasing attack surface that demands adequate countermeasures regarding security issues.

To summarize, it is assumed an automotive system cannot be completely free of hazards and related risks which increasingly does apply as well to ICM. Moreover, human interaction and systems built by human beings cannot be guaranteed to be absolutely free of operational errors and their consequences (ICAO, 2013). With focus on the mitigation of driver distraction, adequate reliability is a key feature of ICM. Due to the fact that this is a constructive aspect, it cannot be effectively added to an already-existing system or introduced at the final stage of development. This gains significance within the context of MCS. The system's architecture and the development process must reflect this from the ground up. Hence, the system's criticality and derived demand for reliability are significant design drivers for the software infrastructure and has to be considered throughout

[5]also driven by vehicular ad-hoc communication

development and maintenance. This demands adequate means to structure the overall system, including features to isolate particular components while still allowing for efficient inter-communication to utilize advantages of a common hardware platform.

2.2.6 User interface

From a user perspective, the UI is the actual point of contact with the ICM system. The design of the UI has to cover demands for an appealing front-end to foster a positive user experience (UX) (Buxton, 2007).

UX is non-tangible and depends on aspects such as the actual user's experiences, culture, and objectives. It goes even beyond usability, as it incorporates the user's emotion within the context of using the system. However, 'perceived usability' is one aspect of UX. This implies that the actual perception of the user regarding the UX may change over time and is unique to an individual. The UX is addressed by a product's design process (i.e., user centred design), while within this context, 'design' is not restricted to the product's 'visual characteristics'. Roto et al. (2011) provide a comprehensive overview on UX. From an economic viewpoint, the UX affects a product's user acceptance and hence may influence the customers purchasing decision. For this research, no detailed discussion regarding UX is provided. However, it is used to express the use of an appealing, attractive, enjoyable and fun to use UI and ICM system, which adequately fulfils demands stemming from the in-vehicle context.

Further, the UI has to reflect the system's functional purpose with regard to the safety-critical environment. The question of how ICM systems can efficiently exploit their functionality and the capabilities of current hardware platforms is of prime importance. This implies the UI takes on a very important role for ICM systems due to their nature of highly interactive systems. It has to combine the content of very heterogeneous content providers in terms of multi-sourced and mixed-criticality components to form a uniform, domain specific and appealing front-end to the occupants. The blending of different content has to operate in such a way that the 'user' does not take notice of the separation into components, as illustrated in Figure 2.4.

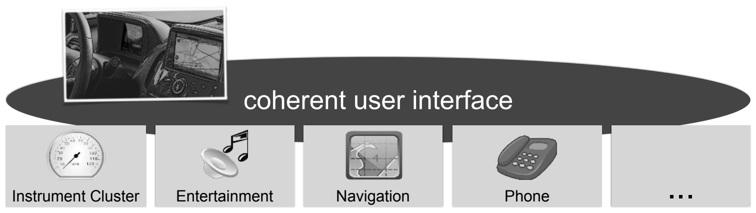

Figure 2.4: Automotive user interface

The design space for interaction is analysed by Kern and Schmidt (2009). They focus on the role of the 'driver'. However, in particular ICM is operated and used by different 'users' occupying different roles such as 'driver', 'front-seat passenger', or 'rear-seat passenger'. These different roles call for different variations of the UI with respect to usability, vehicular context, and legal framework. As an example, the driver need not be able to view a movie while the car is moving, while a rear-seat passenger must not change the seat-position for the driver. Schmidt et al. (2010) provide a comprehensive overview on challenges regarding the human computer interaction as related to automotive UI, including interaction with built-in information and entertainment systems as well as smart and autonomous function.

A significant part of the UI relies on visual information and feedback. By technical means, next-generation automotive UI consists of seven (or more) displays connected to the ICM system or respectively its head-unit to visualize content as depicted in Figure 2.5.

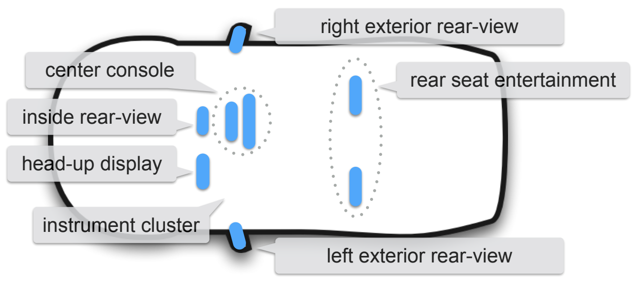

Figure 2.5: In-vehicle displays connected to ICM

Additionally, acoustic signals and voice synthesis (i.e., text to speech) is used to minimise visual distraction. Accordingly, multi-modal input capabilities are provided, including buttons, switches and knobs linked with both fixed and context-sensitive functionality (i.e., 'soft-buttons'), touch-displays/-areas and voice recognition. Recently, (touch-)displays have been becoming larger and incorporating an increasing amount of functionality beyond route navigation, exemplified with the 17-inch touch screen of the Tesla Model S (Rümelin and Butz, 2013). Generally, these multi-modal options for input to operate and use the system are available as well for all occupants simultaneously. The provided features that compose the UI must be assembled in such a way as to foster a positive UX because they have become an integral part of how the car appears to the driver, passengers, and prospective buyer. However, the ultimate aim is to provide a well-arranged visualisation and clear control to support an unobscured and intuitive usage of the system while minimising driver distraction (Hudelmaier, 2014; Pfleging et al., 2014). The demands for an ICM UI based upon such a wide range of available options for input and output in combination with derived requirements due to the automotive domain are not comparable to any CE device.

2.2.7 Connectivity

ICM systems already provide data links to other communication partners inside and outside the vehicle. In-vehicle communication includes Bluetooth and WiFi connections with mobile phones, Internet tablets, laptop computers, and portable media players to support use-cases like hands-free telephony, address synchronisation with the navigation system, or streaming/synchronisation of personal media content.

Connectivity to 'outside' infrastructure using cellular access networks allows next-generation ICM to update content during operation. Wireless broadband technologies of the third generation of mobile telecommunications technology (3G) (a.k.a. Universal Mobile Telecommunications System (UMTS)) and the succeeding Long Term Evolution (LTE) (3GPP, 2010) provide the enabling infrastructure. In particular, the latter contributes a promising current technology providing high data rates and low latencies even for high velocity vehicles (Araniti et al., 2013). These communication paths are used to incorporate satellite images into the navigation and guidance system and to provide access to other Internet services. Moreover, they facilitate the update of, e.g., geographical maps or real-time traffic information to improve existing functionality. Moreover, connectivity also enables the implementation of new use-cases, like for example intelligent personal assistants with natural language/conversational UI comparable to Apple 'Siri' (Aron, 2011) and real-time ride-sharing applications (Amey et al., 2011). Although such functionality is already available within the automotive context by use of mobile devices, an integration into the head-unit and composition with already existing functionality may help to reduce driver distraction and facilitate the use of vehicular information with the aim to improve the perceived UX. Further, systems may use this connectivity even to update exiting or install new functionality; this is discussed in more detail in Section 2.3.6.

Additionally, a continuous connection to the Internet using cellular networks allows relocating certain functionality of the vehicle into centralized infrastructure. This may have a positive effect on the maintainability as well as enabling the implementation of new use-cases that incorporate both data and functionality of the vehicle. In particular, the latter allows a shift of computational requirements into infrastructure of remote service providers, given network communication facilities that provide sufficient coverage, bandwidth and latency. A promising approach is described by Glaab et al. (2014a) and Glaab et al. (2014b), who describes an Automotive Service Delivery Platform based on Machine-to-Machine (M2M) communication. Off-loading functionality from the head-unit into 'the cloud' might be beneficial for a range of applications that currently are computed on-board. However, critical functionality with demands for low latency results independent of network coverage must reside physically within the vehicle. That means future ICM might incorporate software components that appear to be computed locally but are physically computed at a service provider (e.g., at the OEM).

A further benefit of the vehicle's connectivity to the Internet is the ability to provide gateway functionality to mobile devices using WiFi communication. These may benefit from improved connectivity due to the vehicle's outside antenna. Also, capabilities to communication peer-to-peer between in-vehicle mobile devices can be achieved using wireless networks which may also include communication with ICM functionality. This means the car acts as access point for non-automotive devices which introduces both new use-cases regarding passenger information and entertainment as well as new requirements for maintaining adequate robustness against improper use and thus prevent violation of the required dependability.

Additionally, future vehicles will provide capabilities to communicate with each other without the need for infrastructure using Vehicular Ad-Hoc Networks (VANETs), also referred to as Car2X[6] communication. Vehicles spontaneously form a network while travelling with other vehicles (V2V) or infrastructure (V2I). This type of communication is standardized with IEEE 802.11p as amendment to the IEEE 802.11 family for implementing wireless local area networks (IEEE 802.11p, 2010). It forms the base for the higher layer standard IEEE 1609 - Wireless Access for Vehicular Environments (WAVE) and Management and Security of WAVE (IEEE 1609, 2007). Uzcategui and Acosta-Marum (2009) provides an overview on the standards related to this type of vehicular peer-to-peer communication.

With the ability to directly communicate with other traffic participants, the vehicle's sensory capabilities reach a new level (Dar et al., 2010). This is basically related to active safety demands by enhancing the sensory capabilities beyond the limits of a single vehicle and independent of (the driver's) direct visual or audible perception of traffic hazards. However, it also enables provisioning of data to realise other use-cases such as improvement of transport efficiency, e.g., reducing travel time by avoiding traffic jams realized through enhancing route navigation with real-time traffic information. This may further include location-based advertisements regarding points of interest (POI), update of (local) maps, parking payments, and automatic tolling services. However, while these already affect ICM, they also facilitate new use-cases related to entertainment, such as multi-player gaming, multimedia content provisioning, multicast audio communication, in between vehicles without the need for infrastructure (Tonguz and Boban, 2010; Olariu, 2007; Bucciol et al., 2008; Amadeo et al., 2012). Moreover, beside data, also functionality (i.e., 'apps') can be provisioned on a peer-to-peer basis, e.g., a petrol station provides an app as part of the fill up by use of 'V2I' communication. Although VANETs and in particular V2V interaction rely on a sufficient number of 'communication partners' to achieve adequate usability of such use-cases, its introduction instantly affects the software of ICM to maintain adequate robustness against improper use in order to ensure the required dependability.

This research does not focus on communication technologies using wireless networks. However, next-generation ICM will have to cope with the effects that arise with dynamic update of data and functionality. Such support has to be reflected within the constructive aspects of such systems to adequately mitigate risks introduced by the new communication ports and which thereby increase attack surface for such systems.

The automotive use of additional and external computing platforms has been common practice for years. Current ICM provides basically the synchronisation of contacts, usage of the telephone stack and streaming of music using wireless communication with mobile

[6]The X is a placeholder for 'car' or 'infrastructure' (e.g., road-side units like traffic signals).

devices. Moreover, Portable Navigation Devices (PND) can be seen in many cars, either to have route navigation as the vehicle's ICM system (if available) does not provide any, or to 'update' the existing one by adding the PND and not using the built-in any more. As route navigation is also provided by current mobile phones accompanied with a media player, personal contacts, telephony and connectivity to wireless cellular networks, PNDs are often substituted, as the driver carries a mobile phone in any case. As the PNDs were designed for automotive use, the mobile phones and their UIs are usually not, which is related to potential issues regarding driver distraction – not considering legal constraints of the use of mobile phones while driving. The solution here is provided by concepts that use the vehicular UI to visualise content, render audio of, and interact with the mobile device. This means the functionality of the mobile device is blended to or even substitutes for the built-in, while the actual applications are computed on the mobile device. Exemplary technologies are Nokia 'Terminal Mode', 'Mirror Link', and more recent solutions like Google 'Android Auto' or Apple 'iOS Carplay'. The normal UI of the device is customized to automotive usage concepts when connected to the vehicle (e.g., reduced functionality, no text presented) to reduce driver distraction (Bose et al., 2010; Apple, 2014; Google, 2014). Such 'integration' of mobile devices provides benefits to the users by having their current custom device accessible while driving, using the vehicles UI capabilities (e.g., buttons on steering wheel, display integrated to centre console). However, this covers only a portion of an ICM system's set of use-cases which include, for example, the multi-display and multi-user management and control of automotive body functions. Although this might be partly solved through the provisioning of apps that cover the OEM or car model specifics, the usage concepts and criticalness of ICM systems does currently differ from the one of mobile phones to substitute the ICM system within the near future. Using the viewpoint of an OEM, it may be unlikely to give up and pass the 'control' over the car's centre console to a manufacturer of mobile devices. Such 'integration' can be seen more as add-on or temporary substitution of in-vehicle functionality. Nevertheless, the question as to what has to be done to replace the ICM by use of a mobile device is another current field of research (Hüger, 2011).

2.2.8 Hardware

To counter issues raised to limited thermal dissipation capabilities and availability of energy, MC architectures do provide adequate help. They have been common in the high-performance computing (HPC) sector for decades. In the more recent past, they have emerged and proved applicability also in server and desktop market segments, to solve the need for more computational power while improving energy efficiency. This is mainly driven by the fact that the increase of clock speeds to improve performance reached a physical barrier due to current limits in transistor technologies and the necessary thermal dissipation facilities.

Nowadays, this also applies to the domain of embedded systems where special purpose cores support the main processing unit to form a heterogeneous System-on-Chip (SoC)

MC architecture. But also homogeneous MC architectures are already available for different instruction set architectures (Levy and Conte, 2009). Those provide a number of advantages, some of the most prominent of which are outlined in (Smit et al., 2008) as follows:

Scalability is supported, as the architecture itself does not grow in complexity with future technologies. Only the number of provided computational cores increases, depending on the density of the integrated circuits and the size of the silicon. The computational power of MC CPUs scales direct proportional with the number of integrated cores, although the exploitation will suffer slightly due to necessary overhead.

Energy efficiency can be obtained by switching off unused cores to reduce the static power consumption. Also, the clock speed might be dynamically adapted to current needs for computation tasks which do not have to fulfil hard real-time constraints for determinism. Energy efficiency increases with reduced clock speeds, resulting in a lower thermal footprint. However, dynamic change of the provisioned computational resources decreases predictability of behaviour, which has impact on the testability and analysability.

Independency of computational tasks is realised by space division on MC architectures in contrast to the time division manner of multi-tasked software systems executing on single-core systems. That means that MC systems support parallel processing. In contrast, single-core systems have to perform jobs concurrently. Nevertheless, MC systems still have to compete for shared resources. Functional dependencies are realised by using an inter-core communication bus or network for exchange of information between the cores.

For ICM - as well as (mobile) CE devices - features such as the increased integration density, improved computational power, energy efficiency and functional features of current embedded SoC designs enable the creation of cost-efficient systems. However, to utilise the facilities of multiple cores, the software has to address issues raised through parallel execution (Sutter and Larus, 2005; Cantril and Bonwick, 2008; Holt et al., 2009). Eventually, MC CPUs were basically introduced to avoid the physical problem of increasing clock speeds to enhance computational power and not as a new feature to provide more parallelism which software developers will have to cope with. The new opportunities and implications with the use of MC hardware architectures are discussed in more detail in Section 4.2.3.

2.3 Requirements

System requirements correlate with the targeted software quality. A requirement is basically a statement that identifies a necessary quality or characteristic of a system. This means a requirement facilitates a formal specification of the systems quality. The ISO/IEC/IEEE standard for systems and software engineering vocabulary defines software quality as the capability of a software product to satisfy stated and implied needs when used under specified conditions (ISO 24765, 2010, p334). This does not include the process quality related to the development of the product, although the process quality may affect the product quality. The herein referred requirements are related to the software product quality and not the process quality of the development. However, this does not imply any limitations for application of process quality models to system development (i.e., 'use the right processes'). Moreover, the use of a stable software architecture that adequately reflects the characteristics of the targeted systems and the development environment during the development may support the process quality. But for this research the focus is set to constructive aspects of the system under development (i.e., 'develop the system right').

Based on the presented characteristics for ICM, a selection of significant requirements for compositing such systems are detailed in the following, also derived from relevant product qualities. Although the list of presented characteristics is not extensive, it highlights significant architectural triggers with respect to modularity due to multi-source development while considering adequate dependability, portability and adaptability. The characteristics are selected and justified based on the review of relevant literature. Moreover, those form the basis for evaluation of an adequate software structure (i.e. architecture) to address demands for next-generation ICM.

With the ISO standard for Software product Quality Requirements and Evaluation (SQuaRE) (ISO 25010, 2011; Wagner, 2013) a mature software quality model is available, derived of the former (ISO/IEC 9126, 2001). It defines high-level characteristics that capture major aspects of the system's quality which are refined by sub-characteristics to capture finer-grained quality aspects. Kalaimagal and Srinivasan (2008) compare different quality models derived from ISO 9126 within the context of CBSE and propose the one defined by Alvaro et al. (2005) to be most consistent and suited to the software component domain. They propose software-quality framework that relies on characteristics refined to sub-characteristics and further refined to quality attributes with the aim of efficiently evaluating the quality of the software components. Beside others, this model explicitly utilises sub-characteristics for self-containment and reusability. In further work (Alvaro et al., 2010), they also propose means for evaluation. However, a high-quality software component is not worth much when the remaining software infrastructure and components are of a lower grade. All parts of the system have to contribute to fulfil the required product quality. Hence, this research adapts the more general ISO 25010 model, with particular focus on the qualities that support integration of components into a whole (cf. Table 2.1; following Wagner (2013, p62)). Therefore, the focus is not limited to components, but includes the infrastructure that binds them to support the achievement of the particular quality attributes. When using quality models such as the one defined by ISO 25010, it has to be considered that neither the characteristics nor the sub-characteristics can be regarded free of interdependencies. For example, improved security has potential impact on the system's performance efficiency and usability.

In the following, a subset of the characteristics are detailed and related to this research. Therefore the quality model of ISO 25010 provides the foundation, although it has been

Characteristics	**Sub-Characteristics**
Functional Suitability	Functional completeness Functional correctness Functional appropriateness
Reliability	Maturity Availability Fault tolerance Recoverability
Performance Efficiency	**Time behaviour** **Resource utilisation** **Capacity**
Usability	Appropriateness Recognisability Lernability Operability User error protection User interface asesthetics Accessibilty
Maintainability	**Modularity** **Reusability** **Analysability** **Modifiability** **Testability**
Security	Confidentiality Integrity Non-repudiation Accountability Authenticity
Composability	**Compatibility** **Interoperability**
Portability	**Adaptability** Installability Replaceability

Table 2.1: Quality model following ISO 25010

adapted to this research. Originally, the standard defined a characteristic ‘compatibility’ with the derived sub-characteristics ‘co-existence’ and ‘interoperability’. This has been substituted with ‘Composability’ and the respective sub-characteristics as colour highlighted in Table 2.1. This corresponds to the herein-defined dependencies of characteristics according to Peter G Neumann (2004) and emphasises the component-based nature of ICM.

2.3.1 Composability

Composability is a central quality characteristic within the context of this research. It is usually not significant for a user, but its potential direct effect to other quality aspects pushes its significance through all phases of software development. That applies in particular to ICM systems, which are not designed and implemented monolithically from scratch. They are composed of different software components using concepts and principles of CBSE while, reflecting a heterogeneous set of MOC (cf. Section 2.2.3). The combination of component-based architecture and the need for resource efficient software implementations raises demands for approaches that allow a rapid development with low risks. This is related to the specification, design, and implementation, while considering both hard- and software (Martin, 2002).

Provided the correctness of a system's constituent parts (i.e., components) is given, it is necessary to emphasise the validity of the composition with the aim of a safe and reliable system. Therefore both functional and non-functional requirements (or properties) must be considered (Crnković et al., 2011a). They are fundamental to the provided quality of a software system and hence to software engineering and the related processes.

Chung and Prado Leite (2009) provide a comprehensive overview on the treatment of NFR. They manifest the need to consider both functional and non-functional characteristics with insufficient usable (i.e., not useful) functionality. Further, they provide definitions to clarify terms related to NFR, describe the differences to non-software systems, and reason about the lop-sided emphasis in functionality with the short history behind software engineering, the demand for swiftly producing running systems which fulfil the basic necessity, and the 'soft' nature of NFR. Hence, most attention is usually centred on notations and techniques for definition and implementation of functional requirements.

This demonstrates the need for a more thorough consideration of NFRs during system building and maintenance. Hence, NFRs must also be reflected when defining a common software infrastructure to improve the predictability of both the effort and success of component integration. Otherwise the components' temporal behaviour during runtime is rather based on coincidence than planning and predictability. Current software frameworks do not sufficiently foster the predictable combination of interdependent multi-sourced components (cf. Section 2.4).

Multimedia systems have to deliver the information in a predefined quality to the user. This includes the processing, the communication and the presentation of information and media. These operations are usually not performed by a single system component. Thus, considering all relevant components, the end-to-end path has to deliver a certain service quality. This concept is defined as quality of service (QoS) and refers to operational deadlines which have to be met (Steinmetz and Nahrstedt, 2004, p9 ff.). It applies in particular to continuous information such as, for example audio and video, but is also prerequisite to fulfilling certain protocols and interface standards of the relevant communication facilities and an adequate responsiveness to user interaction. Due to the ICM systems' applications' heterogeneity and interdependencies, the compliance to a certain QoS related to a given use-case depends on multiple applications. Further, it has to be considered that several use-cases are performed in parallel, utilising common applications, which again share common resources (e.g., the audio device for an incoming phone call while listening to music and speech synthesis of the navigation system's route guidance). Which use-case currently is of top priority may depend on the current state of the system, which is also affected by the environment (e.g. the vehicle's state). Hence, the applications have to reflect the actual priority in accordance to a given use-case and the current environmental conditions. However, this does not mean task priorities have to be adjusted dynamically during runtime. But the system's state and state transitions have to comply with required and predefined system behaviour, e.g., formalised by use of hierarchical finite state machine (FSM) (Wietzke and Tran, 2005, p215 ff.).

The temporal system behaviour is of significant importance, especially when considering a system as having failed when it is not able to execute the critical workload in time, although the hardware and software may be working properly from a functional viewpoint (Krishna and Y. Lee, 1991). This applies to embedded systems that are part of a physical process, i.e., cyber physical systems. Hence, such systems have to provide real-time behaviour. Following the definition of (R. I. Davis and Burns, 2011), a system is referred to as real-time when its correct behaviour depends not only on logically correctly performed operations, but also on the time at which the operations are performed and respectively when their results are available. Buttazzo (2011, p26) distinguishes real-time into three categories: hard, firm and soft. Concerning firm real-time, results available after a certain 'deadline' are useless for the system, but this does not cause any damage which is the case for hard real-time systems. Soft real-time implies such late results may have some utility to the system, but do cause quality degradation. For instance audio media may be affected by jitter, delay, distortion, or cracks and crunches when not meeting the end-to-end path's deadlines, which result in unintelligible route guidance, arduous phone calls, or low-quality music play back. Although such cracks and crunches do not directly affect the safety, other use-cases related to driving are safety critical and hence classified using an ASIL category, for example the visualisation of certain elements of the instrument cluster. High system load situations issued due to a non-critical use-case must not affect a critical one. This demands a predictable temporal resource reservation, whereas resources include CPU, memory, and I/O.

All this can be summarized into the need to render an integrated whole out of more or less distinct and heterogeneous components. To form such 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 and attributes to making the system useful and usable under stated conditions.

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, detailed in the following sections (see also Figure 2.6). Unfortunately, most quality models (a.k.a. classification schemes) for NFRs are inconsistent with each other (Chung and Prado Leite, 2009) and do not sufficiently recognize potential interactions between requirements. Although this means a single NFR cannot be addressed without affecting other required system qualities, in the following sections the focus is placed on composability as much as possible for reasons of comprehensibility.

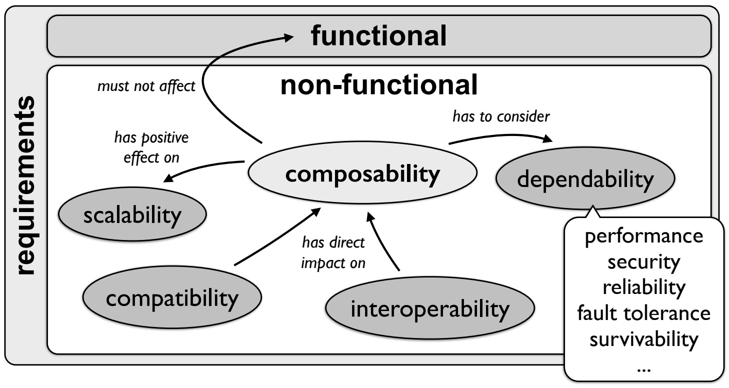

Figure 2.6: Qualities related to composability

Composability mainly depends on ‘compatibility’ and ‘interoperability’. Following Peter G Neumann (2004), compatibility implies the possible coexistence of different components (or entities) without adverse side effects. In contrast, interoperability addresses the ability of those different components to work constructively with one another. Both compatibility and interoperability are constructive aspects of software engineering and therefore have to be already considered during the system design phase. This also implies that a system that lacks compatibility and interoperability might not be able to be refactored for improvement of those qualities without significant efforts. Hence, consideration of relevant NFRs permeates an adequate architectural design (Chung and Prado Leite, 2009).

This results in the following system requirements to describe ICM software components more formally, which were defined as part of this research and represent the foundation of architecture for future ICM:

REQ-1 Components shall not interfere with each other during runtime unless it is explicitly specified.

This puts emphasis on the prevention of unwanted interference to mitigate adverse side effects. Although such interference most obviously is related to functional interfaces, also the NFR must be covered (e.g., temporal behaviour to fulfil a required QoS). On the contrary, wanted interference is basically related to functional dependencies. This means components still have to communicate in order to achieve the overall system’s desired functionality. However, the communication has to be restricted using well-defined functional interfaces (also referred to as ports).

REQ-2 Components shall provide defined functional ports to enable inter-component communication.

Although these are broad definitions that are applicable to a wide range of system domains and types, they represent fundamental aspects for future ICM systems. Being constructive aspects, they must be adequately considered throughout system development and in particular by the system's architecture.

Further, compatibility and interoperability can have a positive effect on the 'scalability' of a modular system. This is mainly related to the extendibility regarding the number of composed components. However, following Peter G Neumann (2004, p32), performance may degrade unpredictably with multiple conjoined components. Such a performance decrease may extend from linear, multiplicative, to exponential relative to the number of composed components. He explains that in practice the impact is even worse, caused by design or implementation flaws or indirect effects of the composition as of unrecognized dependencies. Such adverse side effects have to be mitigated. That is also a driver for well-defined interfaces. Given fulfilment of REQ-1 and REQ-2, the number of coexisting components does not necessarily result in an exponential growth of mutual interference. Hence, the system's 'extendibility' regarding the number and nature of composed components also benefits from improved composability (cf. Section 2.3.6).

Also, the 'reusability' of the system's components increases with well-defined interfaces. This means positive effects on the efficiency of the development process due to the possible reuse of already-existing or legacy components. This addresses demands for the capability of independent design and implementation of the multi-sourced developed software components. That can even be expanded to the build and deployment of those to support dynamic functionality (cf. Section 2.2.7).

2.3.2 Functional suitability

The system's functional suitability is directly related to the purpose of the system. It is usually the primary focus when defining a system, because it expresses whether it fits to the functional needs of the user. Hence it lays the foundation for use-cases descriptions, derived services and applications realised within or in collaboration of software components. However, providing adequate functionality is usually not directly related to the system's architecture, properties and constraints a system has to cope with, or its performance efficiency. Moreover, it must not be affected by parallel development or software reuse. Within a component-based software system, the major part of functionality is provided by the components and not the infrastructure. Nevertheless, the components rely on the infrastructure and hence may be dependent on the functional suitability of infrastructural software components.

Although for an actual system the functional suitability is of prime importance, this research does not focus on functionality. In the following it is assumed that adequate methods and technologies are applied to ensure that the right functionality is provided correctly by the targeted system. Nevertheless, given components that provide the right functionality and adhere to agreed functional interfaces does not necessarily imply the targeted system is usable, maintainable and composable. Therefore functional suitability does not mean a system is capable to serve the intended purpose well. NFRs must be considered with at least the same magnitude.

2.3.3 Usability

Usability can be seen as the second-best characteristic for a software product from a user's viewpoint. According to Alvaro et al. (2005) it can be subsumed for expressing the ability to be understood, learned, used, configured and executed under specified conditions. This may include the consideration of the users' capabilities as well as the environment. Referred to ICM systems that are operated within a safety-critical environment by untrained[7] users, demands for adequate usability gain significance. Beside the UI, this also is related to the system's superordinate usage concept that has to combine the diversity of provided functionality. That also includes the necessity to consider the variety and multi-modal abilities to interact with the system as well as multi-user operation and multi-display and multi-audio rendering (i.e., simultaneous visualisation and different displays and sound generation at different speaker locations). Hence, the complex component-based systems that have to provide an integral usage concept with multiple means for in-/output of the usability acquire a constructive aspect.

2.3.4 Performance efficiency

Efficiency expresses the relationship between the level of provided performance and the amount of resources used under stated conditions (Alvaro et al., 2005). That includes the system's response times and compliance against real-time constraints, while considering limited hard- and software[8] facilities available. Within the context of ICM, in particular the system startup time and responsiveness have to be considered. In particular the latter is of interest due to the interactive nature of those systems when operated within a safety-relevant environment (cf. driver distraction in Section 2.2.5). This has to be reflected in the system architecture and design. Albeit one common driver for CBSE is reuse (e.g., within another composition) which prerequisites the adaptation of a component to the actual infrastructure. However, such adaptation (or configuration) should be put into action most likely before runtime. This means either during deployment or even better: during 'compile-time'. Although optimisation might be deferred until it is necessary,

[7]It is rather unusual to thoroughly read a manual or get any other training before using an ICM system.

[8]Within the domain of embedded systems, it is not unusual that libraries and system APIs do only provide a reduced functionality (e.g., OpenGL ES for graphics).

premature design decisions may prevent the chance of achieving an efficient system or in actuality a system that provides the required efficiency after integration of all components.

2.3.5 Reliability

Reliability expresses the ability to maintain a defined level of performance under stated conditions for a stated period of time. This may include several of the other quality characteristics. That includes maturity, availability, fault tolerance and recoverability. Importance is emphasized for mission and life critical systems and has to be ensured for long runtime and untrained users. Following Wagner (2013, p11), failures in the functionality of the software to be considered differ from performance problems, for example. This makes reliability a well-defined quality characteristic for which statistical models and measurements exist.

Reliability is directly related to the often referred-to term ‘dependability’. According to the classification of Avižienis et al. (2004), the attributes for dependability basically correspond to the characteristics of reliability used for this research adopted from ISO 25010 (cf. (Wagner, 2013, p16); Section 2.2.5).

With regard to the operation within a safety-relevant environment and containing safety-critical functionality, the ICM system’s ‘dependability’ takes on a special role. This has to be considered throughout system development, maintenance and deployment of dynamic functionality. Qualities affected are the system’s security, reliability, fault tolerance, survivability and performance. However, although this list covers significant qualities, it is not extensive.

Due to the mixed-criticality of the components and the integration to a shared platform, a failure within an uncritical component potentially propagates to a critical one. This may be considered as adverse interference (covered by REQ-1). Nevertheless, such error propagation is enabled through the components’ connectors (or ports) which are necessary to achieve interoperability. Mehta et al. (2000) present a comprehensive classification framework and taxonomy of software connectors derived from analysis of existing component interaction. They define types for communication connectors, including ‘procedure call’, ‘event’, ‘data access’ and ‘stream’. These types are further developed by Manadhata and Wing (2005), who define metrics for computing ‘attackability’ as a cost-benefit ratio within the context of an ‘attack surface’ metric. Although the size of a system’s attack surface should not be confused with vulnerability or dependability, such a metric provides a quantitative method to express damage potential and effort. Within the context of this research, an offender does not necessarily cause ‘damage’. Although this is a valid trigger, damage is potentially attributed to (undiscovered) incompatibilities, error propagation, misinterpreted communication and transmission errors. They assume the order ‘event’ (1), ‘data access’ (2), ‘procedure call’ (3), ‘stream’ (4), with (1) low and (4) damage potential.

Hence, the ports (required with REQ-2) should be defined as narrowly as possible:

REQ-3 An interface port shall be reduced to the particular needs of the respective communication (based on type 'event' or 'data access').

This means an interface may implement a predefined protocol, ideally without the need of interpreting received data and making use of fixed message sizes (e.g., uniform event messages). This calls for detailed and explicit interface specification (at best at an early development stage) to reduce chances of misinterpretation during development or runtime. Interpretation may provide some flexibility, but also potentially decrease the efficiency – again of development and the system performance. Furthermore, such a non-abstracted interface may improve testability and assessment for critical components (e.g., when comparing a Simple Object Access Protocol (SOAP) based web service against fixed-size event messages to communicate between the route navigation system and the ASIL-B classified instrument cluster). Admittedly, this may be counterproductive for interoperable evolving systems. Nevertheless, a best practice protocol family for such communication is still the Hyper Text Transfer Protocol (HTTP) and its relatives that feature flexible message structures, interoperability and mature infrastructures. However, interpretation (i.e., 'decisions') should be made at best before runtime with respect to the particular needs of the respective communications and performance efficiency (cf. Section 2.3.4).

2.3.6 Portability

As the complexity of multi-sourced and in-parallel operated ICM components consistently rises, this brings with it the demand for a mature hardware abstraction layer integration to a single platform including the use of shared resources. This issue is addressed by use of an OS and its 'kernel'. The kernel forms the core of an OS and provides the necessary means to cope with the complexity, parallelisation and heterogeneity by use of a set of fundamental functionalities following the micro-kernel concepts of Liedtke (1995):

- Address spaces
- Threads and inter process communication (IPC)
- Unique identifiers

Upon those core concepts, further abstractions can and will be realized to provide, for instance, memory management and device drivers. This aims for adequate usability at the application level, which these system services are provided to by use of an operating system interface. This special form of an API defines the actual interface between the application level and the kernel, also referred to as user space and kernel space with focus set on the respective memory region. Depending on the actual OS and its implementation of the concepts and abstractions, this interface may vary and hence limit portability between different types of OSs, although they provide similar functionalities. Portability gains significance when considering reuse of already-existing components on different platforms or decisions to switch platform during development (e.g., because the target platform is not available when development starts, change of platform supplier due to technical or strategic/management decisions, etc.). Basically, portability may help to protect investment already made on software development.

An appropriate solution is the use of a standardised interface like the Operating System Interface (POSIX). It defines system API of mature quality, proved applicability, and provides adequate portability to facilitate a reuse for software components specified simultaneously by the IEEE and The Open Group (ISO/IEC/IEEE 9945, 2008; Walli, 1995). It is supported by various OSs[9], although they usually provide additional features beyond the defined standard. Nevertheless, POSIX support the applications portability at source code level. This still applies when considering that its scope does not cover areas like graphics interfaces, object and binary code portability, system configuration, and resource availability. It lays the foundation for enhancing the provided interface to cover those areas as well, i.e. by implementing a domain-specific software framework (cf. Section 2.4) that streamlines the provided services and capabilities to a special purpose. Hence, for this research, POSIX is used as the least common denominator for system implementation to foster adequate portability.

REQ-4 Either the utilized software framework or the components shall comply with POSIX.

This means at least the framework (if used) must be portable to another POSIX compliant OS, which implies the components are portable to that OS as well. In result, an implementation of components is independent of a particular (UNIX like) OS as long as it complies with the POSIX programmers interface.

Nevertheless, especially within the context of automotive embedded systems a great share of a vehicle's functionality is implemented without the utilisation of a POSIX compliant OS (e.g., OSEK-OS (ISO 17356-3:2005, 2005)), or even without an OS by running the code directly on a microcontroller hardware. This is attributed to the fulfilment of hard real-time requirements and the intent to decrease hardware costs. Although the ICM head-unit usually employs a POSIX compliant OS to counter the complexity, the OEMs intent to reduce the number ECUs may require an integration of 'non-POSIX' applications for next-generation ICM. This research does not explicitly address the integration of non-POSIX software components. Nonetheless, except of REQ-4 the herein specified characteristics of the target system are still valid. For the integration of such components an appropriate runtime environment must be available on the head-unit. An integration to a POSIX-conform framework and respective framework-conform software components might be achieved by use of a 'proxy-component' that adapts the non-POSIX software-component to the software framework.

A sub-characteristic of portability is adaptability. This is important, as for ICM systems, a relatively long development process elapses until they actual are built into a car to be

[9]Including, e.g., QNX Neutrino, Wind River VxWorks, GNU/Linux, Green Hills Integrity, SYSGO PikeOS, etc.

delivered to the customer. Although the nature and basic tasks of ICM are quite clear due to the automotive context, the users' demands are not. Admittedly, in all probability they all expect a set of current state-of-the-art functionality, including, e.g., instrument cluster, media player, hands-free telephone, satellite based route navigation. However, it is not possible to foresee the particular demands of all drivers and passengers for the whole product life cycle. The fulfilment of the users' demands correlates with user acceptance. Hence, the ability to adapt an ICM to changing demands resulting from divergent users who might also change their demands over time, gains importance for next-generation systems.

Even though existing ICM systems are configurable, they provide limited capabilities to adapt their function set to the actual needs and demands of their passengers. Up to now, the provided functionality is defined during the initial development process and fixed latest at SoP. This means the functionality is static during the whole product-life cycle. However, there are user demands that call for a change of that policy, not least because of the capabilities provided by devices within the CE domain (as detailed below), although usual product lifecycles are distinctly shorter compared to an automotive system. The evolution from systems with static functionality extends to ICM devices that can be updated and enriched regularly or on-demand and this is a major change in the automotive domain. The UX can be efficiently maintained through the entire vehicle lifetime. Hence, this evolution may have significant impact on the user interaction, as a current user is basically limited to configure functionality delivered with the vehicle (i.e., as initially shipped). Next-generation ICM enable their users to adapt a portion of the provided functional extent to personal needs or favourites. This creates a new dimension of customization.

To adapt the function set to individual needs and demands of a vehicle's passengers, ICM have to provide the capabilities to install/deploy applications dynamically (e.g., after-market). This puts emphasis on agreed upon inter-application communication facilities and functional interfaces to achieve the necessary interoperability. The relations to other quality characteristics have to be considered as well, depicted in Figure 2.7.

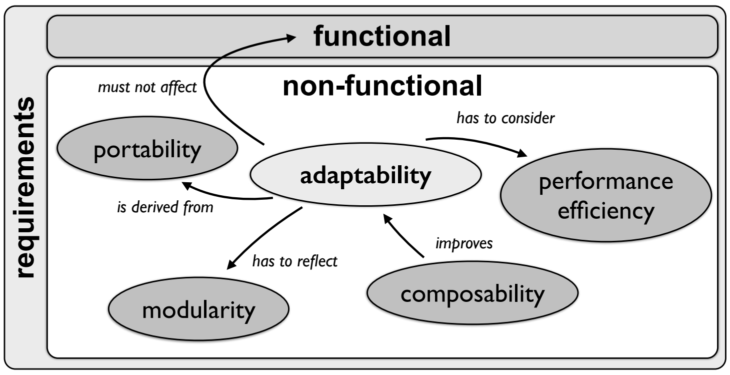

Figure 2.7: Qualities related to adaptability

Such functional flexibility is an already-established and approved approach by use of so-called ‘Application Stores’ and ‘Application Markets’ in the CE market domain of smartphones, Internet tablets, and desktop computers. They enable their users to install new functionality into their devices as well as remove unwanted applications (Pope and Muller, 2011). With increasing functionality introduced to ICM systems, the demand for innovative digital contents and user-individual functionality rises for such systems as well. This issue is already under research at several relevant OEMs and Tier-1 OEMs (Smethurst, 2010; Pflug and Frederick, 2011). Nevertheless, the software system’s complexity increases due to almost infinite permutation by recombination of different apps, as well as with the possibility of defining and implementing new aftermarket functionality which the initial system designer cannot foresee but has to consider to ensure a dependable system that fulfils the requirements of an ICM system. (Charette, 2011; Quain, 2011).

As detailed in Section 2.2.7, the current and next-generation systems are already connected to wireless networks. This provides the necessary channels for distribution (or deployment) of such dynamic function sets. Hence, the availability of wireless communication facilities enables after-market enhancement of ICM functionality even during operation. Again, such functional enhancement amplifies demands regarding composability, which puts emphasis on the software’s infrastructure and temporal predictability. Nevertheless, the need for adaptability is defined as follows:

REQ-5 The system shall provide capabilities for updating and installing components on user demand.

2.3.7 Maintainability

Maintainability characterises the effort needed to make specified modifications. The effect is not limited to the maintenance phase, as adequate maintainability is necessary right from the beginning of the development for project efficiency. This is of great significance in particular for multi-sourced software projects with various organisations involved that run for several years. That implies, for example, staff exchange, potential change requests at every project stage, multiple integration phases combined with fixed delivery dates due to concerted development and production with other vehicular subsystems.

As a software project benefits from a maintainable system, while maintainability is also a constructive aspect. Hence, considering it carefully at the beginning can significantly enhance the entire subsequent life cycle and reduce development costs. Maintainability is difficult to satisfy once the development is advanced if it had not been included in early planning.

Although essential for all software development projects using a developer’s and even more a manager’s viewpoint, this research does not explicitly focus on maintainability. Moreover, it is presumed that the respective sub-characteristics are considered according to good architecture, design and coding style to achieve maintainable ICM systems. In practice, for the maintenance until EOP, about 30% of the total number of human resources that were employed for the development of an ICM system are necessary, according to the experience made in recent projects Section 3.2.

2.3.8 Security

Security issues are gaining significance within the domain of ICM systems due to connectivity to cellular networks combined with integration of after-market functionality. Although characteristics such as reliability are related to integrity derived from security, this research does not cover security-related system qualities. However, the herein proposed architecture may affect security aspects, especially the structuring software components as detailed in Chapter 4. An investigation of isolation mechanisms and potential attack vectors within the context of ICM is an active field of research (Schnarz et al., 2013; Schnarz et al., 2014b; Schnarz et al., 2014a).

2.4 Software frameworks

An ICM system is composed of different interdependent software components. A quality model to verify the individual components' quality is useful as starting point when integrating multi-sourced parts. Components of adequate quality are necessary but not sufficient to achieve a composition of the required quality (i.e., satisfy stated and implied needs when used under specified conditions, according to ISO 24765 (2010, p334)).

The components are computed asynchronously which allows for parallelisation depending on the hardware platform's available facilities. This presumes the availability of common synchronisation and communication facilities. A common infrastructure eases the development and maintenance process, especially for projects with independent, parallel and multi-sourced development. It supports both the design phase and the implementation phase by domain-specific abstractions of the resources provided by underlying system layers to foster both the quality of the system under development and the productivity. Such an infrastructure is provided by use of a software framework in combination with an agreed upon guideline on how to use it. It provides capabilities to support the development and maintenance process by defining an infrastructure in combination with approved methods and techniques to achieve efficient inter-application communication and synchronisation. That can be realised by a domain-specific abstraction of the facilities of the underlying system layers, including both software and hardware.

There is a wealth of definitions for the term framework. Different target platforms come with their individual ones, possibly with their specific OS and HMI. Riehle and Gross (1998) provide the following definition:

> *"Frameworks are a central concept of large-scale object-oriented software development. They promise increased productivity, shorter development times, and higher quality of applications."*

Within the domain of automotive software systems, and in particular for ICM systems, Wietzke and Tran (2005, p10)[10] provides an applicable motivation taking into account the parallel development and division of labour:

> *"As soon as larger groups work on one or more projects, perhaps dispersed over different locations, rules and solutions must be defined, which have to be recorded within a framework."*

A software framework is recommended for parallel development, involving a large number of developers, probably working at different locations. A software framework is usually domain-specific, which applies in particular when targeting efficient embedded solutions such as ICM systems. It may define coding conventions, provides common communication and synchronisation facilities, rules for memory usage, and further hardware abstraction. Therefore, the framework implements a set of base functionality and provides an API using abstract base classes. The implementation relies on object-oriented principles using C++. It helps to ensure certain qualities of the targeted system. These include portability, maintainability, reusability, extensibility, and interoperability. A software framework predefines performance and efficiency of the system and further has impact on the efficiency of the development process. If not already part of the framework, the system's architecture and design is at least highly influenced by it. It constructs the foundation, which has to be elaborated for a particular target system. Thus, it is a critical component and should receive appropriate attention (Wietzke and Tran, 2005, p10 f.).

When considering the evolvement from decoupled architectures to an integrated one, the utilised frameworks receive increased significance. The former independent components have to share a common infrastructure, which requires rules and solutions to ensure interoperability and extensibility. This applies even more in regard to the complex functional behaviour and the required characteristics for safety, time and reliability due to the context of use (Di Natale and Sangiovanni-Vincentelli, 2010). An appropriate framework is able to provide the necessary support for the development process of highly complex and integrated ICM systems.

The use of a software framework supports the design phase by providing well-defined application interfaces, as well as the implementation and maintenance due to the availability of a set of mature base functionalities. A developer benefits from its use by being able to limit focus onto application logic that makes up the core of the functionality. Hence, the use of software frameworks aids the separation of different concerns. It is possible that

[10]Translation by A. Knirsch: Sobald größere Gruppen, eventuell sogar verteilt über Standorte, an einem oder mehreren Projekten arbeiten, müssen Regeln und Lösungen festgelegt werden, die im Framework niedergelegt werden.

a framework will also ease the reuse of mature and well-tested applications within future systems which rely on compatible frameworks.

Using a more applied developer's perspective, a software framework represents an infrastructural system component that supports both the assembly of application components (horizontal binding) and the deployment to a particular platform (vertical binding). Interoperability and compatibility must be considered for both horizontal and vertical binding. Using a software framework, the respective implementation parts that support the assembly and deployment are kept separate from the implementation of the applications, i.e., the use-cases. This means application components are 'glued' together using a domain-specific infrastructure that offers well-defined interfaces. The decomposition of the system into infrastructure and use-case implementation follows the paradigm of 'separation of concerns' and fosters a relatively independent in-parallel development and maintenance, given an agreed upon common interface. Such division into different development domains (cf. Figure 2.8) improves the comprehensibility of the implementation. This can have positive effects on the quality characteristic 'maintainability' and the derived modularity, reusability, analysability, modifiability and testability (cf. (ISO 25010, 2011); Section 2.3.7). This applies all the more when the domains evolve at a different pace.

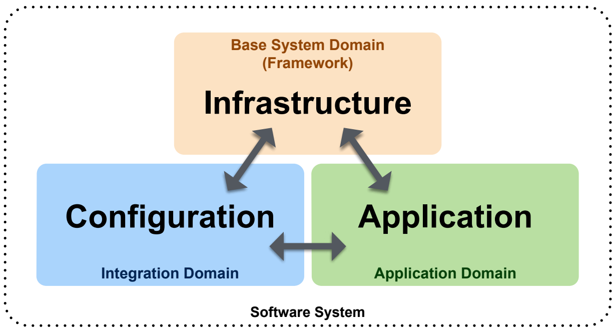

Figure 2.8: Separated software development domains

A comprehensive software framework represents the infrastructure that provides the base for configuring or adaption of application components to achieve a compatibility and interoperability.

2.4.1 AUTOSAR

An application of the idea of CBSE (cf. Section 2.2.3) to embedded and real-time systems is the Automotive Open System Architecture (AUTOSAR) (Bunzel, 2011). AUTOSAR is the outcome of a consortium consisting of international automobile manufacturers[11], suppliers and producers of development tools. It is a standardized architecture, development approach and API. Beside many other features, it also provides mature means for

[11]AUTOSAR members include the following core partners: BMW, Bosch, Continental, Daimler, Ford, General Motors, PSA Peugeot Citroen, Toyota, Volkswagen. Further 48 premium, 103 associate, 25 development partners and 12 attendees are affiliated to the AUTOSAR association.

partitioning different components (Mössinger, 2010). These support the shift from the 'one function per ECU' paradigm to more centralized architecture designs (Monot et al., 2012), also addressing issues of mixed criticality (cf. Section 2.2) (Burns and R. Davis, 2013). Following Jensen and Mascolo (2010), AUTOSAR provides key concepts to enable collaborative development of automotive software in large inter-company development groups, using a standardized design for integration and software reuse, implementation specification and inter-organisational communication.

The technical strategy of AUTOSAR covers collaboration at vehicle system level and component level. Whereas vehicle system level is related to interoperability for networked and interdependent ECUs, the component level covers compatible and interoperable intra-ECU software co-location. Both are facilitated by use of a 'virtual bus' called Run Time Environment (RTE) to provide the necessary means for intercommunication independent of the deployment of the software components (i.e., collocated within a common ECU or dispersed to distinct ECUs). Low-level basic software provides the necessary infrastructure, while the so-called AUTOSAR software components (SWC) implement the application logic. This corresponds to a separation of development domains as detailed in Section 2.4. However, the base system is divided into a service layer that relies on an ECU abstraction layer that in return relies on a hardware abstraction layer. The service layer acts as the interface for the application components to provide system services, memory services and communication services. Additionally, that integral structure AUTOSAR enables the 'standardised means to circumvent the standard' by use of a Complex Device Driver that enables (legacy) code to run collocated next to a AUTOSAR system. The downside is that the Complex Device Driver has no communication ports to the AUTOSAR base system's stack.

With the primary focus set on automotive control functions (Kindel and Friedrich, 2009, p62), it is rather static and does not provide the necessary capabilities to integrate software components (i.e., dynamic functional content) which originate from the CE domain or integrate mobile devices. This also has an effect on the available communication facilities between the AUTOSAR SWC. Although with release 4.0 (and substantially enhanced with release 4.1), its architecture and API support the use of MC platforms (cf. (AUTOSAR, 2013, p27 ff.) and (AUTOSAR, 2014, p88 ff.)), it is very restrictive regarding communication to foster strict partitioning. However, it focuses on mechanisms regarding the communication between applications running on different processor cores. A utilization of shared resources by applications deployed onto different cores is not supported (AUTOSAR, 2014, p112):

> *"AUTOSAR RESOURCES cannot be shared between TASKs/ISRs on different cores."*

The communication between different SWC is basically limited to stream-oriented socket communication. Although this preserves the degree of freedom of where to deploy a particular SWC to (i.e., collocated vs. different ECUs), it limits liberty regarding the structuring of SWCs. This is a disadvantage for systems consisting of many inter-dependent components, in particular for such that rely on high-bandwidth communication for multimedia such as, for instance, ICM. The related implications of partitioning are detailed in more depth in Chapter 4.

As of several OEMs and Tier-1 OEMs already relying on AUTOSAR-conform ECUs, future in-vehicle software systems may have to support at least compatible communication facilities (using the AUTOSAR RTE). This can be realised using concepts that feature co-existence of AUTOSAR implementation and other applications/OSs on a single hardware platform (Hergenhan and Heiser, 2008; Nett and J. Schneider, 2013; J. Schneider, 2013; Thiebaut, 2011), also supported by the specification for AUTOSAR OSs with requirement SWS_OS_00576 (AUTOSAR, 2014, p93):

> *"It shall be allowed to use only a subset of the cores available on a μC for the AUTOSAR system."*

Respective system environments that focus on partitioning of resources for isolation of software components are detailed in Chapter 4.

2.4.2 GENIVI

Another multi-organisation initiative consisting of a wide range of OEMs and their suppliers with the aim to foster collaborative in-vehicle software is the GENIVI[12] Alliance (GENIVI, 2014). It was announced in early 2009 with eight founding members[13]. As of this writing, over 180 members are collaborating within the non-profit organisation. In contrast to AUTOSAR, this one explicitly sets its focus on the domain of infotainment, adopting the idea of 'open-source' by their standards and relying on a Linux-based OS as base hardware abstraction. The proposed GENIVI IVI platform does not address highly competitive areas such as UI and logic that defines the UX.

Following Germonprez et al. (2013), such an open-source community enables partnerships, advance technology, and seeks opportunities even among competitors. Similar to the aims of AUTOSAR the objective is to collaborate on the development of the basics of software platforms to enable the individual organisations to differentiate among products based on usability and features instead of wasting time on platforms. Smethurst (2010) details the motivation for GENIVI and its operation principals 'Code', 'Platform' and 'Reference', following their approach of adopting from community (80%), adapting CE functionality (15%) and creating automotive specific GENIVI code (5%). Following this '80-15-5 assumption', several existing open source projects have been adopted and adapted to meet defined requirements for their free and open source software (FOSS) platform, including an improved 'D-Bus' for component intercommunication, 'Common API C++' IPC abstractions, Linux containers (LXC) for resource isolation and control, a layer management

[12]GENIVI is an artificial term that combines the pronunciation oft the city Geneva and IVI.

[13]GENIVI founding members (OEMs, tier-1 OEMs, silicon and OS vendor): BMW, PSA Peugeot Citroen, General Motors, Delphi, Magneti-Marelli, Visteon, Intel, Wind River.

to blend the visual output of different components, audio management, and many more (GENIVI Alliance, 2013; *LXC - Linux Containers* 2014). GENIVI basically provides a Linux-based foundation that is already compliant to a wide range of hardware platforms, ready to build an ICM system on top.

In 2013, BMW demonstrated the applicability of the proposed concept by starting production of vehicles with GENIVI-based head-units in several product lines, four years after the official start of the alliance (GENIVI Alliance, 2014).

A close relative to GENIVI is Tizen and its IVI profile architecture (the successor of MeeGo IVI) (Tizen Project, 2014b). It also follows the idea of FOSS and provides a fully-equipped OS driven by the requirements of the GENIVI alliance, and although incorporated into their platform, it does not fully share the same core components. Tizen also supports profiles for other device categories including mobile phone, wearable computer, and TV. Therefore it distinguishes between platform and application domain, providing software development Kits (SDK) dedicated for application development only. Thus it emphasises the UX, featuring both a web technology-based or native UI API[14] and comprehensive design guides (Tizen Project, 2014a; Tizen Project, 2014c).

With GENIVI, an extensive platform to build ICM systems is available, designed and maintained by OEMs and Tier-1 OEMs. The Linux kernel-based LXC provides ('container' based) means for temporal isolation of different components using a lightweight virtualisation using kernel namespaces with 'near native' performance (Xavier et al., 2013; Calarco and Casoni, 2013). However, without the need to employ multiple OS instances, the degree of isolation may not be strict enough to satisfy demands of MCS, as well as to cover different components' life cycles. Respective implications are detailed in Chapter 4, and regarding LXC in particular in Section 4.3.4.

2.4.3 Android

Within the recent past, the system platform Android continuously increased its presence within CE devices. This most significantly applies to mobile phones where it dominates the market with a share of 78.1% and nearly 793.6 million devices sold in 2013[15] (IDC, 2014). One success factor might be its open source license that enables different competing manufacturers to use it as software platform for their devices. A further positive aspect is the key feature that enables the user to install and update application components to customize the provisioned features of the particular device's life cycle. The platform is developed and maintained by Google, which also runs an application market as distribution point for new functionality that is comparable to the Apple App Store for their iOS platform.

[14] Tizen's web UI refers to HTML5 and JavaScript while the native UI is related to the integration of Qt.

[15] By comparison: Apple iOS had a share of 15.2% with 153.4 million sold devices.

The Android platform relies on a Linux kernel and enables the application developer to realize the functionality using a specialised Java runtime environment (Dalvik VM), eased due to availability of a SDK. It is further possible to run native (i.e. C/C++) code on the platform using the Native Development Kit (NDK) which can be connected to, for example, a Java-based UI. The particular functionality for one application is encapsulated within a 'sandbox'. This improves compatibility due to the prevention of adverse interference of different applications, but limits interoperability between different 'apps'.

Although Android does not fulfil the demands for an ICM platform regarding mixed criticality, its user acceptance within the CE domain, license model and capabilities for dynamic functionality also affects the roadmaps for next-generation ICM systems and evolve new research areas (Macario et al., 2009). Several OEMs and Tier-1 OEMs are currently running projects to integrate Android or are even already using Android as platform for their systems (Pflug and Frederick, 2011; Continental, 2014; Renault, 2014). As Android needs connectivity to the Internet and application market(s) to unfold its capabilities for customisation, it might be necessary to decouple it from automotive and critical functionality (e.g., running it side-by-side with another software platform). Respective features are detailed in Chapter 4.

2.4.4 OpenICM

OpenICM is an academic software framework developed and maintained at the ICM Labs of the Faculty of Computer Science at the University of Applied Sciences Darmstadt, freely available due to open-source licensing (ICM Labs, 2010). It implements the concepts described in (Wietzke and Tran, 2005) by use of the POSIX API (cf. Section 2.3.6). Basically, it supports the implementation of software components by providing a flexible and domain-specific infrastructure and guidelines for its use. OpenICM have been influenced by several industry-cooperation and inter-institutional projects for more than a decade. Further, it contains the experience made within several task forces at Tier-1 OEM to rescue projects in difficulties. Although its focus is primarily set on ICM, its concepts and implementation are also applicable for other target domains where the system under development has to address comparable issues.

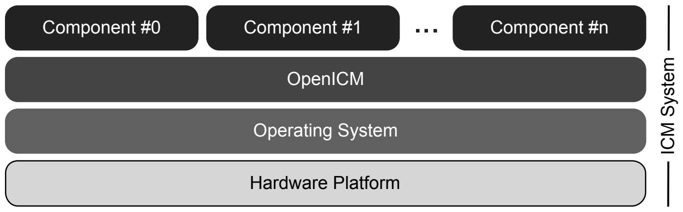

Figure 2.9: Abstraction layers using OpenICM

To ease the parallel and distributed development and maintenance, OpenICM fosters a loose coupling between software components (e.g., UI and application logic) in combination with corresponding rules and guidelines to achieve adequate interoperability and compatibility. This is realised by abstractions of an underlying OS layer and following the idea of reducing the overall complexity to interacting subsystems by use a policy of 'divide and conquer' as depicted in Figure 2.9. Within this context, interoperability basically relies on the availability of efficient and unambiguous means for communication. That includes inter-process communication (IPC) and synchronisation of concurrent or parallel executed components and their tasks respectively. That fundamentally implies primitives for mutual exclusions and signalling, message queues and shared memory for data exchange. The use of dynamic memory (heap) is avoided to prevent fragmentation of the limited memory and to allow the prediction of necessary memory during runtime. The predefined and limited means of memory usage can also have a positive impact on the system start-up time as detailed in (Knirsch, 2009), which is of high importance for automotive applications.

The implementation of OpenICM utilises object-orientated concepts and the programming language C++. Due to the use of the POSIX API, it is portable to different target platforms, while GNU/Linux and QNX Neutrino are the actively maintained ones. An essential part of OpenICM is the abstraction of functionality by use of a thread-based runtime model. The adaptation of the components and threads is realised by use of a static central configuration (i.e., not changeable during runtime), including memory layout and thread priorities. Threads are labelled with names in plaintext to ease system monitoring and analysis. For efficient inter-component communication, event-triggered queues for fixed sized messages in combination with a 'dispatcher', component are provided (Wietzke and Tran, 2005, p308). The message queues rely on a shared memory region using so-called 'component contexts' that realise three different priority levels per component. For complex data exchange, the component context can also contain so-called 'data container' in shared memory (Wietzke and Tran, 2005, p347). It is ensured that even before a particular component is started, no message addressed to this component can get lost. The dispatcher and the uniform message structure supports monitoring and analysis. Figure 2.10 illustrates the static relations between component contexts and application components that realise the inter-component communication.

Additionally, OpenICM provides abstractions for domain-specific hardware resources. This unifies, for example, the access to field bus systems like CAN or MOST, as well as supporting the development by use of those abstractions to realise the access to physical devices (e.g., FM radio, CD player, amplifier). This means that in addition to the components that realise the application logic, further infrastructure components are available to encapsulate, e.g., inter-component messaging or component monitoring. Such infrastructural 'base components' use the same architecture as the application components.

Within this rEsearch, OpenICM is used as portable OS abstraction layer to address ICM specific requirements for evaluation of the herein provided concepts. Therefore OpenICM were enhanced and modified (cf. Chapter 7).

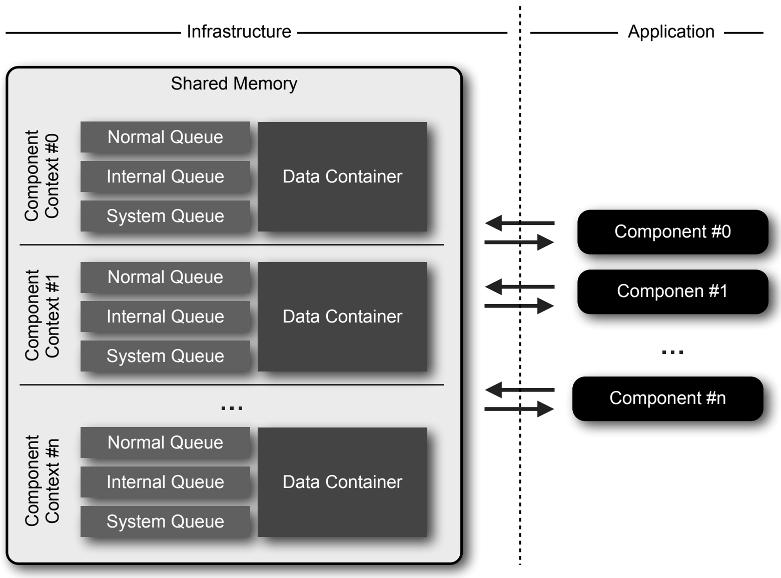

Figure 2.10: OpenICM communication infrastructure

2.5 Summary

ICM systems inherit characteristics from various domains, which include embedded, automotive, and multimedia. In detail, this places a demand for systems that are able to cope with real-time requirements, the operation in harsh environments, limited hardware resources, and need for adequate dependability due to their operation within a safety-relevant environment. In parallel, ICM evolved to a key differentiator with significant economic relevance to the OEM. This is a result of the feature-rich applications provided to the user, comparable to a mixture of functions as available in current cutting-edge consumer devices. Their development is realised in parallel by different independent parties and has to be coordinated with all other parts of the vehicle. Moreover, next-generation ICM has to incorporate means to support a dynamic function set to address the users' needs for individual functional adaptation. The platform's resources are not sufficient to run all software components in parallel without violation of usability, reliability and performance efficiency due to overcommitment of memory and processing power. A degradation of the perceived usability is the lesser evil: due to the inherent criticality of ICM, an inadequate software architecture has a potential effect on the occupants' and others' safety. This is emphasised by the rising importance of ICM, as it constitutes the integration platform for other vehicular software systems and realises connectivity beyond the vehicular boundaries.

The approaches applied for the development of past and current systems are not applicable for the rising integration density at the software level. This has resulted in a demand for design and implementation concepts that support the development of valid compositions, predictable both on functional and temporal behaviour while considering changes during the whole product lifecycle.

Therefore a set of essential requirements is presented (to be enhanced within the following sections) with the aim of considering the characteristics of ICM as well as improving the predictability of the success of the development. Here software integration is seen as a crucial quality for the successful rendering of future ICM systems.

In parallel to this research, industry-recognised, formulated and evolved standardised platforms for mixed criticality and infotainment systems that consider multi-sourced software have been developed for the automotive domain (cf. Section 2.4.1 and Section 2.4.2). They focus on particular solutions that offer only limited compatibility with application for a comprehensive ICM architecture. This is further detailed within the subsequent chapters.

3

Applied software composition for ICM

"The only source of knowledge is experience." (Albert Einstein)

ICM is a converging point for functional complexity and safety relevance. From this emanates the challenge of rendering integral, consistent, and dependable software systems while integrating heterogeneous components developed in parallel by independent parties with different interests. Under these constraints, the successful integration of automotive software components becomes a major challenge of the development process (Pretschner et al., 2007; Sangiovanni-Vincentelli and Di Natale, 2007). This implies the parts have to be composed into an integral and dependable system. Given the correctness of the constituent parts, it is necessary to emphasis the validity of the composition.

The significance of the integration issue increases with evolving functionality and complexity. The amount of software within a current upper market car has already reached the mark of 10^6 lines of code, where up to 70% can be addressed to future ICM systems (Charette, 2009; Smethurst, 2010). Although this metric is not appropriate to directly express the systems complexity, it provides a first impression to anticipate the challenges the Tier-1 OEMs will counter during the development process in general and the integration process in particular.

Additionally, the software has to be developed according to the schedule of all other parts of an automobile, as at SoP all vehicular components have to be ready for production. This requires a deterministic development process for all components. In this context, the term 'deterministic' is related to the duration of the development process, the efforts of the integration process, and fulfilment of the required functional and non-functional qualities of the targeted system. This again puts emphasis on the system development and composition which this chapter covers from a more practical viewpoint.

3.1 Development approach of automotive systems

ICM is the contribution of the automotive industry to the general trend for ubiquitous computing. Within this context, it has already gained significance in regard to the OEMs' economic success. While in the past, prospective customers focused primarily on vehicular characteristics like related fuel consumption, engine power or safety systems, now the 'FM radio's descendant' increasingly affects their purchase decision. Car magazines already compare vehicles based on their ICM systems. This is the logical result of the market penetration of CE devices that also foster ubiquitous computing, the availability of wireless access networks with high bandwidths, and the fact that younger drivers are already growing up with everyday computing. However, ICM is still a relatively young domain within the automotive development and manufacturing process. Two decades ago, built-in satellite navigation, speech recognition, in-vehicle touch displays, Internet media streaming and hands-free telephones within both lower and upper market cars were still dreams of future. Although some of these functionalities may also rely on specialised hardware devices, the usability is achieved by use of an increasing amount of software. The development and production of both is subject to processes that have evolved during the last decades and which have significant impact on the development approaches applied to ICM systems' software.

The development of ICM, and respectively its software, follows the rules of automotive manufacturing, and in particular those for supply production processes. Those are in accordance with Quality Management Systems (QMS) which gained significance in the 1940s in the USA and 1950s in Japan. In the evolution of QMS within the automotive industry, Deming played a significant role and shaped the term Total Quality Management (TQM). In (Deming, 2000) he describes the effect of improved quality as chain reaction as illustrated in Figure 3.1.

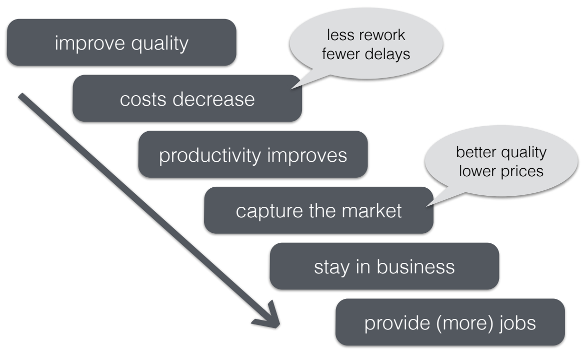

Figure 3.1: Chain reaction following improved quality according to Deming

In the 1980s, the evolved QMSs were standardised by multiple national norms while aligning with ISO/TS 16949 (2009) that replaced or at least covered the requirements of the different quality systems (QS) (Hoyle, 2005, p95 ff.). With this standard, a global management system fosters continuous improvement, defect prevention and waste in the supply chain by applying a process approach, commitment to quality by the management and emphasis on customer focus (Kartha, 2004). Hoyle (2000, p48 ff.) highlight differences between automotive QSs' requirements using ISO/TS 16949 as reference. All members of the International Automotive Task Force (IATF) insist that their suppliers develop and produce in accordance with this standard. The same applies to Japan Automotive Manufacturers Association (JAMA), which encourages their members' suppliers to be certified to the ISO/TS 16949. While a most of the industry's distinguished OEMs in Europe, United States and Japan are either members of IATF[1] or JAMA[2], it is the de facto standard that all producers of ICM are obliged to implement. This means, ICM is developed and produced in accordance to the same process standards as the other vehicular components.

In practice, at the start of a new project, the specifications and requirements are unclear. This prevents the applicability of a top-down approach based on a well-defined set of requirements. However, the project timeline is already defined, aligned with the development of the other vehicle's subsystems. This project environment requires a robust change management that supports the development (Schleuter et al., 2009). An approved and widely-applied approach is staged prototyping that relies on evolutionary A-, B-, C-, and D-models (a.k.a. prototypes) as illustrated by Schäuffele and Zurawka (2013, p240 f.) and Borgeest (2014, p308 ff.). Hoyle (2005, p438) describes a prototype program as part of the design and validation of quality automotive systems, to ensure whether it is the right design to meet the requirements defined up to that point (i.e., verify the design[3]) which are necessary for satisfying ISO/TS 16949 . Possible benefits are reduced time and costs in accordance with customer involvement (i.e., the OEM). Each development cycle has a duration of 3 to 12 months and ends with an acceptance test and an examination by the customer. Based on the examination of the early development models (i.e. A, B, C), the customer may refine the requirements and specifications for the next stage prototype following the principle of 'I don't know what I want, but I will know it when I see it' (IKWISI) (Borgeest, 2014, p300). That implies a truly integrated supply chain where the buyer (i.e., OEM or Tier-1 OEM) integrates suppliers' engineers and designers into the decision-making process as described by Tan et al. (2002).

Each integration stage relies on a complete development cycle, usually following a software development process model such as the V-model XT (VM-XT) (Rausch et al., 2005; Manfred Broy and Rausch, 2005). Following Maurer (2013) the application of the V-model for automotive systems' engineering leads to a significantly more structured way of development with OEMs and their suppliers. However, due to the staged process, the

[1]IATF members include the following OEMs: BMW, Chrysler, Daimler, Fiat, Ford, General Motors, PSA Peugeot Citroen, Renault, Volkswagen (*IATF - International Automotive Task Force Global Oversight* 2014).

[2]JAMA members include the following OEMs: Daihatsu, Honda, Isuzu, Mazda, Mitsubishi, Nissan, Suzuki, Toyota, and others (*JAMA - Japan Automobile Manufacturers Association, Inc.* 2014).

[3]Validation proves it is the right design, while verification proves the design is right (Hoyle, 2000, p264).

development follows a bottom-up approach driven by past experience with damage, rather than a preventative top-down approach (Winner, 2013). This puts emphasis on efficient quality systems and even more process models such as VM-XT to improve the projects' transparency by use of process documentation and thus provide uniform communication throughout all model phases (Rausch et al., 2005).

Each model phase may differ in provided functionality and integrated domains regarding the targeted system. The ICM system's model phases are usually aligned with the model phases of the vehicle's other subsystems. As an initial starting point, an A-model often reuses already available (i.e., legacy, previous series) systems to provide basic usability built into a housing that may not perfectly fit into the car's dashboard. Due to the limited available specifications, it is basically a very quick and pragmatic development, and differs significantly from the final product that is produced in very few numbers. Subsequent B- and C-models are more sophisticated preliminary evolutions of the product 'ready for production'. The B-model is characterised by a more experimental nature, whereas the C-model already represents a pre-series version. At the latest with the start of the requirements phase of the D-model, the OEM must define the complete list of features and qualities that the targeted system has to provide. The D-model represents the version ready for production for a four phase prototype and hence named 'product' instead of 'prototype' (cf. (Borgeest, 2014, p309 f.)). It is usually reviewed for a period of three months. However, this may vary as premium market segment OEMs test the D-model for up to 12 months to ensure high quality, whereas as been recently noted, others reduce it to less than a month. A delay in any phase has effect on the overall schedule, because the requirement phase for a particular model (except for A-model) depends on the outcome of review and examination of the preceding one, as illustrated in Figure 3.2.

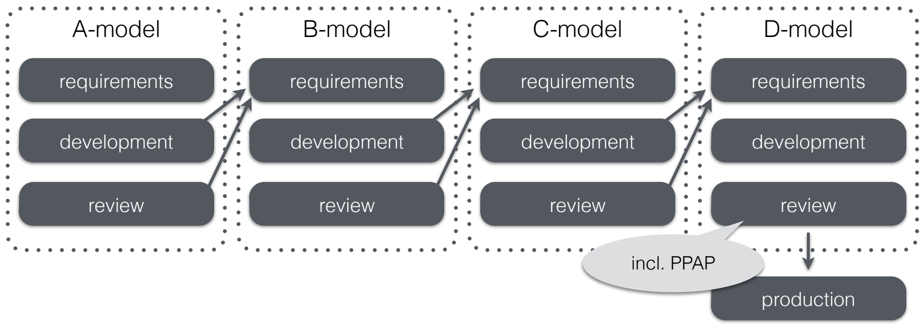

Figure 3.2: Sequential phases, following the prototype model of (Borgeest, 2014)

Nevertheless, review and examination of the respective phases for this evolutionary approach with initially unclear specifications is an important task. This is addressed by the applied QS by documenting results with an Initial Sample Inspection Report (ISIR). With a successfully tested D-model, the product gets approved for production with a Production Part Approval (PPAP) according to the regulations of ISO/TS 16949 (Borgeest, 2014, p310). Following this approach, the loop between process and product certification becomes connected: ISO/TS 16949 defines the process that requires suppliers to document the quality of their products with a PPAP. The aim is to certify their capability and competence (i.e., qualification) of producing products in accordance with specified requirements (Hoyle, 2005, p461). This proceeding is applied beyond ICM to all domains (i.e., subsystems) in automotive systems development. Moreover, it is derived from vehicular mechanical engineering to the E/E domains. Although this approach might appear inadequate or at least not state-of-the-art for software development, it has to be adopted due to the supply chain regulations of the OEMs. This view can also be transferred to the application of rich process models in software engineering. Rausch and Kuhrmann (2011) derived some principles and values for rich process models and came to the conclusion that there is a need for process frameworks to organize projects while allowing the use of project-specific methods and tools:

> *"This is the way rich process models can be of advantage: A comprehensive framework to host projects, but also giving freedom to the teams to work without having to pay much attention to the development process model itself."*

In summary, the development processes provide the environment for software engineering. The multi-organisation and multi-level supplier chain for automotive systems necessitate the management of the process quality, taking into account a hierarchy of national, international and organizational norms (aligned to ISO/TS 16949). The goal is to produce a valid design certified for conformity to defined needs. Both the applied regulations and applied approaches within the context of staged prototyping provide an integral process framework to organize the interaction in between the OEM, Tier-1 OEMs, and their suppliers ('Tier-2 OEMs'), distributed development teams and individual developers. However, regulations and process definition may prepare an adequate environment and a valid system design. Nevertheless, they can neither ensure that the qualities of the components are defined sufficiently nor that the system under development will in the end operate as required in relation to the defined qualities. From experience of real-world projects, this does in particular apply to software-intensive systems such as ICM as detailed in the following section.

3.2 Exemplary real-world projects

The previous chapter basically took an academic view of the problem domain of building software for ICM. However, the issues described correlate with recent engineering projects in industry. Empiric information on applied software engineering of ICM systems is essential to understand current practice (Segal, 2003; Segal et al., 2005) and must be considered for applicable solutions.

An inside view was granted through two past long-term inter-institutional research projects with two of the leading Tier-1 OEMs for ICM systems (Wünderlich, 2007). For further projects, only snapshots for the project state were made available, or they have not yet been completed. In the following, some basic information regarding those projects is provided to illustrate the current practice of software development for ICM beyond academia and in order to put emphasis on the relevance of this research. The project names (‘Alpha’ and ‘Beta’) have been changed so as to maintain confidentiality.

3.2.1 Project Alpha

Project Alpha did run from Q3/2007 until Q4/2009. The ICM system makes use of more than 1,000 tasks[4] at run-time, issued from a source base with 86,000 files (or 3.1 GByte), and produced by 235 software engineers at 13 different locations and affiliated to 9 different organisations. The ICM system had more than 2,000 open issues (i.e., bugs or suspect system observations) at SoP, whereas 8,600 where rejected (including not reproducible system behaviours) and 26,000 were processed, which adds up to 17,300 developer days for issue solving. The resulting software system was successfully deployed and is now operational in 45+ variants.

Even though this development project can be seen as a success, the applied process relied on a wasteful fix-up phase, resolved by a subsequent installed task force, consisting of 10 additional developers. That team was co-located to reduce communication loss and ease coordination to encounter the problem of parallel and independent development.

3.2.2 Project Beta

The ICM system of project Beta was under development for two years and was finished Q3/2013. It involved 220 software engineers at 32 different locations and affiliated to 28 different organisations. During the development, more than 22,000 issues were processed with ˜3,500 open issues at SoP. Sixty developers (27%) were assigned to the engineering of the HMI. Furthermore, decisions on the systems’ HMI technologies were discussed over a period of six months. Three independent design studies were ordered to reflect the increasing frequency of innovations throughout such a project. Specifications mainly relied on pictures, which may obfuscate the traceability to requirements and implementation. The overall development finished three months behind schedule, which affected the duration of the review phase. The resulting software system was successfully deployed and is in operation in 15 variants. However, the maturity of the initial production system was not satisfactory, which led to a stabilisation phase for vehicles that had already been delivered where daily updates were submitted to the OEM and transferred to the car dealers on a weekly base.

No dedicated integration team with adequate insight into component bindings, interdependencies and knowledge about relevant quality characteristics was installed throughout the development phase. This was not changed before the project was very far advanced,

[4]The term task represents a single computational context with, e.g., an individual stack and program counter. It is used interchangeably here with the terms thread and process.

again by a 'task-force' to rescue the project. Even at this stage, automated tests were useless because the system crashed within the first minutes after startup. However, 'monkey tests' (a.k.a. 'free-play testing') proved to be an efficient means to stabilise the system within a short time. The total cost for the development reached USD 80 million.

3.2.3 Other projects

Another long-term development project has been setup for several generations, with a two-year development cycle for one generation. About 200 developers affiliated with 12 organisations are involved. Although the produced variants have already been present on-road for three years, the software maintenance is still busy with stabilisation. Exemplary issues are system failures due to deadlocks caused due to misuse of inter-component synchronisation (e.g., relying on remote-method invocations) and locking mechanisms (e.g., recursive mutual exclusions), which are hard to reproduce and analyse.

A further project with deferred SoP to Q1/2015 has still more than 20,000 issues to solve in Q3/2014, which will require the Tier-1 OEM to process more than 500 issues per week, not considering a review phase for the production system. Even in very optimistic scenarios, an expensive fix-up and stabilisation phase is to be expected after deployment to market, degrading (or even eliminating) the Tier-1 OEM's economic success. Currently the total cost for the development is estimated to reach USD 100 million.

3.3 Interpretation of the reviewed projects

Based upon the information gathered through involvement in several multi-national development projects of ICM systems at different OEMs, it can be observed that the industry does not apply a comprehensive approach to achieve composable systems. Interpreting those numbers leads to the assumption that it is possible to create a state-of-the-art ICM system, consisting of heterogeneous functionalities from different vendors, but at the cost of unforeseeable efforts necessary to put these together. With an overall cost for the development of an ICM system of up to USD 100 million, the economic success of such a project is questionable, as the revenue is highly dependent on the quantities sold. These are difficult to predict, as they are related to multiple other aspects of the vehicle as well as the success of the marketing that affects customer acceptance.

3.3.1 Management

Although the reasons for the illustrated problems could be categorized as organisational or communication problems during the development process, a solution at that level is difficult due to incongruent interests of the involved parties. A supplier usually is only able to provide a competitive offer as long as the efforts necessary can be reused for multiple clients. A custom-made solution developed from scratch is not negotiable for economic reasons. A supplier of such a third-party component would not necessarily display much interest in easing the integration efforts if his own return of investment will suffer. Further, he may not have an interest in providing much detail about the internals of his part of the product in order to secure his intellectual property for economic reasons. A mutual consent between suppliers and integrator is most likely based upon the functional profile rather than distinct timing behaviour. This approach is derived from the distributed architecture utilised for previous generations of ICM systems, consisting of interconnected electronic component units (ECU), where the least common denominator was defined by the fieldbus system and the communication protocol. Those communication facilities were the shared resources of the architecture of such distributed ICM systems. The agreed specifications are good and necessary but seldom sufficient to ensure a predictable integration of the components.

With a consolidated architecture, integrating the functional components into one hardware unit (i.e., the head-unit), the components have to share additional resources, including computational power, memory, and I/O devices. Beside the functional dependencies, this introduces new dependencies regarding the temporal behaviour. The lack of details about the internals results in a lack of an overall understanding of the subsystems' interplay, and therefore decreases the predictability of the integration process which is countered by heuristic and ad hoc approaches (Sangiovanni-Vincentelli and Di Natale, 2007). This in particular has an effect on mixed-critical systems like ICM.

Based on the gathered experience, the applied business project management tools or techniques with focus on the development process do not provide sufficient assistance. They may help to mitigate effects by adding transparency and traceability but also obfuscate the root cause: insufficient addressing of composability throughout the constructive phases of the system development, i.e., the product or system under development.

This also applies to the use of coding standards. 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. Hence, coding standards are an important building block for development and maintenance of complex systems. Coding standards can be used for automated static checking of the components' sources for compliance and producing clear results to support the reliability of the system under development (Holzmann, 2013). This can have a positive effect on the system's dependability. However, this is of limited significance for the components' composability. This means, composability lies beyond non-architectural implementation rules defined through coding standards.

Due to problems encountered, delays during the projects have become the standard. Unfortunately, the development of ICM is coordinated with the other vehicular subsystems which makes a project delay unacceptable. Practice shows the review phase planned for the production ready system (i.e., D-model) becomes a buffer that shrinks continuously as the project advances. This results in immature systems delivered to the customers and an 'in-field' stabilisation phase to minimise claims for compensation by the OEM. An estimated increasing complexity for next-generation ICM may accentuate this situation.

To summarise, the called for integration of the system components is affected by management decisions with primary focus on the process rather than on the targeted system, organisational issues inherited due to parallel and distributed development, and insufficient requirements because of incongruent interests and lack of knowledge of the overall system details. Unfortunately, a solution at that level is not in sight due to the described strategic or economic reasons. Therefore, the subsequent development stages will be forced to cope with those circumstances which in the past were achievable more or less successfully as detailed with the exemplary real-world projects.

3.3.2 Domain knowledge

Beside organisational differences, the developing parties have heterogeneous expertise regarding ICM, also referred to as different domain knowledge. With a rising number of functionalities already available within CE devices, even software suppliers with no expertise in automotive systems are getting involved in ICM. For the herein exemplary projects described, this applies for features such as satellite navigation, voice recognition and speech synthesis.

One could argue that this may have a positive effect based on the multi-domain knowledge combined in ICM. On the contrary, this introduces 'language' barriers due to different use of terminology, lack of knowledge concerning implicit requirements, and inadequate assessment of requirements due to the particularities of automotive systems (e.g., low-power situations during motor ignition, coexistence with other components of different criticality, start-up behaviour).

3.3.3 Component interfaces

Although the component interfaces were defined within an early development phase, the definition mainly focused on the functionality to achieve interoperability. NFR were not addressed adequately. This may have an effect on the temporal behaviour of the overall system. 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 momentary state of the components and depending on the integrated components at that point in time. 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 temporal behaviour is not usually 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. Unfortunately, this may usually occur in proximity to SoP, which issues additional pressure on the overall project and the stakeholders respectively. Also with dynamic functionality, that may for instance be realised by after-market 'apps' and facilitated due to the head-unit's connectivity, the system's qualities are affected. This applies in particular to the software components' compatibility with effect on the systems performance efficiency, reliability and usability (cf. Table 1). This potentially causes unpredictable system behaviour throughout the whole product-lifecycle. A system behaviour that does not correspond with a user's learnt empirical knowledge features a disruptive influence on the user (Göschel, 2012). Put more simply: unpredictable system behaviour may cause driver distraction. Hence, unsatisfactory compatibility of software components poses a safety risk.

3.3.4 Integration

The process of integration requires the availability of the components to assemble, which puts the process of integration to the final stage of the development process. The root cause of the described problems is not the integration process. Unfortunately, this is the late development stage where the problems usually appear. Whether problems got unveiled or not still depends on the efforts and coverage of the integration tests. If the problems appear after the integration stage, such as after SoP or in use by the customer, the impact of those problems will be even more drastic due to the difficult maintenance situation within the automotive domain.

Within this context, the term integration is of particular importance. Generally speaking, integration means the assembly of separate components to form a (new) whole. Integration as used within the following is associated with the non-functional quality composability. It refers to correctness of a system that integrates different components. Within the domain of real-time systems, correctness includes the fulfilment of both functional and temporal requirements. Regarding ICM systems, erroneous behaviour might be observed by restarting components due to missed watchdog triggers, cracks in audio rendering of the phone, route navigation, or music component, inadequate responsiveness to user input, or elusive after effects. Whereas this enumeration is not exhaustive, the impact of these issues is amplified due to very difficult reproducibility.

Erroneous behaviour caused by an inadequate shared use of common hardware facilities also occurs at the very end of the development process during system integration. That includes the interference through concurrent use of computational power. Such erroneous behaviour may only be observed for high system load situations, but still within the range of real-world use-case scenarios. At that time, the separated work packages will have to be joined together and will have to prove their qualities by use of an integration test, while the development project has not much buffer time until SoP. This makes the integration a critical stage of the development process, even though the cause for problems can be deferred to the design phase and the system's architecture respectively. Kopetz and Obermaisser (2002) put criticism on inadequate interface specification and describes the problem as follows:

> *"A fundamental problem in the domain of distributed real-time systems relates to the constructive design of large systems out of independently developed prevalidated components. At the core of this problem is the provision of precise interface specifications, both in the value domain and in the temporal domain."*

When occupying an organisational or management viewpoint, the components to be integrated are supplied by internal or external development teams, providing software artefacts. With a more technical view, those artefacts might be delivered as source code, binary object files linked into the targeted system during compile time, binary library objects files linked into the targeted system during run-time, or binary files executed by the targeted system. Due to the interdependencies of the woven ICM software components, the integration efforts are affected by the overall system's complexity. Therefore it evinces good practice for the integrating Tier-1 OEM to install an integrator (usually represented by a team of senior developers and architects), which is assigned for the assembly of the targeted system's components. Such integration goes beyond just setting up the build-environment correctly and placing the delivered components in the right place. It presumes explicit expertise about the components' qualities and features so as to be able to actively and proactively detect issues and suspect interoperation as early as possible. The staged development model for automotive systems put even more emphasis on integration, because there are four (or more) models to be integrated and reviewed. Instead of a non-deterministic 'big-bang' integration at the end of several years of development, preliminary versions of the components have to pass the integration process for the A-, B-, and C-models. But this does not imply these integration processes are free of conflicts. Hence, there is still valid need for increased determinism for those staged integration processes. However, the final integration for the D-model is still critical, because this stage firstly has to cover all requirements.

Good practice showed the installation of a continuous integration approach, to have a running (and testable) system available right from the early development stage. This still cannot substitute a review phase for the D-model, but may uncover principle problems related to compatibility and interoperability early. This especially applies to such multi-source development with potentially imprecise interface specifications.

Independent of the applied quality system or process model, the efficiency of an evolutionary prototype approach improves with reuse of substantial components of the product under development. Such a reuse prerequisites a stable software infrastructure that scales well with the introduced changes due to the sequential requirements phases (cf. Figure 3.2). This means while a QS following ISO/TS 16949 in combination with a process model such as VM-XT provides a framework to support the development processes and communication in between developing parties, a software framework may support the multi-sourced development and integration with focus on the product under development by providing the necessary infrastructure.

3.4 Summary

Understanding current practice in large-scale industry projects is important to support research of applicable solutions considering the requirements of next-generation ICM. Therefore, empiric information was gathered and discussed.

The integration of ICM systems' components is a process that combines heterogeneous software parts to form an integral system. The validity of the resulting composition depends on the functional and temporal correctness of the interacting software parts as well as the integrated overall system. An isolated pre-validation of components has limited conclusiveness on a correct functional and even less on the temporal behaviour within the targeted system.

However, the success of the integration is dependent on the provided components and the utilized infrastructure rather than on the integration itself. This implies a high risk due to multi-sourced components and inadequate or premature software frameworks and affects predictability on the validity of the result of integration process.

The different domain backgrounds of the involved parties, communication barriers, and incongruent interests affect the development process negatively, although this is not the root cause. It is rather the increasing complexity caused by additional functionalities and an unpredictable temporal behaviour of the targeted software system. The system's 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 to fulfil the required interoperability. This is necessary but not sufficient to achieve composable systems. The components compatibility (i.e., coexistence without adverse interference) is neither addressed adequately nor supported by the system's infrastructure. With an increasing extent of the system under development, such an issue treatment can be expected to be less successful. This means the current integration approach of ICM systems with its downstream[5] fix-up phase does not scale with the anticipated rise of both the functional features and the independent developing parties involved.

While next-generation ICM will rely on additional and even dynamic functionality, the software architecture's complexity is ever-increasing. In result, this puts emphasis on REQ-1 (cf. Section 2.3.1) to 'enforce compatibility'. This demands new concepts to

[5]i.e., during the phases assigned to the D-model, at 'the end of the development', or even after SoP

support the development and assembly of the interdependent software components. In particular, this applies to high-system load situations during operation. A comprehensive approach in combination with an appropriate architecture is needed. The aim is to support the development and integration of fine-grained and heterogeneous software components into a dependable and comprehensive whole while considering a predictable temporal behaviour. Or put more simply: reduce complexity by use of composable components.

4

Structuring software components

"At the heart of every well-engineered software system is a good software architecture." (Medvidovic and Taylor, 2010)

Essential qualities to achieve a composable system are interoperability and compatibility. These are constructive aspects which have to be considered through all development phases, and in particular by the software architecture. The latter defines how the components are structured. This also includes arrangement into abstraction layers and groups of components (i.e., agglomerates), definition of communication paths that represent vertical and horizontal bindings in between the components and to underlying system services. Following Medvidovic and Taylor (2010), such constructive aspects permeate all major facets of a software system. They define its architecture as set of principal design decisions made during its development and any subsequent evolution.

Within this context, the prevention of interference between tasks allocated to different components is relevant, in particular for implementation of MCS (Burns and R. Davis, 2013). Although this behaviour is relevant for all systems that host several different applications, it gains significance for such that are operated within a safety-relevant environment or contain mixed criticality applications or both (such as ICM). To address a system's and its component's criticality the overall system complexity has to be controlled to avoid faults. Following Sha (2001) this can be achieved by simplicity, as with higher complexity it is more difficult to specify, design, develop and verify, as he relates to avionic systems and respective standards of the avionic industry. Basically, this means partitioning a complex system into 'understandable' (i.e. less complex) parts can help to address the overall system's criticality. However, simplicity may contradict with performance. This implies practical limits and trade-offs to partitioning and containment respectively, not further detailed here. However, the concepts applied in avionics systems can be transferred also to automotive and in particular to ICM systems of mixed criticality. With appliance component containment, the components take care of the functional aspects while the containers take care of NFRs by preventing unwanted interference. This can be classified as an exogenous management of NFR following the scheme proposed by Crnković et al. (2011b).

The implementation of such an architecture can be supported by use of a software framework (cf. Section 2.4) to separate the infrastructure from the actual functionality provided by the structured components. Such an architecture and the framework respectively have to fulfil the requirements defined in Section 2.3.

In the following, a concept to structure components is discussed, focusing on efficient inter-component communication while preventing adverse interference of components (due to shared use of resources).

4.1 Inter-component communication

The temporal behaviour and performance of an integrated software system is highly dependent on efficient messaging and memory access. This is also proposed by Kopetz and Obermaisser (2002) with their third principle of composability: 'performability of the communication system'. This is especially applicable for cost-efficient solutions that do not allow an extensive communication overhead introduced by an abstraction layer to support a loose coupling. This basically applies to the interdependent software components of an ICM system. They also require efficient, clear and unambiguous inter-component communication. This is fundamental to achieving the required interoperability despite the segmentation of the overall system's functionality into distinct components.

Interoperability relies on inter-component communication and hence implies data flow. Such communication includes IPC of concurrent and parallel executed components and their subsequent tasks. Related POSIX primitives are mutual exclusions, semaphores, and condition variables. They provide mature intra-OS mechanisms to achieve synchronisation between different components' tasks, whereas message queues and shared memory regions enable data exchange. For more complex data transfer between components, the use of shared memory remains the method of choice in efficient environments opposed to more loosely coupled technologies, such as socket stream base communication that requires appropriate interpretation at the receiver, serialization and deserialization. This especially applies to data flow in between mixed criticality components, i.e., respectively from low to high critical components and tasks. This creates an attack surface because a high-critical component has to cope with potentially unreliable data which may affect its temporal behaviour[1]. The latter applies also to vice versa data flows for synchronous communication when the more critical component is delayed. Following Sha (2009), such behaviour where a high-critical component depends on a less critical one is also referred to as 'dependency inversion'.

[1]Other attacks 'by intention' due to an offending component or due to other vulnerabilities are not detailed here, but are a current field of research that is relevant in particular for systems supporting 'dynamic functionality' (cf. (Schnarz et al., 2013; Schnarz et al., 2014b)).

With rising complexity, the manageability of the components' dependencies decreases. Furthermore, with depending components, this complexity becomes nonlinear (cf. (Leveson, 2011, p4)). The reasons are potentially indirect relationships between cause and effect. In some measure the relationship is not obvious because the structural decomposition is inconsistent with the functional decomposition: 'decompositional complexity'. Such adverse behaviour is usually assisted by complex intercommunication such as remote function calls/method invocation or complex data protocols that allow loose coupling but require extensive interpretation at the receiver. Interpretation at runtime degrades efficiency. Hence, in particular for inter-component communication, a simple event-based communication utilising fixed-size messages supports the manageability (i.e., maintainability and traceability) and reduces the related complexity. This is backed by the experience made within practical testing as part of the projects detailed in Chapter 3, where complex communication repeatedly issued locked components with opaque relationships between cause and effect.

Beside the components that implement the application logic, there may exist infrastructure components which affect communication-related functionalities such as central message dispatching and system monitoring (i.e., message tracing). The distinct components are connected by defined 'communication channels' connecting the components-defined 'communication ports'. These are accessible by other components through an unambiguous abstraction. This is part of the infrastructure code (i.e., the software framework).

Such infrastructural facilities are provided with the event-based message system of OpenICM (cf. Section 2.4.4). The provided message queue abstraction fosters flexibility in combination with a deterministic behaviour and efficiency due to the use of shared memory (Wietzke and Tran, 2005, p197 ff.). For this research, the communication means provided by OpenICM are utilised because they rely on the standardised POSIX API and hence are portable to many platforms and operating systems. Further, the open-source licensing allows for reproducing the herein provided practical evaluation of concepts. However, this prerequisites the availability of shared memory accessible from the communicating components. One the one hand, this may limit the applicability of certain technologies for partitioning; on the other hand, the use of shared memory provides very efficient data exchange between components. The latter is in particular relevant for multimedia content that requires high bandwidths and low latency to achieve the required QoS.

4.2 Component partitioning

When beginning to structure a system with predefined functionality where the architect possesses all degrees of freedom and comprehensive knowledge about all details of the system (no predefined architectural restrictions, COTS software or legacy code to consider, no 'black box' deliverables to be integrated), it is suitable to decompose the functionality into fine-grained separate tasks using software engineering technologies to transfer abstract functionality to implementation level. In this context, a task is defined as the smallest execution item. Based on certain constraints, they can be partitioned into groups of tasks which make up the components of the system. The constraints can be derived from the attributes of the individual tasks in association with their interdependencies. An appropriate way is to differentiate the tasks and the related implementation (if already available) based on their internal architecture (i.e., dependencies), their vendor (especially for COTS parts), and their lifecycle or ability to be reused. Considering these factors will lead to a generally coarser-grained segmentation (i.e., fewer components) in comparison to a design with all degrees of freedom and detailed insight.

Internal dependencies can be rated by use of design structure matrices (DSM). Those reflect the amount, frequency and direction of information exchanged between tasks (Sangal et al., 2005). Such dynamic information most certainly will change during system operation. Therefore, an adequate system profiling is advisable to achieve expressive and effective results.

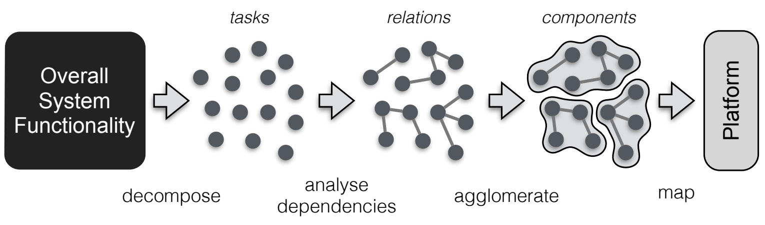

Figure 4.1: Functional decomposition with all degrees of freedom

In the 'real world', the decomposition and development commonly does not look like the process depicted in Figure 4.1. To manage the complexity of the overall system, functionality can be described using use-cases. Accompanied by further high-level design information, the implementation can be separated into distinct chunks to be provided by suppliers. The task of the integrator is to unite the resulting chunks and arrange these onto the available platform as depicted in Figure 4.2. A software framework may provide support (cf. Section 2.4). Architectural design concepts like use-cases, components and modules, for proper abstraction and modelling a system, help to cope with complexities. But the resulting design artefacts still have to be transferred to the level of implementation utilising the API of the OS and programming languages and comply with software frameworks.

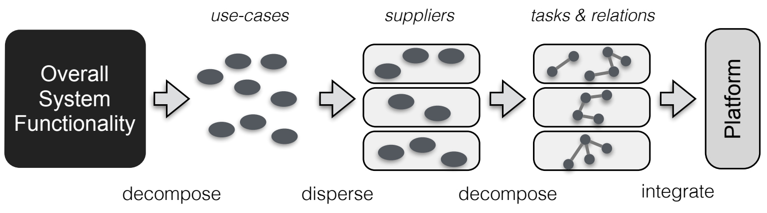

Figure 4.2: Functional decomposition using use-cases for multi-source development

Using distributed (also referred to as 'federated') hardware architectures for previous generations of ICM systems had the advantage of clearly separated software components that correspond with the decomposition of the use-cases, enforced through distinct platforms within a federated architecture. With consolidated architectures, these platforms' hardware boundaries were abandoned (cf. Figure 4.3), providing both advantages and disadvantages. From the viewpoint of the software, the most significant of the advantages are improved communication abilities; of the disadvantages are unpredictable temporal behaviour and complex temporal dependencies. These are all caused by shared use of common resources. Scheduling algorithms and the implementation of these in terms of schedulers enable configurable reservation of certain resources with the aim of improving predictability of runtime-behaviour despite random occurrence of event-triggered tasks and independently developed components that have not been harmonised with a system global scheduling scheme (cf. Section 4.2.1). This means a system without scheduling and implementing a 'run-loop approach' is basically more predictable but less manageable in terms of integration of multi-sourced software components that address different use-cases and has to fulfil both event- and time-triggered requirements. In all practicality, the scheduler supports the integration of heterogeneous functionality while featuring adequate predictability. Prerequisite for such scheduling is the abstraction into distinct execution units. In the following, the approved concept of execution units and a selection of solutions to create segregated 'domains' for software components are discussed.

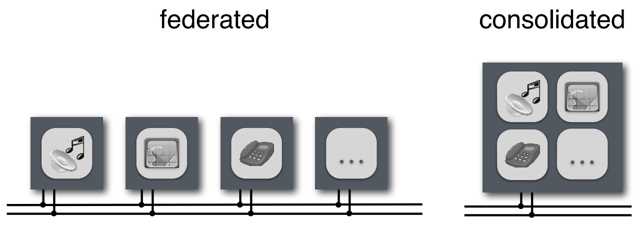

Figure 4.3: Federated vs. consolidated system architecture

4.2.1 Scheduling

Before describing approaches to effectively partition a system's functionality, an OS's native features for temporal control of execution units during runtime are detailed. The most fundamental concept here is scheduling. Scheduling in combination with disjoint address spaces is an abstraction of a hardware platform's resources into a task-based model (cf. Section 2.3.6). The task-model also abstracts the functionality into execution units. That means an application is executed by use of a set of parallel or concurrent threads. A process represents the frame for one or more threads that share a common memory region. This leads to the definition of a task, which is the hyponym for addressing an executional unit consisting of several operations represented by a thread at the implementation level.

For the following, threads are used as an execution unit. They may have access to both shared and private memory regions. They are arranged as the performing entities of processes, whereas one process contains at least one thread. Each software component may utilise a number of processes and hence threads to fulfil its respective job. An application may consist of one or more software components.

A 'scheduler' implements scheduling and usually is a core component of an OS kernel. Generally speaking, a scheduler assigns (i.e., 'schedules') a given resource for a certain timespan to an execution unit (i.e., a thread). Within this context, a resource is basically a hardware device or an abstraction of such. This may also include virtual devices provided by an underlying system layer.

Within this context, the CPU is the most central resource within a computer system. It provides computational power to the software layer. The OS's 'thread-scheduler' basically assigns the computational power of the available CPU cores to the threads ready for computation. In the following it is assumed that a thread-scheduler assigns the hardware platforms' processing capacity to the applications' threads independent of their belongingness to processes (i.e., independent of any semantic relations in between the threads).

The challenge here is to determine which thread has to be assigned next for computation and on which core for MC hardware platforms). Therefore the thread-scheduler uses a 'scheduling-policy' that follows an algorithm that defines which thread is scheduled when and where. The scheduling-policy may consider both static characteristics and runtime behaviour of a given thread. This may include thread priorities, CPU utilisation in the past, anticipated CPU utilisation in the future, or the thread's deadline. Scheduling-policies are classified into (a) pre-emptive and (b) cooperative. For (a), the scheduler grants a thread access to the computational resource for only a limited timespan[2] (a 'timeslice' a.k.a. 'quantum') or until a thread of higher priority gets ready for computation, while for (b) the thread does not return from computation before it is finished, it releases the CPU 'at own will', or until a thread of higher priority gets ready for computation. In pre-emptive systems, the program locality suffers when cache misses increase due to cache invalidations as result of iterating context switches (Buttazzo, 2011, p15). However, it fosters the system's reactivity by ensuring that a higher priority task interrupts low priority ones, as well as granting computation time to equal prioritized threads in turn. Hence, different scheduling policies are utilized in regard to their field of operation. For example, the demands for scheduling in a server system are optimised for resource utilisation (i.e., using large time-slices to reduce overhead introduced with context switches) while a desktop system might be optimised for responsiveness. Embedded systems are usually part of a physical overall system ('cyber-physical') where the threads' computation usually has to meet strict deadlines to fulfil requirements regarding real-time behaviour. Common real-time scheduling policies are round robin (RR), first in first out (FIFO) (Kerrisk, 2010; Love, 2010) and earliest deadline first (EDF) (a.k.a. Horn's algorithm (Horn, 1974; Buttazzo, 2011)). Whereas RR supports a fully pre-emptive scheduling where threads are arranged in priority based waiting queues, FIFO follows a cooperative scheduling as long as no higher prioritised thread becomes ready. This means that while for RR two threads

[2]time division multiplexing of the resource

of the same priority are executed alternately until they are finished (or blocked), for FIFO the one which is executing keeps running until it is finished (or blocked); when for both scenarios no higher priority thread becomes ready. This means with FIFO it is possible to reduce thread synchronisation due to its sequential running order, although this is not recommended when aiming for robust implementations and potential reuse within similar scenarios within different environments. EDF is a dynamic policy that gives higher priority to urgent tasks. EDF improves resource utilisation by use of fine-grained scheduling and reduced interference, as the particular tasks' requirements are considered. Therefore the attributes runtime, deadline and period have to be defined which allows a dynamic adaptation. The different policies provide benefits depending on particular use-cases or a software component's internal architecture. As a result, the scheduler may utilise different scheduling policies for different groups of threads, as each component may use the best fitting policy according to the actual use-case implementation. That implies different scheduling-policies may coexist during runtime. This means different components potentially have an adverse effect on each other's task scheduling which may affect the overall system's temporal behaviour. This is detailed in the following.

Especially when composing software with both aperiodic and periodic timing characteristics, the scheduling of the respective threads is a challenging issue. Aperiodic is also referred to as event based, whereas external triggers (e.g., interruptions due to user input or received messages) or the progress of preceding threads (e.g., termination or waiting for I/O) determines the start or computation. If the trigger that issues a thread is dependent on the progression of a certain timespan or when a particular point in time is reached, the start of computation of a thread can be classified periodic or time triggered (Kopetz and Obermaisser, 2002).

The number of coexisting threads sharing the computational resources has significant impact on the overall system's temporal behaviour independent of the utilised scheduling policies. Threads do not switch context instantaneously. Liu and Solihin (2010) distinguishes between direct and indirect overheads. Whereas direct overhead includes saving and restoring CPU registers, flushing CPU pipeline, and executing the OS scheduler, indirect overhead refers to perturbation of the cache and TLB states. The latter is related to the scenario where a thread-pause computation (i.e., due to pre-emption) occurs during the subsequent running thread and may bring its own working set to the cache and overwrite/invalidate at least partially the set of the former one. Hence the earlier thread has to rebuild the cache and TLB when it resumes execution. This behaviour may reiterate for each context switch. Hence, with an increasing number of threads, the overhead also increases due to context switching. Further, with more threads, the access to the computational resources is reduced and delayed for an individual one. Given the 1,000 threads of Project Alpha (cf. Section 3.2.1) and assuming there are 10 percent ready for computation during a high load situation, all using the same priority level and a timeslice of 4 ms (realistic value for QNX Neutrino real-time OS; the default timeslice for RR with Linux based OSs is 100 ms) and context switch time of 0.001 ms. The temporal cost of context varies due to cache interference that is also referred to as 'indirect costs' of context switching, as well as the available hardware architecture and features (Li et al., 2007). For this example the context switch time is insignificant but considered for completeness.

$$t_{\text{wait}} = \sum_{1}^{n-1} (t_{\text{timeslice}} + t_{\text{context switch}}) \text{ ; with } n \text{ ready threads} \tag{4.1}$$

$$t_{\text{wait}} = \sum_{1}^{(1000*1/10)-1} (4ms + 0.001ms) = 396.099ms \tag{4.2}$$

Following Equation 4.1, the introduced wait time (also referred to as 'invocation interval') for the exemplary scenario adds up to 396 ms for rescheduling a pre-empted thread (cf. Equation 4.2) not considering any additional costs introduced due to cache invalidations. The wait time causes missed deadlines for the individual threads T as illustrated in Figure 4.4. This may cause inadequate responsiveness even for non-critical automotive use-cases like visual or audible feedback for user input when considering the mitigation of driver distraction due to delayed feedback. According to Stevens et al. (2002, p21), a 'timely response' should be given within 250 ms for in-vehicle information systems with respect to usability. However, requirements for such systems define responsiveness with less than 50 ms for feedback on user input. A neatly layered software architecture for interactive systems (e.g., following the model view control (MVC) pattern) may obviate fulfilment of temporal requirements without considering scheduling policies and thread priorities. The wait time measures linear with the number of threads. Nevertheless, the behaviour of an uncoordinated scheduling is not predictable due to aperiodic tasks and unforeseeable permutations of components' states within the complex architecture of ICM systems.

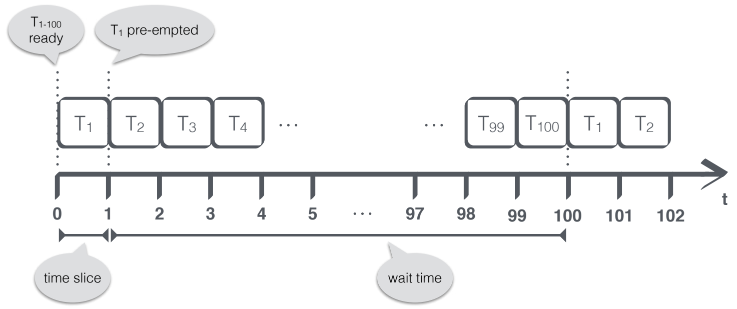

Figure 4.4: RR scheduling with 100 threads using the same priority

Although this can be controlled by use of suitable scheduling policies and thread priorities related to the particular importance of the respective threads, the low prioritised ones have to cope with even less access and increasing delays. However, the overhead introduced due to context switches still reduces the overall efficiency.

Although priority-based scheduling provides a flexible means to adapt the system's threads according to their importance, the mapping of timing constraints into a particular priority scheme is not simple for complex systems. Buttazzo (2011, p10) claims this is also a result of the limited number of priority levels, while on the other hand deadlines may vary over a considerable wider range. Although this might theoretically apply to complex systems that have to fulfil a wide range of use-cases, by practical means an increase of priority levels has a negative effect on the system's maintainability. The typically available 128 to 256 priority levels already introduce a high degree of freedom which the developing organisations within a multi-sourced environment have to agree upon. On the contrary, due to the experience gathered in industrial projects, it is beneficial to reduce the number of priority levels by use of a specification to improve comprehensibility and support predictability and traceability of the system's behaviour. Further, a dynamic mapping of priorities also provides limited support, as on the arrival of new and the termination of existing tasks a remapping might be necessary. Such remapping causes additional overhead that may affect the system's overall temporal behaviour and complexity.

With the transition from a federated to a consolidated architecture (cf. Figure 4.3), more software components, and therefore more execution units (i.e., threads), compete for the 'shared resource' CPU. Third parties supply the software components of ICM systems. Those components usually utilise a scheduling-policy and thread priorities according to their individually required behaviour. This implies that they configure the scheduler independently of the remaining system components which are provided by other parties. The coordination of scheduling policies and thread priorities is only possible with a deep insight into all individual software components while considering the overall system's timing behaviour. With more software components to consolidate, complexity rises.

The implicit issue regarding the integration of different components can be illustrated with only three threads. Therefore it is assumed that a component's functionality can be disassembled into multiple threads. Those threads might be of different criticality and deadlines to support mixed QoS and hence are assigned different priorities. Those priorities assigned to the threads of a single component constitute a self-contained priority scheme.

In the following, C denotes the components of the system S (cf. Equation 4.3), while each C contains multiple threads T (cf. Equation 4.4). Further, each T has a specific priority P within limits defined by the scheduler or the scheduling policy respectively (i.e., the priority levels available; cf. Equation 4.5).

$$S = \sum_{i=1}^{n} C_i \text{ ; with system } S \text{ and component } C \tag{4.3}$$

$$C = \sum_{j=1}^{m} T_j \text{ ; with component } C \text{ and thread } T \tag{4.4}$$

$$(P_{min} \leq P(T) \leq P_{max}) \wedge (P(T_j) = P_j) \text{ ; with Priority } P \tag{4.5}$$

The scheduler handles the threads according to their priorities independently of their affiliation to a given component. Given a system of at least two components (*C1* and *C2*) with one of them having at least two threads (here *C1*), the priority schemes of different components may overlap (cf. Equation 4.6).

$$\Big((n > 1) \wedge (m_{C_1} > 1) \wedge (m_{C_2} \geq 1)\Big) \Rightarrow \Big(P_{1_{C_1}} \leq P_{1_{C_2}} \leq P_{2_{C_1}}\Big) \tag{4.6}$$

Different priority-schemes defined within different components are not necessarily compatible. This means they may interfere with each other in an adverse way. Furthermore, they may not reflect the respective components' overall criticality. This applies for example, if *C2* is of higher criticality (i.e., higher overall priority) in relation to *C1*. This becomes even more complex if one component utilises a wide or even the complete range of priority levels defined through (P_{min}, P_{max}).

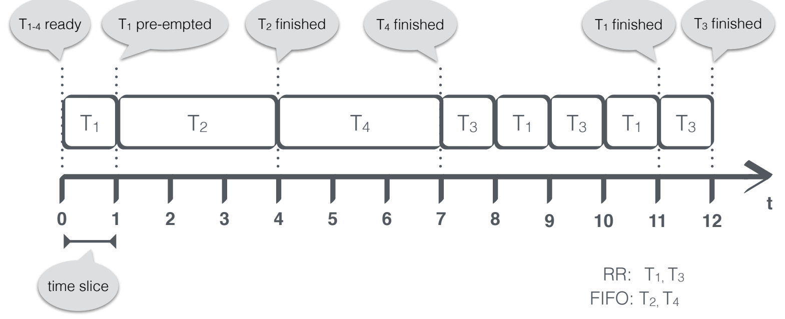

Figure 4.5: Mixed policies assigned to threads with the same workload and priority

Incompatibility also applies for mixed scheduling-policies, such as pre-emptive and cooperative policies, such as RR and FIFO using the same priority and assigned to the same CPU core. A FIFO thread will occupy the CPU until it is finished, a higher priority thread becomes ready, or has to wait for I/O. This means for a schedule derived from a taskset consisting of four threads (*T1-4*), with two threads using FIFO and two threads using RR, the RR threads compute after the FIFO threads have finished (the RR threads may compute for one timeslice; cf. Figure 4.5 and Section A.1).

A homogenisation of incompatible priority schemes and scheduling-policies may imply extensive efforts of coordination at both the organisational and development levels, which could result in the need to reengineer substantial parts of given software components that already fulfil all (component-local) functional and most[3] non-functional requirements.

[3]They actually do not meet the requirements for compatibility, if available.

The initial requirements could be blamed for incompleteness, but this does still not cover legacy or contributed off-the-shelf (COTS) software, provided as-is where the integrator is not willing to pay for any changes, or the contributor is not willing or not able to apply any changes.

4.2.2 Partition scheduling

Partition scheduling provides a solution for incompatible priority-schemes or scheduling-policies. The idea is to assign the software components to separate 'vertical' partitions which are able to apply different scheduling schemes and thread priorities without negatively interfering with each other in terms of the utilisation of the computational resources. This can be achieved by use of a partition scheduler which is able to separate the computational power of the underlying hardware into distinct partitions by means of agglomerates of threads. All threads of a single partition have to compete for the computational share which is granted to the partition. This share may be assigned statically by use of a fixed quantum or dynamically relative to the current system load, whereas the latter decreases determinism in terms of timing behaviour.

The approach of partitioning the platform by use of a scheduler supports the integration of components into a single software system while the platforms resources can be shared. This includes temporal and spatial partitioning according to the definition of Y.-H. Lee et al. (2000):

"Spatial partitioning implies that a partition cannot access other partitions resources, like memory, buffers, and registers. On the other hand, temporal partitioning guarantees a partitions monopoly use of a pre-allocated processing time without any intervention from other partitions."

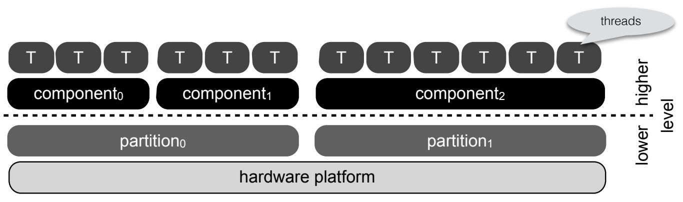

Figure 4.6: Partition scheduling with three components assigned to two partitions

As effect, components assigned to different partitions have exclusive access to the platform's resources (albeit only for a certain timespan if partitioned temporal), significantly reducing a conflicting interference. Kim and Y.-H. Lee (2002) provide a detailed view to real-time capable integration of time- and event-triggered tasks by use of partitioning with an evaluation of exemplary algorithms while having a strong focus on the integration in complex systems. Therefore they define partitioning as a two-level hierarchical scheduling approach, allocating a processing capacity for each partition.

Such a two-level processor scheduling relies on a lower layer and a higher layer, as depicted in Figure 4.6 for a setup consisting of three components and two partitions. The low layer separates the processing power into partitions by use of a predefined monotonic cyclic schedule. A predefined processing capacity is granted to each partition in fixed intervals (a.k.a. cycles). Within this context, processing capacity serves as abstract concept for measures such as CPU cycles, timeslice, or instructions. This allows calculating the 'schedulability' of periodic tasks which are assigned to a predetermined partition. Additionally, the predictability of the latency of aperiodic tasks increases. This leads to an improved determinism of the targeted system and implies a more probable compliance with the required real-time behaviour derived from the QoS of ICM applications.

The high layer is characterised through the partitions and their local scheduler. The partitions are defined by their assigned tasks which determine the actual scheduling algorithm. This implies that different scheduling algorithms are allowed to coexist on the high layer. Although temporal interference is prevented by the monotonic cyclic schedule of the low layer underneath, the behaviour of a certain application is highly dependent of the processing capacity and the invocation interval assigned to the predetermined partition (Kim and Y.-H. Lee, 2002).

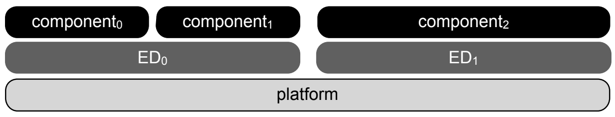

Figure 4.7: Partitioning of components using execution domains

To summarise, the partitions defined by the low layer effectively create separated scheduling domains, in the following referred to as execution domains (ED), as depicted in Figure 4.7 for the previously defined setup.

Definition *Execution Domain (ED)*

An ED is a scheduling domain configured by use of the PU-affinity feature of an OS's task scheduler. Tasks allocated to an ED are scheduled independently (regarding task-priorities and scheduling-policies) of tasks not allocated to that ED.

These improve the compatibility of mixed criticality and multi-sourced components. This is proposed as a fundamental concept for the integration of next-generation ICM systems. Unfortunately, OSs normally do not feature partition scheduling. But in combination with MC hardware architectures under certain conditions, an efficient and portable solution becomes available, detailed in the following section.

4.2.3 Multicore architectures

MC hardware is characterised by providing multiple processing units (PU) in form of CPU cores. They have been common in the High Performance Computing (HPC) sector for decades. In the recent past they have emerged and proved applicability also in server and desktop market segments to solve the need for more computational power while improving energy efficiency. This is mainly driven by the fact that an increase of clock speeds to improve performance reached a physical barrier due to current limits in transistor technologies. This is also valid for the domain of embedded systems, where special purpose PUs support the main processing unit to form a heterogeneous SoC MC architecture. But also homogeneous MC architectures are already available for different instruction set architectures (Levy and Conte, 2009). These provide a number of advantages, some of the most prominent of which are outlined by Smit et al. (2008): scalability, energy efficiency and independency of computational tasks.

Scalability is supported, as the architecture itself does not grow in complexity with future technologies. Only the number of provided PUs increases, depending on the density of the integrated circuits and the size of the silicon. The computational power of MC CPUs scales direct proportional with the number of integrated PUs, although the exploitation will suffer due to necessary overhead.

Energy efficiency can be obtained by switching off unused PUs temporarily to reduce power consumption. Also the clock speed might be dynamically adapted to current needs for computation tasks that do not have to fulfil real-time constraints. Energy efficiency increases with reduced clock speeds, resulting in a lower thermal footprint. However, dynamic adaptation of clock speeds and switching off PUs massively affect the determinism and might not be adequate for very complex systems within a safety-relevant environment.

The most relevant feature in regard to this research is the independency of computational tasks. For MC systems, this is realised by space division on MC architectures in contrast to the time division manner of multitasked software systems executing on single-core systems. That means that MC systems support parallel processing whereas single-core systems have to perform jobs concurrently ('as if they were parallel'). However, MC systems still have to compete for shared resources. Functional dependencies are realised by using an inter-PU communication bus or network for routing information between the PUs.

Eventually MC CPUs were basically introduced to avoid the physical problem of increasing clock-speeds to enhance computational power and not as a new feature to provide more parallelism which the software developers have to cope with. According to Pham et al. (2011), the increased power comes at cost of increased complexity for software development. Although with the use of an OS that features Symmetric Multi-Processing (SMP; detailed below) and hence abstracts multiple available PUs with a single scheduler (Wietzke, 2012, p151), the execution of dependent and independent tasks at the same time has to be considered throughout the development. Basically, the utilisation of MC requires adequate parallelisation and hence synchronisation, which according to Sutter and Larus (2005) and Cantril and Bonwick (2008) produces additional challenges for implementation. These may include finding and implementing parallelism, advanced debugging techniques to analyse deadlocks and race conditions, and eliminating performance bottlenecks (Pham et al., 2011). However, with the availability of MC hardware architectures, new capabilities for structuring software are provided (Kopetz et al., 2007). They facilitate an opportunity to reflect a parallel software design in hardware. This opportunity was taken up for this research.

In regard to the concept of a two-level partition scheduler, a use of multiple computational PUs has significant effect on the invocation interval of predefined partitions and their threads respectively. They are not necessarily in temporal dependence due to the opportunity to compute tasks in parallel instead of concurrent execution. The summed up processing capacity within a given timespan increases, while the invocation period decreases. This has a positive effect on the partitioned tasks latencies and therefore on the overall (RT) behaviour.

With SMP, a single OS abstracts the hardware resources and employs a single scheduler to disperse computational resources. The scheduler reduces the additional complexity caused due to multiple PUs and the derived parallel computation. For an MC system, this implies that threads ready for computation are allotted to available computational PUs considering an equal dispersion of load while reflecting given priorities. Usually neither the coherence of threads with respect to their components nor the communication flow between the tasks influences this distribution. In best case, an already-scheduled thread might be kept on a particular PU for further computation to decrease cache-bouncing effects related to context switch and reschedule on a PU with distinct cache hierarchy. Cache-bouncing increases cache invalidations which cause read and write misses and therefore rising latencies during memory access (Tam et al., 2007).

The PUs feature interconnected but independent partitions which the scheduler makes available transparently to the applications. The focus here is to maximise the system performance. Regarding the scenario related to the wait time as depicted in Figure 4.4, the use of MC divides the invocation interval by the number of available PUs (cf. Figure 4.8).

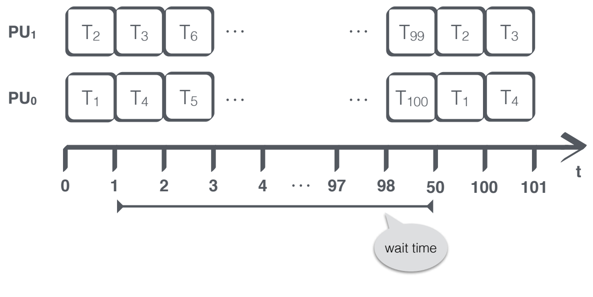

Figure 4.8: RR scheduling with 100 threads on same priority using two PUs

Although this has significant impact on the temporal behaviour, i.e., the system's responsiveness to interaction and a higher probability to fulfilling deadlines, the underlying problem of non-deterministic scheduling due to incompatible components is not solved. With respect to the history in computer systems, more computational power will be utilised with additional and more complex features (i.e., components and applications).

Hence, even if additional available PUs have a positive impact on the wait time, it is very likely that future systems have to cope with even more threads. That implies that merely adding computational power will improve the system only in the short term, if at all.

This also applies to the scenario related to mixed scheduling policies as illustrated in Figure 4.5. Here any additional PU also improves the temporal behaviour, but it is still not foreseeable whether deadlines of particular tasks can be met, as depicted in Figure 4.9. Moreover, parallel computing might introduce new problems for cooperative scheduling (i.e., FIFO with threads using the same priority level) when synchronisation relies on the sequential computation. This is exemplified by use of a set of practical experiments, detailed in Section A.1.

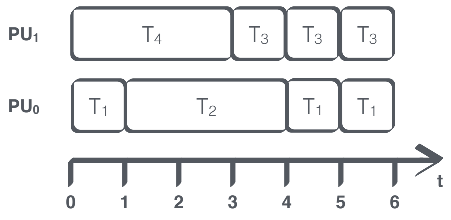

Figure 4.9: Mixed scheduling policies on two PUs

In summary, within an SMP system the OS scheduler's decision regarding the targeted PU for computation depends neither on affiliation to a component nor on the assigned scheduling policy and priority. It basically may optimize the dispersion of threads for process locality with respect to improving trashing effects and load balancing to fully utilize the system's capacity. Although these are valid goals for some system types (e.g., non-interactive batch processing with focus on high throughput), they do not apply to highly interactive MCS.

The additional computational power introduced with MC hardware has to be adapted to the software system to improve benefits. This can be achieved by partitioning the available hardware by means of the available PUs aligned to partitioning the software system. With respect to performance efficiency this can be realised with the herein-proposed concept regarding segregated scheduling using EDs based on PU-affinitiy, as they employ already OS features without the need for an additional management layer (cf. Section 4.2.2).

For SMP systems, the OS's scheduler can be configured by use of 'thread-affinity' on the granularity of threads to assign them statically to a specific set of CPU cores (Love, 2003; Love, 2010; Wietzke, 2012; Nagarajan and Nicola, 2009). This is also referred to as Bound Multi-Processing (BMP)), CPU-, core-, or PU-affinity. A set of PUs may include any number from one to all PUs. The default affinity assigned to a thread usually includes all available PUs, which grants the scheduler the freedom to decide where it is actually to be executed. Limiting this freedom into defined PU sets by use of a static configuration according to the component's peculiarities effectively allows for partitioning the computational resources. Both incompatible scheduling policies as well as priority schemes can be integrated onto a common platform while mitigating adverse temporal interferences related to shared PU resources. The components' threads are allotted to different PUs, computed in parallel and therefore do not have to compete for shared PU resources, as exemplary illustrated in Figure 4.10 for a dual-core hardware platform.

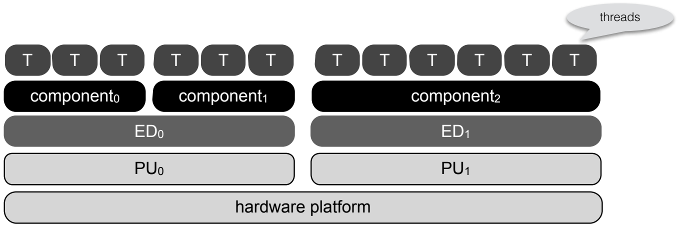

Figure 4.10: Realisation for execution domains using MC hardware

Such an architecture based upon EDs that exploits the features of MC hardware fosters the components' compatibility without the need to modify the components. The partitioning is transparent to the of the components' suppliers. This enables an integrator to define an ED in form of a dedicated PU or set of PUs by adapting the scheduler's allocations depending on implemented use-cases, independent of the software's internals. This applies also for components which are not available in source code. Further, it is possible to safely integrate components that are not prepared for parallel computation such as legacy code that does not employ adequate synchronisation mechanisms. A component bound to a single-PU behaves very similarly in timing as to what is observed when it is executed exclusively on a single-PU system, similar to a separate ECU, as systems were designed in the past (but without additional housings, power supplies, etc.). In contrast, the components need not rely on parallel computation or individually assign their threads statically to particular PUs. This limitation can be enforced by use of a robust software framework that interferes with the corresponding API calls for configuring thread affinities. However, a software framework may not only incorporate enforcement of limitations but can also exploit the features for creating EDs by use of a portable abstraction of partitioning components. This affects both the composability of such and improves the maintainability (i.e., usability for the integrator). This can help to mitigate risks related to unforeseeable integration expenses and thus help to predict the success of multi-sourced development projects.

Moreover, predefined collocations of particular threads with respect to their inherent interdependencies or affiliation to components may positively affect the system's performance. This can be achieved by taking advantage of the underlying hardware's (coherent) cache hierarchies (Schnarz et al., 2014a). Co-location of communicating threads can reduce memory access latencies by decreasing cache misses. That means the throughput can be improved when communication partners use a shared cache instead of the main memory. Information stored within shared caches can be accessed about 10 (or more) times faster than memory connected via system bus. This means optimised arrangement of tasks can improve overall system performance (Tam et al., 2007).

For the successful isolation using EDs, a set of essential capabilities has to be provided by an underlying software framework in association with the utilised OS. Those were identified as part of this research:

- Threads and groups of them can be statically bound to certain PUs.
- That binding is inherited to dynamically created sub-threads.
- The scheduler supports the parallel execution of tasks which make use of different priorities on different computational PUs.
- The static mapping of tasks to PUs appears transparent to the application developers and is managed only by the system integrator.

As part of this research, those features were incorporated into OpenICM. The API for the application developer did not need to be changed, whereas the integrator is now empowered to cluster the components on certain PUs with minimal efforts. Only one additional argument for indicating the targeted PU is necessary for the subsystems (i.e., components) start routine. The implementation abstracts the peculiarities and deviations from POSIX for Linux and QNX.

As a result of such a static configuration, the tasks are scheduled as defined by the integrator based on the interdependencies and predefined characteristics rather than 'only' depending on an equal dispersion of workload. That means different software is separated. Distinguishing characteristics may include the particular vendor, change rate, internal structure, internal priorities, internal scheduling strategies, and mission. Imminent conflicts are effectively reduced, without the need for changing the components' internals. A task with high priority does not displace a low priority task as long as both are defined for different execution domains which implies they could make use of different priority schemes. This probably does not support a most optimal performance, but improves the deterministic behaviour and helps to reach a higher grade of stability. Erroneous behaviour is not necessarily propagated beyond the boundaries of one computational PU or one set of computational PUs predefined with a bit-mask. According to Aggarwal et al. (2007), depending on the implementation, it is even possible to handle failures of affected subcomponents for improving availability.

To summarise, EDs realised through the exploitation of the capabilities of MC hardware in combination with an SMP-based OS is an effective way to partition heterogeneous components for mitigation of adverse temporal interference.

4.2.4 Interrupt affinities

An interrupt signals the OS's kernel about an event. This event is handled by use of an interrupt handler routine (a.k.a. Interrupt Service Routine (ISR)) as part of the kernel or device driver. The occurrence of an interrupt changes the instruction sequence independently of the scheduling policy or thread priority currently being used for executed user space applications (as well as kernel functionality for some OSs). This means an ISR blocks the systems processing asynchronously and temporarily takes control over the PU where it is executed: it ‘interrupts’ the current processing context and immediately switches to another one.

Besides the discussed scheduling overhead, the interrupt latency is an important characteristic for a system’s real-time behaviour. The interrupt latency is defined by the occurrence of the signal until executing the ISR. As Wietzke (2012, p114) states, this latency subsumes the time necessary for the completion of the current instruction, save of CPU registers, disabling potential concurrent interrupts and change context. Further overhead is necessary to enable interrupts and restore the context and registers after completion of the ISR. Depending on the current version of OS, further tasks related to memory management might be necessary. The time needed for preparation, execution and completion of an ISR affects the computation of applications and may have significant impact on the system’s temporal behaviour depending on frequency of interrupts. However, interrupts are necessary to ensure the timeliness (e.g., timer interrupts used for scheduling) and responsiveness (e.g., signal user input or receive of fieldbus messages) of the system.

To reflect the different types of signals, interrupts can be divided into two types: hardware interrupts and software interrupts. Whereas the former are related to events originated by attached hardware (e.g., input devices, clocks, etc.), the latter one are related to events (implicitly) issued by the processed applications to signal the OS. They may differ in their ISR’s execution mode, derived privileges and reactivity to the actuating event, also depending on the actual implementation of the OS. Comprehensive insight to interrupts and ISRs are provided by Wietzke (2012) and Love (2010). However, both types share the same impact to the system with introducing delay for processing applications.

The delay introduced due to ISRs is no problem as long as the trigger is as the same or of higher criticality as the current processed thread. There are no means to map an event’s criticality to the actual processing of an ISR, because it is executed independently of any thread priority. This is attributed to keep the interrupt latency as small as possible. However, within a MCS like ICM, for example, a user input must not delay the processing of an ASIL-relevant software component.

With the use of MC systems and the static configuration of the component’s threads’ affinities, the relation between critical processing and interrupt handling is also adaptable. The interrupt requests (IRQ) are processed by a programmable interrupt controller , which is an integrated circuit (IC) that basically collects several interrupt sources (several lines) and passes them to the CPU(s) using less connections (often only one). For MC hardware, a PIC usually allows programmable routing of interrupts to dedicated PUs (ARM, 2014). This feature realises PU affinity for IRQs and hence provides the capability to control where the related ISRs are processed, comparable to the configurable thread affinity using the scheduler. This enables the separation of critical processing and interrupt handling.

Beside separation, IRQ affinity enables co-location of coherent ISRs and components.

This may apply, for example, to input devices and the ICM's corresponding HMI software components, or for communication devices and the ICM's corresponding software components to handle incoming messages. This can improve the overall performance due to exploitation of shared caches and hence more efficient intercommunication in between ISR and the corresponding software components, comparable to thread clustering as proposed by Tam et al. (2007).

In summary, using IRQ affinity, the component structure of the software can be mapped to the interrupt handling to ensure critical components are not delayed due to uncritical interrupts, as well as to improve reactivity including the full path form IRQ to software component due to shared caches. To put this more simply: as long as an IRQ is related to particular software components, it is also related to the respective ED which the software component is assigned to. This means the IRQ has to be assigned to the ED of the respective software component it belongs to. As with thread affinity, for IRQ affinity no changes to the OS or the hardware are necessary as long as a reprogrammable PIC is available.

4.3 Related concepts

The proposed use of MC hardware architectures to realise isolated EDs features only one opportunity to structure software components. In the following, some notable alternatives and complementing concepts are discussed. The aim is to emphasis the appropriateness of the EDs as well as point out the composability within or next to other architectural means for partitioning software.

4.3.1 Service orientation

A great deal of research for structuring and governing complex software systems has been done in the field of service-oriented architectures (SOA). Following Krüger et al. (2004), automotive software systems do not pose an exception. However, with increased abstraction of system complexity, the overhead during runtime may increase. This is attributed to flexibility of composition of services which relies on the paradigm of register, find, bind and execute (Zhu, 2005). Although SOA may support the integration of heterogeneous software components, it lacks efficient exploitation of the underlying hardware architecture's features and neglects issues caused by conflicting scheduling policies and priority schemes. SOA may not necessarily be seen as a contrary architectural concept. Moreover, services may substitute components within the herein proposed concept for ED to support compatibility of heterogeneous services for temporal behaviour due to mitigated interference. Hence, the availability of EDs may introduce an additional layer for fine-grained deployment on a single hardware platform to complement the features of SOA for improved compatibility of multi-sourced services. Eichhorn et al. (2010) propose a flexible in-vehicle HMI based upon SOA to foster platform independency, scalability, faster time to market and lower development costs using web technologies, but do not address mixed criticality.

The OSGi Alliance propagates with 'OSGi' an open dynamic component platform to assure interoperability of applications and services. The platform addresses the integration issue of software provided by different vendors with respect to reliable operation, shared resources and the ability to add functionality dynamically during runtime. With a sophisticated service model, it follows a service-oriented approach, relying on a Java Virtual Machine (JVM). Kriens (2008) provides a comprehensive overview on OSGi. Comparable to SOA, it fosters interoperability. OSGi does not cover the hardware's features for partitioning or addressing compatibility issues due to conflicting scheduling policies and thread priorities. But again, the concept of EDs may complement OSGi to mitigate adverse temporal interference in between heterogeneous components.

4.3.2 Budgeting

Budgeting is related to agglomerate tasks into defined partitions and grants those partitions access to the platform's resources repeatedly up to a certain time. That time is derived from a temporal budget assigned to the respective partition, comparable with a pre-emption scheduler for groups of tasks with heterogeneous time slices.

QNX Software Systems provide the capability to partition software by use of resource budgets with their Adaptive Partitioning Scheduler (APS) (Johnson et al., 2006; QNX Software Systems, 2010). This thread scheduler supports the creation of such 'virtual' partitions that are configured during runtime with a specific budget that corresponds to a guaranteed portion of resource usage. Software components are assigned to those partitions while utilising application-specific scheduling policies (e.g., RR, FIFO, etc.) and priorities. This basically corresponds to a two-level partition scheduling approach with configurable partitions at the lower level that incorporates dynamic adaptation to actual utilisation. The latter implies a high load (a.k.a. 'full load') partition is allowed to borrow unused budgeting of another partition (referred to as 'underload partition'). The borrowed time must be returned back to the lending partition as much as the scheduler 'remembers' (i.e., only the borrowing that occurred in the last 'cycle'). However, partitions'-guaranteed budgets take precedence over the contained threads' priority. Nevertheless, threads can be defined as 'critical' that allow them to run even if its partition is already over budget. For such cases, 'critical time' is billed against the respective partitions. If the billed critical time exceeds a partition's critical budget, 'bankruptcy' occurs that forces the initiation of a recovery policy. As this is considered as design error, it significantly affects the related threads' behaviour and overall systems behaviour. Depending on the recovery policy this means, e.g., either turn-critical budget to zero, a forced reboot, or a pause of the scheduling of the partition.

Such an approach fosters the exploitation of the platform's computational power (compared to cyclic toggling between partitions using fixed time slots – cf. synchronous Time

Division Multiple Access in Section 5.3). But the combination of budgets, priorities and adaptive behaviour may contradict each other and affect predictability in high load situations. As exemplified by Vergata et al. (2010), this means APS does not support independent, non-coordinated priorities and scheduling schemes in different partitions. Furthermore, budgeting with APS is also affected by IRQ handling. The additional temporal efforts related to handle an IRQ are deducted from the budget of the current computing partition, regardless of any dependencies between the respective IRQ and the computing partition. Moreover, the inherent complexity due to the adaptive behaviour produces new challenges for trace and debugging the system behaviour. This is emphasised for systems that may change their functionality during product lifecycle (i.e., dynamic functionality).

4.3.3 Virtualisation

Virtualisation describes a concept of simulating hardware in software, which has been a research topic for decades (Goldberg, 1974). It basically supports the provisioning of one (or more) virtual hardware platform by use of a single physical one (Neiger et al., 2006). Following Popek and Goldberg (1974), this efficient, isolated duplicate of a real machine is realised by use of a virtual machine monitor (VMM) (a.k.a. hypervisor). The VMM defines and manages the capabilities of the virtual platforms which are referred to as virtual machines (VM). The runtime environment appears for the software (e.g., OS, bare-metal applications) executed on the VM like deployed on physical hardware, except degradations in speed. This means a VMM may host several VMs, while each VM represents a partition with a different OS instance, which in turn may host multiple application components. Following Kanda et al. (2010), this realises a multi-OS environment.

The use of different OS instances implies different scheduling domains and hence decoupled scheduling policies and priority schemes. Adverse temporal interferences are mitigated. This behaviour can be improved when allotting the VMs to distinct PUs (Kanda et al., 2010; Vergata et al., 2012).

Furthermore, such a multi-OS environment provides the capability to employ different OSs, with each providing dedicated features for certain purposes. This may include real-time capabilities, UI features, compatibility to CE platforms, or domain-specific application frameworks.

Nevertheless, the introduced abstraction layer to provision the virtual hardware platforms introduces overhead as well as the operation of multiple OSs. Such overhead include additional startup time to initialize the guest-OSs, memory resources for the extra OSs instances within persistent and non-persistent memory and also additional latency during runtime. This is caused due to the indirection introduced with the abstraction layer of the VMM and additional guest-OSs and also includes additional efforts for intercommunication between different OS instances. An empiric comparison regarding the runtime overhead is detailed in Table 4.1 using quantitative measurements based on a Linux based QEMU/KVM VMM (Kivity et al., 2007). Those are gathered using the benchmark-suite 'lmbench' (revision 3.0-a9), which relies on a set of reproducible and transparent test cases to compare latency of e.g. system calls, process creation, math operations and signalling (Staelin, 2005; McVoy and Staelin, 1996). Basically, such overhead reduces the integrated system's effectiveness, as computational power is spent to operate infrastructural system parts such as VMM and VMs. The efficiency can be improved with the availability of hardware-supported virtualisation features as detailed by Neiger et al. (2006). Such features are also available for current automotive platforms. Furthermore, the guest-OSs can be optimized for the virtualisation layer to reduce the performance degradation. The measurements presented in Table 4.1 were collected using x86 hardware platform that features such virtualisation support (using a 64bit Linux kernel 3.6.37-mainline, a magnetic hard-drive and eight processing cores with 1995 MHz each). Although those show a near-native performance can be achieved, there is overhead which affects system performance. This does not apply for the EDs based on PU-affinity, because the indirection and the need for infrastructural system parts are not needed. ED-based partitioning of components effectively is equivalent to the measurements for native execution, deducted from quantitative empiric tests. Static partitioning (i.e. static allocation of components to scheduling domains) is most effective with regards to avoiding performance degradation by use of EDs.

Heiser (2008) details virtualisation use cases, its limits and technologies for the domain of embedded systems and presents a microkernel (named 'OKL4') that offers hypervisor functionality. In further work, Heiser (2011) refers to virtualisation and exemplifies this with the collocation of infotainment and AUTOSAR using a single hardware platform, following the architecture proposed by Hergenhan and Heiser (2008). Also, Kaiser (2011) anticipates virtualisation as an adequate means to cope with the challenges of upcoming complex embedded systems, despite the fact that most of current VM environments are still targeted for desktop and server systems. However, increased manageability due to the use of multiple isolated VMs hosting different subsystems applies for embedded, desktop and server systems.

Virtualisation may provide an alternative for ICM systems to structure different software components into isolated containers as proposed by Vergata et al. (2010). However, even with hardware-supported virtualisation, speed degradations are measureable. Moreover, the isolation of components using VMs in combination with the operation of multiple (instances of) OSs inhibits the utilisation of mature and efficient intra-OS synchronisation and communication infrastructures. Such inter-OS communication between different VMs is basically comparable to communication in between different hardware platforms. However, there are technologies to signal and allocate shared memory regions accessible by multiple VMs to realise efficient data interchange, as proposed by Macdonell (2011) with the Nahanni system.

Despite the drawbacks of virtualisation due to the additional overhead the provided multi-OS environment in combination with thread-affinity-based EDs enables a flexible system architecture. A host-OS may provide the device drivers and runtime environment for

Measurement	**Unit**	**Native**	**Virtual**	**Overhead**	
Syscall read	microsec	0.1198	0.1198	0.0000	0.0 %
Syscall write	microsec	0.1170	0.1172	0.0002	0.2 %
Syscall open/close	microsec	1.3059	1.4481	0.1422	10.9 %
Signal handler installation	microsec	0.1855	0.1865	0.0010	0.5 %
Signal handler overhead	microsec	1.1826	1.2114	0.0288	2.4 %
Process fork+exit	microsec	155.8485	191.1667	35.3182	22.6 %
Process fork+execve	microsec	412.3077	573.1000	160.7923	38.6 %
Process fork+/bin/sh -c	microsec	958.8333	1351.2500	392.4167	39.2 %
integer bit	nanosec	0.50	0.50	0.00	0.0 %
integer add	nanosec	0.25	0.25	0.00	0.0 %
integer mul	nanosec	0.15	0.15	0.00	0.0 %
integer div	nanosec	12.06	12.07	0.010	0.1 %
integer mod	nanosec	11.55	11.57	0.020	0.2 %
int64 bit	nanosec	0.50	0.50	0.00	0.0 %
uint64 add	nanosec	0.25	0.25	0.00	0.0 %
int64 mul	nanosec	0.15	0.15	0.00	0.0 %
int64 div	nanosec	22.31	22.33	0.02	0.1 %
int64 mod	nanosec	21.11	21.53	0.42	2.0 %
float add	nanosec	1.50	1.51	0.01	0.7 %
float mul	nanosec	2.01	2.01	0.00	0.0 %
float div	nanosec	7.47	7.48	0.01	0.1 %
double add	nanosec	1.50	1.51	0.01	0.7 %
double mul	nanosec	2.01	2.51	0.50	24.9 %
double div	nanosec	11.48	11.49	0.01	0.1 %
float bogomflops	nanosec	7.02	7.03	0.01	0.1 %
double bogomflops	nanosec	11.03	11.04	0.01	0.1 %

Table 4.1: Runtime overhead for virtualisation

critical components with real-time demands. The latter can directly interact with the host OS and the platform's devices. Further, the host OS provides the runtime environment for the VMM. According to the definition of Goldberg (1973, p22), this means the hypervisor is a type-2 VMM. In contrast, a type-1 VMM runs on 'bare-metal' without the need for a host-OS. Such a VMM incorporates basic functionality of a kernel such as memory management, task abstraction and scheduling. Although a type-1 VMM may reduce the overall system overhead, host-OS based architectures provide the capability to deploy non-virtualised low-level components. The integrator configures the VMM and the related OS-partitions based upon individual VMs. Each VM can be 'equipped' with n virtual PUs (vPU) to correspond with the number of EDs deployed to the respective guest OS. By this means, a virtual MC platform within a VM is realised, introduced as VMMC (Virtual Machine MC). This provides the capability to allot EDs to the nvPU to achieve independent scheduling domains, as depicted in Figure 4.11 and exemplified in Section A.2. The guest-OSs are interconnected by use of a Nahanni based inter-VM shared memory. As part of this research, this integral architecture-incorporating virtualisation and EDs were presented by Vergata et al. (2012)[4].

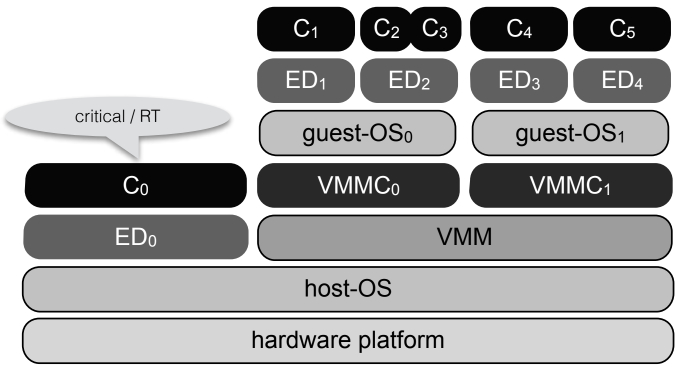

Figure 4.11: Architecture combining EDs and virtualisation

Virtualisation within ICM system is a promising on-going field of research. It can effectively complement the herein proposed concept for EDs on MC hardware.

4.3.4 Application containers

Virtualisation provides a virtual hardware platform to a VM that may run an individual guest OS, also referred to as hardware-level virtualisation. In contrast, 'application containers' rely on a mechanism that sets into an OS kernel that supports application containers. This technology is also known as virtual engines, virtual private servers, or OS–level virtualisation. By practical means, (hardware-level), virtualisation is related to the provisioning of a virtual instruction set architecture (ISA), whereas applications containers rely on the availability of a contained application binary interface (ABI), as depicted in Figure 4.12, modelled after Smith and Nair (2005, p10).

[4]cf. Appendix B - 6

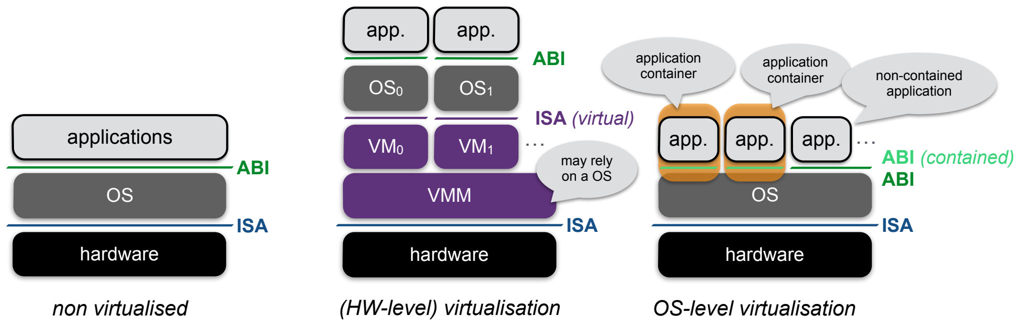

Figure 4.12: Comparison of virtualisation and application containers by their interfaces

Basically, with application containers, 'sandboxes' on top of a single kernel are realised. These provide isolation that aim to reduce adverse interference between applications running within distinct containers. Depending on the actual implementation, it enables resource and security isolation by realising defined share and reservation of hardware-facilities such as available PUs, memory, and I/O (Soltesz et al., 2007). Reshetova et al. (2014) compares several OS-level virtualization systems, discusses their security and identifies gaps of current solutions with focus mainly set on the Linux kernel.

Compared to hardware virtualisation, with such 'OS virtualisation' and the need to run only a single OS that handles multiple containers, the startup time and overhead for partitioning related to intercommunication and hosting multiple OS instances is reduced (Strauss, 2013; Tsai et al., 2014). However, it does not support a heterogeneous multi-OS environment that enables the integration of software components targeted for different OSs or different revisions of an OS. Hence it is comparable rather to the concept of EDs using PU affinities in which application containers are assigned to a dedicated PU or PU-set. However, application containers are not available for each OS kernel, as this approach requires built-in features within the kernel. Moreover, there is no standardized API to configure and control the runtime of 'OS-level virtualisation' (e.g., for Linux different layered APIs or frontends are available to improve the usability of the kernel's built-in 'control groups' and 'namespaces').

An ICM related platform that utilises application containers is GENIVI. Therefore it exploits the Linux kernel's built-in features by use of the abstraction layer LXC (cf. Section 2.4.2). In particular, these include kernel namespaces (related to, e.g., IPC, process IDs, networking, user privileges, etc.), chroot (to change the root file system), control groups (budgeting resource utilisation) (Bustos-Jimenez et al., 2014). Despite isolation, inter-partition communication via shared memory and inter-partition synchronisation can be configured, as evaluated in Section A.3.

As this approach is only available for a subset of relevant embedded OSs for ICM, it is not considered as a portable solution for structuring software components. However, application containers pose a valid supplement for a comprehensive software structure. Also, an enhancement of existing API (e.g., POSIX) or framework that abstracts the core features would improve maintainability and portability (cf. Table 1) and address future demands for flexible OS-level virtualisation in complex software systems with minimal management overhead. Moolenbroek et al. (2014) argue that OS-level and hardware-level virtualisation are extremes in a continuum of virtualisation boundaries in which new alternatives have the potential of combining the good properties of both. This is still a valid field of research, not further detailed within this research project.

4.3.5 Asymmetric Multi-Processing

With Asymmetric Multi-Processing (AMP), the platform's hardware features are strictly separated to different OS instances or applications when no OS is available or necessary (Wietzke, 2012, p151). This means a hardware device is dedicated to only one partition and cannot be shared. The allocation to different partitions is statically defined and realised during the system's startup sequence.

In contrast to SMP that utilises all available PUs for one OS, AMP allows the parallel execution of multiple OSs. As Schnarz et al. (2013) states, this fosters a stricter decoupling of resources compared to the concept of EDs using SMP in combination with thread affinity. Similar to virtualisation, an AMP-based multi-OS environment implies the lack of low overhead intra-OS synchronisation features such as semaphores, mutual exclusions, condition variables and message queues. Within an AMP environment, inter-OS communication may rely on signalling (i.e., to trigger IRQs), message-based hardware interconnects, or network sockets (Mücke, 2014; Daub, 2012). It may utilize dedicated shared memory regions for data interchange (Wietzke, 2012, p150). Also as with virtualisation, a multi-OS environment based on AMP means the dissipation of computational power (overhead) for the operation of multiple OS instances. This power is not available for application components. Summarized, AMP allows a strict decoupling of components, but comes at the cost of limited and more restricted communication facilities and increased overhead. Additionally, it features the parallel deployment of different OSs and so-called 'bare-metal applications', again comparable to virtualised system architectures, but without the need to run a VMM.

Related to the concept of EDs, AMP is not necessarily a substitute. For a layered system architecture that has to cope with different levels of decoupling (or segregation), it may complement PU affine EDs for AMP partitions with multiple PUs and virtualisation. Fischer (2009) provides a comprehensive discussion on an AMP-based multi-OS environment within the context of ICM, including a prototype implementation and evaluation using an ARM-based MC platform.

4.4 Applied structuring

Segregation of different functionality can be seen as a key to achieving dependable mixed-criticality systems that have to provide adequate means for updating particular software components without re-accessing the whole. Figure 4.13 abstracts the elements and interdependencies, divided into the layers architecture, OS, application and hardware (arrows define the direction for the given association accompanied with a label and cardinality). Therefore the segregation is abstracted by the element 'Partition' that is specialised by 'Execution Domain' and 'Operating System', comparable to an object-oriented strategy pattern. The component 'Operating System' can be further divided related to the actual implementation of the respective environment, such as native on hardware, virtualised, and using AMP.

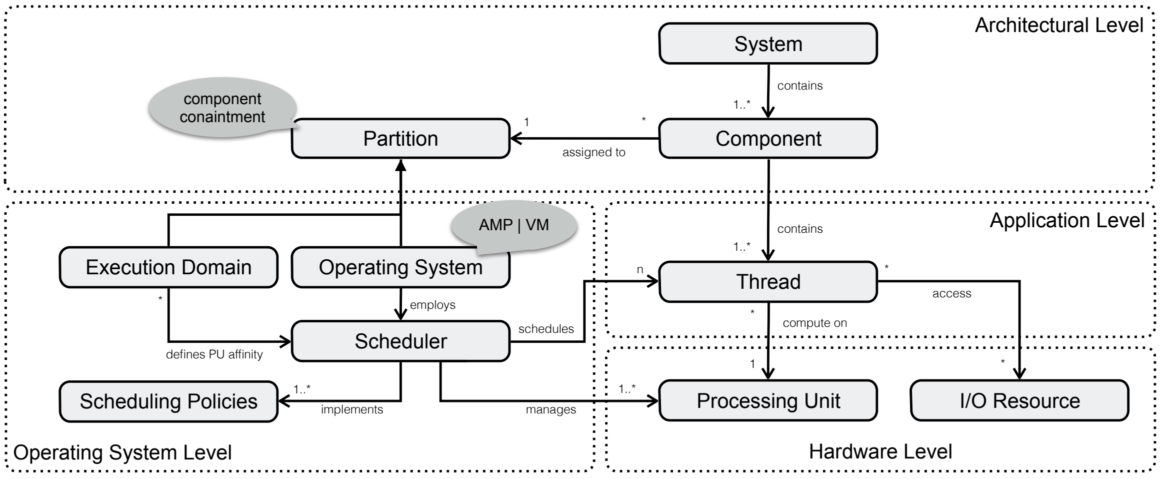

Figure 4.13: Layered view on structuring system components

Such component containment improves composability due to improved compatibility at the architectural level. This is achieved due to enforced decoupling of execution at the application level by use of segregated scheduling domains. Hence, asynchronous components and their threads must not be synchronised with one another. This applies both to the process of system integration by an integration team and also to system runtime by use of a scheduler and respective policies and priority schemes. Components that are not designed for compatibility are to be kept separated. Compatibility is not a characteristic that is easily applied to already available and functional components, if not considered as constructive aspect. For ICM systems, this also gains significance due to the high division of labour during the development process. Further, composability achieved through containment may have positive affects on the system's maintenance as thus the update cycle for partitioned components can be segregated. Further aspects on compatibility by containment are discussed in Section 7.1.1 for the context of an integral architecture.

Structuring can be achieved using different technologies and concepts which also includes a combination of these. A selection of the herein discussed ones with interdependencies on requisites and opportunities is depicted in Figure 4.14 (technologies are highlighted in blue; arrows stand for 'provides' using the given cardinalities). For example, if the physical platform does not provide a sufficient number of PUs to segregate using EDs, a virtual hardware platform is able to incorporate the necessary vPUs as proposed in

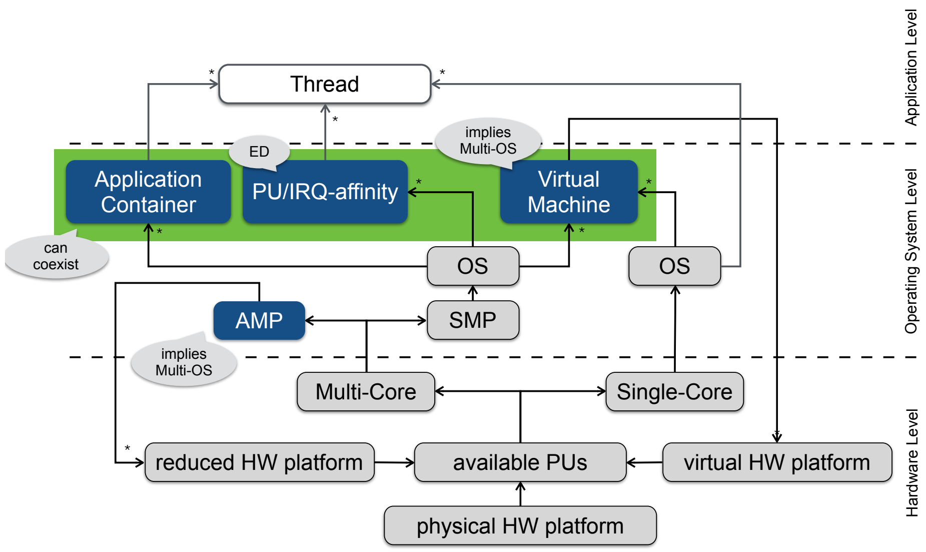

Figure 4.14: Interdependencies between technologies for structuring threads

Section 4.3.3. Further, PU-affinity based EDs can be combined with VMs, AMP and application containers, whereas a decision for a particular technology should correspond with its respective characteristics and the related requirements derived from the components' use-case and criticality.

Such a mix of different technologies for containment is shown in Figure 4.15. It illustrates the nested partitions using bounding boxes. For this example a combination of PU affinity, virtualisation and AMP is used. It further demonstrates the provisioning of additional virtual PUs to enable a sufficient number of scheduling domains and parallel operated OS-instances.

Different technologies and concepts imply different characteristics regarding isolation (such as self-containment, overhead, communication facilities) which may be in contrast to one another as depicted in Figure 4.16. Hence, it is reasonable to classify the different concepts with respect to shared platform facilities. Therefore the 'Containment Level' (CL) is introduced as designator, mapped to particular concept's characteristics in Table 4.2. The CL represents the degree of containment using a scale from zero to four, while CL0 does not provide software component isolation and CL4 means very strict partitioning.

A minimum requirement for adequate containment within the context of this research is non-shared PUs due to the inability of enforcing the software components' incompatible temporal requirements. CL1 to CL4 do provide this capability, whereas the particular levels are differentiated with regards to the availability of shared memory (SHM) or shared I/O. For the context of ICM systems, the focus may be set on CL1 and CL3. They both feature isolated PUs in combination with a shared memory region that allows for efficient inter-partition communication. Shared I/O might be avoided to mitigate concur-

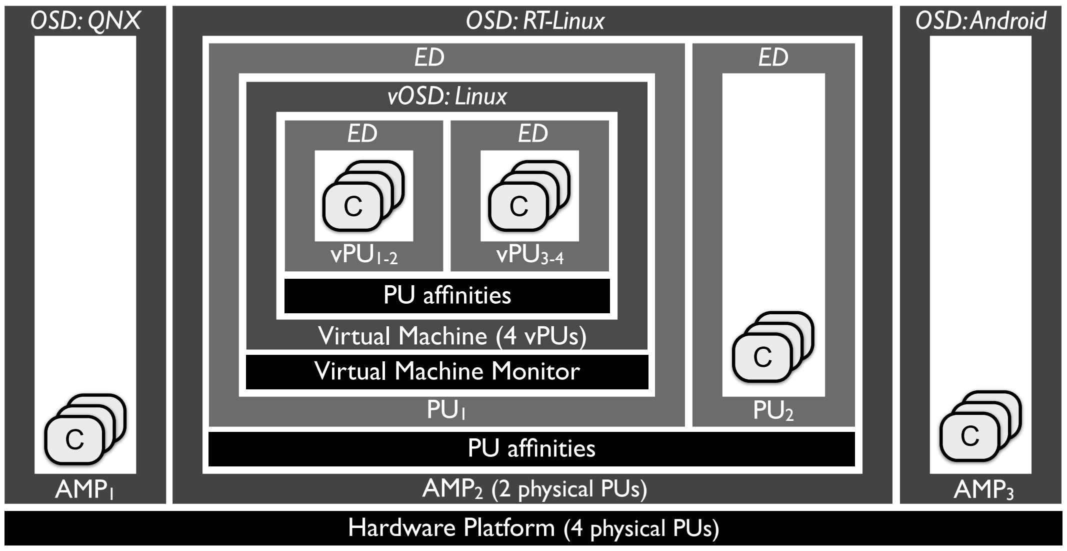

Figure 4.15: Applied containment using mixed technologies for a multi-OS envrionment

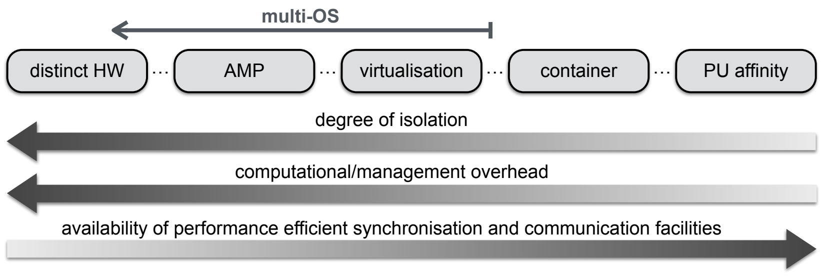
multi-OS
distinct HW
AMP
virtualisation
container
PU affinity
degree of isolation
computational/management overhead
availability of performance efficient synchronisation and communication facilities

Figure 4.16: Contrary characteristics for selected segregation concepts

rent access that affects the temporal behaviour of the competing software components. However, depending on the use-case or applied access management (i.e., arbitration; cf. Chapter 5), shared I/O can be a beneficial option. In contrast, CL2 and CL4 do not feature a SHM region. This implies they are more suited for software components that must not support interoperability (e.g., using efficient intercommunication), but still have to fulfil requirements for compatibility due to integration onto a single hardware platform.

Moreover, as exemplary depicted in Figure 4.17 (evolved from Figure 4.11), different CLs can be combined to address the actual need for isolation using a layered architecture. A higher CL may introduce additional overhead (e.g., due to the need to run multiple OS instances). Further, a low level containment may contain a higher level containment. In the example, ED_0 and the VMM use SMP-based PU-affinity that isolates according to CL1 on tier 1. This does not imply an inner container can increase the CL against an outer container, as depicted with $VMMC_0$ and $VMMC_1$ on tier 2 with CL3. Although both VMMCs are still isolated against each other using CL3, they are isolated against the software component C_0 (deployed to ED_0) with CL1. But while the components on

Containment Level	Shared Processing Units	Shared Memory	Shared I/O Resources	Technology / Concept
CL4	○	○	○	distinct HW; AMP
CL2	○	○	●	VMs with PU-affintiy
CL3	○	●	○	AMP with SHM; VMs with (PU-affinity & SHM)
CL1	○	●	●	SMP with PU-affinity; VMs with (PU-affinity & shared I/O & SHM)
CL0	●	○	○	VMs
CL0	●	○	●	VMs with shared I/O
CL0	●	●	○	-
CL0	●	●	●	SMP

○ = isolated; ● = shared

Table 4.2: Mapping of Containment Levels to concept characteristics

tier 3 within the VMMCs are isolated using CL1 (e.g., C_1 against C_2), the components of the different VMMCs are still isolated using CL3 (e.g., C_1 against C_4).

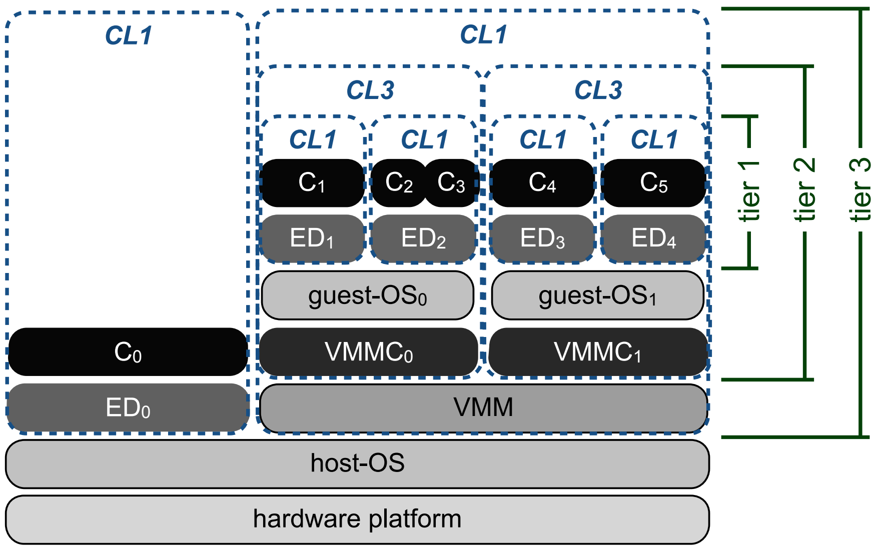

Figure 4.17: Multitier CL architecture

A tree spanned of the different CLs and software components can express this in a more formal abstraction. A node represents a CL while a leaf-node represents a software component. A CL is connected to at least two other nodes. Highlighting the shortest path between the respective leaf nodes and identifying the CL at the lowest tier can determine the effective isolation between two software components. Software components collocated within a single leaf node are assumed as not isolated against each other (i.e., CL0). A graph corresponding to the previous example is depicted in Figure 4.18.

Additional to the CL, it might be necessary to decide whether it is necessary or useful to employ multiple OS instances which increase overall startup time, require additional memory with each instance and require efficient intra-OS synchronisation facilities when using intra-OS SHM regions. The freedom of choice is limited to CL1, which can be implemented using EDs or VMs (with SHM and shared I/O; cf. Table 4.2).

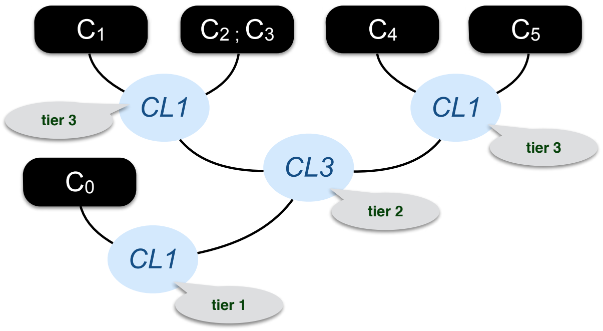

Figure 4.18: Visualisation of effective CL in multitier architecture

Such a formal representation can support the validation of an architecture to structure software components. The effective CL between particular components should match their respective required characteristics (e.g., criticality, responsiveness).

To subsume, different concepts can be combined to address the mixed requirements of different software components. For this research, their structuring relies basically on isolation using segregated containments within the runtime environment. While the concepts are different, they all reduce temporal dependencies to improve the components' compatibility by adequate preparation of their runtime environment. Simplified, this means not the component must be adapted to the system – the system can be (statically) adapted to the components. This leads to the need for an adequate base system (i.e., software framework) that abstracts the respective peculiarities of the concepts to encapsulate their complexity and improve usability for developers and integrators. Admittedly, abstractions that, for example, both run and manage an AMP-based multi-OS and provide high-level thread abstraction might not be incorporated into a single software library. However, inter-OS synchronisation mechanisms necessary for a multi-OS environment are well suited for such a software framework that enables the integrator to have the full degree of freedom to deploy a particular component using the best suited concept.

For this research, a subset of the described abstractions were incorporated into OpenICM as described in Section 4.2.3. This implementation relates to PU-affinity-based EDs to achieve CL1. The thread abstraction mechanism of OpenICM allows for subdivision of the architecture into different (development) domains, depicted in Figure 4.19. With the static definition of the software component's runtime configuration within a 'Context Description' from its actual implementation, an integrator effectively can adapt predefined system behaviour to the system. The configuration includes the PU affinity to define the particular component's ED. This is possible without the need for modification of either the base system (i.e., the software framework) or the component's internals. The implementation of the PU affinity is encapsulated within the framework's thread abstraction that puts the configuration into action, separating the configuration from the application. All this is assembled during compile time to keep the overhead low during the system start. Further, this architecture does not require any OS or kernel modifications and hence is portable to multiple platforms.

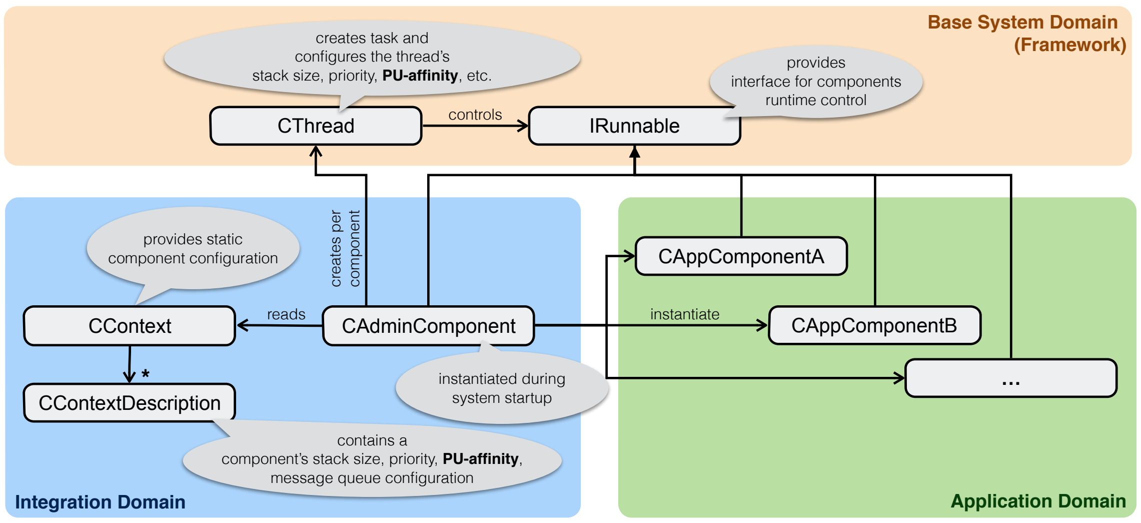
Base System Domain (Framework)
creates task and configures the thread's stack size, priority, PU-affinity, etc.
provides interface for components runtime control
CThread
controls
IRunnable
provides static component configuration
creates per component
CContext
reads
CAdminComponent
*
CContextDescription
instantiated during system startup
contains a component's stack size, priority, PU-affinity, message queue configuration
Integration Domain
CAppComponentA
instantiate
CAppComponentB
...
Application Domain

Figure 4.19: Segregated domains using OpenICM's runtime control of software components

Besides the implementation of EDs, a VM based multi-OS environment (CL1/CL3) was setup, detailed for the 'pilot case' in Chapter 7.

4.5 Summary

With focus on the application layer and the anticipated increased complexity of future ICM systems, the structuring of components and their necessary communication is of great significance. The management of concurrent resource access is identified as a significant issue when structuring integrated systems that rely on components of mixed criticalness which have to fulfil a certain timing behaviour and were developed independently without inside knowledge of the other components. The management of shared computational resources and the handling of external events are necessary features of an infrastructure for complex component-based software systems such as ICM systems.

As part of this research the concept of ED for partitioning software components is introduced and proposed for next-generation ICM. The utilisation of MC hardware architectures to retain the structure of the software system by use of ED can support the compatibility of heterogeneous software components. With EDs, the available computational capabilities are partitioned to create temporal isolated scheduling domains.

A software framework supports the development process with a tested and approved infrastructure, which a concrete application can build upon. The proposed architectural concept is lightweight enough to be usable in practice and proved practicality by adopting it to OpenICM. However, the problems of concurrency remain for unique peripheral resources shared by multiple computational resources.

5

Arbitrate resource utilisation

Following the concept proposed within the previous chapter, the availability of multiple PUs forms the basis for an approach to structure software components through the use of EDs. The utilization of a single OS avoids additional overhead introduced through a virtualisation- or AMP-based approach for structuring components. With EDs, formerly physically isolated software systems can be integrated into a highly integrated head-unit. The temporal behaviour of the targeted system also becomes predictable for high system load situations.

Even though MC platforms provide multiple computational cores, there are still a number of resources that are only available once. Access to such shared resources is realized using a concurrent behaviour, meaning each accessing component has to compete with others for the shared resource. On a single-PU system, such access is implicitly arbitrated by the task scheduler and controlled using thread-priorities: access to a certain resource is only granted as long as the accessing thread is scheduled for computation by the scheduler and when the state is changed to ‘running’. This relies on the limitation that in a single core system, only one thread in the state ‘running’ is possible. If there are multiple PUs available, multiple threads can be computed in parallel and they can also potentially access the same resource in parallel. The temporal order of the latter is not deterministic. This applies in particular for multiple available thread scheduling domains (cf. Section 4.4). This affects the temporal behaviour and predictability of those accessing tasks. If a low critical component reserves a shared resource that is also requested by a high critical component, the high critical component is delayed. This is comparable to the issue of ‘priority inversion’ in task scheduling. Although a guideline for strict partitioning to obviate concurrent access to a shared resource from mixed criticality components might be the most obvious solution, it is not applicable for all resources. This applies in particular to such which are only available once or in less quantity than (potentially) needed. Therefore, the correct functioning of time-critical tasks cannot be guaranteed, especially in high system-load situations. This might lead to the perception that the use of single core systems can ease the integration of multiple software components due to implicit arbitration using the operating system's task scheduler. However, with such systems, the 'bottleneck' is the scheduling of tasks using incompatible priority schemes and scheduling strategies, not forgetting to mention the insufficient computational power. The move to MC systems is unavoidable for CPU-intensive parallel applications, which implies increased complexity in system design. The consequence of this evolution to MC architectures is new challenges which, according to Torres et al. (2011), make it impossible to design current systems using the same approaches applied 20 years ago. They further motivate the need for synchronised and protected access to shared memory in multiple-instruction architectures according to the classification of Flynn (1972), which can be enhanced to shared resources. This basically implies the consideration of the move of 'the bottleneck' (i.e., concurrency) during system design, where multiple tasks compete for a single resource in parallel. A configurable arbitration of such concurrent access is not available for the OSs usually used for ICM.

In the following, an approach is presented to arbitrate the access to shared resources in a parallel computer- and component-based software system that is configurable for the integrator. The main objective is to improve the predictability of the temporal behaviour, especially for high system-load situations and therewith to improve the reliability of the integrated ICM system. For this purpose, further detail is included on the problem definition, and a set of requirements for a resource arbiter are specified. A prototype has been implemented based on these specifications which is also presented.

5.1 Parallel computing versus shared resources

The arbitration of resource access for shared resources is not a new field of research (Manfred Broy and Streicher, 1991). Generally speaking, for systems that rely on different tasks and must fulfil a certain QoS, the management of resource access has to be done at some point. This can be achieved by interrupting the processing of an accessor, or at the other extreme: delegated to the targeted system's user, such as, multiple different audio streams consumed by the user in parallel. If the arbitration has to be predefined, based on the system requirements, the access control needs to be reflected by the system. With multiple computational resources and accessors, this control has to be available for the integrator. Hence, the utilization of MC systems in combination with component-based systems requires a practicable solution to manage the temporal order of concurrent access.

With MC hardware platforms for structuring heterogeneous software components, the parallel utilisation of resources in addition to the PUs has to be considered. Buttazzo (2011, p31) provides the following definition of a resource:

> *"From a process point of view, a resource is any software structure that can be used by the process to advance its execution. Typically, a resource can be*

a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be private, whereas a resource that can be used by more tasks is called a shared resource."

The use of MC platforms reduces the competition for computational resources of the CPU due to increased number of available resources (i.e., PUs). But other resources are still available only once or can only be accessed by one software component at a time. This may include memory (e.g., RAM, NOR/NAND flash, HDD), graphical processing unit (GPU), network interfaces, devices attached via serial communication channels, or various other I/O devices. For single-PU architectures, the OS's scheduler grants processing capacity to threads, which implicitly control the access to resources by use of the configured priorities for the threads. This means the priority assigned to threads affects the order of access to shared resources with only a single PU available. This is not applicable to MC architectures, because threads are able to compute at the same point in time independent of the respective priority, as long as they are scheduled for different PUs, as illustrated in Figure 5.1. Here the access order related to shared resources is non-deterministic and follows a FIFO policy, independent of any affiliations to a particular software component or thread priority. The scheduler is still limited to managing reservations for the PUs. This applies to both static PU affinity and dynamic assignment based on load-balancing algorithms as facilitated for SMP by an OS scheduler. This situation exacerbates with the number of accessors that compete for a shared resource such as for flash file systems, fieldbus transceivers, etc.

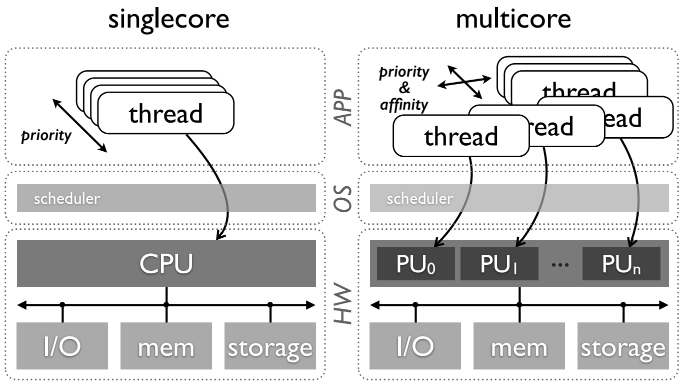

Figure 5.1: Implicitly scheduled and concurrent access to shared resources

Furthermore, for shared resources that have to be used exclusively for a given point in time a global lock mechanism incorporated to the OS and the related device driver respectively prevents pre-emption of low priority accessors. For MC architectures this also applies beyond the scheduling domain of a given PU. This means a high priority thread that wants to access a resource which is currently being used by a low priority one has to wait until the low priority one is finished. This issue gains significance for systems that rely on components with limited compatibility due to the absence of an agreed scheduling policy or priority scheme. Component containment using segregated scheduling domains, such as EDs and multiple OS instances (cf. Section 4.4) do not provide a solution for temporal management of resource access. Nevertheless, waiting for resources and I/O introduces latency to the software components behaviour and therefore undermines the priorities and scheduling policies of the related threads. Such competition for a shared resource represents a bottleneck that may degrade the system's overall performance.

For shared resources, only limited priority-based access control is usually provided by common embedded OSs. The Linux kernel allows the use of an I/O scheduler for block-oriented devices (e.g., hard-disk drives), mainly targeted to improve the performance for concurrent disk access. The driver of such random access devices implements a job scheduler to hide latencies; as an example, for a hard disk: the I/O scheduler virtualises the disk among multiple outstanding requests from different client applications to reduce disk seeks and improve overall performance (Love, 2010, p297-304). Therefore different implementations are available, even with the ability to adjust request orders by use of priorities. They all have in common that they are only targeted for block devices and do not support stream- and character-oriented resources.

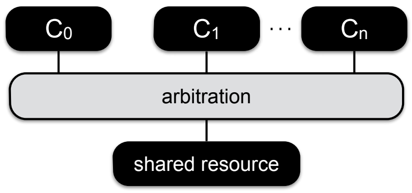

Figure 5.2: Abstraction layer for arbitration of access to shared resource

That necessitates abstraction of the shared resource in form of an additional scheduling instance, as depicted in Figure 5.2 for parallel-executed software components that compete for a single resource, derived from Schranzhofer (2011, p55). This instance has to provide access to shared resources other than the PUs for defined groups of tasks, based on predefined priorities. For the components, the resource appears as having direct access. Although such abstraction has similarities to virtualisation approaches, an arbiter that can be configured to manage the access on the granularity of a single shared resource is beyond the abilities of current virtualisation solutions. This allows and forces an integrator of the system's software components to set priorities that are based on the semantics of the components, independent of their internal prioritisation of threads and implementation details. Resource control can be implemented at integration time to reflect the required behaviour of the system as derived from the specified use-case scenarios.

5.2 Requirements for a resource access arbiter

Before defining requirements, the designated environment and derived limits of the herein proposed arbiter are detailed. Basically, the concept was initially developed on the assumption that the underlying OS features an SMP-based scheduling. Herewith additional overhead costs due to multiple instances of OSs are circumvented in comparison to AMP or virtualisation-based system designs. However, this does not imply the concept is limited to SMP. On the contrary, the herein proposed approach is generally also applicable to such architectural designs, if not even required when using such. This means within AMP and virtualization-based systems, concurrent access to shared resources might be necessary, depending on the integration density and hence the shared use of common resources.

Further, the static scheduling of tasks is presumed to improve predictability and therefore maintainability due to simplified analysis capabilities during runtime. Additionally, these can be processed principally within their Worst Case Execution Time (WCET), although this condition might be limited to a certain threshold due to the mix of time- and event-triggered task characteristics. This implies the software system is a 'schedulable taskset' as long as the competing access to shared resources is not considered and the system is not operating within a high-load scenario. This basically implies that during such a high system load, neither low prioritized tasks nor low prioritized access is scheduled for computation. The use of dynamic scheduling (e.g., based on deadlines calculated during runtime) would improve the efficiency with regard to processing cores and resource utilization, but decrease predictability. Within the context of ICM systems that employ various tasks of differing importance, triggered either by time or events and clustered into components that are developed in parallel by independent organizations, the need for predictability prevails to foster a deterministic temporal behaviour of the overall system. This is further supported by the problematic and costly maintenance of vehicular systems after they have left the production line.

The shared resources addressed here exclude multiply available general purpose PUs. Those are managed by the OS scheduler. The focus for the arbiter is set on I/O devices, such as, automotive fieldbus connections like CAN and MOST, serial connections, files and file systems respectively. It is further presumed that these resources are utilized using an OS and available hardware drivers.

Based on this environment, the requirements for next-generation ICM systems are enhanced. Basically, the system is required to provide a certain degree of determinism related to resource-access latency to support a predictable temporal behaviour of the accessing threads and components respectively.

REQ-6 The latency related to the access of a shared resource shall be predictable.

Further, the management of the resource access has to appear transparent for the supplier of the components. The decision as to which component is granted high-priority resource access (i.e., to achieve low latency) is delegated to the integrator. Thus, the access and the management of the access are clearly separated to improve the components' reusability and allow the integration of legacy components.

REQ-7 The resource access shall be manageable without the need for any modifications of third-party components.

Following the prioritized thread scheduling, an arbiter orders the access requests strictly according to the accessor's criticality, expressed by static priorities to improve both predictability and maintainability.

REQ-8 The access to shared resources shall be temporally ordered using static-defined priorities.

This enumeration defines the most essential architectural driver for the implementation of a resource arbiter.

5.3 Strategies for accessing shared resources

Similar to the policies implemented by a task scheduler (cf. Section 4.2.1), an arbiter follows a particular strategy that defines behaviour regarding contention for a certain shared resource. To detail strategies, at first related characteristics for a particular resource access are to be defined.

The resources are considered to be stateless. The access to a (shared) resource requires a certain amount of time for completion and is modelled following a division into three distinct subsequent phases. These are employed as 'request', 'in-use' and 'clearance' (cf. Figure 5.3). The phase 'request' is related to the management and basically refers to the introduced overhead due to arbitration. It differs related to the particular access prioritization (if available) and applied arbitration policy. The phase 'in-use' refers to the timespan where the accessor performs the particular operations on the resource, while 'clearance' covers efforts regarding the return of a success value and data. An access is granted for - at the most - one accessor at a time, which implies necessary blocking for any other succeeding access until the one being currently processed is 'in-use'. Hence, the 'in-use' phase represents a critical section. Furthermore, an access at phase 'in-use' cannot be pre-empted. This means the completion of phase 'request' is delayed at least until the current accessors finish 'in-use', and at most until no other accessor is 'in-use'. The contention is handled within the 'request' phase. If more than one accessor is in phase 'request', it depends on the applied policy as to which accessor actually enters 'in-use'.

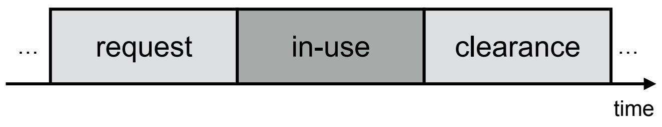

Figure 5.3: Modelled phases of resource access

A strategy that provides good predictability and hence offline computable schedulability analysis for time-triggered or sequential resource access is synchronous Time Division Multiple Access (TDMA) (Schranzhofer et al., 2010). A resource is repeatedly assigned to an accessor for a predefined time slot with monotonic frequency. Hence, TDMA is a static arbitration policy, as its parameters 'time slot length' and 'time slot frequency' are usually defined before runtime and do not change. This requires a minimum TDMA time-slot of the maximum 'in-use' phase for a particular resource, as the resource access is considered to be not pre-emptible. A TDMA policy provides deterministic (contention caused) interference in accordance to a linear complexity. Also, the delay within phase 'request' scales linear with the number of accessors and time-slots respectively. In contrast, it potentially degrades overall performance, as time-slots are assigned to accessors that actually have not requested an access (for the current slot). Although such behaviour might be optimized by asymmetric assignment of time-slots to accessors which might incorporate the criticality of accessors, it is still challenging to forecast when and in which order the resource is accessed. This applies even more for multi-sourced software components and (user-)event-triggered resource access. Therefore TDMA is not an appropriate solution to improve composability for the domain of ICM, although it might be applicable for software components that have to fulfil (hard) real-time requirements while sharing resources. Schranzhofer (2011) provides in-detail assessment on TDMA within the context of resource arbitration for time-triggered and sequential tasks.

An alternative strategy is First Come First Serve (FCFS), also referred to as First In First Out (FIFO). Accessors are queued and processed subsequently according to their temporal order of 'requests'. This dynamic strategy adapts during runtime to occurring 'requests' and is performance-efficient, as no 'time slot' is reserved for 'potential' accessors. However, as FCFS does not incorporate any priorities to reflect the accessors mixed criticality, the latency also due to the still-necessary blocking within phase 'request' is not predictable as it depends on the current number and 'in-use'-duration of earlier queued accessors. The latter is related to the availability of a priori knowledge regarding point in time, order of access and potential differing duration of 'in-use' phases. As this cannot be assumed, FCFS does not provide adequate features for implementing an arbiter for multi-sourced event-triggered systems of mixed criticality.

Another dynamic strategy is Round Robin (RR) with fixed priorities. Comparable to the RR scheduling policy for OS tasks, 'requests' are queued according to their occurrence and priority level. The priority level is assigned statically before runtime, while the order of processing for a single priority level follows an FCFS strategy. The queued 'requests' of lower priority are not processed as long as higher-prioritised accessors are waiting for 'completion'. Considering the above detailed characteristics regarding resource access, an accessor 'in-use' cannot be pre-empted by another requesting accessor. This also applies to scenarios where a low-prioritised accessor actually is in phase 'in-use' while a high-priority accessor enters 'request'. The low-prioritised must at least enter phase 'completion' before the high-prioritised is granted access to the shared resource. However, the subsequent processing of resource access is ordered according to predefined priorities while fostering performance efficiency to ensure that no 'time slots' are wasted. Although the complexity of the strategy is higher than a static policy, RR arbitration can improve predictability due to static priorities even for unpredictable points in time, order and duration of resource access. This qualifies RR arbitration for the use within the context of mixed-critical and multi-sourced ICM systems.

A strategy that mixes static and dynamic components to form an adaptive policy is derived from the FlexRay protocol. It relies on both a static and optional dynamic segment. Monotonically repeated slots following the strategy of TDMA are on demand succeeded by a fixed number of slots that follow an FCFS policy (so called 'mini-slots') (Hagiescu et al., 2007; Lukasiewycz et al., 2009; R. Schneider et al., 2011). Hence, both deterministic and dynamic requirements are addressed. However, FlexRay is intentionally targeted for fieldbus communication where slots are represented by segments within a fixed-size message to achieve real-time behaviour in data transport. Adapted to arbitration, the TDMA segment is utilised for accessors that require real-time access, while a certain amount of TMDA slots is used to make use of a different strategy. The inherent complexity related to the real-time accessors is low, which does not apply to the FCFS slots. Such a strategy might be promising for a shared resource with many competitors. However, the overall complexity reduces the predictability, at least for those accessors that rely on the use of FCFS slots. This is emphasised for accessors that employ both TDMA of the first segment and FCFS of the second segment. That affects reproducibility, analysability and maintainability of the configuration, which has to reflect the predefined mixed criticality within the resource arbitration. That leads to further challenges regarding multi-sourced ICM systems, also with perspective on dynamic functionality.

Other dynamic strategies like EDF or Least Completion Time (LCT) consider the maximum allowed response time or respectively the estimated 'in-use' timespan to order the resource access (Jaouani et al., 2012). Although these may provide adequate results in particular for multimedia scheduling (i.e., soft RT), these strategies do not reflect a mixed criticality of concurrent accessors. This means they may appropriate for accessors of common criticality that have to fulfil dynamic temporal requirements. Due to their dynamic prioritisation, their behaviour is difficult to reproduce and hard to analyse. Accessors from different software components may define deadlines and 'in-use' independently of their respective criticality. As long as a feasible schedule exists, EDF is able to find it (Buttazzo, 2011, p59). But for multi-sourced ICM, there is not a guarantee that a feasible schedule during high-load situations exists. For such, the respective deadlines are second-order, whereas a static configuration reflecting the components' criticality is top-tier. Comparable to the deadline for EDF, this also applies to LCT and the then secondary completion time (i.e., duration of 'in-use' phase).

To summarise, arbitration strategies can be classified 'static', 'dynamic' and 'adaptive', whereas the latter combines static and dynamic components (Schranzhofer, 2011). Static arbitration policies are generally related to linear complexity, good repeatability and hence good analysability for schedulability. This makes the behaviour predictable and allows for offline analysis of WRCT regarding the access to shared resources. However, these are mainly applicable for time-triggered or sequential task models without requirements for responsive processing of event-based trigger. Whereas static arbitration is related to time-triggered or sequential resource access, dynamic arbitration covers event-triggered access. Dynamic arbitration strategies increase complexity and reduce predictability, also affected due to decreased repeatability. Hence such dynamic approaches imply more challenging analysability. Although such policy may improve performance, it also decreases isolation due to increased potential interference related to mutual blocking, causing increased contention during accessors' 'request' phases. Dynamic arbitration can be further subdivided into dynamic priority scheduling (DPS) and fixed priority scheduling (FPS) (Jaouani et al., 2012). While DPS depends on deadlines and estimated 'in-use' phases calculated during runtime, FPS utilises fixed configured priorities to reflect the accessors' mixed criticality. Although the latter requires (manual) configuration, it enables the integrator to directly affect the respective strategies during runtime to realise a predefined access order.

For this research project, dynamic arbitration with FPS represents a promising approach for addressing performance efficiency and providing adequate means for configuration or accessors related to the integration of mixed-critical and multi-sourced software components. Based on the constraint regarding non-pre-emptive resource access, a short 'in-use' phase has to be considered.

5.4 SHARB: a prototype architecture and implementation

The specified requirements were effected and validated for applicability by use of a prototypical implementation, the Shared Resource Arbiter (SHARB).

It provides a solution for managing resource access by abstracting the access. All accessing instances have to utilise this managing resource and must comply with its respective interface. This is a common approach within domain-specific software frameworks that offer abstract services to a higher-level application layer. It does not presuppose a modification or even reimplementation of legacy or third-party components to fulfil the respective interfaces of the resource abstraction.

SHARB both utilise OpenICM as software infrastructure. SHARB not only makes use of the existing facilities of OpenICM, but it is also a suitable enhancement to OpenICM to provide the necessary abstraction for a predefined temporal behaviour of the concurrent access to shared resources.

5.4.1 Architecture constraints and design decisions

With consideration of REQ-6, SHARB avoids the use of dynamic memory and employs efficient communication facilities for internal synchronization and data transfer. Therefore, the abstractions provided by OpenICM are utilized for shared memory, message queues and the parallel execution of tasks.

The interface to the application layer on top of SHARB as well as the interface to the operating system below conforms to the POSIX API to achieve portability. With the use of the library interposition mechanism as presented by Nakhimovsky (2001), the arbitration layer introduced with SHARB is hidden to software components that access managed shared resources. Applying this mechanism, also referred to as interposing, introduces an intermediate layer that provides capabilities to modify, prevent or substitute the functionality of referenced libraries. This appears transparent to both the caller (the application) and the callee (library), only based upon the configuration of the loader and runtime linker. The loader overloads resource-access relevant symbols during runtime for the dynamic binding with the use of symbols provided by the interposing resource arbiter. This means SHARB forms an additional layer on top of the operating systems and system libraries to intercept certain calls, reinterpret them and redirect them to available system libraries where appropriate. This obviates any functional changes regarding the access to resources from the application layer. Hence, through the use of SHARB, no modifications to the already-existing software are necessary. This implies there is no recompilation of supplied binary software required, with respect to REQ-7. Further, no change within the layer of the operating system is necessary because SHARB operates in user space. In combination with its conformance to POSIX, this eases a port to other system platforms.

Through OpenICM as an infrastructural software framework for the application layer, the creation of threads is abstracted and unified. This includes the association of contextual data of a particular software component with the thread by use of the POSIX API. Thereby SHARB is able to identify the context of a particular software component by the implicit identifier of an accessing thread. In combination with the identifier of the accessed resource, SHARB determines the priority of a certain access based on statically-defined priorities as part of the configuration associated with the software components' contextual data.

5.4.2 Architecture and functional principle

In the following section the internal functional principles of SHARB are illustrated to describe the arbitration of concurrent access to shared resources. The essential architectural components are depicted in Figure 5.4. Applicational software components are allotted to the herein proposed EDs. For the prioritization of resource accesses, the Device Manager (DM) delegates all relevant calls from EDs and affiliated threads to a Service Driver (SD). Relevant calls include primitives like open, read, write and close, as well as those used for the control and initialization of resources as defined within the POSIX API. For each association between an ED and a resource, a dedicated SD is created which is executed as a thread within the context (i.e., process frame) of the associated ED. An SD is connected to a Device Instance (DI) to actually perform the access to the resource abstraction provided by the OS. The details of the access to a certain resource are encapsulated within its associated DI which is executed as a thread in an independent context. This corresponds to the ‘in-use’ phase introduced in Section 5.3.

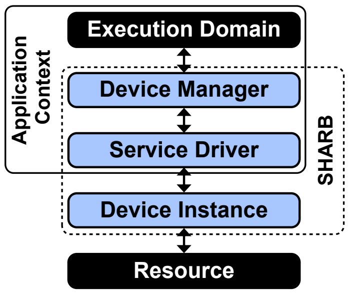

Figure 5.4: Architectural layers of SHARB

Within this context, a thread-pool manages standby worker threads which receive prioritized jobs (DI-job) triggered by events. A DI-job represents a read or a write access to a certain resource. With the use of the thread-pool, additional costs for thread creation can be avoided during runtime. The communication between DM, SD and DI relies on shared memory, POSIX message queues and binary semaphores. The prioritization is realized using static thread priorities given to the DI-jobs and the SDs, whereas all DI-jobs and SDs associated to a common DI are bound to a common PU. This PU affinity supports both a deterministic order of resource access and an efficient communication within SHARB through the use of a common cache memory hierarchy. The foundation for the prioritization of DI-jobs and SDs is the utilization of the OS’s CPU scheduler by use of an RR scheduling strategy with fixed priorities. Additionally, the messages between the components are marked with priorities that are considered during their handling. The use of RR scheduling does not imply the accessor must also be configured for RR scheduling. Nevertheless, this may require the segregation of SHARB using a separate scheduling domain depending on the compatibility of scheduling policies and defined priorities, as detailed in Section 4.4.

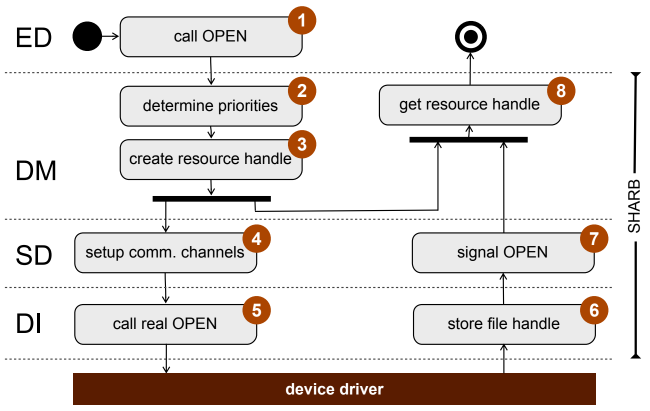

Figure 5.5: Main activities to process a call to open()

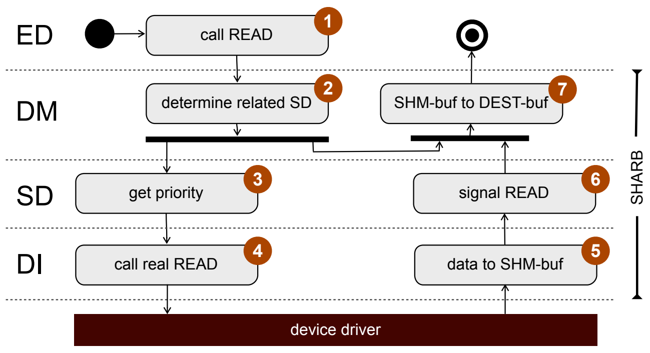

Figure 5.6: Main activities to process a call to read()

Figure 5.5 and Figure 5.6 provide an alternative view of the prototype's design. They depict the activities for two exemplary scenarios for a call to *open()* and *read()*. These are partitioned into the elements of the arbiter to denominate the execution contexts of the respective activities.

For an *open()*, the effective access priorities are determined, depending on the calling ED and the requested resource, and a logical handle is created which is used as the identifier for the association between ED and the resource (Figure 5.5 (activities 2-3)). After creating a new thread for the SD, communication channels are established for internal message events, followed by the call of the *open* routine using the device driver API from the DI (Figure 5.5 (activities 4-5)). The DM is blocked until the result of the 'real *open()*' is signalled. Subsequent calls to *open()* for the same resource will not affect the device driver as long as there is at least one active SD.

For a *read()*, the DM delegates the call to the related SD by use of the logical handle which was created during the *open()* (Figure 5.6 (activity 2)). The SD is already aware of the access priority which is used to prioritize an available worker instance of the thread-pool within the DI (Figure 5.6 (activities 3- 4)). The DI copies any read data to an internal shared memory buffer (SHM-buf), which is copied into the destination buffer (DEST-buf) by the DM on the signal from the respective SD (Figure 5.6 (activities 5-7)).

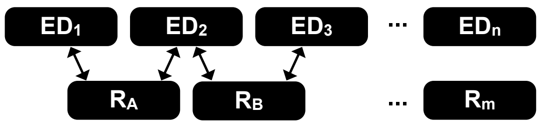

Figure 5.7: Resource access without arbitration

In Figure 5.7, three exemplary EDs are depicted which access two resources. It illustrates the co-operation of multiple accessors competing for shared resources (R). For this scenario the following assumptions are made:

- ED_1 utilizes R_A
- ED_2 utilizes both R_A and R_B

- ED_3 utilizes R_B

This means R_A and R_B are shared resources. The order of access is not defined for EDs executed in parallel, each on a different PU.

SHARB introduces a new abstraction layer as depicted in Figure 5.8. For each shared R, a separate DI is created during initialization. Further, for each association between ED and R, a separate SD is created during runtime. Also, for each DI n connections are handled during runtime. This implies a management overhead by means of an extra thread for each SD, for each DI and for each DI-job that is held in readiness by use of the thread-pool.

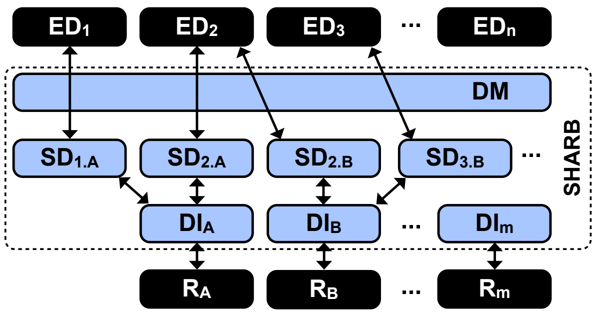

Figure 5.8: Resource access with arbitration

Derived from the example introduced in Figure 5.8, the effective threads for the access of ED_1 and ED_2 are depicted in Figure 5.9 with their SHARB Priority Levels (SPL). The SPLs are a concept to order the access-priority in dependency of the relation between a certain ED and R. The OS's task scheduler implements this concept. SPL-0 supersedes all other SPLs and is reserved for the management of the effective resource access effected by DIs. The subsequent SPLs correspond to the respective access priority derived from the relation between ED and R. A single SPL represents a task queue. The task queues related to a single resource must be bound to a single processing core.

Figure 5.9 illustrates both the assignment of threads to SPLs and the partitioning into distinct scheduling domains using PU_1 and PU_2. Therefore, the behaviour based upon the initial example shown in Figure 5.7 is refined by assuming the following:

- ED_1 has higher access priority than ED_2 for R_A
- ED_3 has higher access priority than ED_2 for R_B

For the OS's task scheduler, the threads of the arbiter are stringed on different priority scheduling queues as depicted in Figure 5.10. The DI-jobs represent effective access calls to the corresponding resource (i.e., ED_1 is accessing R_A two times; ED_2 is accessing R_A three times; both are assigned to PU_1). The OS's priority levels utilized for the SPLs may supersede the priorities assigned to EDs which are collocated on the same processing

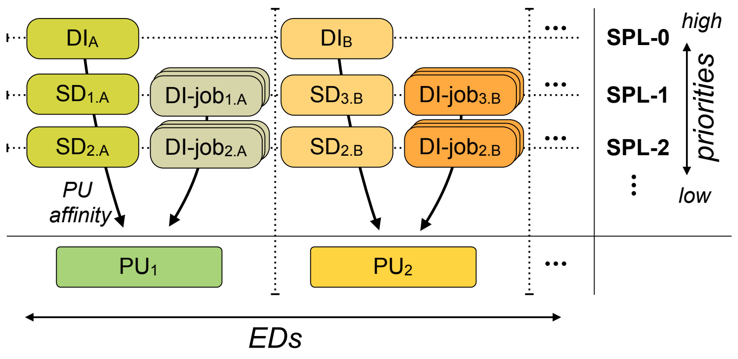

Figure 5.9: Prioritization of resource access using SPLs and PU affinity

core. Although this is not a prerequisite, it prevents unwanted temporal interference between SHARB and the applications. Alternatively, dedicated PUs might be reserved for SHARB, as for example general-purpose cores with a reduced clock rate or features within a heterogeneous MC hardware architecture (a.k.a. 'big-little' architecture).

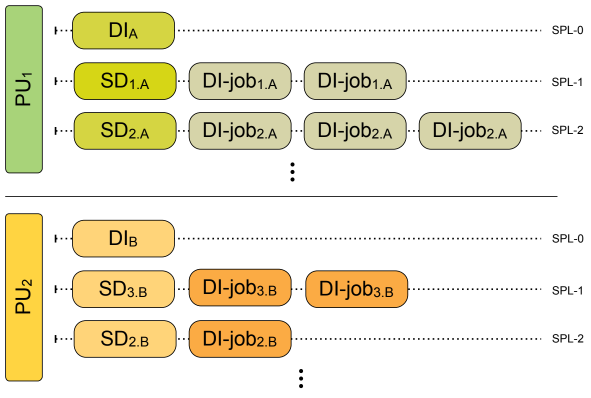

Figure 5.10: Prioritized scheduling queues per PU

5.4.3 Impact

With the described approach, the access to resources can be prioritized to achieve a more predictable temporal behaviour. A prerequisite is the definition of a static configuration which provides the capability to specify priorities for resources (and groups of them) in need of the accessing software components. An access priority is not specified for an ED or a resource, but for a combination of both of them. Hence, an ED could make use of different priorities for different resources. This offers the required degree of freedom to the integrator which is necessary to achieve predefined temporal system behaviour without the need to modify the implementations of the accessing software components.

Further, it is possible to arbitrate the access to a selected set of resources. This implies that additional costs for the management only occur when necessary. With this selective deployment of SHARB, the overhead introduced through the additional indirection is kept to a minimum, with the focus being on the overall system. It is also possible to combine SHARB with the implementation of another resource arbiter through selective resource configuration.

Additionally, the attributes and strategies used to access a specific resource can also be optimized using the decoupling of software components and the resources. This means, for example, the use of a new hardware with legacy software is feasible without the need to change the software. Apart from the demultiplexing of accesses, SHARB can enforce an optimal configuration for the abstracted resources or hardware devices (e.g., rate of transmission, byte order, synchronization methods) through the intercepting of control calls. This might provide further freedom during the integration process.

Moreover, the described abstraction provides the capability for achieving different access strategies with regard to the resource or the accessing software component. This applies to read access in particular. The following enumeration lists a selection of such strategies for stream-oriented resources:

- A call of *read()* starts at the position after a previous read() of an access of the same component.
- A call of *read()* starts at the position after a previous read() of an access of any component.
- A call of *read()* returns the most recent data, independent of any previous access.

Furthermore, the abstraction allows for the implementation of filters which may manipulate, discard or add transmitted data that is transparent for the application layer. In addition, the latter could be used to substitute, simulate or emulate unavailable resources in the early development stages and therefore to reduce risks during system integration.

5.5 Applicability of SHARB

The ICM system's interdependent software components require efficient communication facilities, provided by the use of shared memory regions and adequate mechanisms for synchronization like semaphores, mutual exclusions and condition variables. Such an abstraction may include a usable and domain-specific interface for concurrent and parallel processing, as for example by providing capabilities to define EDs and their priorities to enforce temporal requirements. Regarding the access of shared resources, a framework should also enable an integrator to define the temporal order of accessing these to improve the predictability of the system's behaviour. In relation to ICM systems, the targeted system has to provide the functionality in a coherent and uniform way in agreement with the vehicle's user interface design. This supports achieving the goal of providing the perception of an ensemble in one piece. This also includes the predefined behaviour of the system.

With integration into the OpenICM framework, SHARB becomes usable for a system integrator. The exploitation of the underlying OS's RR task scheduling provides mature means for access arbitration. This includes the applicability of available technology and tools for tracing and debugging.

In the following, the evaluation of the proposed approach based on the described prototypical implementation is discussed.

5.5.1 Test setup

The test system is an x86 MC hardware platform based on two Intel Xeon E5504 processors with four cores each. Even though equivalent performance is not available for current ICM systems, the test environment's characteristics can be compared with next-generation head-units. The operating systems used are GNU/Linux 3.6, GNU/Linux 3.6 patched with PREEMPT RT real-time extension (in the following referred to as PREEMPT RT), and QNX Neutrino 6.5. The shared resource is a dedicated kernel module which implements a character-oriented device driver (char dev) that returns a given number of bytes into a target buffer on a POSIX read request. To simulate latency within a real resource driver and respectively a real device, the module responds to read access with a fixed latency of 50 ms independent of the number of requested bytes or repetitions. Although a real driver or device might not answer using a fixed latency, this testing environment prevents additional variance during the time measurements. This leads to reproducible results. The scope of the evaluation is reduced to the testing of the prototype (and not any driver or device implementations). Further, the module behaves as a blocking device, which implies only one accessor can read at a time.

5.5.2 Prioritization

To evaluate the correctness of the approach, the time to read a predefined number of bytes is measured. Therefore, two components (ED_1 and ED_2) are bound to different processor cores. These components are implemented to read from the described device driver (R_A) concurrently and start simultaneously. The dependencies are illustrated in Figure 5.11. The coordination of the simultaneous start is implemented through the use of a semaphore, triggered for both ED_1 and ED_2. For ED_1 a high- and for ED_2 a low-resource access-priority is configured. Due to the implementation using the library interpositioning mechanism, the arbiter can be activated without much effort by adapting the search path of the dynamic linker. The measurement is performed as a loop to collect values from 10,000 runs to achieve adequate statistical stability.

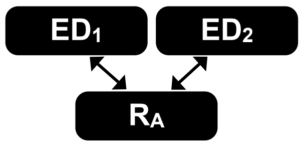

Figure 5.11: Test setup for evaluation of prioritization

Symbol	**Parameter**	**Description**
a_i	Arrival time	time at which a task becomes ready
s_i	Start time	time at which a task starts its execution
f_i	Finishing time	time at which a task finishes its execution
p_i	Preemption time	time at which a task is interrupted due to the arrival of a higher priority task
r_i	Resume time	time at which a task continues its execution after preemption
C_i	Computation time	time needed to compute a task without interruption
D_i	Delay	time introduced through pre-emption ($D_i = r_i - p_i$)
E_i	Execution time	time needed to compute a task including delays ($E_i = C_i + D_i$)
L_i	Latency	time until a ready task starts computation ($L_i = s_i - a_i$)

i corresponds to the related task, e.g. SD or DI-job.

Table 5.1: Definitions of points in time and timespans for scheduling tasks

Scenario	**Computing**	**Arriving**	**Condition**	**Impact**
1	$SD_{1.A}$	$SD_{2.A}$	$f_{SD1.A} > a_{SD2.A}$	L
2	$SD_{1.A}$	DI-job$_{2.A}$	$f_{SD1.A} > a_{DI\text{-}job2.A}$	L
3	$SD_{2.A}$	$SD_{1.A}$	$f_{SD2.A} > a_{SD1.A}$	D
4	$SD_{2.A}$	DI-job$_{1.A}$	$f_{SD2.A} > a_{DI\text{-}job1.A}$	D
5	DI-job$_{1.A}$	DI-job$_{2.A}$	$f_{DI\text{-}job1.A} > a_{DI\text{-}job2.A}$	L
6	DI-job$_{1.A}$	$SD_{2.A}$	$f_{DI\text{-}job1.A} > a_{SD2.A}$	L
7	DI-job$_{2.A}$	DI-job$_{1.A}$	$f_{DI\text{-}job2.A} > a_{DI\text{-}job1.A}$	L
8	DI-job$_{2.A}$	$SD_{1.A}$	$f_{DI\text{-}job2.A} > a_{SD\text{-}1.A}$	L

Impact is either delay (D) or latency (L), whereas D is related to the task's computation and L is related to the task's arrival.

Table 5.2: Permutations of task arrivals for a single scheduling domain

The results of the measurements show that the high-prioritized ED_1 successfully completes read before the low-prioritized ED_2 gets access for every test run when SHARB is activated. With the arbiter deactivated, the number of test runs where ED_1 finishes before ED_2 compared with where ED_2 finishes before ED_1 is uniformly distributed.

In addition to the previously-described empirical methods to prove predictable access prioritization, a theoretical consideration of SHARB is provided in the following:

Therefore, a set of parameters to define points in time and periods of time are provided in

Table 5.1. Furthermore, Table 5.2 lists eight permutations of task arrivals related to the concurrent access of different EDs using different priorities within a common scheduling domain. Hence, they cause pre-emption of, or introduce latency to tasks. The table also specifies the conditions of the concurring tasks, using the symbols of Table 5.1 to support the clear distinction between computing and the arriving task. Corresponding to Figure 2.6, tasks related to ED_1 are of higher access priority than tasks related to ED_2. This means $SD_{1.A}$ and DI-$job_{1.A}$ are configured for a high SPL, whereas $SD_{2.A}$ and DI-$job_{2.A}$ are configured for a low SPL, respectively. A more trivial scenario with only one accessing ED is not considered here because in such a case access prioritization has no affect on the tasks' computation order.

In order to show the correctness of SHARB, three exemplary scenarios selected from Table 5.2 are detailed in the following section. They show either the delay in an already computing task's execution time or a delay in the start of the computation (latency). For reasons of clarity and comprehensibility, any additional latency caused through the tasks' context switches was disregarded. Further, the illustration is reduced to the two fundamental types of tasks derived from the architectural elements of SHARB. These differ in their behaviour regarding pre-emption: the SDs are pre-emptible whereas the DI-jobs are non-pre-emptible. The latter is caused through the DI-jobs main task of accessing the actual resource, which implies the current access must be finished before switching to a subsequent DI-job (although this is highly dependent on the type of the resource and therefore leaves space for further optimization). Following previous notations, Figure 5.12, Figure 5.13 and Figure 5.14 visualize tasks separated by their configured SPL. The focus is on the scheduling order of the subsequent tasks.

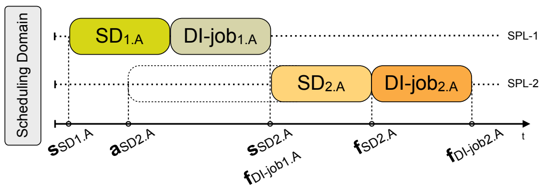

Figure 5.12: Low SPL latency (scenario 6)

$$L_{SD_{2.A}} = f_{DI-job_{1.A}} - a_{SD_{2.A}} \tag{5.1}$$

Figure 5.12 depicts the occurrence of a low-priority access during a computing high-priority access. In particular, scenario 6 of Table 5.2 is addressed here. The low-priority tasks $SD_{2.A}$ and DI-$job_{2.A}$ are delayed until all high-priority tasks are finished. This implies an introduced latency for $SD_{2.A}$ which may affect ED_2 as expressed by Equation 5.1.

$$D_{SD_{2.A}} = f_{DI-job_{1.A}} - a_{SD_{1.A}} \tag{5.2}$$

$$E_{SD_{2.A}} = D_{SD_{2.A}} + E_{SD_{2.A}} \tag{5.3}$$

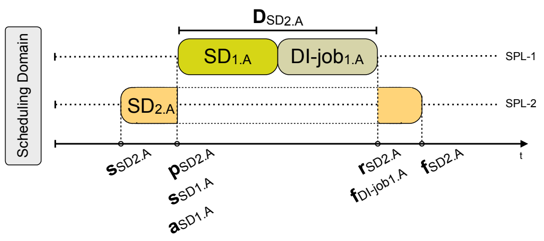

Figure 5.13: Low SPL delay (scenario 3)

Figure 5.13 depicts scenario 3. A high priority SD (and subsequent DI-job) pre-empts a low priority SD. In particular, the task-scheduler of the OS pre-empts $\mathrm{SD}_{2.\mathrm{A}}$ and starts the computation of $\mathrm{SD}_{1.\mathrm{A}}$ immediately after the arrival of $\mathrm{SD}_{1.\mathrm{A}}$. $\mathrm{SD}_{2.\mathrm{A}}$ resumes computation after $\mathrm{SD}_{1.\mathrm{A}}$ and the subsequent DI-job$_{1.\mathrm{A}}$ finishes computation. This causes a delay which may affect ED_2 as expressed by Equation 5.2.

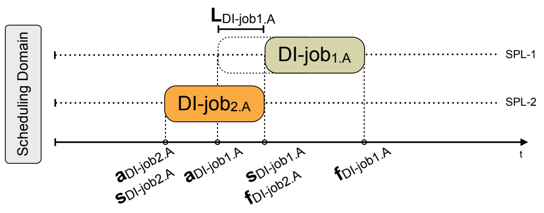

Figure 5.14: High SPL delay (scenario 7)

$$L_{DI-job_{1.A}} = f_{DI-job_{2.A}} - a_{DI-job_{1.A}} \tag{5.4}$$

Figure 5.14 depicts scenario 7. A high-priority DI-job arrives while a low-priority DI-job is computing. The DI-job$_{2.\mathrm{A}}$ locks resource R_A until the current job is finished. DI-job$_{1.\mathrm{A}}$ is delayed until the low-priority DI-job$_{2.\mathrm{A}}$ releases R_A. The maximum latency for a high-priority DI-job is the computation time of a single low-priority DI-job. Respectively, this may affect ED_1 as expressed by Equation 5.4.

To summarize, the temporal order of the concurring access to the shared resource is configurable with SHARB. Hence, with its use, the predictability of the behaviour increases and REQ-3 is met.

5.5.3 Temporal overhead

For the temporal overhead a single ED reads from a single R. The results provided are based on 90,300 measurements. These were taken natively (without SHARB/not prioritized) and arbitrated to visualize the costs in terms of temporal overhead. The amount of data (1-256 byte) and the number of repetitions (1-30) are iterated during the data collection, whereas for each permutation 25 measurements were recorded to achieve reliable and evaluable results. In this context, repetitions are related to the number of calls to *read()* between an *open()* and *close()*. This means a single read consists of the call '*open() - read() - close()*', whereas an access with eight repetitions consists of the calls '*open() - 8*read() - close()*'.

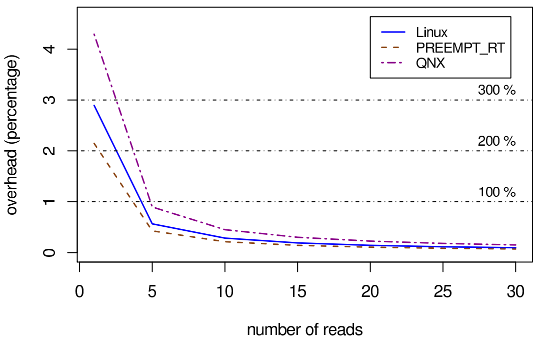

Figure 5.15: Overhead in relation to repetitions in percentages

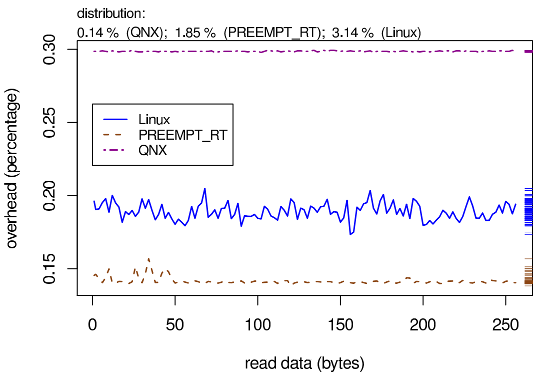

Figure 5.16: Overhead in relation to amount of data in percentages

Figure 5.15 and Figure 5.16 visualize the results as a percentage in relation to non-arbitrated access for Linux, PREEMPT RT and QNX. The respective overhead results from the arithmetic mean of the absolute time measurements for an arbitrated access less the mean of the absolute time measurements for a native access.

Figure 5.15 shows the overhead as the mean of different data sizes in relation to the repetitions. For a few repetitions, a significant percentage of overhead can be observed. This is caused by the initial setup phase to establish the internal administration and communication infrastructure. These temporal costs amortize when accessing the resource five times or more. Further, this effect can be mitigated by the pre-initialization of SHARB (the evaluation was performed by initialization on-demand). Nevertheless, the temporal overhead levels off at about 20% for QNX and Linux.

Figure 5.16 shows the effect of SHARB related to different amounts of data, whereas for each data size the mean duration based on different repetitions of the *read()* access is measured. The overhead with QNX is significantly higher compared to the use with PREEMPT RT. However, the temporal overhead with QNX is considerable more stable in variance and therefore more predictable (distribution with QNX is 0.14%, compared to 1.85% with PREEMPT RT, and 3.14% with Linux).

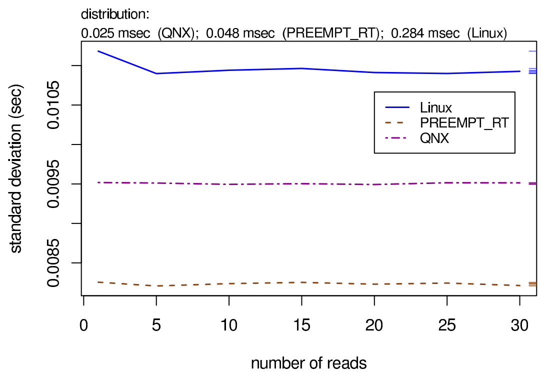

Figure 5.17: Standard deviation in relation to native access and repetitions

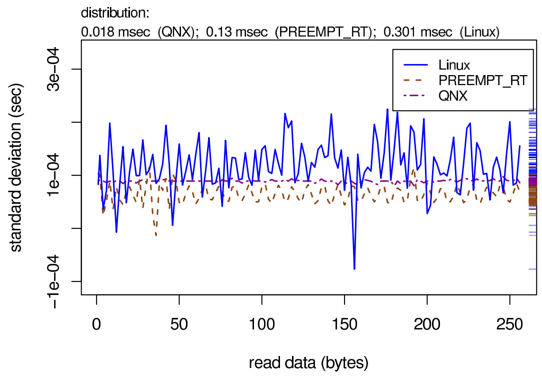

Figure 5.18: Standard deviation in relation to native access and amount of data

Further conclusions about the predictability regarding access latency are founded on the standard deviation. Figure 5.17 and Figure 5.18 show the standard deviation of the time necessary to access a resource using SHARB in comparison to native access, which is also in relation to the repetitions and the amount of data respectively. Nevertheless, the measurements also show that the latency is scattered only slightly more with activated SHARB and is independent of size and repetitions. In particular, the standard deviation built from the arithmetic mean of different sizes for the different number of reads (as shown in Figure 5.17) illustrates a predictable temporal behaviour that is independent of the repetitions of a call to *read()*. For example, both the distribution of the relative standard deviation with SHARB on QNX and also the standard deviation itself is smaller than the measures of SHARB on Linux. However, Linux patched with PREEMPT RT still performs comparably to the measures achieved with QNX.

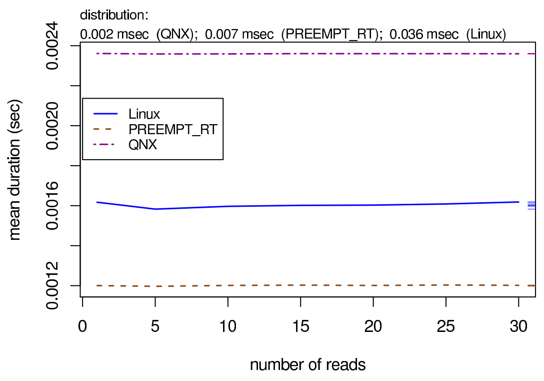

Figure 5.19: Overhead in relation to repetitions

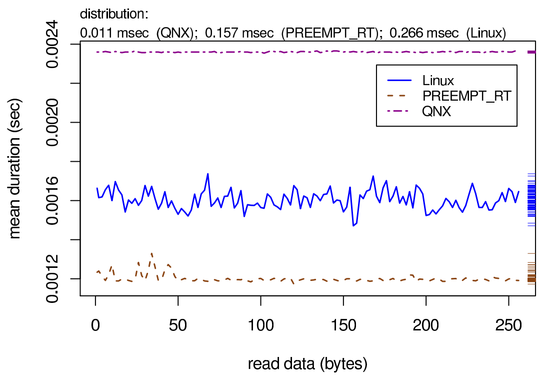

Figure 5.20: Overhead in relation to amount of data

Figure 5.19 and Figure 5.20 provide details regarding the absolute overhead (mean duration arbitrated less mean duration native). They correspond to Figure 5.15 and Figure 5.16 for the observations regarding the comparison of QNX, PREEMPT RT and Linux: with QNX SHARB behaves more deterministically, however introduces more overhead.

Although the additional costs may appear high considering the efficiency of the targeted system, the predictability of the access order prevails with respect to the deterministic behaviour of the overall system. This is the main objective of SHARB.

5.6 Related research and alternatives for resource arbitration

The AUTOSAR API gains importance for automotive software systems (cf. Section 2.4.1). With release 4.0, it also supports MC platforms. The main focus of AUTOSAR is on mechanisms regarding the communication between applications running on different PUs. The utilization of shared resources by applications that are deployed on different PUs is not supported (AUTOSAR, 2014). This limits the degree of freedom regarding the structuring of software components and is thus a disadvantage for systems consisting of many components such as ICM systems.

According to Bini et al. (2011), the ACTORS project addresses embedded software intensive systems in combination with resource utilization and high demands regarding adaptability and efficiency. This problem domain can also be mapped to ICM systems. The project proposes virtualization technologies for software isolation to improve predictability and reliability. The resource management is implemented in user-space and adapts the resource reservation dynamically during runtime while considering optimal system occupancy. Although this promises good utilization of the system's resources, the resource management does not take into account semantic dependencies in relation to the desired temporal system behaviour. This could be achieved through the allocation of static priorities that are defined by an integrator.

Nesbit et al. (2008) predict that the available mechanisms and strategies for managing the resources of future MC systems will be insufficient. They further describe that a shared utilization of resources by tasks executed simultaneously could lead to unpredictable individual durations of the respective threads related to the involved tasks. This may include the violation of requirements and associated QoS. Similar to the ACTORS project, they propose a spatial separation by use of virtualization and feedback channels for the adaptive management of resource utilization during runtime. Their main focus is on the computational resources of the underlying hardware platform.

Waldspurger and Rosenblum (2012) identify the conflation of concurrent access to physical hardware as a central challenge within the context of I/O virtualization. This includes prioritization and arbitration. They specify a minimal additional overhead for the indirection as critical. Although this is unquestioned, one of the most important characteristics of a resource arbiter for the herein proposed concepts 'overhead' is regarded as secondary in relation to the predictability of fulfilling temporal requirements during high system-load situations.

Schranzhofer et al. (2010), Schranzhofer et al. (2011), and Schranzhofer (2011) focus on efficient and predictable resource sharing using MC hardware architectures. Therefore they motivate the need for arbitration related to the concurrent use of shared resources by parallel-executed processing elements, detail the derived interference on their shared usage, and propose approaches for static and hybrid arbitration policies. Within that context, real-time tasks are allocated on predefined processing elements with emphasis put on the proposed policies, their effect on temporal behaviour (i.e., Worst Case Response Time (WCRT)) by application of different models, analysability and schedulability related to memory access. Interactive event-triggered systems that demand responsiveness are not considered in particular. However, the proposed TDMA and adaptive policy following the FlexRay protocol may be valuable for ICM domains with particular necessity for hard real-time behaviour despite the use of shared resources. Moreover, it might be of interest to implement and evaluate those policies in user-space using the herein detailed arbitration infrastructure of SHARB.

5.7 Summary

The integration of different software components into a common platform implies demanding challenges. MC hardware platforms can provide support but at the same time create new challenges due to the additional parallelism. In contrast to the concurrent multi-tasking (quasi-parallelism), concurrent resource access is not predictable in terms of temporal order within the context of multiple scheduling domains that support parallel computation of low and high priority threads. Hence it is necessary to arbitrate the access to shared I/O resources. Requirements were defined which formed the basis for an arbiter: SHARB. It introduces a thin architectural layer in between application components and system libraries for accessing particular resources. SHARB appears transparent to both the application layer and the libraries underneath, which obviates the need to change either the applications or libraries. Furthermore, SHARB resides within user-space and therefore does not require any changes to the OS. A prototypical implementation supports the verification of the presented approach for arbitration. The applicability and introduced overhead of the prototypical implementation were theoretically and quantitatively evaluated. For the latter, different target OSs are compared.

Although SHARB is targeted for use in SMP-based ICM systems utilizing a single OS, the approach is also portable to other usage contexts and architectures with similar requirements. This includes architectures focusing on a more strict isolation of components. These may also require the management of concurrent access to shared resources based on their density of integration or grade of parallelism. This refers, for example, to AMP systems and architectures based upon virtualization using hypervisors. For certain I/O resources within the context of multimedia, including video and audio, alternative arbitration and compositing technologies are discussed within the following chapter.

6

Compositing User Interfaces

"Design is not just what it looks like and feels like. Design is how it works."
(Steve Jobs)

For the vehicle's occupants, the user interface (UI) is the actual point of contact with the vehicle. The design of such systems has to cover demands for an appealing front end to foster a positive UX (cf. Section 2.2.6). Basically, the presentation and use of 'the system' has to address current state-of-art technologies and concepts for user interaction, even beyond approved in-vehicle usage concepts and UIs. Although in practice, the 'visual design' is usually detached from the implementation as this is the domain of automotive (graphic) designers, it has to be transferred to the functional level as part of the software development process. This includes in like manner - independently of any 'look and feel' the UI has - the necessity of reflecting the system's functional purpose with regard to the safety-critical environment. A central question is how the UI of ICM systems can efficiently exploit the integrated functionality and the capabilities of current hardware platforms. That is in particular related to the provided hard- and software infrastructure.

As previously discussed, IVI systems were rather isolated in the past. Their main task was to provide information and entertain the car's occupants. Nowadays they have become an integral part of the in-vehicle system's network and enable the driver to configure and control automotive functions. Moreover, ICM systems feature the communication node between components attached to automotive fieldbus and infrastructure-based wireless communication networks. The interconnection with other systems within the vehicle as well as the environment enables new services and functionality, including future ADAS (Bolle, 2011). Furthermore, interconnectivity allows future ICM systems to update both data and functionality dynamically during operation regularly or on-demand. With respect to the customary system lifecycle of several years, the UX can be efficiently maintained throughout the entire vehicle lifetime. So far, such capabilities have not been available within the automotive domain. With future systems the user will be able to adapt the functional extent to personal needs or desires. This creates a new dimension of customization, adopted from mature deployment models for dynamic functionality in the CE domain.

Previously, respective implications regarding the compatibility of different functionality were addressed, including opportunities for structuring a priori incompatible software due to parallel hardware architectures. But similar to the necessary arbitration-shared resources, partitioned software leads to challenges regarding the provisioning of an integral and homogeneous UI. Components distributed to isolated partitions have to form a uniform presentation that appears to the user as if the overall system is made of one piece (i.e., non-partitioned). In particular, as each of the components provides 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 solution for current systems is to implement the whole UI within a distinct component that relies on distinct 'functional' components. However, such an approach inhibits the modification and extension of those 'functional' components without adaptation of the UI component. Hence such architecture must be thoroughly reconsidered due to the parallel and independent development, decoupled update policies and intervals, and demands to integrate after-market functionality (i.e., 'apps') on user demand. A system-global UI component that implements the entire visualization and user event handling (the 'frontend') to abstract the systems logic (the 'backend') is no longer possible. A UI for next-generation ICM systems has to provide capabilities for removing, replacing or adding UI components. However, a central UI instance (or UI subsystem) may implement the UI logic that reflects the required usage concept.

In summary, a centralised UI approach is not applicable for dynamic functionality that evolves over the complete product life cycle of the vehicle. This chapter presents an alternative that supports both a change of presented functionality and provided capabilities to enable the realisation of adequate UX to address the demands of next-generation ICM.

6.1 Requirements for an in-car user interface

The segregated computation mitigates risks regarding negative interferences between different applications and error propagation due to an infrastructure-based encapsulation. Still, the software system shares a common hardware platform, including shared resources. The allocation and arbitration of such shared resources potentially cause temporal interference as well. For resources that allow only an exclusive usage at any given time, a priority-based arbiter may lead to more predictable system behaviour as discussed in the previous chapter. Time slicing is not appropriate for shared resources that at the same time facilitate multiple accessing applications. This applies especially to data sinks that allow the blending of data streams, such as video and audio. However, these types of data are significant for building an appealing UI. Hence, these have to be considered for establishing a comprehensive infrastructure relying on the segregation of functionality while improving the UX.

Based on the independent development of the software components, their respective portions of the UI are built independently from the core functionalities. Nevertheless, they have to comply with design specifications and guidelines predefined by the OEM to implement a homogeneous 'look and feel' and usage concept to facilitate consistent user interaction. With a rising number of applications and after-market 'apps', a comprehensive UI component covering all visual presentation and user event handling is no longer feasible. Each application has to provide its own UI to be integrated with – or blended into – the existing ones. This creates a demand for a graphics compositing instance as a segregated component that can cope with multiple graphics sources and the related user interaction (the 'back channel') for user presentation and event handling respectively. Such an instance may act as manager and define what to visualize, where, and in which presentation mode, whereas the graphics sources are segregated software components.

REQ-9 A UI compositor shall provide the capability to blend independent portions of the UI.

A conceptual architecture for such an instance is detailed in the following. The goal is to pave an integration path for independently-developed components while enforcing individual run-time polices.

6.1.1 Architectural drivers for compositing

The following constraints lay the foundation for the architecture of a compositing instance for ICM.

The system's functionality is partitioned into segregated partitions to ensure local run-time policies. These include predefined temporal behaviour derived from given priority policies and priority levels. The intention is to prevent effectively negative interference between different functionalities – or applications – deployed to different partitions. The partitions may be implemented using PU affinity-based EDs, or multi-OS environments using virtualization technologies or AMP. This implies that different partitions do not necessarily share a common OS (or kernel space), meaning that the options for inter-process communication (IPC) are limited. However, efficient communication is necessary to utilise and benefit from the common hardware infrastructure and achieve the necessary QoS related to UI responsiveness.

With respect to the varying safety relevance of different applications, the interoperability between partitions has to meet certain security-related requirements. It has to be ensured that a dynamically installed or updated application cannot cause an error within a safety-relevant partition (e.g., containing the instrument cluster) or the compositing partition.

REQ-10 The UI compositing shall support the distribution of UI components onto multiple OS instances.

Generally, the UI provides multiple input and output facilities to enable a multi-modal interaction with various software components in parallel use. Many of these provide a graphical front end using the car's multi-display environment. Nowadays, appealing UIs usually rely on three dimensional (3D) graphic effects. This may also relate to facilitating a positive UX. Therefore hardware accelerators relieve the general purpose PUs from graphics computation. The individual components concurrently utilise such graphical processing units (GPU). By partitioning the system into several partitions that independently render graphics, a single GPU has to be shared between multiple OSs. Alternatively, only one partition benefits from the GPU, while the others have to render their graphics without acceleration. Both options are unsatisfactory. The lack of the availability of multi-GPU platforms implies a bottleneck with potential adverse temporal effects for parallel computed components that rely on graphical output. For efficient compositing, the architecture may employ several GPUs as accelerators for different partitions and OS instances respectively.

REQ-11 A GPU shall be assignable exclusively to a dedicated OS partition.

In summary, the integrated modular architecture applied to highly interactive ICM systems requires partitioning. This is caused through the functionalities' different safety relevance and hence the need for preventing adverse interference. Demands for uniform and compelling UIs create requirements for efficient graphic processing. Using dedicated CPU cores for segregated computation is no solution as long as more than one partition relies on graphic acceleration. Thus, the utilization of multiple GPUs to consistently maintain the segregated architecture for graphics processing as well is proposed. This implies decreasing computational load for the CPUs related to graphics processing and hence more effective utilization of hardware capabilities. Consequently, negative inter-partition interference is mitigated and additional graphics acceleration for future highly interactive ICM UIs is made available.

6.1.2 Architectural drivers for communication

The isolated computation of different software components presents challenges regarding the inter-partition communication. The latter must support interoperability while preserving the components isolation using PU affinity-based EDs or multi-OS variants. In particular, the conveyed data may follow a protocol that obviates interpretation at the receiver to enforce containment for the different components to improve composability and with respect to reliability (cf. Section 2.3.5). In practice this means the rendering of graphics data is preferably performed at the UI-component provider, and the data transmitted can be directly passed on to graphics hardware without any further processing of any graphic directives at the receiver (i.e., the compositor).

REQ-12 A UI component shall provide pixel frames that obviate further interpretation at the receiving compositor.

The communication's performance efficiency has direct impact on the UI's responsiveness and respectively to the system's reactivity. This is related to the limited communication and synchronisation facilities due to software component isolation for multi-OS environments compared to integration onto a single OS, while considering a predefined QoS. Especially for pre-rendered graphics data, the necessary throughput and bandwidth is of importance, as it directly depends on the number of UI-component providers and particular frame size or graphics resolution.

REQ-13 The compositing architecture shall provide performance efficient communication channels.

Besides visualisation, a GUI has to provide means for interaction. Therefore, additional to the transport of graphics data, the compositing architecture has to provide means to route user events to the corresponding UI component provider.

REQ-14 User-event shall be communicated from the compositor to the designated UI component.

The above detailed requirements are seen as the foundation for an ICM-compositing architecture that copes with the introduced barriers for temporal isolation of software components.

6.2 Related architectures and research

AUTOSAR fosters an independent development using well-defined interfaces to enable integration onto shared hardware platforms. Therefore, abstraction layers help to decouple software from hardware specifics that make it appear as underlying platform to software components. Although the target is very similar to the previously-described segregation, it does not detail compositing of graphics to a shared rendering device. Nevertheless, the concept discussed in the following might be transferred to an AUTOSAR-conform ICM system using the provided API of AUTOSAR. However, as intercommunication with AUTOSAR SWCs is restricted to defined ports and does not provide capabilities such as performance- efficient communication via shared memory, this limits the intercommunicational abilities for a composting architecture with certain requirements on bandwidth.

Similar to AUTOSAR, open vehicular software platforms are intended to create abstraction layers that provide access to hardware resources and offer domain-specific software services. They aim for reduction of application complexity while fostering parallel execution and reuse of software components. However, 'open' implies the platform specification is freely available, which enables everyone to develop platform-compatible software components. Prominent open automotive platforms for ICM are AutoLinQ™, GENIVI and

Ford SYNC® (Holle et al., 2011). A side effect of the open platform trend is the introduction of Linux-based OSs into the vehicle, which is also applied for evaluation of the herein-proposed compositing architecture. Despite the fact that no specific platform is addressed by the latter, it may constitute a beneficial enhancement to them to enable independent UIs. However, GENIVI's 'IVI Layer Management' project addresses compositing and separation of HMI and layer management, but does not yet cover efficient inter-OS UI provisioning for multi-OS environments (GENIVI, 2014).

QNX Software Systems propose their QNX CAR HTML5-based HMI framework to ease integration of applications from CE space using web technologies (Gryc and Lapierre, 2012). A compositing of different UI components might be realized by use of different in-vehicle provisioned web services, each offering a particular functionality. A 'browser' acts as a central compositor. This positively affects the development process through the use of web technologies and may ease the transfer of a predefined design to a working UI. Different service providers could be segregated into dedicated partitions with the freedom to utilize different OSs. However, the major part of the UI's content has to be rendered within the partition of the 'browser'. Hence, there is more computational power required for the compositor which therefore may become the bottleneck. Furthermore, a certain service provider may interfere with a more critical one due to the need for interpretation and computation of the content to visualize, which undermines the concept of partitioning. Therefore it does not provide an adequate solution, although within layered system architectures, HTML5 might be applicable as long as the rendering is performed within a segregated partition.

Eichhorn et al. (2010) do also propose an automotive HMI architecture that relies on web technologies. They structure the UI functionality into distinct services, managed by a database-driven 'service manager'. A 'composer' creates and pre-processes graphical primitives to adjust the component's content to the display features. These graphical primitives are forwarded to and displayed by a lightweight 'renderer'. The complexity for processing the graphical content is concentrated to the 'composer', which may become the bottleneck for the architecture, similar to the drawbacks of QNX CAR HMI framework.

Various graphics already exist compositing window managers for different OSs and providing different features. The latter address, for example, improved accessibility, simplified use and so-called 'eye candy' to enhance UX. One of the more recent developments is Wayland (Høgsberg, 2012), which focuses on a lightweight and efficient internal communication and, therefore, is also applicable to resource-constrained embedded systems. It is also incorporated into the 'IVI Layer Management' of GENIVI. Unfortunately, Wayland does not natively facilitate an efficient inter-partition communication. Nevertheless, it is used for the evaluation of the herein proposed concept. Therefore, fundamental communication components were substituted or enhanced.

Hudelmaier (2014) identifies demands for an integrated HMI within the context for professional drivers to combine instrument cluster with infotainment, fleet management, driver assistance, and UIs for special car bodies and equipment. The proposed approach relies on hardware bus technologies (e.g., LVDS) to interconnect the different UI-component provider. Although this might be advantageous for the graphics data transport with regard to preserved isolation and high bandwidths, it limits the system's flexibility related to the functional evolution during the vehicle's lifetime. Decisions concerning what to display at the level on single 'apps' can only be made on the granularity of a whole system partition. This also affects composability, in particular regarding maintenance throughout the system's life cycle.

Holstein et al. (2015) also address the challenges in compositing heterogeneous UIs for automotive use. They define a layer model to distinguish between different levels for compositing: hardware (1), OS (2), application (3) and UI (4). Depending on the layer, different methods of integration and different needs of knowledge for the composited parts are required. They further explain that specifications within lower layers create constraints for the higher ones, while the latter depend on the lower ones. This herein presented research covers multiple layers of this model, in particular (2) with multi-OS approaches, (3) with inter-OS communication frameworks and ED based on PU-affinity and partly (4) by blending the heterogeneous UI parts while considering the constraints of the underlying layers. Nonetheless, although at layer (4) UI-design guidelines and integral-usage concepts are to be considered to achieve a homogeneous UX, this research focuses on the technical infrastructure to composite graphical content and communicate user events to the respective addressee despite containment of the providing UI-apps. This implies that the herein-detailed research project is rather founded on a technical base to support the implementation of compositing on the UI layer while achieving a homogeneous usage concept and frontend design, which is an active field of research.

6.3 An architecture and prototype for compositing segregated UIs

The design is derived from the architectural drivers defined above. Basically, it consists of three conceptual components as detailed in the following and depicted in Figure 6.1:

- UI application (content provider)
- Compositor (content blender)
- Intercommunication (implements the binding between UI application and compositor)

6.3.1 Conceptual components

A 'UI application' (UI-APP) refers to an independent functionality providing a UI artefact (or UI component). Such an artefact (or surface) may implement comprehensive and extensive menu structures providing access to a set of applications, or only a section of a certain UI screen that has to be blended with other UI-APP's artefacts. This implies that

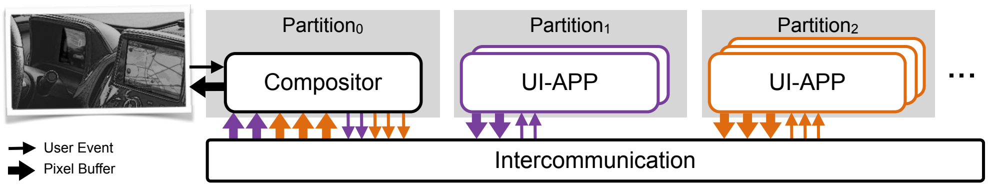

Figure 6.1: ICM UI infrastructure

each application renders its own subset of the UI. UI-APPs are distributed to different partitions, whereas each partition may benefit from a dedicated GPU. The combination of all UI-APPs forms the UI of the ICM system, which means that they represent the source for graphics and the sink for related user events. However, all UI-APPs must comply with the design concept and UI guidelines of the overall system to form a homogenous UI layer.

The 'Compositor' is a super-ordinated instance that blends the artefacts provided by different UI-APPs. Therefore, it may, for example, resize, transpose, and project the graphics with perspective. These artefacts can be regarded as active video streams. This is comparable to applying a texture to a 3D model, whereas here the texture is not an image but an animated and active UI artefact. Active means the UI artefact is still receiving user events (e.g., touch-events). Extensive manipulation of the provided artefacts may demand for a dedicated GPU for the respective compositor. Furthermore, the graphical artefacts are received as plain pixel buffers. This obviates the need to interpret information and hence mitigate security issues such as code injection. However, it also means additional efforts, for instance, to resize the artefacts if the provided dimension or aspect ratio does not fit the display. Nevertheless, using plain pixel buffers has great advantages in terms of loose coupling, maintaining a high degree of freedom for the UI-APPs, but still ensuring compatibility. This means a moderate functional complexity in terms of logic for the compositor, as its main task is to adapt the provided graphical content using primitive graphics processing. Appropriate hardware accelerators can support this to relieve the general-purpose PU. Additionally, the compositor delegates incoming user events to the related UI-APP, comparable with an input-event mapper. An ICM system employs a single compositing instance for each display, communicating with all UI-APPs. Hence, a compositor is aware of what artefacts are actually displayed to the user. Therefore, it also maps generic input events (e.g., buttons on a multifunction steering wheel) that are related to the current system context to the corresponding UI-APP or respective ED. This also applies to input preprocessed by speech- or gesture-recognition. The compositor can also be seen as abstraction for accessing the UI, while the UI is a shared resource. In contrast to other shared resources (cf. Chapter 5) the UI can cope with several accessors in parallel due to blending the content and utilisation of multiple displays. This means the compositor is a specialised resource manager. Due to its important role for visualising UI applications, a compositor is classified at the same level of criticality as the connected UI-APP with the highest level of criticality.

The facilitator of the compositing infrastructure is an efficient communication by use of a shared memory region that is accessible by both UI-APPs and the compositor. Basically, this is needed to transfer pixel buffer information from UI-APPs to the compositor with adequate throughput to achieve predefined frame rates. The intercommunication also transfers user events from the compositor to UI-APPs, which requires low latency to provide appropriate responsiveness.

A central characteristic of this architecture is the location where the graphic primitives are rendered: at the producing UI-APP. This is also a key differentiator to other approved graphics systems which send primitives to a central instance that has to interpret and render them for all applications. Such a central graphics server implies an increased inherent complexity to the system, as the applications that produce graphics primitives are highly dependent on the server's actual graphic features and performance. Such architecture diminishes the respective components' self-containment as their functional interdependency increases. This may have a negative effect on the components' compatibility and hence affect composability. Further, this central graphics renderer constitutes a bottleneck that gains significance with an increasing number of connected UI applications. This becomes even more relevant for MCS that may incorporate dynamic functionality throughout the system's life cycle, such as ICM.

6.3.2 Prototype architecture

The infrastructure as depicted in Figure 6.1 already covers the concept of decoupled content providers, segregated due to the use of three partitions ($\mathrm{P_n}$; with $n = 0 \,..\, 2$). This implies that the criticality of the partition that contains a compositor (here $\mathrm{P_0}$) is at least equal to the highest criticality of the content-providing partitions. $\mathrm{P_0}$ becomes the bottleneck for both UI rendering and event dispersion and hence has to be equipped with adequate computing power and communication facilities. For providing more partitions of different criticality, a hierarchical compositing architecture might be more appropriate. This in particular applies to the relatively wide spectrum of applications within the ICM system domain.

Figure 6.2 depicts an enhanced infrastructure that employs two compositing entities ($\mathrm{Compositor_0}$ and $\mathrm{Compositor_1}$), deployed to distinct partitions. For this example, $\mathrm{Partition_0}$ is of high criticality due to the provisioning of the instrument cluster. It is able to blend UI artefacts of other partitions, provisioned by $\mathrm{Compositor_1}$. Therefore the receiving $\mathrm{Compositor_0}$ has no information about the semantics of the provisioned content. This means $\mathrm{Compositor_1}$ acts as compositor for low-critical UI artefacts (i.e., entertainment, multimedia) provisioned by UI-APPs of $\mathrm{Partition_2}$ and $\mathrm{Partition_3}$, as well as UI-APP for $\mathrm{Compositor_0}$. This allows blending of pre-composited low-critical UI artefacts (created within $\mathrm{Partition_2}$ and $\mathrm{Partition_3}$) with high-critical UI artefacts of $\mathrm{Partition_0}$. The interface between the partitions is reduced to pixel buffer information and predefined fixed-size key events to handle user input and feature synchronisation and flow control of pixel information. Hence functional dependencies are kept to a minimum with respect to the partitioning prevention of adverse interference. The depicted intercommunication component may also be split by use of dedicated shared memory regions to follow the segregation of the UI and application logic.

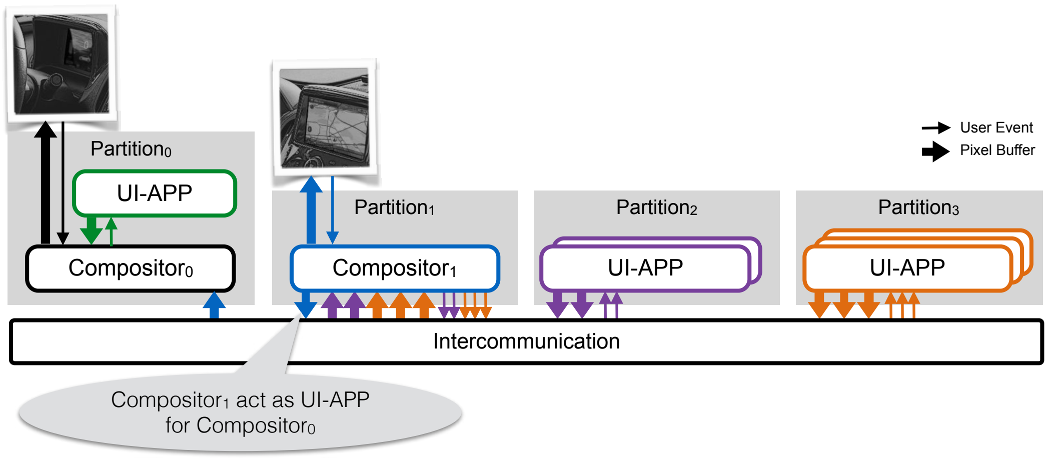

Figure 6.2: ICM UI infrastructure for mixed criticality UI-APPs

Each partition may rely on a different OS to fulfil the different demands of the particular UI-APPs, which may include real-time abilities such as QNX, not connected to the Internet for Partition$_0$ or in contrast a mobile-OS such as Android that allows the use of 'social media' on the same hardware platform. Such 'social media' applications rely on Internet connectivity. Using the proposed ICM UI infrastructure, it is still possible to blend such content into critical partitions[1]. The critical partition's compositor still has full control of how and where to integrate such UI artefacts, even though it has no information about the visualised content's functional details. The loose coupling ease dynamic update of particular UI-APPs without affecting critical (ASIL classified) software components and therefore obviates costly re-validation. Even the update of a particular partition is possible without affecting the others, to support, for example, security updates or maintenance of a mobile-OS. This fosters the ability to maintain the overall system's UX by use of evolving low critical UI-APPs. However, this also provides a means to maintain critical UI-APPs. Such evolution during the life cycle is mandatory for some functionality, which relies on connectivity to third-party service providers with independently-evolving functional interfaces (e.g., web-interface of social-media platforms). For the consumer (i.e., compositor), the content is opaque in terms of merely receiving already pre-computed plain pixel data, handled as textures or frames. In the above example, the update-cycle of Compositor$_0$ is independent of any UI-APP within a partition different from Partition$_0$.

The prototype architecture's components and their relationships can be expressed using the following notation, given that the UI (ICM-UI) consists basically of n UI-APPs and m compositors.

[1]A discussion whether it is sensible to blend particular content like 'social media' into the instrument cluster is not part of this research.

$$\textit{ICM-UI} := (\sum_{1}^{n} \textit{UI-APP}) \cup (\sum_{1}^{m} \textit{Compositor}) \cup (\textit{Intercommunication}) \tag{6.1}$$

$$\textit{UI-APP} \xrightarrow[\textit{Intercommunication}]{} \textit{Compositor} \tag{6.2}$$

$$\textit{Compositor} \xrightarrow[\textit{Intercommunication}]{} \textit{Compositor} \tag{6.3}$$

$$\{\textit{ICM-APP} \mid \textit{Compositor}\} \xrightarrow[\textit{deployed to}]{} \textit{Partition} \tag{6.4}$$

Figure 6.3 provides an alternative formal view on the architecture. It depicts the relationships between the components, modelled after a Unified Modelling Language (UML) class diagram. Therefore, a 'UI-Entity' is introduced that represents the commonalities of the UI-APP and compositor.

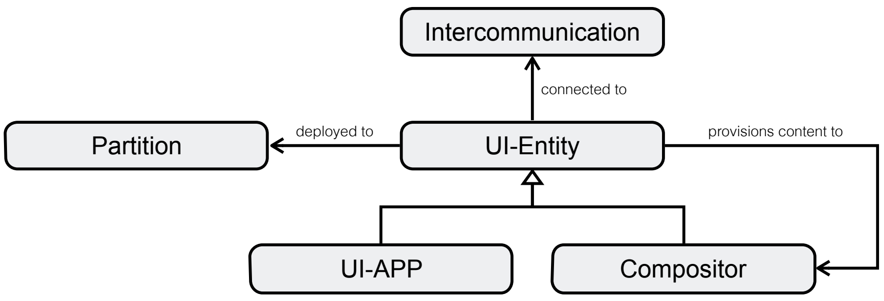

Figure 6.3: Relationships between UI components

The conceptual components arranged to the exemplary infrastructure depicted in Figure 6.2 scale with the requirements. With a compositor's capability of providing already-blended content to other compositors, it is possible to define 'composite UIs' for the occupants' different roles within the vehicle (e.g., driver, passenger, rear-seat passenger) as well as custom UI-setups for individual persons. The later may include what to visualise where and how. Such setup information might be stored within the ICM system, on a portable storage media or CE device, or on a remote infrastructure using the head-unit's wireless connectivity. Furthermore, content can be visualised on multiple displays. Therefore only one pixel buffer has to be rendered at a UI-APP which is distributed to multiple target compositors. The respective target compositors are able to adapt the content by resizing and transposing the pixel information according to the actual display characteristics and predefined setup information. This implies the UI-APP must render graphics using the highest resolution required, as the adaptation is performed by the compositor.

Although the content provisioning based on pixel buffers is basically attributed to functional decoupling of UI-APP and Compositor, it enables an additional feature for the UI design. Due to the capability of resizing the pixel buffer at the consumer (Compositor) without affecting the producer (UI-APP), it is possible to 'iconize' the UI artefacts. This means a navigation- or menubar may consist of minimized 'live views' of the running UI-APPS instead of static icons or 'screenshots'. The computational efforts to prepare the iconized live views are segregated within the limits of the particular partition that hosts the respective UI-APPs.

6.4 Applicability of the ICM-Compositor

A prototype implementation has been built to facilitate evaluation of the proposed design and demonstrate its feasibility. It is not extensive and does not provide all functional capabilities of a real-world ICM system. However, the previously defined architectural drivers are covered to address essential features from an architectural viewpoint.

The prototype implementation has to feature at least n partitions (P), with n greater than 2. P_1 contains a compositor that is blending the independently rendered graphics for visualization on a display. Furthermore, P_1 is connected to a dedicated GPU to support the modification of artefacts. $P_{2..n}$ contain UI-APPs rendering 3D graphics, also using dedicated GPUs for graphics acceleration. All EDs run different instances of an OS (i.e., forming a multi-OS environment) and have access to a shared memory region. This constitutes the minimum criteria to verify the applicability of the herein described concept for graphics compositing. In the following, the implementation and its constraints of the prototype are outlined.

The partitioning in the prototype relies on virtualization, where each partition is encapsulated within a dedicated VM. All VMs are connected to a shared memory region to prepare the prerequisite for the intercommunication component. This is realized using KVM as VMM in conjunction with a virtual inter-VM shared memory PCI device based on Nahanni (Kivity et al., 2007; Macdonell, 2011). The platform of the host system provides several GPUs passed through to respective VMs for dedicated acceleration.

A GNU/Linux based OS is utilized for the compositor's and UI-APPs' partitions. The prototype also supports Android OS based partitions acting as UI-APP to demonstrate the blending of graphical artefacts which are rendered by different OSs.

The intercommunication is realized by using Wayland with enhancements to utilize inter-VM shared memory. Admittedly, the use of Wayland violates REQ-12 which requires a rudimentary communication based on pixel information to obviate the need for any further interpretation. This is due to additional bidirectional control messages for synchronisation, a keep-alive mechanism, and forwarding of user-events. However, except for these disadvantages the features of Wayland represent the best available COTS solution to support the other requirements. Moreover, its open-source licensing allows for the required enhancements to enable inter-OS communication. Nevertheless, as part of this research, alternative non-interpreted communication was also evaluated (cf. Section A.4).

Within Android, the system services for rendering the UI is modified to clone and route surfaces to the compositor using the intercommunication component of the proposed design detailed by Theis (2013). The surfaces are routed without changing the Android applications.

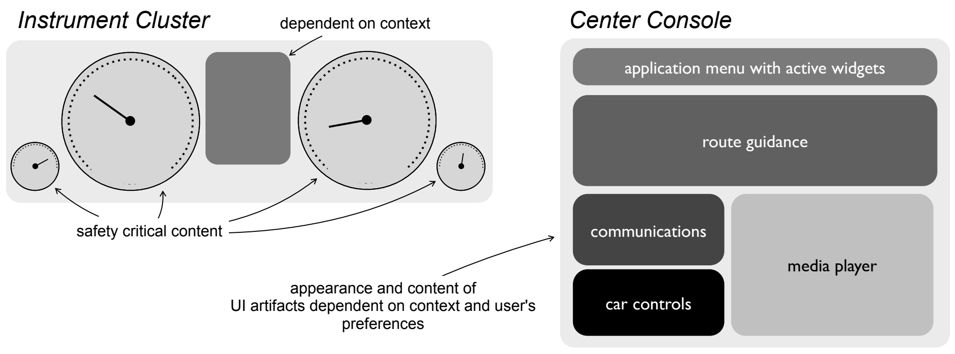

Figure 6.4: Exemplary ICM UI with different UI-APPs

Figure 6.4 depicts a prototype UI layout that relies on various UI artefacts rendered by different partitions. The selection and appearance of the content within the centre console is adaptable, whereas the instrument cluster must comply with regulations and laws. All UI artefacts are fully active and may be transposed in size and perspective by the compositor instance independent of the UI-APP. Certain content is additionally displayed on the instrument cluster, dependent on the vehicle's or application's context or user interaction.

The prototype to all intents and purposes demonstrates how a compositor along with graphic acceleration could enable modular UIs without breaching partitioning concepts.

The compositing component is implemented using Qt5 (*Qt Project* 2014) with the QtWayland module that acts as wrapper for the enhanced Wayland implementation to provide inter-OS communication based on shared memory. The compositor utilises on a scene graph for efficient handling of multiple displayed UI-APPs. Figure 6.5 depicts significant software layers of the prototype implementation, mapped to the conceptual components of the proposed compositing architecture. A preliminary prototype that founds on the herein proposed architectural principles is discussed by Bienias (2013).

For multi-headed compositors, a session compositor layer abstracts the access to the OS graphics layer (here kernel mode setting (KMS)) to foster flexibility regarding configuration of connected displays on the particular partition or GPU respectively. Hence, a single compositor can blend UI artefacts for multiple displays. Further, a compositor may also render to an 'off-screen' buffer that is provided to another compositing instance. This provides the capability to establish a hierarchy of compositing instances to reflect mixed criticalities of assigned UI-APPs and scale with the number of available displays. The prototype is equipped with multiple GPUs. A single GPU is assigned to a dedicated partition (Fries et al., 2013; Fries, 2013).

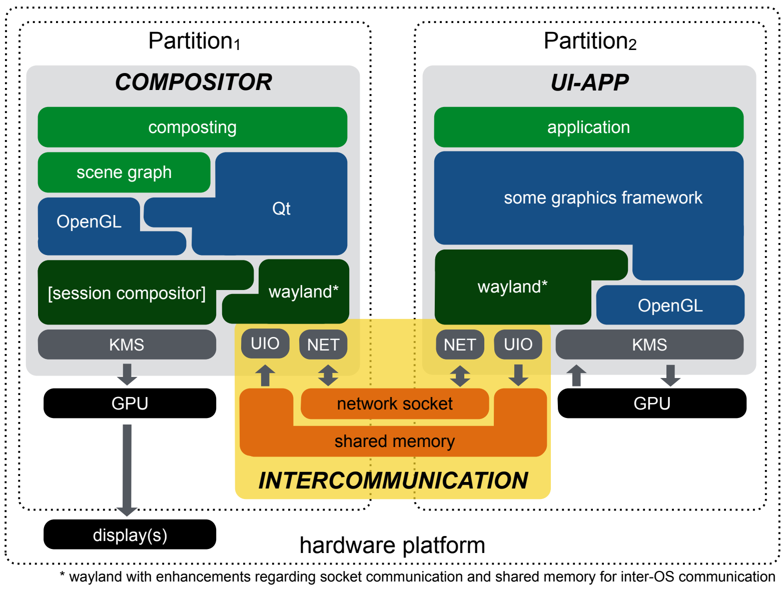

Figure 6.5: Conceptual components mapped to prototype implementation

As an alternative to the compositing architecture that relies on the Wayland protocol, a ‘frame-grabber’ component may intercept and copy rendered pixel-buffer data to a shared memory region. This can be implemented as a separate application or system service that accesses the (intermediate) frame-buffer of the graphics framework in-use, as depicted in Figure 6.6. Respective prototypes are available for Android’s ‘Surface Flinger’ (Theis, 2013) and QNX ‘Screen’ Section A.4. This allows for ‘grabbing’ and use of the whole (virtual) display output of an OS partition, but also supports the use of individual applications. A frame receiver that is connected to the compositor consumes the provided frame-buffer. The compositor blends the received UI artefacts with other content, provided by further frame-receiver or using the Wayland protocol and respective abstractions such as QTWayland.

Although the herein proposed concept focuses on unidirectional communication for the UI artefacts, the technologies used allow also for a bidirectional transport of graphic content. This means a UI-APP may incorporate precomposited content, which is then returned to the same or forwarded to another compositor. Such bidirectional communication means increased flexibility to the architecture that, for example, allows the use of a corporate overlay design that includes application-specific UI functionality. However, such a multipoint communication data-flow also implies an increased complexity due to mutual dependencies and may affect the UI responsiveness due to additional latency caused by the extra communication efforts and processing steps. Hence, bidirectional communication for UI artefacts is not considered for this research.

For the prototype implementation, the control channel relies on network socket communication. This improves traceability and flexibility due to the use of existing abstractions within utilised related software frameworks. However, this offers a starting point for

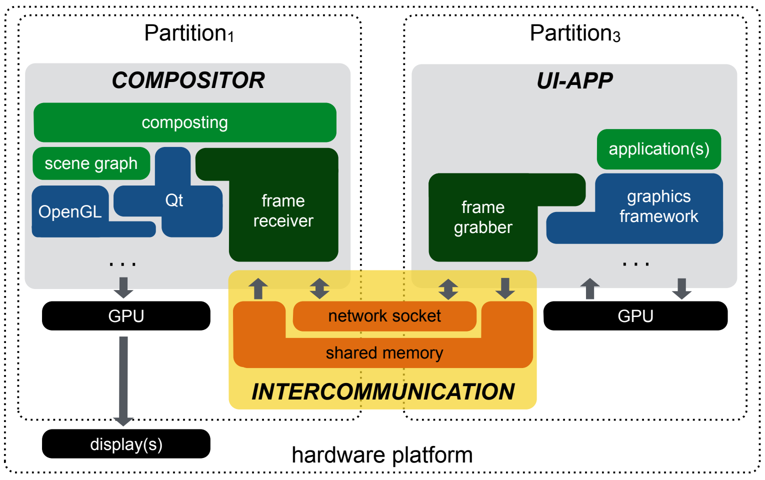

Figure 6.6: Enhancements to incorporate non-Wayland compatible UI-APPs

optimisation. The network socket can be substituted by an event-based message queue placed within a shared memory region (i.e., next to the buffer for the UI artefacts). This requires the existence of an inter-partition interrupt facility to signal the availability of new messages, such as the doorbell functionality offered by the virtual PCI device with the Nahanni-based inter-VM shared memory implementation (cf. Section 4.3.3).

6.5 Audio compositing

This research mainly focuses on the compositing of graphical UI components. However, a current ICM system heavily relies on audible user interaction. A preliminary investigation was carried out to evaluate the feasibility of mixing multiple audio provider using efficient communication facilities beyond OS boundaries. Therefore, a prototype audio compositor based upon a virtualised multi-OS environment that relies on OpenICM, inter-partition shared memory, and the gstreamer multimedia framework (*gstreamer - open source multimedia framework* 2014) was implemented (Gathmann, 2013). Basically, similar requirements apply to such an audio compositor when compared with above-proposed graphical compositing architecture. The compositing instance decides which audio stream is actually blended (a), played at which location (b) (i.e., speaker channel) and at which volume (c). Here also the respective criticality of the audio source must influence the compositor's decision (e.g., park distance control might be more important than route guidance and hence have least affect (a) or (b)).

However, an integration of graphics and audio composition into a comprehensive UI compositor is not part of this project.

6.6 Summary

Appealing UIs are important features of future ICM systems. First of all, this is addressed to the automotive (graphic) designers. Nevertheless, the visual concepts must be transferred to the software level. In parallel, the increasing extent of functionality integrated into such systems creates new challenges. Functionality varies in criticality in terms of safety. This leads to time/space-separated software architectures to enable strong enforcement of run-time policies. Such a partitioned architecture counteracts the implementation of a comprehensive, coherent, and compelling UI, which has to appear as an ensemble of one piece. This is amplified as long as only one graphic accelerator is available that has to be shared by applications executed in parallel on multiple PUs and structured using technologies for containment, such as EDs or VMs. The architectural-design approach presented addresses this issue and provides an integration path for individually-developed software components of different criticality. Relevant architectural drivers are discussed and the essential design components are illustrated. A prototype implementation supports the evaluation of the design by use of a functional proof-of-concept. It basically reflects defined key requirements for such an architecture to support the implementation of next-generation graphical UI designs.

7

Interoperation of approaches and evaluation

"The whole is greater than the sum of its parts." (Aristotele)

To fulfil the demands of next-generation ICM systems, a new integration path is necessary. The main objective here is to build dependable MCSs that rely on graphic-intensive and multi-modal UIs. Moreover, with changing functionality throughout the product lifecycle, preceding approaches for integration of multi-sourced components are not feasible anymore.

Within the previous chapters, a set of distinct approaches were presented that form architectural building blocks to address some fundamental issues related to the development and maintenance of future ICM. This chapter collects those concepts to form an integrated and holistic architecture aiming for a more predictable integration due to improved composability, in particular related to temporal behaviour while providing the necessary flexibility for dynamic functionality. Moreover, these are assembled using a prototype demonstrator to showcase their practicability.

7.1 An architecture to construct next-generation ICM

To address the herein described issues, the proposed architecture covers basically the design principles investigated as part of this research. These are briefly recapitulated in the following sections and basically consist of the design ideas and patterns that permeate a system's architecture. The assembly of the architectural components is illustrated by use of an abstract view on a system that consists of multiple application components that utilise multiple PUs, a set of shared resources and provide a uniform UI, as depicted in Figure 7.1.

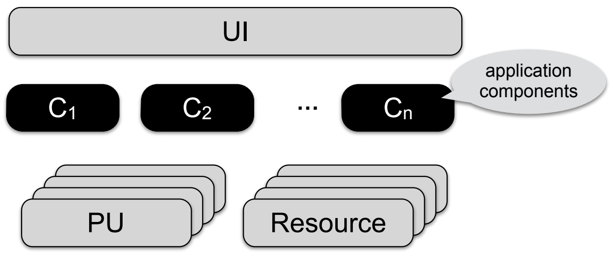

Figure 7.1: Abstract view on a component based system

7.1.1 Compatibility by containment

A key to foster deterministic component integration while preventing adverse interference in between components of mixed criticality during system operation is separation through containment. Due to the components' scope on the functional features, already-existing applications can be integrated without the need to adapt them to a system-global framework. This provides freedom to make use of legacy and third-party functionality and positively affects the portability of such. A redevelopment of applications is no longer required as long as a suitable component container is available that forms the local runtime environment.

Furthermore, a strict segregation of different functionality enables the implementation of different and independent update policies to reflect the criticalness of the particular contained functionality. Also, the different evolution pace of components can be reflected without affecting each other. Critical software may only be updated through secure and trusted communication channels with the option of validating the success of the update, as for example during the regular service at the automotive repair shop. In contrast, noncritical software may use similar distribution channels for after-market functionality and updates as approved for CE devices (i.e., so-called 'market places' or 'app stores'), using wireless access networks (a.k.a. 'over-the-air' updates). The separation ensures that both the update itself as well as the introduced or updated functionality within a container does not affect the operation performed within the other ones.

As detailed in Chapter 4, component containment can be implemented using different approaches. This allows for reflecting the actual needs due to the system's architecture and the available hardware capabilities. Applicable means are the use of scheduling policies (a), ED using PU affinities (b), containment models like LXC for Linux or APS for QNX (c), type 1 or type 2 virtualisation (d) or strict hardware partitioning in the style of AMP (e). In particular, (d) and (e) provide a multi-OS environment to enable effective containment with strong encapsulation. However, (a) to (e) all feature an explicitly controlled use of MC hardware in terms of mapping the software architecture to PUs and shared resources to foster a more deterministic behaviour. Improved compatibility positively affects composability, while the overall complexity is reduced due to the encapsulation/containment of heterogeneous software components. This is anticipated as a necessary prerequisite for next-generation ICM systems with their continuously rising complexity and number of provided functionalities.

The component containment within the demonstrator is achieved by selection and combination of representatives of the enumerated approaches. Containment at the OS level is implemented using type-2 virtualisation and at the user-space level using PU-affinity-driven EDs. This does not mean a limitation regarding the possibility of enhancing the showcase with integration of other approaches or substituting selected with matching alternatives. However, it supports multi-OS environments and is portable to different hardware platforms to improve the demonstrator's reproducibility. This is also fostered by the use of FOSS components, including a Linux kernel-based OS as host-OS and KVM as VMM/hypervisor.

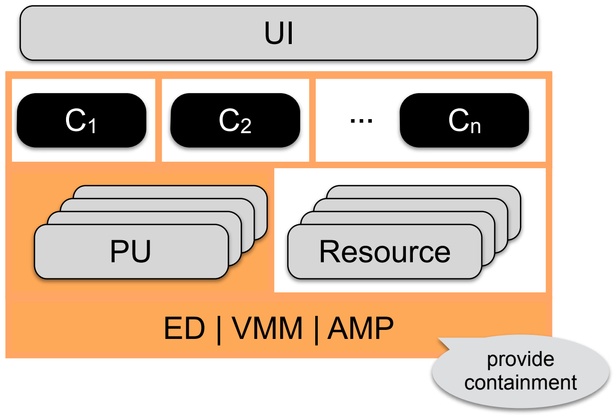

Figure 7.2: Containment of application components

7.1.2 Managed access to shared resources

Despite the isolation into containments, the integration and deployment of software components to a common hardware platform implies necessary access to shared resources. To foster compatibility by mitigation of adverse interference, in particular by a less-critical component, the access to and usage of shared resources has to be managed. This applies especially to parallel access enabled by MC hardware architectures.

Containment of software components already improves the system behaviour, but it unfolds its potentials even more when the partitioned functionality is also temporally managed regarding the access for shared resources considering the respective criticalities. This implies the managed access of shared resources is a necessary completion for the proposed technologies to structure mixed criticality and heterogeneous software components.

For the demonstrator, with SHARB a low-level software resource arbiter is available to enforce access priorities according to the component's criticality as detailed in Chapter 5. With the integration into OpenICM, it builds upon a mature software framework for embedded systems.

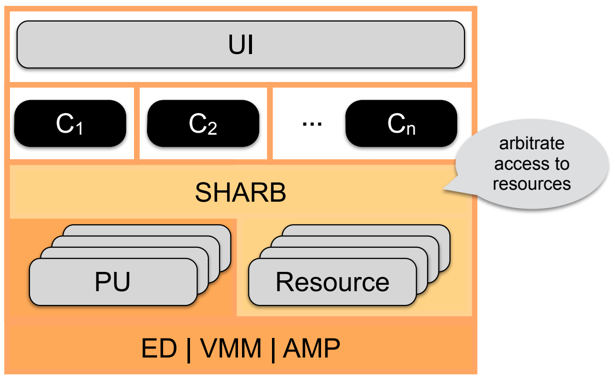

Figure 7.3: Arbitration of concurrent resource access

7.1.3 Compositing UI

For ICM systems, the UI represents a very important role. It must combine the content of heterogeneous content provider (i.e., mixed criticality components within multi-OS environment) and form a uniform, domain-specific and appealing front-end to the occupants. Especially due to the containment of the component, the blending of different contents has to work in such a way that the user does not take notice of the partitioned computation behind the visualization.

For the demonstrator, a compositor was developed that basically implements an integral usage concept and controls what to visualize on which displays by blending UI artefacts. UI applications render these artefacts within dedicated components, and compositors are assigned to different containments, while several UI-APPs may share a single containment (e.g., UI-APPs of similar criticality, with strong mutual dependencies, provided by the same supplier, etc.). This implies the UI-APPs are basically independent of each other related to their criticality. To support adequate hardware acceleration for graphic processing, particular containments have exclusive access to non-shared GPU, which applies also for the containment that contains the compositing component.

7.1.4 Interoperability by efficient communication

Separation to achieve compatibility may contradict interoperability. Interoperability addresses the ability of different components to work with one another, e.g., the functional bindings in between the components. As with compatibility, interoperability is also a constructive aspect of software engineering and an essential quality to be considered when aiming for composable systems. Hence, a system design relying on segregation through component containment has to incorporate means to achieve interoperability. The proposed architecture reflects this by use of well-defined interfaces using an event-based communication via shared memory regions, mapped to the particular component container. This is implemented by enhancing Linux KVM with QEMU/Nahanni inter-VM shared memory for efficient intercommunication.

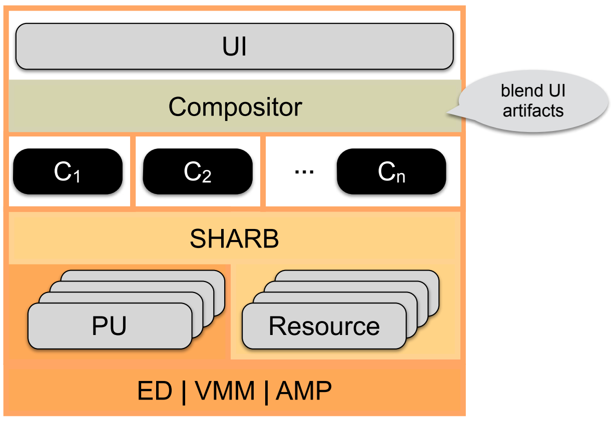

Figure 7.4: Compositing of an uniform UI

The communication related to the UI compositing requires particular consideration due to the transport of relatively big amounts of data. Hence, the communication of UI artefacts and user events relies on efficient data transport to achieve adequate responsiveness with the focus on usability. The pixel buffer information is provided to the consuming compositor using a format that obviates further computation to reduce the attack surface (cf. Section 2.3.5).

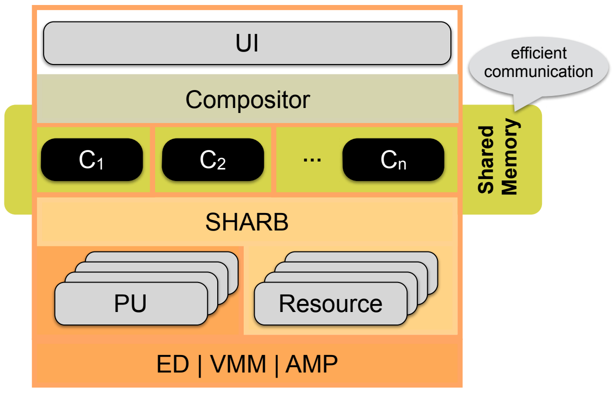

Figure 7.5: Communication facilitated by use of shared memory regions

7.2 Evaluation

Per Eklund et al. (2012), the initial design decisions that led to a particular product's architecture ('as-is') also have relevance for similar products. This means they basically can be applied to a whole product family or product line. From these general design decisions, details of a 'reference architecture' can be derived. According to the definitions provided by Angelov et al. (2012), this is considered a 'non-structured reference architecture'. However, the goal of such reference architecture is to capture rationale and design principles that form the base for a particular product's architecture but is applicable to a wider range of products or allow usage in different contexts. As the design principles are discussed and evaluated in isolation within the previous chapters, an implementation is utilised to 'pilot-case' their feasibility for a product. Further, their applicability puts emphasis on the relevance for the whole product family of ICM systems.

A set of requirements is used to express necessary qualities of future ICM (cf. previous chapters; assembled in Table 7.1). These express the rationale behind the principles in a very condensed form and are applicable to the whole 'product family'. Although the usage context of this research focuses on ICM, the defined requirements are sufficiently abstract to allow transfer or reusing of them within a different context. Further, this research does not provide a comprehensive reference architecture for ICM, but it may contribute to the 'reference architecture details' that detail the architectural strategies, as defined by Eklund et al. (2012). The defined set of requirements for this research presently enables a verification of the implemented demonstrator that can be viewed as 'architecture implementation'. Therefore its implementation is detailed in the following.

7.2.1 Implementation details of the demonstrator

The main objective of the demonstrator is to cover the requirements assembled in Table 7.1. Moreover, its design and related technology decisions consider the product qualities maintainability (in particular analysability) and portability (cf. ISO 25010 (2011) and Table 1) with the focus on reproducibility. Hence the use of FOSS is emphasised whenever possible and a common hardware architecture/platform is selected. Albeit not explicitly discussed, the use of portable technologies fosters support of a wide range of target platforms.

The host-OS is based on a Linux kernel, using KVM for creating OS partitions to achieve CL3 (cf. Table 3). The build-system to render the host-OS is based upon the Yocto build system, which allows configuration regarding what and how to build the target system by use of interdependent 'recipes' (Yocto Project, 2014). These enable a reproducible build of both the toolchain and the target system images, including kernel, modules, libraries and user-space applications. The respective recipes are structured by use of distinct layers. These represent a modular system configuration that contains optimizations for the used target hardware by use of a board support package (BSP), as well as defines substitutions or complements to recipes of subsequent application layers. This allows for the building of a whole meta-system and target-system by use of a single configuration using a customary Linux distribution. Although the demonstrator is primarily targeted for an x86 based hardware platform, the build-system used fosters only portability by substitution of the BSP (exemplary supported platforms relevant for ICM systems include Texas Instruments OMAP5, Intel Atom Zxx/Exx, Renesas RCar).

The x86-based hardware platform was selected because of easily accessible hardware and reproducibility due to a customary architecture. Moreover, it provides modularity by use of PCI-express-based expansion cards to integrate multiple GPUs, as well as automotive

ID	Requirement	Short Description
REQ-1	Components shall not interfere with each other during runtime unless it is explicitly specified.	prevent component interference
REQ-2	Components shall provide defined functional ports to enable inter-component communication.	provide functional ports
REQ-3	An interface port shall be reduced to the particular needs of the respective communication (based on type 'event' or 'data access').	simplified ports
REQ-4	Either the utilized software framework or the components shall comply with POSIX.	conform to POSIX
REQ-5	The system shall provide capabilities for updating and installing components on user demand.	support dynamic functionality
REQ-6	The latency related to the access of a shared resource shall be predictable.	predictable resource access latency
REQ-7	The resource access shall be manageable without the need for any modifications of third-party components.	transparent resource management
REQ-8	The access to shared resources shall be temporally ordered using static-defined priorities.	resource access priorities
REQ-9	A UI compositor shall provide the capability to blend independent portions of the UI.	compositing UI-APPs
REQ-10	The UI compositing shall support the distribution of UI components onto multiple OS instances.	multi-OS compositing
REQ-11	A GPU shall be assignable exclusively to a dedicated OS partition.	dedicated GPU
REQ-12	A UI component shall provide pixel frames that obviate further interpretation at the receiving compositor.	frame-based communication
REQ-13	The compositing architecture shall provide performance efficient communication channels.	efficient communication channel
REQ-14	User-event shall be communicated from the compositor to the designated UI component.	backchannel for user-events

Table 7.1: Requirements for an ICM architecture.

bus systems like MOST and CAN. The utilised platform is equipped with a CPU that provides four PUs and features hardware-accelerated OS-virtualisation. Figure 7.6 depicts the demonstrator's hardware configuration.

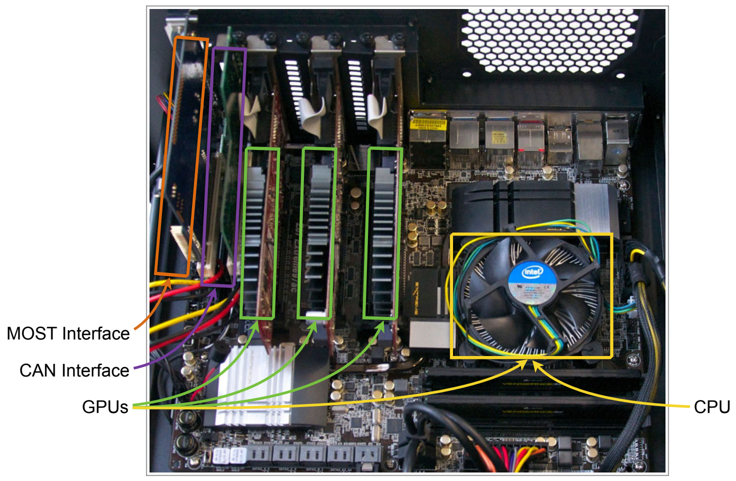

Figure 7.6: Hardware configuration of the demonstrator

The demonstrator hosts four OS partitions (OSP), each running a separate OS instance within a VM (cf. Figure 7.7):

- OSP_0 contains the VMM and the compositor , while both rely on Linux.
- OSP_1 contains an instrument cluster that builds upon QNX Neutrino
- OSP_2 contains a media player that builds upon Linux
- OSP_3 contains navigation and IP telephony that builds upon Android

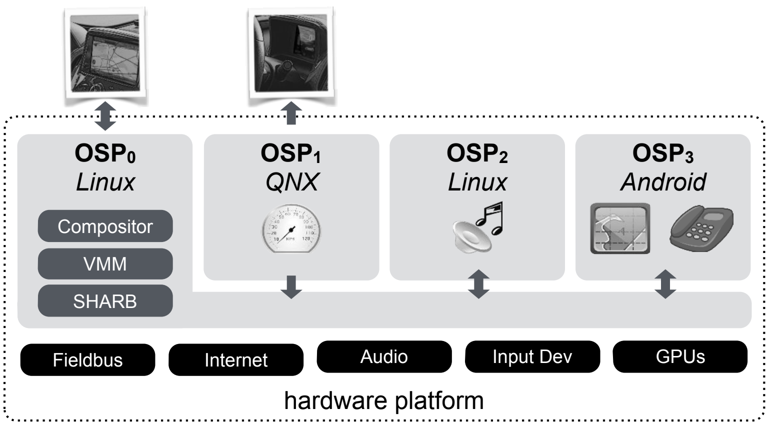

Figure 7.7: The demonstrator's four OS partitions

OSP_0 issues and controls the startup sequence of $OSP_{1\text{-}3}$. This is supported by use of the features of OpenICM to assign the respective VMs to dedicated PUs and provide means to manage access to shared resources by use of the integrated SHARB. The use of a software framework for control of the startup sequence and observation of the runtime behaviour positively affects the systems maintainability in terms of analysability. This can be achieved, i.e., by use of a software-watchdog (cf. (Wietzke, 2012, p60 ff.)) and continuous system traces during test phases.

The GPU assigned to OSP_0 is connected to a display that provides multi-touch capabilities for user interaction. $\text{OSP}_{0\text{-}3}$ are interconnected using shared memory and socket communication, whereas the latter is implemented using virtual network sockets in between the VMs accelerated by hardware support for virtualisation.

The compositor deployed to OSP_0 is implemented with Qt5 and its abstractions for Wayland and OpenGL, following the layered architecture detailed in Figure 6.5 (cf. Section 6.4). This also includes the employment of a session compositor to abstract the configuration of the multi-display environment. Within $\text{OSP}_{1\text{-}3}$, the tasks are structured using EDs that rely on PU-affinity, provided by OpenICM.

In summary, the demonstrator utilised multiple technologies to fulfil the specified requirements. These technologies can be classified into three conceptual groups that correspond to the main contribution of this research:

- Partitioning components for temporal containment (cf. Chapter 4).
- Management of shared resources (cf. Chapter 5).
- Compositing of UI artefacts (cf. Chapter 6).

Table 7.2 maps the technologies to the specified requirements for the ICM architecture. Several requirements are related to multiple technologies that complement each other. This implies that most requirements are achieved by use of a set to technologies.

The features of the OpenICM framework were not introduced but enhanced with this research. Further, the distinct technologies were evaluated individually, while their assembly into an integral demonstrator basically evinces their compatibility and interoperability. They complete each other to form a comprehensive architecture for improving the composability of interactive software components.

7.2.2 Discussion

Based on the evidence of applied software engineering practice for ICM (cf. Chapter 3), containment of heterogeneous components provides the architectural means to counter the rising integration density at the software level of next-generation ICM. The demonstrator covers the architectural requirements for fostering such modularised architecture with temporally separated software components and providing the infrastructure to render a uniform HMI integration layer. Taking the viewpoint of the user and emphasing the system's usability, the HMI basically constitutes the integration point of potentially heterogeneous components. This implies an architecture that focus on structuring software

ID	**Short Description**	**OpenICM contexts**	**OpenICM events**	**OpenICM data container**	**EDs (PU-affinity based)**	**VMMC**	**inter-VM SHM**	**SHARB**	**UI-compositing**
REQ-1	prevent component interference	•			•	•		(•)	
REQ-2	provide functional ports		•	•			•		
REQ-3	simplified ports		•	•					
REQ-4	conform to POSIX		•	•	(•)			•	
REQ-5	support dynamic functionality				(•)	•			
REQ-6	predictable resource access latency				•		•		
REQ-7	transparent resource management							•	
REQ-8	resource access priorities	•						•	
REQ-9	compositing UI-APPs								•
REQ-10	multi-OS compositing					•	•		•
REQ-11	dedicated GPU					•			•
REQ-12	frame-based communication						•		•
REQ-13	efficient communication channel						•		•
REQ-14	backchannel for user-events								•

Table 7.2: Map requirements to technologies

components to improve composability must consider features to composite components at the UI level.

The herein detailed demonstrator reflects selected key features of current ICM systems, but is not comparable to a system that is rendered by >200 software developers within >2 years (cf. Section 3.2). This includes the functional features as well as the visual design of the HMI. However, this research serves as pilot case regarding the software architecture that consists of the proposed concepts. Within that context, some implementation-specific decisions might not apply to a 'productive' ICM system which are discussed in the following.

Linux on OSP_0 provides only limited real-time features. However, for this evaluation it features portability and good hardware support. Basically, the superordinate user-space layers enable substitution of Linux with a POSIX conform OS like QNX. Furthermore, the compositor can be moved to a VM instance to free the host-OS from all tasks except the VMM/hypervisor. The next consecutive step is to remove the hypervisor by an AMP-based multi-OS architecture, where the segmentation and startup is coordinated by an enhanced bootloader. First evaluation showed promising results on MC architecture that relies on ARM ISA. The herein described general architecture must not be altered to support that shift to AMP-based multi-OS. Especially for MCS, a reduced or even unnecessary VMM layer eases the efforts related to the required verification process and decouples independent software evolvements.

For both the compositor and client UI-APPs, the Qt framework was used (*Qt Project* 2014). Qt provides extensive support in particular for the implementation of UIs. The utilisation for the compositor makes the used Qt libraries critical components of the system. The same applies to the software-based instrument cluster, which for the demonstrator also relies on Qt. Due to the number of provided functionalities, a formal verification of both the prototype compositor and instrument cluster is improbable. However, Qt enables precompiled native rendering of UIs. This means the code must not be parsed and interpreted during runtime, as required for UIs that rely on web technologies such as HTML5 (Gryc and Lapierre, 2012).

The demonstrator relies on the Wayland protocol for UI compositing, with enhancements on the libraries to support inter-VM shared memory communication. This is a prerequisite of the availability and integration of the enhanced Wayland library to all guest OSs. This is also related to the required driver to connect to the inter-VM shared memory region. An alternative approach is to enhance the VMM to provide a virtual hardware platform that implements the Wayland protocol within the virtual graphics hardware. This would obviate any modification to a guest-OS, system libraries and applications executed within a virtual environment. This means, compositing is possible without altering a self-contained VM. Such an approach limits the degree of freedom related to the arrangement of multiple UI-APPs within the compositor, as the virtual display of the VM containing those UI-APPs prearranges/composites their visual output. However, for the herein detailed demonstrator modifications are limited to ‚user-space', i.e. system libraries and applications. Modifications of the OS were avoided with respect to portability and reproducibility.

The use of an integral build system improves the ability for maintaining and reproducing the demonstrator and adapting it to alternative hardware platforms. Here, this is featured by the Yocto project that is utilised for both the OSP_0 and OSP_1 to formalise the building and configuration of all parts of the system.

7.3 Summary

The herein described demonstrator illustrates the practicality of the proposed approaches and concepts regarding containment, intercommunication, arbitration of resource access and UI compositing. Each viewed separately provides only limited support. This means, although each of them is incomplete on its own, they finally complete each other. Through their combination, their full potential to improve composability for next-generation ICM systems, supporting the parallel use of multiple OSs, mixed criticality components and dynamic functionality with predefined temporal behaviour is exploited.

8

Conclusions and future work

"The reward for work well done is the opportunity to do more." (Jonas Salk)

With software, new functionality can be provided at low cost (Göschel, 2012) and maintained throughout a vehicle's product life cycle. While the replication costs for software are insignificant, the related development costs are increasing dramatically (Manfred Broy et al., 2007). Integration of new functionality has implications for the fulfilment of the overall system's required qualities, in particular regarding composability, performance efficiency, usability and maintainability. The rising complexity with next-generation ICM systems demands a reconsideration of current system architectures, as traditional engineering perspectives are no longer adequate (Sommerville et al., 2012). This relates to interdisciplinary challenges that range from management of distributed development teams via clearly defined requirements to decisions and implementations driven by technical criteria. Additionally the systems' software components' mixed criticality generates cumulative challenges for the development, verification and update cycles. At the organisational level, distributed development involving multiple organisations all over the world implies an additional dimension of complexity. At the same time, the development of these systems has to be aligned with the development processes of all other vehicular components even beyond the limits of the E/E domain to a common SoP. Current practice does not adequately address these issues.

8.1 Achievements of the research

This research is based on an investigation of the current situation covering the increasing complexity and current requirements for ICM systems. Therefore, literature within related fields of research, as well as common practice within recent and on-going industrial projects were reviewed. The gathered information was used to highlight the significance and challenges regarding composability and related qualities of ICM software components and supporting infrastructure by means of a software framework.

A set of architectural features were investigated and assembled to form a proposed architecture that improves the relevant qualities. The main objective is to support the integration of heterogeneous functionality that varies by means of criticality, temporal requirements (for both time and event triggered tasks), and demands for software updates, while fostering an integral UI that provides a positive UX. This architecture also addresses the implications that arise during the development process, in particular due to a parallel and independent software development that generates the need for integration through a coordinator, as for example the (Tier-1) OEM.

The proposed architecture essentially consists of the following three basic concepts. These were motivated, detailed and implemented using a pilot case to demonstrate both their feasibility and practical impact:

1. Isolate software components into partitions while exploiting the features of current hardware architectures and preserving efficient inter-component communication facilities.
2. Manage the access to shared resources by arbitration to reflect the mixed criticality of software components computed in parallel.
3. Composite segregated UI/software components to enable the implementation of homogeneous and integral UI.

None of these individually solves the issues regarding composability. But combined and integrated into a software framework, they provide holistic support to integrate complex systems that behave deterministically even during situations of high system load.

Similar concepts with different emphases developed and enhanced in parallel to this research address either a more strict decoupling with the focus on non-user interactive systems (cf. Section 2.4.1), or only partly address rising demands for mixed criticality and demands for multi-OS environments of future ICM systems (i.e., with focal point strictly set on infotainment; cf. Section 2.4.2). Hence this research can be regarded as bridgework between these two currently distinct domains, paving the way for an increased integration of formerly distinct vehicular hardware platforms.

8.2 Limitations

Although the research objectives have been met, a number of limitations associated with the project can be identified. The key limitations of the research are summarised below.

The focus of the research was to increase the determinism of the behaviour at runtime while improving the predictability of the integration. All measures proposed here basically share the same principle of abstraction of the underlying hardware or software level and introduce a management layer. This affects the overall system efficiency, both in terms of speed and memory usage, due to introduced management overhead. Exceptions are EDs that utilize the task scheduler's existing feature for PU-affinity and strict hardware partitioning that follows the idea of AMP. In any case, these also reduce the overall system efficiency due to static allocation of computational resources and not considering load-balancing strategies[1]. This means increased determinism comes at the cost of introduced overhead and reduced exploitation of the available computational resources. Nevertheless, these expenses facilitate the integration of mixed criticality components onto a single hardware platform and improve the system's determinism during high-load situations.

The concepts scale with the number of components and their containments. In particular, the number of parallel containments is - depending on the technology up to a certain degree - limited by the physical features of the HW platform. Besides the number of available PUs, this is related to the shared on-chip/on-system infrastructure, layered memory architectures (i.e., including shared caches and main memory), and other shared resources. This means the applicability of the concepts improves with the availability of hardware features. Currently available MC-architectures within the embedded and automotive system domain may only partly exploit the potential of the proposed concepts and technologies.

For the number of available PUs, it can be assumed that the currently available SoCs are only the beginning of parallel platforms within the domain of embedded computing. This research contributes to this evolution by proposing to set focus on the enforcement of predefined temporal behaviour instead of 'merely' increasing computational throughput.

Although an arbiter may improve determinism for accessing shared resources, such management will increase delays for low priority accessors with an increasing number of accessors. At some point, the low priority accessors will not be able to provide the required QoS, which may only be acceptable for a short timespan (e.g., during a high-load situation). Such behaviour must not prevail. This means increasing the number of accessors for a shared resource may violate required qualities regarding usability and performance efficiency, at least for low priority components. This requires either the provisioning of additional resources to reduce the degree of concurrency or system redesign. However, it is important to define the components' particular criticality to derive their QoS and priorities while considering the limits of the available hardware platform in terms of computational power, memory and I/O resources. Despite the potential integration of a virtually infinite number of functionalities, the execution of the software components is constrained by physical limitations.

Aside from the accessed resources, the on-chip/on-system infrastructure also has to cope with an increased amount of processed data with regards to both throughput and fre-

[1]In practice, it is even counterproductive to exploit the full potential computational power of the PUs, as some embedded platforms are not capable of running at full load for longer than a given time period without damaging the hardware due to thermal issues. Although such implications were not examined as part of this research, the effects have been observed, e.g. with the Texas Instrument OMAP5 SoC.

quency. The related effects depend on the particular components' demands for intercommunication and resource utilisation and scale at least linearly with the number of integrated components. This necessitates hardware platforms with high throughput infrastructures and emphasises the proposed use of simplified event based (fixed) message protocols to reduce complexity related to the components intercommunication.

For the proposed UI, compositing the transmitted pixel buffer information consumes additional memory. Although the related footprint is predictable, the amount of memory actually utilized might be extensive. It can basically be calculated before runtime based on the components that actively provide UI artefacts as well as on their individual resolution (which may differ from the displayed resolution). However, the memory usage and utilisation of infrastructure to transmit the pixel buffer information scales linearly with the number of active UI components. While dynamic approaches such as on-demand downgrading of the resolution of non-critical applications might be helpful to improve efficient memory utilisation, they can also cause decreased determinism and may affect the UX. It might be more beneficial to set a fixed limit for active UI components related to the HW platform's memory facilities. Nonetheless, such an investigation was not conducted as part of this research.

Additionally, the consideration of MCS basically refers to the integration of components that have to be verified to provide a certain degree of functional safety, as for example in accordance with ASIL classifications. However, this research did by design not extensively cover related areas such as requirements and verification of 'functional safety' as defined with ISO 26262 (2011).

Furthermore, this research did not, by design, provide in-depth details regarding process and maturity models supporting the product life cycle right from the start of development. It is assumed that currently-applied related approaches may adequately support the development and integration at the organisational level (CMMI Product Team, 2010; Automotive SIG, 2010). The proposed concepts neither contradict nor conflict with such. On the contrary, these concepts even support a multi-sourced and multi-organisational development by complementing the process models with an integral infrastructure on the level of development.

8.3 Suggestions and scope for future work

This research programme has advanced the field of software engineering of ICM with a strong focus on components' composability by proposing an infrastructure that exploits characteristics of parallel hardware architectures. However, a number of areas of scope for future work exist, specifically related to this research and more generally within the area of mixed-criticality and highly interactive embedded systems. These suggestions are detailed below:

1. Improving multi-OS architectures with decreased overhead to foster strict resource partitioning is necessary, advancing the proposed architecture of Fischer (2009). This may also include the utilisation of multiple GPUs without the utilisation of a virtualisation layer. Also, the enforcement of the component containment of multi-OS architectures and respective approaches for assurance of such is an active field of research (Schnarz et al., 2013; Schnarz et al., 2014b; Schnarz et al., 2014a). Both are prerequisites for performance-efficient and verifiable MCS.

2. Further, resources could be spent on the integration of 'real-time data' and functionality provided by infrastructure (i.e., 'off-board'). This may include the consideration of in-vehicle proxy services and remote delivery platforms to which former on-board functionality is off-loaded, as proposed by Glaab et al. (2014a). This must be reflected by the ICM systems' architecture, whereas the containment concepts described here may provide an adequate foundation to decouple such remote functionality from (critical) on-board functionality. Beyond that, the use of remote data as well as the incorporation of remote functionality into a vehicular E/E system that is interconnected with safety-critical ECUs generates further implications that are related to secure communication. Potential effects of a violation of the ICM's integrity may affect the car's occupants' physical health, not comparable with the results of security issues related to mobile phones or desktop computers which are basically limited to financial loss and loss of data.

3. Further investigation of ICM system's UX while considering reduction of driver distraction and support for dynamic functionalities is necessary. This research project is focussed primarily on an architectural view of ICM. Although architecture is important for system design, this must not be confused with the design that appears to the user. Although the implementation of a UI relies on the system design by exploiting of the features provided, the realisation of an appealing and usable UI is beyond infrastructural concepts. An adequate UX requires an integral usage concept that covers the vehicular context as well as the mixed criticality of heterogeneous applications and use cases. This also includes the consideration of multi-modal input and multiple displays and speakers for spatially distributed output. The proposed compositing architecture covers some significant issues but is only the provisioning infrastructure that has to be instrumented by an actual UI implementation that sustains a homogeneous UX (Holstein et al., 2015). The demonstrator described within Chapter 7 basically describes features but not the possible impact to a user regarding sub-characteristics of usability (e.g., appropriateness, user error protection, aesthetics, accessibility – cf. Table 1).

4. Further investigation on the applicability of the concepts for compositing of graphics to audio is necessary. The proposed compositing architecture's focus is primarily set to graphical UIs. An in-vehicle UI also relies on audible user-interaction (cf. Section 6.5). Although preliminary investigations were carried out to demonstrate

feasibility, the combination of audible and graphics within an integral ICM compositing architecture is still an open issue. However, the essential requirements for graphics compositing may correlate with audio compositing.

5. Further research on a formal approach to transfer components' importance to threads' priorities that reflect a predefined system behaviour is necessary. The foundation for mapping mixed importance from design to implementation level is related to (static) prioritisation and scheduling or arbitration that consider the individual priorities. Incompatible priority policies and schemes can be resolved following the proposed approach for decoupled scheduling domains through containment. The question of how a system can be prioritised efficiently at the design level, including consideration of how to transfer such information to the implementation level using a formal approach, is still an open issue. This is a prerequisite for automating such interpretations and 'translating' to achieve the predefined system behaviour.

6. Further research on appropriate system profiling to determine correct partitioning with respect to the predefined system behaviour is necessary. Although the containment of functionality and derived tasks on the implementation level is fundamentally related to their heterogeneity (i.e., regarding supplier, criticality, other characteristics), there might still be a certain degree of freedom, in particular regarding how to partition the overall system.

7. Furthermore, additional research might be applied to approaches for determining the correct allocation and best parallel use of shared resources. Related results may improve the system's overall efficiency regarding the concurrent use of shared resources. Again, appropriate system profiling might be a useful building block for such research, which may also influence the partitioning of functionality.

8. Further, resources could be dedicated to reviewing the relationship between a component-based software architecture for mixed-critical/interactive systems and the engineering processes defined by maturity models such as CMMI and Automotive SPICE (Software Process Improvement and Capability Determination) (CMMI Product Team, 2010; Automotive SIG, 2010). This research focused on constructive aspects for the development of complex software systems that form a coalition of systems. In practice, such development might be organised by the use of abstract process models with a focus on quantitative management and optimisation.

8.4 The future of In-Car Multimedia

The root of ICM systems was the FM radio. Within a relatively short time - not much more than one decade - it expanded to a computer system with significant computational power that addresses multiple use cases. This is an on-going evolution. Increased interconnectivity with on-board units as well as to outside infrastructures provides the necessary capabilities for new applications.

At the same time, ADAS means a fundamental change to driving. Functionalities like assisted or semi-automatic parking, lane departure warning, lane change assistance, collision avoidance, traffic sign recognition and adaptive cruise control with steering assist are already available (Fleming, 2013). Although there is still a significant time gap until autonomous driving is ready for everyday use (including the necessary clarification of legal issues), the transition by the use of such assisting functionalities has already started (Coelingh and Solyom, 2012). Ross (2014) pictures the evolution of the assistance supplied by the car as follows:

> *"In 5 years, cars will be quicker to intervene; in 20, they won't need your advice; and in 30, they won't take it."*

Giving credit to car manufacturers like Nissan, a line of autonomous driving cars will already start to be available in 2020. Even if their introduction is delayed by a few years, the results of the past years' autonomous vehicle races sponsored by Defense Advanced Research Projects Agency (DARPA), as well as various field tests of different OEMs, demonstrate the feasibility of 'driverless' travel (Ross, 2014). This puts additional emphasis on ICM systems that combine entertainment of the passengers, 'operation planning' of travel, and an in-vehicle communication centre. By practical means, in the future, an ICM system will not distract from driving, instead, driving will distract from using the ICM: the descendent of the FM radio may outlive the accelerator, brake and steering wheel.

A more imminent change for the automotive industry is the increased use of battery-powered electrical drives for both economic and ecological reasons. These will reduce practicable driving distances before a 'refill' is necessary, while such a 'refill' currently requires significantly more time than fuelling up with petrol. This has an effect on driving, as it requires a 'planned operation' of the vehicle to avoid a breakdown due to an empty battery. Although battery technology is evolving, it may still take many years until cars with electrical drive provide their passengers a similar flexibility for travel as already provided by petrol-driven ones (also with respect to the infrastructure for 'refill'). ICM systems and in particular their route navigation subsystem may assist in such planning, especially with correlations to information related to current battery charge, traffic conditions, driving behaviour, climatic conditions, remaining driving distance, alternative charging stations, etc. Hence, the ICM system will become a more integral part of the travel and therefore gain importance. A prerequisite for such improved driver assistance is an increased interconnectedness of vehicular subsystems of mixed criticality and the use of 'real-time' information via wireless networks. This poses a substantial challenge for vehicular E/E systems and in particular for next-generation ICM.

The use-case detailed above regarding electrical drives represents only one example for assisting systems that rely on data provided by different providers such as driving dynamics, up-to-date geographical information, past behaviour of the driver, current (or estimated) traffic and climate conditions. Sources are vehicular ECUs, 'off-board' servers, and the

ICM system. The latter bridges the different worlds of on-board and off-board systems and is able to process new information based upon the combined data, to support applications that offer improvements related to fuel economy, active driver safety, traffic efficiency, and parking space allocation; or more generally: 'an improved driving experience'. They all share an increased interconnectedness of formerly distinct systems to offer new opportunities while also introducing new risks.

In conclusion, the requirements and user expectations regarding the number of functionalities and services provided by ICM are continuously growing. At the start of this research programme, such an evolution was only speculated upon, but now it has become reality. Unfortunately, the applied technologies and in particular the evolution of architectural concepts did not keep pace with current demands. Today, OEMs and tier-1 OEMs are encountering unpredictable integration efforts of such multi-sourced and mixed-criticality SW systems and are forced to bear cost-intensive downstream fix-up phases with significant negative effects on economic success. However, with further developments, such as vehicles' connectivity to infrastructure, vehicular ad-hoc networks, (semi-) autonomous driving and electrical drives, this evolution is still in its early stages. A solution at the management/organisational level is not foreseeable, as with inadequate architectural concepts, the current problems are to be seen at the technical level.

This research programme contributes a solution to relieve such architectural deficits by proposing a set of integral concepts for a predictable integration, maintenance and operation of ICM. It adapts composability to the domain of ICM and envisages this as a key characteristic that has to be manifest in a system's architecture. The concepts and detailed technologies for containment, shared resource arbitration and compositing perfectly complement each other to improve composability and reduce complexity, while exploiting the features of parallel HW architectures. Their interoperability has been proven with a demonstrator. Transferring these concepts into practice can build a foundation for a more predictable and efficient development of next-generation ICM.

Bibliography

1. 3GPP (Apr. 2010). *Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN).* Standard, TS 36.300, Rel. 8. 3GPP. URL: `http://www.3gpp.org/DynaReport/36300.htm` (cit. on p. 27).

2. Aggarwal, Nidhi, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith (June 2007). "Configurable isolation: building high availability systems with commodity multi-core processors". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, pp. 470–481 (cit. on p. 83).

3. Alvaro, Alexandre, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira (2005). "Quality Attributes for a Component Quality Model". In: *10th WCOP/19th ECCOP, Glasgow, Scotland* (cit. on pp. 31, 37).

4. Alvaro, Alexandre, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira (Jan. 2010). "A Software Component Quality Framework". In: *SIGSOFT Softw. Eng. Notes* 35.1, pp. 1–18. DOI: `10.1145/1668862.1668863` (cit. on p. 31).

5. Amadeo, Marica, Claudia Campolo, and Antonella Molinaro (Mar. 2012). "Enhancing IEEE 802.11P/WAVE to Provide Infotainment Applications in VANETs". In: *Ad Hoc Netw.* 10.2, pp. 253–269. DOI: `10.1016/j.adhoc.2010.09.013`. URL: `http://dx.doi.org/10.1016/j.adhoc.2010.09.013` (cit. on p. 28).

6. Amey, Andrew, John Attanucci, and Rabi Mishalani (Dec. 2011). "Real-Time Ridesharing - Opportunities and Challenges in Using Mobile Phone Technology to Improve Rideshare Services". In: *Transportation Research Record: Journal of the Transportation Research Board* 2217.1, pp. 103–110. DOI: `10.3141/2217-13` (cit. on p. 27).

7. Angelov, Samuil, Paul Grefen, and Danny Greefhorst (Apr. 2012). "A Framework for Analysis and Design of Software Reference Architectures". In: *Inf. Softw. Technol.* 54.4, pp. 417–431. DOI: `10.1016/j.infsof.2011.11.009` (cit. on p. 145).

8. Apple (2014). *Apple iOS Carplay.* URL: `https://www.apple.com/ios/carplay/` (visited on Oct. 10, 2014) (cit. on p. 29).

9. Araniti, Giuseppe, Claudia Campolo, Massimo Condoluci, Antonio Iera, and Antonella Molinaro (May 2013). "LTE for Vehicular Networking: A Survey". In: *Communications Magazine, IEEE* 51.5, pp. 148–157. DOI: `10.1109/MCOM.2013.6515060` (cit. on p. 27).

10. ARM (May 2014). *ARM CoreLink GIC-500 Generic Interrupt Controller.* Technical Reference Manual Revision r0p0, Issue B. ARM (cit. on p. 84).

11. Aron, Jacob (2011). "How innovative is Apple's new voice assistant, Siri?" In: *New Scientist* 212.2836, p. 24. DOI: `10.1016/S0262-4079(11)62647-X` (cit. on p. 27).

12. Automotive SIG (May 2010). *Automotive SPICE - Process Reference Model.* Standard, Version 4.5. The SPICE User Group. URL: `http://www.automotivespice.com/fileadmin/software-download/automotiveSIG_PRM_v45.pdf` (cit. on pp. 156, 158).

13. AUTOSAR (Oct. 2013). *Requirements on Operating System.* Standard, AUTOSAR (cit. on p. 46).

14. AUTOSAR (Mar. 2014). *Specification of Operating System.* Standard, AUTOSAR (cit. on pp. 46, 47, 122).

15. Avižienis, Algirdas, Jean-Claude Laprie, and Brian Randell (2001). *Fundamental Concepts of Dependability.* Tech. rep. UCLA CSD Report no. 010028, LAAS Report no. 01-145, Newcastle University Report no. CS-TR-739. UCLA Computer Science Dept. Univ. of California, LAAS-CNRS, Dept. of Computing Science Univ. of Newcastle upon Tynea (cit. on pp. 18, 188).

16. Avižienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr (Jan. 2004). "Basic concepts and taxonomy of dependable and secure computing". In: *Dependable and Secure Computing, IEEE Transactions on* 1.1, pp. 11–33. DOI: `10.1109/TDSC.2004.2` (cit. on pp. 18, 19, 38).

17. Bienias, Tim (June 2013). "Erstellung und Evaluierung eines Compositor-Bedienkonzepts in einem Multi-Client-System". Bachelor's Thesis. Faculty of Computer Science, h_da Hochschule Darmstadt, University of Applied Sciences (cit. on p. 137).

18. Bini, Enrico, Giorgio Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Arzen, Vanessa Romero, and Claudia Scordino (May 2011). “Resource Management on Multicore Systems: The ACTORS Approach”. In: *Micro, IEEE* 31.3, pp. 72–81. DOI: 10.1109/MM.2011.1 (cit. on p. 123).

19. Bolle, Michael (2011). “Connected Vehicle: i2Car or Car2i?” In: *carIT-Kongress - Mobilität 3.0*. Media-Manufaktur (cit. on pp. 2, 3, 17, 21, 24, 125).

20. Borgeest, Kai (2014). *Elektronik in der Fahrzeugtechnik*. 4th. Springer. DOI: 10.1007/978-3-8348-2145-4 (cit. on pp. 17, 55, 56).

21. Bose, Raja, Jörg Brakensiek, and Keun-Young Park (2010). “Terminal Mode – Transforming Mobile Devices into Automotive Application Platforms”. In: *Proceedings of the 2nd International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. AutomotiveUI ’10. Pittsburgh, Pennsylvania: ACM, pp. 148–155. DOI: 10.1145/1969773.1969801 (cit. on p. 29).

22. Bräuer, Mathias (2011). “Skalierbare Systemlösungen für Instrument Cluster”. In: *Hanser Automotive* 10.1-2, pp. 20–23 (cit. on p. 24).

23. Broy, M. (July 2005). “Automotive software and systems engineering”. In: *Formal Methods and Models for Co-Design, 2005. MEMOCODE ’05. Proceedings. Third ACM and IEEE International Conference on*, pp. 143–149. DOI: 10.1109/MEMCOD.2005.1487905 (cit. on p. 22).

24. Broy, Manfred (2006). “Challenges in Automotive Software Engineering”. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06)*. Shanghai, China: ACM, pp. 33–42. DOI: 10.1145/1134285.1134292 (cit. on pp. 10, 16).

25. Broy, Manfred, Samarjit Chakraborty, Dip Goswami, S. Ramesh, M. Satpathy, Stefan Resmerita, and Wolfgang Pree (2011a). “Cross-layer Analysis, Testing and Verification of Automotive Control Software”. In: *Proceedings of the Ninth ACM International Conference on Embedded Software*. EMSOFT ’11. Taipei, Taiwan: ACM, pp. 263–272. DOI: 10.1145/2038642.2038683 (cit. on p. 13).

26. Broy, Manfred, Inglof H. Krüger, Alexander Pretschner, and Christian Salzmann (Feb. 2007). “Engineering Automotive Software”. In: *Proceedings of the IEEE* 95.2, pp. 356–373. DOI: 10.1109/JPROC.2006.888386 (cit. on pp. 1, 2, 12, 153).

27. Broy, Manfred and Andreas Rausch (2005). “Das neue V-Modell® XT”. German. In: *Informatik-Spektrum* 28.3, pp. 220–229. DOI: 10.1007/s00287-005-0488-z (cit. on p. 55).

28. Broy, Manfred, Günter Reichart, and Lutz Rothhardt (2011b). “Architekturen softwarebasierter Funktionen im Fahrzeug: von den Anforderungen zur Umsetzung”. In: *Informatik-Spektrum* 34 (1). 10.1007/s00287-010-0507-6, pp. 42–59 (cit. on pp. 1, 16).

29. Broy, Manfred and Thomas Streicher (1991). “Specification and Design of Shared Resource Arbitration”. In: *International Journal of Parallel Programming* 20 (1). 10.1007/BF01407930, pp. 1–22. DOI: `10.1007/BF01407930` (cit. on p. 102).

30. Bucciol, Paolo, Frederico Ridolfo, and Juan Carlos De Martin (Apr. 2008). “Multicast Voice Transmission over Vehicular Ad Hoc Networks: Issues and Challenges”. In: *Networking, 2008. ICN 2008. Seventh International Conference on*, pp. 746–751. DOI: `10.1109/ICN.2008.107` (cit. on p. 28).

31. Bunzel, Stefan (2011). “AUTOSAR – the Standardized Software Architecture”. In: *Informatik-Spektrum* 34 (1), pp. 79–83. DOI: `10.1145/1978802.1978814` (cit. on p. 45).

32. Burns, Alan and Rob Davis (2013). “Mixed Criticality Systems - A Review”. In: vol. 3rd Edition. Department of Computer Science, University of York. York, UK (cit. on pp. 4, 13, 15, 23, 46, 67).

33. Bustos-Jimenez, Javier, Rodrigo Alonso, Camila Faundez, and Hugo Meric (Sept. 2014). “Boxing experience: Measuring QoS and QoE of multimedia streaming using NS3, LXC and VLC”. In: *Local Computer Networks Workshops (LCN Workshops), 2014 IEEE 39th Conference on*, pp. 658–662. DOI: `10.1109/LCNW.2014.6927717` (cit. on p. 91).

34. Buttazzo, Giorgio C. (2011). *Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Applications*. 3rd. Real-Time Systems Series. New York: Springer (cit. on pp. 34, 72, 75, 102, 108).

35. Buxton, Bill (2007). *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann (cit. on p. 25).

36. Calarco, Giorgio and Maurizio Casoni (2013). “On the effectiveness of Linux containers for network virtualization”. In: *Simulation Modelling Practice and Theory* 31, pp. 169–185. DOI: `10.1016/j.simpat.2012.11.007` (cit. on p. 48).

37. Cantril, Bryan and Jeff Bonwick (2008). “Real-World Concurrency”. In: *ACM Queue* 6, pp. 16–25 (cit. on pp. 30, 79).

38. Carroll, Lewis (1887). *Through the Looking-glass*. Plain Label Books (cit. on p. 187).

39. Charette, Robert N. (Feb. 2009). “This Car Runs on Code”. In: *IEEE Spectrum* (cit. on pp. 1, 3, 10, 13, 53).

40. Charette, Robert N. (Feb. 2011). “Smart Cars and the Need for Smarter Software”. In: *IEEE Spectrum* (cit. on p. 42).

41. Chung, Lawrence and Julio Cesar Sampaio Prado Leite (2009). "On Non-Functional Requirements in Software Engineering". In: *Conceptual Modeling: Foundations and Applications*. Ed. by Alexander T. Borgida, Vinay K. Chaudhri, Paolo Giorgini, and Eric S. Yu. Vol. 5600. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 363–379. DOI: `10.1007/978-3-642-02463-4_19`. URL: `http://dx.doi.org/10.1007/978-3-642-02463-4_19` (cit. on pp. 33, 35).

42. CMMI Product Team (Oct. 2010). *CMMI for Development*. Technical Report CMU/SEI-2010-TR-033, V1.3, Software Engineering Institute, Carnegie Mellon. URL: `http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9661` (cit. on pp. 156, 158).

43. Coelingh, Erik and Stefan Solyom (Nov. 2012). "All Aboard the Robotic Road Train". In: *IEEE Spectrum* 49.11, pp. 34–39. DOI: `10.1109/MSPEC.2012.6341202` (cit. on p. 159).

44. Continental (2014). *AutoLinQ*. URL: `http://www.conti-online.com/www/automotive_de_en/themes/passenger_cars/interior/connectivity/pi_autolinq_en.html` (visited on Oct. 10, 2014) (cit. on p. 49).

45. Crnković, Ivica, Séverine Sentilles, Aneta Vulgarakis, and Michael R. V. Chaudron (2011a). "A Classification Framework for Software Component Models". In: *Software Engineering, IEEE Transactions on* 37.5, pp. 593–615. DOI: `10.1109/TSE.2010.83` (cit. on p. 33).

46. Crnković, Ivica, Judith Stafford, and Clemens Szyperski (May 2011b). "Software Components beyond Programming: From Routines to Services". In: *Software, IEEE* 28.3, pp. 22–26. DOI: `10.1109/MS.2011.62` (cit. on pp. 15, 68).

47. Dar, Kashif, Mohamed Bakhouya, Jaafar Gaber, Maxime Wack, and Pascal Lorenz (May 2010). "Wireless Communication Technologies for ITS Applications". In: *IEEE Communications Magazine* 48.5, pp. 156–162. DOI: `10.1109/MCOM.2010.5458377` (cit. on p. 28).

48. Daub, Daniel Tobias (Jan. 2012). "Synchronisation von Kernel-based Virtual Machines durch Interrupts". Bachelor's Thesis. Faculty of Computer Science, h_da Hochschule Darmstadt, University of Applied Sciences (cit. on p. 92).

49. Davis, Robert I. and Alan Burns (Oct. 2011). "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems". In: *ACM Comput. Surv.* 43 (4), 35:1–35:44. DOI: `10.1145/1978802.1978814` (cit. on p. 34).

50. Deming, W. Edwards (2000). *Out of the crisis*. MIT Press (cit. on p. 54).

51. Di Natale, Marco (2008). “Design and Development of Component-Based Embedded Systems for Automotive Applications”. In: *Ada-Europe ’08: Proceedings of the 13th Ada-Europe international conference on Reliable Software Technologies*. Venice, Italy: Springer, pp. 15–29. DOI: `10.1007/978-3-540-68624-8_2` (cit. on pp. 17, 19).

52. Di Natale, Marco and Alberto Sangiovanni-Vincentelli (Apr. 2010). “Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools”. In: *Proceedings of the IEEE* 98.4, pp. 603–620. DOI: `10.1109/JPROC.2009.2039550` (cit. on p. 44).

53. Dodig-Crnković, Gordana (Apr. 2002). “Scientific Methods in Computer Science”. In: *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden* (cit. on p. 6).

54. Ebert, Christof and Capers Jones (2009). “Embedded Software: Facts, Figures, and Future”. In: *Computer* 42, pp. 42–52. DOI: `10.1109/MC.2009.118` (cit. on p. 10).

55. Ebert, Christof and Jürgen Salecker (2009). “Guest Editors’ Introduction: Embedded Software”. In: *IEEE Software* 26, pp. 14–18. DOI: `10.1109/MS.2009.70` (cit. on p. 13).

56. Eichhorn, Michael, Martin Pfannenstein, and Eckehard Steinbach (2010). “A Flexible In-Vehicle HMI Architecture based on Web Technologies”. In: *Proceedings of the 2nd international workshop on Multimodal interfaces for automotive applications*. MIAA ’10. Hong Kong, China: ACM, pp. 9–12. DOI: `10.1145/2002368.2002374` (cit. on pp. 85, 130).

57. Eklund, Ulrik, Niklas Jonsson, Jan Bosch, and Anders Eriksson (2012). “A Reference Architecture Template for Software-intensive Embedded Systems”. In: *Proceedings of the WICSA/ECSA 2012 Companion Volume*. WICSA/ECSA ’12. Helsinki, Finland: ACM, pp. 104–111. DOI: `10.1145/2361999.2362022` (cit. on pp. 145, 146).

58. Fischer, Clemens (Aug. 2009). “Examining the Potential of Multiple-Operating-System Architectures for Multicore Automotive Embedded Systems”. MA thesis. Darmstadt, DE: Faculty of Computer Science, h_da Hochschule Darmstadt, University of Applied Sciences (cit. on pp. 92, 157).

59. Fleming, Bill (2013). “Advanced Automotive Electronics [Automotive Electronics]”. In: *Vehicular Technology Magazine, IEEE* 8.4, pp. 4–12. DOI: `10.1109/MVT.2013.2281677` (cit. on p. 159).

60. Flynn, Michael J. (Sept. 1972). “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9, pp. 948–960. DOI: `10.1109/TC.1972.5009071` (cit. on p. 102).

61. Fries, Stephan (Aug. 2013). "Hardwarerendering innerhalb virtualisierter Systeme & Composited User Interfaces". Master's thesis. h_da Hochschule Darmstadt, University of Applied Sciences (cit. on p. 137).

62. Fries, Stephan, Andreas Theis, and Andreas Knirsch (Dec. 2013). *Benutzeroberflächen von Anwendungen aus unterschiedlichen VMs mittels Wayland komponieren.* Technical Report ICM-TR 2013-02. ICM Labs, Faculty of Computer Science, h_da University of Applied Sciences Darmstadt (cit. on p. 137).

63. Gathmann, Dustin (Nov. 2013). "Entwurf und Entwicklung eines kontextgesteuerten Audiokompositors für ein verteiltes eingebettetes Multimediasystem". Master's thesis. Faculty of Computer Science, h_da Hochschule Darmstadt, University of Applied Sciences (cit. on p. 139).

64. GENIVI (2014). *GENIVI Alliance.* URL: `http://www.genivi.org` (visited on Sept. 10, 2014) (cit. on pp. 47, 130).

65. GENIVI Alliance (2013). *GENIVI Open Source Project List.* Whitepaper. San Ramon, CA, USA: GENIVI Alliance. URL: `http://www.genivi.org/sites/default/files/Open_Source_Projects_Flyers_10252013.pdf` (cit. on p. 48).

66. GENIVI Alliance (2014). *BMW Case Study.* Case Study. San Ramon, CA, USA: GENIVI Alliance. URL: `http://www.genivi.org/sites/default/files/BMW_Case_Study_Download_040914.pdf` (cit. on p. 48).

67. Germonprez, Matt, J. P. Allen, Brian Warner, Jamie Hill, and Glenn McClements (Nov. 2013). "Open Source Communities of Competitors". In: *interactions* 20.6, pp. 54–59. DOI: `10.1145/2527191` (cit. on p. 47).

68. Glaab, Markus, Woldemar Fuhrmann, and Joachim Wietzke (May 2014a). "Transparent Data for the M2M Service Capability Layer: Benefits and Approaches". In: *International Conference on Telecommunications: Cooperation for a United World (ICT2014).* Lisbon, Portugal: IEEE Communications Society (cit. on pp. 27, 157).

69. Glaab, Markus, Woldemar Fuhrmann, Joachim Wietzke, and Bogdan V Ghita (July 2014b). "A M2M-based Automotive Service Delivery Platform for Distributed Vehicular Applications". In: *Proceedings of the Tenth International Network Conference (INC 2014).* Plymouth, UK, pp. 35–45 (cit. on p. 27).

70. Goldberg, Robert P. (Feb. 1973). *Architectural Principles for Virtual Computer Systems.* Thesis ESD-TR-73-105. Cambridge, Massachusetts: Div. Engineering & Applied Physics, Harvard University (cit. on p. 90).

71. Goldberg, Robert P. (June 1974). "Survey of Virtual Machine Research". In: *Computer* 7.6, pp. 34–45 (cit. on p. 87).

72. Google (2014). *Android Auto.* URL: `http://www.android.com/auto/` (visited on Oct. 10, 2014) (cit. on p. 29).

73. Göschel, Burkhard (Sept. 2012). "Connectivity - Treiber für den Wandel der Automobilindustrie". In: *carIT-Kongress - Das Automobil der Zukunft ist vernetzt.* Media-Manufaktur (cit. on pp. 2, 62, 153).

74. Gryc, Andy and Marc Lapierre (June 2012). *Why HTML5 Is Becoming the HMI Technology of Choice.* Whitepaper (cit. on pp. 130, 151).

75. *gstreamer - open source multimedia framework* (Sept. 10, 2014). URL: `http://gstreamer.freedesktop.org` (visited on Sept. 10, 2014) (cit. on p. 139).

76. Hagiescu, Andrei, Unmesh D. Bordoloi, Samarjit Chakraborty, Prahladavaradan Sampath, P. Vignesh V. Ganesan, and S. Ramesh (June 2007). "Performance Analysis of FlexRay-based ECU Networks". In: *44th ACM/IEEE Design Automation Conference, 2007. DAC '07.* Pp. 284–289 (cit. on p. 108).

77. Heiser, Gernot (2008). "The role of virtualization in embedded systems". In: *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems.* Glasgow, Scotland: ACM, pp. 11–16. DOI: `10.1145/1435458.1435461` (cit. on p. 88).

78. Heiser, Gernot (2011). "Virtualizing Embedded Systems: Why Bother?" In: *Proceedings of the 48th Design Automation Conference.* DAC '11. San Diego, California: ACM, pp. 901–905. DOI: `10.1145/2024724.2024925` (cit. on p. 88).

79. Heling, Günther, Jochen Rein, and Patrick Markl (2012). "Koexistenz von sicherer und nicht-sicherer Software auf einem Steuergerät". German. In: *ATZelektronik* 7.7, pp. 62–65. DOI: `10.1365/s35658-012-0216-9` (cit. on p. 24).

80. Hergenhan, André and Gernot Heiser (Nov. 2008). "Operating Systems Technology for Converged ECUs". In: *6th Embedded Security in Cars Conference (escar).* Hamburg, Germany (cit. on pp. 47, 88).

81. Hoare, Tony and Robin Milner (2005). "Grand Challenges for Computing Research". In: *The Computer Journal* 48.1, pp. 49–52. DOI: `10.1093/comjnl/bxh065` (cit. on p. 22).

82. Höfer, Andreas and Walter F. Tichy (2007). "Status of Empirical Research in Software Engineering". In: *Empirical Software Engineering Issues. Critical Assessment and Future Directions.* Vol. 4336. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 10–19. DOI: `10.1007/978-3-540-71301-2_3` (cit. on p. 6).

83. Høgsberg, Kristian (2012). "Wayland - A new graphics architecture". In: *Free and Open source Software Developers' European Meeting (FOSDEM)* (cit. on p. 130).

84. Holle, Jan, André Groll, Christoph Ruland, Hakan Cankaya, and Marko Wolf (2011). "Open Platforms on the Way to Automotive Practice". In: *8th ITS European Congress*. Lyon (cit. on p. 130).

85. Holstein, Tobias, Markus Wallmyr, and Joachim Wietzke (2015). "Current Challenges in Compositing Heterogeneous User-Interfaces for Automotive Purposes". In: *Human-Computer Interaction, 17th International Conference, HCI International 2015*. Lecture Notes in Computer Science. forthcoming. Springer (cit. on pp. 131, 157).

86. Holt, Jim, Anant Agarwal, Sven Brehmer, Max Domeika, Patrick Griffin, and Frank Schirrmeister (2009). "Software Standards for the Multicore Era". In: *IEEE Micro* 29, pp. 40–51. DOI: 10.1109/MM.2009.48 (cit. on p. 30).

87. Holzmann, Gerard J. (2013). "Landing a Spacecraft on Mars". In: *Software, IEEE* 30.2, pp. 83–86. DOI: 10.1109/MS.2013.32 (cit. on p. 60).

88. Horberry, Tim, Janet Anderson, Michael A. Regan, Thomas J. Triggs, and John Brown (2006). "Driver distraction: The effects of concurrent in-vehicle tasks, road environment complexity and age on driving performance". In: *Accident Analysis and Prevention* 38.1, pp. 185–191. DOI: 10.1016/j.aap.2005.09.007 (cit. on p. 20).

89. Horn, W. A. (1974). "Some simple scheduling algorithms". In: *Naval Research Logistics Quarterly* 21.1, pp. 177–185. DOI: 10.1002/nav.3800210113 (cit. on p. 72).

90. Hoyle, David (2000). *Automotive Quality Systems Handbook*. Butterworth-Heinemann (cit. on p. 55).

91. Hoyle, David (2005). *Automotive Quality Systems Handbook*. 2nd Edition. Butterworth-Heinemann (cit. on pp. 54, 55, 57, 189).

92. Hudelmaier, Philipp (2014). "Funktionen vereint – Kombiinstrument, Infotainment und Flottenmanagement". German. In: *Vernetztes Automobil*. Ed. by Wolfgang Siebenpfeiffer. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, pp. 185–190. DOI: 10.1007/978-3-658-04019-2_27 (cit. on pp. 26, 130).

93. Hüger, Fabian (2011). "User Interface Transfer for Driver Information Systems: A Survey and an Improved Approach". In: *Proceedings of the 3rd International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. AutomotiveUI '11. Salzburg, Austria: ACM, pp. 113–120. DOI: 10.1145/2381416.2381435 (cit. on p. 29).

94. Huth, Nathalie and Christian Spahr (2011). *Studie Automobil – ITK im Auto und Elektromobilität*. Study. Berlin, DE: BITKOM – Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. (cit. on p. 1).

95. *IATF - International Automotive Task Force Global Oversight* (2014). URL: `http://www.iatfglobaloversight.org` (visited on Aug. 1, 2014) (cit. on p. 55).

96. ICAO (2013). *Safety Management Manual (SMM)*. Tech. rep. Doc 9859, 3rd Edition. International Civil Aviation Organization (cit. on pp. 18, 24).

97. ICM Labs (2010). *OpenICM Framework*. Tech. rep. http://fbi.h-da.de/˜openicm, last checked 19.04.2011. Faculty of Computer Science, University of Applied Sciences Darmstadt (cit. on p. 49).

98. IDC (Feb. 2014). *IDC Worldwide Mobile Phone Tracker*. market research report. International Data Corporation (IDC) (cit. on p. 48).

99. IEC (2014). *IEC 61508: Functional Safety - IEC 61508 Explained*. URL: `http://www.iec.ch/functionalsafety/explained/` (visited on Aug. 1, 2014) (cit. on p. 188).

100. IEC 61508 (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems*. Standard, IEC 61508 ed2.0, Parts 1 to 7. Geneva, CH: International Electrotechnical Commision (cit. on p. 23).

101. IEEE 1609 (June 2007). *Wireless Access in Vehicular Environments (WAVE)*. Standards Series, IEEE 1609. IEEE Vehicular Technology Society (cit. on p. 28).

102. IEEE 802.11p (2010). *IEEE Standard for Information Technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, Amenedment 6: Wireless Access in Vehicular Environments*. Standard, 802.11p-2010. New York, NY, USA: IEEE Computer Society (cit. on p. 28).

103. ISO 17356-3:2005 (2005). *Road vehicles — Open interface for embedded automotive applications — Part 3: OSEK/VDX Operating System (OS)*. Standard, ISO 17356-3:2005. ISO/TC 22 Road vehicles. Geneva, CH: International Organization for Standardization (cit. on p. 40).

104. ISO 24765 (Dec. 2010). *Systems and software engineering – Vocabulary*. Standard, ISO/IEC/IEEE 24765:2010(E). Geneva, CH: International Organization for Standardization. DOI: `10.1109/IEEESTD.2010.5733835` (cit. on pp. 31, 43, 187, 190).

105. ISO 25010 (2011). *Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) - system and softwarwe quality models*. Standard, 25010:201. Geneva, CH: International Organization for Standardization (cit. on pp. 31, 45, 146).

106. ISO 26262 (2011). *Road vehicles — Functional safety*. Standard, ISO 26262-1:11 to 9:11 and ISO 26262-10:12. ISO/TC 22 Road vehicles. Geneva, CH: International Organization for Standardization (cit. on pp. 4, 11, 23, 156, 188).

107. ISO 7736:84 (1984). *Road vehicles — Car radio for front installation — Installation space including connections.* Standard, ISO 7736:1984. ISO/TC 22 Road vehicles. Geneva, CH: International Organization for Standardization (cit. on p. 2).

108. ISO/IEC 9126 (2001). *Software engineering – Product quality – Part 1: Quality model.* Standard, ISO/IEC 9126-1:2001. revised by ISO/IEC 25010:2011. Geneva, CH: International Organization for Standardization (cit. on p. 31).

109. ISO/IEC/IEEE 9945 (2008). *Information technology - Portable Operating System Interface (POSIX).* Base Specifications, Issue 7, First edition IEEE Std 1003.1-2008. IEEE and The Open Group. DOI: `10.1109/IEEESTD.2009.5393893` (cit. on pp. 40, 189).

110. ISO/TS 16949 (2009). *Quality management systems – Particular requirements for the application of ISO 9001:2008 for automotive production and relevant service part organizations.* Standard, ISO/TS 16949:2009. ISO/TC 176 Quality management and quality assurance. Geneva, CH: International Organization for Standardization (cit. on pp. 23, 54).

111. *JAMA - Japan Automobile Manufacturers Association, Inc.* (2014). URL: `http://www.jama-english.jp` (visited on Aug. 1, 2014) (cit. on p. 55).

112. Jaouani, Houda, Rim Bouhouch, Wafa Najjar, and Salem Hasnaoui (June 2012). “Hybrid Task and Message Scheduling in Hard Real Time Distributed Systems over FlexRay Bus”. In: *International Conference on Communications and Information Technology (ICCIT).* IEEE Computer Society, pp. 21–26. DOI: `10.1109/ICCITechnol.2012.6285795` (cit. on pp. 108, 109).

113. Jensen, Mark and Tony Mascolo (2010). “AUTOSAR as a Key Enabler for Collaborative Product Development”. In: *SAE International Journal of Passenger Cars-Electronic and Electrical Systems* 3.2, pp. 193–203 (cit. on p. 46).

114. Johnson, Kerry, Jason Clarke, Paul Leroux, and Robert Craig (2006). *OS Partitioning for Embedded Systems.* Tech. rep. QNX Software Systems (cit. on p. 86).

115. Kaiser, Robert (2011). “Applicability of Virtualization to Embedded Systems”. In: *Solutions on Embedded Systems.* Vol. 81. Lecture Notes in Electrical Engineering. Springer, pp. 215–226. DOI: `10.1007/978-94-007-0638-5` (cit. on p. 88).

116. Kalaimagal, Sivamuni and Rengaramanujam Srinivasan (Oct. 2008). “A Retrospective on Software Component Quality Models”. In: *SIGSOFT Softw. Eng. Notes* 33.6, pp. 1–10. DOI: `10.1145/1449603.1449611` (cit. on p. 31).

117. Kanda, Wataru, Yu Murata, and Tatsuo Nakajima (2010). “SIGMA System: A Multi-OS Environment for Embedded Systems”. In: *Journal of Signal Processing Systems* 59.1, pp. 33–43. DOI: `10.1007/s11265-008-0272-9` (cit. on p. 87).

118. Kartha, C. Peeth (2004). “A comparison of ISO 9000:2000 quality system standards, QS9000, ISO/TS 16949 and Baldrige criteria”. In: *The TQM Magazine* 16.5, pp. 331–340. DOI: `10.1108/09544780410551269` (cit. on p. 55).

119. Kern, Dagmar and Albrecht Schmidt (2009). “Design Space for Driver-based Automotive User Interfaces”. In: *Proceedings of the 1st International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. AutomotiveUI ’09. Essen, Germany: ACM, pp. 3–10. DOI: `10.1145/1620509.1620511` (cit. on p. 25).

120. Kerrisk, Michael (2010). *The Linux Programming Interface*. No Starch Press (cit. on p. 72).

121. Keul, Steffen and Helmut Brock (July 2013). “Mixed-ASIL-Systeme praktisch realisieren”. In: *Elektronik Funktionale Sicherheit* 1 (cit. on p. 23).

122. Keul, Steffen, Eduard Metzker, and Dieter Lederer (2013). “Seamless Implementation of ECU Software based on ISO 26262”. English. In: *ATZelektronik worldwide* 8.5, pp. 10–15. DOI: `10.1365/s38314-013-0192-8` (cit. on p. 23).

123. Kim, Daeyoung and Yann-Hang Lee (2002). “Periodic and Aperiodic Task Scheduling in Strongly Partitioned Integrated Real-time Systems”. In: *The Computer Journal* 45.4, pp. 395–409. DOI: `10.1093/comjnl/45.4.395` (cit. on pp. 77, 78).

124. Kindel, Olaf and Mario Friedrich (2009). *Softwareentwicklung mit AUTOSAR*. dpunkt.verlag (cit. on p. 46).

125. Kivity, Avi, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori (2007). “kvm: the Linux Virtual Machine Monitor”. In: *Proceedings of the 2007 Linux Symposium*. Vol. 1, pp. 225–230 (cit. on pp. 87, 136).

126. Knight, John C. (May 2002). “Safety Critical Systems: Challenges and Directions”. In: *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pp. 547–550 (cit. on p. 4).

127. Knirsch, Andreas (Sept. 2009). “Fast-Startup Concept for Embedded Systems”. Master’s thesis. h_da Hochschule Darmstadt, University of Applied Sciences (cit. on p. 50).

128. Knirsch, Andreas, Pierre Schnarz, and Joachim Wietzke (2012a). “Prioritized Access Arbitration to Shared Resources on Integrated Software Systems in Multicore Environments”. In: *3rd IEEE International Conference on Networked Embedded Systems for Every Application (NESEA)*, pp. 1–8. DOI: `10.1109/NESEA.2012.6474014` (cit. on p. 235).

129. Knirsch, Andreas, Pierre Schnarz, and Joachim Wietzke (Dec. 2013a). “SHARB: Shared Resource Arbitration in Partitioned Multicore Systems via Library Interposition”. In: *International Journal of Design, Analysis and Tools for Integrated Circuits and Systems (IJDATICS)* 4.2, pp. 18–27 (cit. on p. 235).

130. Knirsch, Andreas, Andreas Theis, Joachim Wietzke, and Ronald Moore (2013b). “Compositing User Interfaces in Partitioned In-Vehicle Infotainment”. In: *Mensch & Computer 2013 - Workshopband*. Ed. by Susanne Boll-Westermann, Susanne Maaß, and Rainer Malaka. München: Oldenbourg Verlag, pp. 63–70 (cit. on p. 235).

131. Knirsch, Andreas, Sergio Vergata, and Joachim Wietzke (2012b). “Strukturierung von Multimediasystemen für Fahrzeuge”. In: *Echtzeit*. Ed. by Wolfgang A. Halang. Informatik Aktuell. Springer, pp. 69–78. DOI: 10.1007/978-3-642-33707-9_8 (cit. on pp. 189, 235).

132. Knirsch, Andreas, Joachim Wietzke, Ronald Moore, and Paul S. Dowland (Nov. 2010). “An Approach for Structuring Heterogeneous Automotive Software Systems by use of Multicore Architectures”. In: *Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*. Plymouth, UK, pp. 19–30 (cit. on p. 236).

133. Knirsch, Andreas, Joachim Wietzke, Ronald Moore, and Paul S. Dowland (Nov. 2011). “Resource Management for Multicore Aware Software Architectures of In-Car Multimedia Systems”. In: *Informatik schafft Communities*. Ed. by Hans-Ulrich Heiß, Peter Pepper, Holger Schlingloff, and Jörg Schneider. Vol. P-192. Lecture Notes in Informatics (LNI). Berlin: Gesellschaft für Informatik (GI), p. 216. URL: http://www.user.tu-berlin.de/komm/CD/paper/060142.pdf (cit. on p. 236).

134. Knirsch, Andreas, Joachim Wietzke, Ronald Moore, and Paul S. Dowland (July 2014). “Connected In-Car Multimedia: Qualities affecting Composability of Dynamic Functionality”. In: *Proceedings of the Tenth International Network Conference (INC 2014)*. Plymouth, UK, pp. 81–93 (cit. on p. 235).

135. Kopetz, Hermann and Roman Obermaisser (Aug. 2002). “Temporal composability”. In: *Computing Control Engineering Journal* 13.4, pp. 156–162. DOI: 10.1049/cce:20020401 (cit. on pp. 5, 63, 68, 73).

136. Kopetz, Hermann, Roman Obermaisser, Christian El Salloum, and Bernhard Huber (2007). “Automotive Software Development for a Multi-Core System-on-a-Chip”. In: *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*. IEEE Computer Society, pp. 2–7. DOI: 10.1109/SEAS.2007.2 (cit. on p. 80).

137. Kreiker, Jörg, Andrzej Tarlecki, Moshe Y. Vardi, and Reinhard Wilhelm (2011a). “Dagstuhl Manifesto: Formal methods – just a Euroscience?” In: *Informatik-Spektrum* 34.4, pp. 413–414. DOI: 10.1007/s00287-011-0552-9 (cit. on p. 15).

138. Kreiker, Jörg, Andrzej Tarlecki, Moshe Y. Vardi, and Reinhard Wilhelm (2011b). "Modeling, Analysis, and Verification - The Formal Methods Manifesto 2010 (Dagstuhl Perspectives Workshop 10482)". In: *Dagstuhl Manifestos* 1.1. Ed. by Jörg Kreiker, Andrzej Tarlecki, Moshe Y. Vardi, and Reinhard Wilhelm, pp. 21–40. DOI: `10.4230/DagMan.1.1.21` (cit. on pp. 15, 23, 190).

139. Kriens, Peter (2008). "How OSGi Changed My Life". In: *ACM Queue* 6.1, pp. 44–51. DOI: `10.1145/1348583.1348594` (cit. on p. 86).

140. Krishna, C.M. and Y.H. Lee (1991). "Real-Time Systems". In: *Computer* 24, pp. 10–11. DOI: `10.1109/MC.1991.10041` (cit. on p. 34).

141. Krüger, Ingolf H., Edward C. Nelson, and K. Venkatesh Prasad (2004). "Service-based software development for automotive applications". In: *Convergence International Congress & Exposition On Transportation Electronics*. SAE International (cit. on p. 85).

142. Lederer, Dieter and Christof Ebert (2008). "Funktionale Sicherheit–Das Gesamtsystem Fahrzeug". In: *Hanser Automotive* 10, pp. 20–24 (cit. on p. 23).

143. Ledinot, Emmanuel, Jean-Marc Astruc, Jean-Paul Blanquart, Philippe Baufreton, Jean-Louis Boulanger, Hervé Delseny, Jean Gassino, Gérard Ladier, Michel Leeman, Joseph Machrouh, et al. (Feb. 2012). "A Cross-Domain Comparison of Software Development Assurance Standards". In: *Procedings of Embedded Real Time Software and Systems (ERTS^2 2012)*. Toulouse, FR (cit. on p. 24).

144. Lee, Edward A. (May 2006). "The Future of Embedded Software". In: *ARTEMIS Conference*. Graz (cit. on p. 13).

145. Lee, Edward A. (May 2007). *Computing Foundations and Practice for Cyber- Physical Systems: A Preliminary Report*. Technical Report UCB/EECS-2007-72. EECS Department, University of California, Berkeley (cit. on p. 13).

146. Lee, Yann-Hang, Daeyoung Kim, Mohamed Younis, Jeff Zhou, and James McElroy (2000). "Resource Scheduling in Dependable Integrated Modular Avionics". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 14–23. DOI: `10.1109/ICDSN.2000.857509` (cit. on p. 77).

147. Leveson, Nancy G. (1995). *Safeware: System Safety and Computers*. Boston, MA, USA: Addison Wesley (cit. on p. 18).

148. Leveson, Nancy G. (2011). *Engineering a Safer World: Systems Thinking Applied to Safety*. Cambridge, Massachusetts: MIT Press (cit. on pp. 18, 21, 23, 69, 187).

149. Levy, Markus and Thomas M Conte (June 2009). "Embedded Multicore Processors and Systems". In: *IEEE Micro* 29, pp. 7–9. DOI: `10.1109/MM.2009.41` (cit. on pp. 30, 79).

150. Li, Chuanpeng, Chen Ding, and Kai Shen (2007). “Quantifying the Cost of Context Switch”. In: *Proceedings of the 2007 Workshop on Experimental Computer Science*. ExpCS ’07. San Diego, California: ACM. DOI: `10.1145/1281700.1281702` (cit. on p. 74).

151. Liedtke, Jochen (1995). “On μ-Kernel Construction”. In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. SOSP ’95. Copper Mountain, Colorado, United States: ACM, pp. 237–250. DOI: `10.1145/224056.224075` (cit. on p. 39).

152. Liu, Fang and Yan Solihin (Dec. 2010). “Understanding the Behavior and Implications of Context Switch Misses”. In: *ACM Trans. Archit. Code Optim.* 7.4, 21:1–21:28. DOI: `10.1145/1880043.1880048` (cit. on p. 73).

153. Love, Robert (July 2003). “CPU Affinity”. In: *Linux Journal* Issue 111 (cit. on p. 81).

154. Love, Robert (2010). *Linux Kernel Development*. 3rd. Developer’s Library. Pearson Education (cit. on pp. 72, 81, 84, 104).

155. Lukasiewycz, Martin, Michael Glaß, Jürgen Teich, and Paul Milbredt (2009). “FlexRay Schedule Optimization of the Static Segment”. In: *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS ’09. Grenoble, France: ACM, pp. 363–372. DOI: `10.1145/1629435.1629485` (cit. on p. 108).

156. *LXC - Linux Containers* (Oct. 10, 2014). URL: `https://linuxcontainers.org` (visited on Oct. 10, 2014) (cit. on p. 48).

157. Macario, Gianpaolo, Marco Torchiano, and Massimo Violante (July 2009). “An In-Vehicle Infotainment Software Architecture Based on Google Android”. In: *IEEE International Symposium on Industrial Embedded Systems (SIES ’09)*, pp. 257–260. DOI: `10.1109/SIES.2009.5196223` (cit. on p. 49).

158. Macdonell, A Cameron (2011). “Shared-Memory Optimizations for Virtual Machines”. PhD thesis. Edmonton, Alberta: University of Alberta (cit. on pp. 88, 136).

159. Machrouh, Joseph, Jean-Paul Blanquart, Philippe Baufreton, Jean-Louis Boulanger, Hervé Delseny, Jean Gassino, Gérard Ladier, Emmanuel Ledinot, Michel Leeman, Jean-Marc Astruc, et al. (Feb. 2012). “Cross domain comparison of System Assurance”. In: *Procedings of Embedded Real Time Software and Systems (ERTS^2 2012)*. Toulouse, FR (cit. on p. 24).

160. Mahlke, Sascha (2005). “Understanding users’ experience of interaction”. In: *Proceedings of the 2005 annual conference on European association of cognitive ergonomics*. EACE ’05. Chania, Greece: University of Athens, pp. 251–254. URL: `http://dl.acm.org/citation.cfm?id=1124666.1124702` (cit. on p. 190).

161. Manadhata, Pratyusa and Jeannette M. Wing (July 2005). *An Attack Surface Metric.* Tech. rep. CMU-CS-05-155. Pittsburgh, PA, USA: School of Computer Science, Carnegie Mellon University (cit. on p. 38).

162. Martin, Grant (2002). "UML for Embedded Systems Specification and Design: Motivation and Overview". In: *Proceedings of the Conference on Design, Automation and Test in Europe.* DATE '02. Washington, DC, USA: IEEE Computer Society, pp. 773–775. DOI: 10.1109/DATE.2002.998386 (cit. on pp. 15, 16, 33).

163. Marwedel, Peter (2011). *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems.* 2nd Edition. Springer. DOI: 10.1007/978-94-007-0257-8 (cit. on p. 12).

164. Maurer, Markus (2013). "Automotive Systems Engineering: A Personal Perspective". English. In: *Automotive Systems Engineering.* Ed. by Markus Maurer and Hermann Winner. Springer Berlin Heidelberg, pp. 17–35. DOI: 10.1007/978-3-642-36455-6_2 (cit. on p. 55).

165. McVoy, Larry and Carl Staelin (1996). "lmbench: Portable Tools for Performance Analysis". In: *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference.* Usenix Association (cit. on p. 88).

166. Medvidovic, Nenad and Richard N. Taylor (2010). "Software Architecture: Foundations, Theory, and Practice". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2.* ICSE '10. Cape Town, South Africa: ACM, pp. 471–472. DOI: 10.1145/1810295.1810435 (cit. on p. 67).

167. Mehr, Nicole (Jan. 2014). *HARMAN Announces Next-Generation Scalable Infotainment Platform.* Press Release. HARMAN International Industries (cit. on p. 24).

168. Mehta, Nikunj R., Nenad Medvidovic, and Sandeep Phadke (2000). "Towards a Taxonomy of Software Connectors". In: *Proceedings of the 22Nd International Conference on Software Engineering.* ICSE '00. Limerick, Ireland: ACM, pp. 178–187. DOI: 10.1145/337180.337201 (cit. on p. 38).

169. Mitis, Francesco and Dinesh Sehti (2013). *European Facts and Global Status Report on Road Safety 2013.* Fact Sheet. Copenhagen, DK: World Health Organisation, Regional Office for Europe (cit. on p. 19).

170. Monot, Aurélien, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion (2012). "Multi-source software on multicore automotive ECUs - Combining runnable sequencing with task scheduling". In: *IEEE Transactions on Industrial Electronics* 59.10, pp. 3934–3942. DOI: 10.1109/TIE.2012.2185913 (cit. on pp. 2, 4, 13, 20, 46).

171. Moolenbroek, David C. van, Raja Appuswamy, and Andrew S. Tanenbaum (2014). "Towards a Flexible, Lightweight Virtualization Alternative". In: *Proceedings of International Conference on Systems and Storage*. SYSTOR 2014. Haifa, Israel: ACM, 8:1–8:7. DOI: 10.1145/2611354.2611369 (cit. on p. 92).

172. Mössinger, Jürgen (2010). "Software in Automotive Systems". In: *IEEE Software* 27.2, pp. 92–94. DOI: 10.1109/MS.2010.55 (cit. on pp. 3, 4, 46).

173. Mücke, Marc (July 2014). "Weck- und Synchronisationsmechanismen auf AMP-Systemen". Bachelor's Thesis. Faculty of Computer Science, h_da Hochschule Darmstadt, University of Applied Sciences (cit. on p. 92).

174. Nagarajan, Shiv and Vulpe Nicola (2009). *Processor Affinity or Bound Multiprocessing? Easing the Migration to Embedded Multicore Processing.* QNX Software Systems. 175 Terence Matthews Crescent, Ottawa, Ontario, Canada (cit. on p. 81).

175. Nakhimovsky, Greg (July 2001). "Building library interposers for fun and profit". In: *Unix Insider* (cit. on p. 110).

176. Navet, Nicolas, Aurélien Monot, Bernard Bavoux, and Françoise Simonot-Lion (July 2010). "Multi-source and multicore automotive ECUs - OS protection mechanisms and scheduling". In: *IEEE International Symposium on Industrial Electronics (ISIE)*, pp. 3734–3741. DOI: 10.1109/ISIE.2010.5637677 (cit. on p. 20).

177. Neiger, Gil, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig (Aug. 2006). "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". In: *Intel Technology Journal* 10.3. DOI: 10.1535/itj.1003 (cit. on pp. 87, 88).

178. Nesbit, Kyle J., Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith (May 2008). "Multicore Resource Management". In: *IEEE Micro* 28 (3), pp. 6–16. DOI: 10.1109/MM.2008.43 (cit. on p. 123).

179. Nett, Tillmann and Jörn Schneider (2013). "Running Linux and AUTOSAR side by side". In: *7th Junior Researcher Workshop on Real-Time Computing*. Sophia Antipolis, Frankreich, pp. 29–32 (cit. on p. 47).

180. Neumann, Peter G (2004). "Principled assuredly trustworthy composable architectures". In: *Final report for Task* 1 (cit. on pp. 3, 15, 18, 32, 35, 36, 187–189).

181. Neumann, Peter G. (2006). "System and network trustworthiness in perspective". In: *Proceedings of the 13th ACM conference on Computer and communications security*. CCS '06. Alexandria, Virginia, USA: ACM, pp. 1–5. DOI: 10.1145/1180405.1180406 (cit. on p. 5).

182. NHTSA (2012). *Visual-Manual NHTSA Driver Distraction Guidelines for In-Vehicle Electronic Devices.* NHTSA Guideline Docket No. NHTSA-2010-0053. National Highway Traffic Safety Administration (NHTSA), U.S. Department of Transportation (DOT) (cit. on pp. 20, 188).

183. NHTSA (Apr. 2013). *Distracted Driving 2011.* Research Note DOT HS 811 737. Washington, DC, USA: National Highway Traffic Safety Administration (NHTSA), U.S. Department of Transportation (DOT) (cit. on p. 20).

184. Olariu, Stephan (2007). “Peer-to-peer Multimedia Content Provisioning for Vehicular Ad Hoc Networks”. In: *Proceedings of the 3rd ACM Workshop on Wireless Multimedia Networking and Performance Modeling.* WMuNeP ’07. Chania, Crete Island, Greece: ACM, pp. 1–1. DOI: `10.1145/1298216.1298217` (cit. on p. 28).

185. Perrow, Charles (2011). *Normal Accidents: Living with High Risk Technologies.* Princeton University Press (cit. on pp. 21, 22).

186. Pfleging, Bastian, Stefan Schneegass, Dagmar Kern, and Albrecht Schmidt (2014). “Vom Transportmittel zum rollenden Computer – Interaktion im Auto”. German. In: *Informatik-Spektrum* 37.5, pp. 418–422. DOI: `10.1007/s00287-014-0804-6` (cit. on pp. 20, 26).

187. Pflug, Enno and Sue Frederick (Jan. 2011). *Continental at CES 2011 in Las Vegas. The Car of the Future is Always On: Apps, HMI and Personalization will fuel Driving Experiences.* Press Release. Schwalbach, Auburn Hills: Continental AG (cit. on pp. 42, 49).

188. Pham, Dac, Jim Holt, and Sanjay Deshpande (2011). “Embedded Multicore Systems: Design Challenges and Opportunities”. English. In: *Multiprocessor System-on-Chip.* Ed. by Michael Hübner and Jürgen Becker. Springer New York, pp. 197–222. DOI: `10.1007/978-1-4419-6460-1_9` (cit. on pp. 79, 80).

189. Pope, Simon and Trudy Muller (July 2011). *Apple’s App Store Downloads Top 15 Billion.* Press Release. Cupertino, CA, USA: Apple Inc. (cit. on pp. 16, 42).

190. Popek, Gerald J. and Robert P. Goldberg (July 1974). “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Commun. ACM* 17.7, pp. 412–421. DOI: `10.1145/361011.361073` (cit. on p. 87).

191. Pretschner, Alexander, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner (2007). “Software Engineering for Automotive Systems: A Roadmap”. In: *FOSE ’07: 2007 Future of Software Engineering.* Washington, DC, USA: IEEE Computer Society, pp. 55–71 (cit. on pp. 1, 53).

192. QNX Software Systems (2010). *QNX Neutrino Adaptive Partitioning.* URL: `http://www.qnx.com/developers/docs/6.5.0/topic/com.qnx.doc.adaptive_partitioning_en/bookset.html` (visited on Nov. 2, 2014) (cit. on p. 86).

193. *Qt Project* (2014). URL: `https://qt-project.org` (visited on Sept. 10, 2014) (cit. on pp. 137, 151).

194. Quain, John R. (Jan. 2011). "Pursuing Cars That Won't Be Outdated". In: *The New York Times*, AU4 of the New York edition (cit. on p. 42).

195. Randell, Brian (June 1975). "System Structure for Software Fault Tolerance". In: *IEEE Transactions on Software Engineering* SE-1.2, pp. 220–232. DOI: `10.1109/TSE.1975.6312842` (cit. on p. 22).

196. Rausch, Andreas, Christian Bartelt, Thomas Ternité, and Marco Kuhrmann (2005). "The V-Modell XT Applied – Model-Driven and Document-Centric Development". In: *3rd World Congress for Software Quality.* Vol. 3, pp. 131–138 (cit. on pp. 55, 56).

197. Rausch, Andreas and Marco Kuhrmann (2011). "A Proposal for Principles and Values from the Perspective of the German Standard It-development Process V-Modell XT". In: *Proceedings of the 2011 International Conference on Software and Systems Process.* ICSSP '11. Waikiki, Honolulu, HI, USA: ACM, pp. 230–233. DOI: `10.1145/1987875.1987917` (cit. on p. 57).

198. Renault (2014). *R-Link, Renault's onboard multimedia system.* URL: `http://group.renault.com/en/passion-2/innovation/renault-a-born-innovator/r-link/` (visited on Oct. 10, 2014) (cit. on p. 49).

199. Reshetova, Elena, Janne Karhunen, Thomas Nyman, and N. Asokan (2014). "Security of OS-Level Virtualization Technologies". In: *Secure IT Systems.* Lecture Notes in Computer Science. Springer, pp. 77–93. DOI: `10.1007/978-3-319-11599-3_5` (cit. on p. 91).

200. Riehle, Dirk and Thomas Gross (Oct. 1998). "Role Model Based Framework Design and Integration". In: *ACM SIGPLAN Notices* 33 (10), pp. 117–133. DOI: `10.1145/286942.286951` (cit. on p. 43).

201. Ross, Philip E. (June 2014). "Robot, you can drive my car". In: *Spectrum, IEEE* 51.6, pp. 60–90. DOI: `10.1109/MSPEC.2014.6821623` (cit. on p. 159).

202. Roto, Virpi, Effie Law, Arnold Vermeeren, and Jettie Hoonhout, eds. (Feb. 2011). *User Experience White Paper - Bringing clarity to the concept of user experience.* Result from Dagstuhl Seminar on Demarcating User Experience, September 15-18, 2010. URL: `http://www.allaboutux.org/uxwhitepaper` (cit. on p. 25).

203. Rümelin, Sonja and Andreas Butz (2013). "How to Make Large Touch Screens Usable While Driving". In: *Proceedings of the 5th International Conference on Automotive User Interfaces and Interactive Vehicular Applications.* AutomotiveUI '13. Eindhoven, Netherlands: ACM, pp. 48–55. DOI: `10.1145/2516540.2516557` (cit. on p. 26).

204. Sangal, N, E Jordan, V Sinha, and D Jackson (2005). “Using dependency models to manage complex software architecture”. In: *OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* San Diego, CA, USA: ACM, pp. 167–176 (cit. on p. 70).

205. Sangiovanni-Vincentelli, Alberto and Marco Di Natale (Oct. 2007). “Embedded System Design for Automotive Applications”. In: *Computer* 40.10, pp. 42–51. DOI: `10.1109/MC.2007.344` (cit. on pp. 53, 60).

206. Schäuffele, Jörg and Thomas Zurawka (2013). *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen.* German. 5th. ATZ/MTZ-Fachbuch. Springer. DOI: `10.1007/978-3-8348-2470-7` (cit. on p. 55).

207. Schleuter, Willibert, Johannes von Stosch, Anne Jacoby, and Christof Horn (2009). *Die sieben Irrtümer des Change Managements: Und wie Sie sie vermeiden.* Campus (cit. on p. 55).

208. Schmidt, Albrecht, Anind K. Dey, Andrew L. Kun, and Wolfgang Spiessl (2010). “Automotive user interfaces: human computer interaction in the car”. In: *CHI ’10 Extended Abstracts on Human Factors in Computing Systems.* CHI EA ’10. Atlanta, Georgia, USA: ACM, pp. 3177–3180. DOI: `10.1145/1753846.1753949` (cit. on p. 26).

209. Schnarz, Pierre, Clemens Fischer, Joachim Wietzke, and Ingo Stengel (Aug. 2014a). “On a Domain Block Based Mechanism to Mitigate DoS Attacks on Shared Caches in Asymmetric Multiprocessing Multi Operating Systems”. In: *Information Security for South Africa (ISSA), 2014*, pp. 1–8. DOI: `10.1109/ISSA.2014.6950494` (cit. on pp. 43, 82, 157).

210. Schnarz, Pierre, Joachim Wietzke, and Ingo Stengel (Dec. 2013). “Co-Processor Aided Attack on Embedded Multi-OS Environments”. In: *International Conference on IT Convergence and Security (ICITCS)*, pp. 1–4. DOI: `10.1109/ICITCS.2013.6717818` (cit. on pp. 43, 68, 92, 157).

211. Schnarz, Pierre, Joachim Wietzke, and Ingo Stengel (2014b). “Towards Attacks on Restricted Memory Areas Through Co-processors in Embedded multi-OS Environments via Malicious Firmware Injection”. In: *Proceedings of the First Workshop on Cryptography and Security in Computing Systems.* CS2 ’14. Vienna, Austria: ACM, pp. 25–30. DOI: `10.1145/2556315.2556318` (cit. on pp. 43, 68, 157).

212. Schneider, Jörn (2013). “Overcoming the Interoperability Barrier in Mixed-Criticality Systems”. English. In: *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment.* Ed. by Josip Stjepandić, Georg Rock, and Cees Bil. Springer London, pp. 1093–1104. DOI: `10.1007/978-1-4471-4426-7_92` (cit. on p. 47).

213. Schneider, Reinhard, Dip Goswami, Samarjit Chakraborty, Unmesh Bordoloi, Petru Eles, and Zebo Peng (June 2011). "On the quantification of sustainability and extensibility of FlexRay schedules". In: *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 375–380 (cit. on p. 108).

214. Schneiderman, Ron (Jan. 2013). "Car Makers See Opportunities in Infotainment, Driver-Assistance Systems [Special Reports]". In: *Signal Processing Magazine, IEEE* 30.1, pp. 11–15. DOI: `10.1109/MSP.2012.2219681` (cit. on p. 1).

215. Schranzhofer, Andreas (Mar. 2011). "Efficiency and predictability in resource sharing multicore systems". Diss. ETH No. 19556. PhD thesis. ETH Zurich (cit. on pp. 104, 107, 108, 123).

216. Schranzhofer, Andreas, Jian-Jia Chen, and Lothar Thiele (2010). "Timing Analysis for TDMA Arbitration in Resource Sharing Systems". In: *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS '10. Washington, DC, USA: IEEE Computer Society, pp. 215–224. DOI: `10.1109/RTAS.2010.24` (cit. on pp. 107, 123).

217. Schranzhofer, Andreas, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo (Apr. 2011). "Timing Analysis for Resource Access Interference on Adaptive Resource Arbiters". In: *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 213–222. DOI: `10.1109/RTAS.2011.28` (cit. on p. 123).

218. Segal, Judith (Sept. 2003). "The Nature of Evidence in Empirical Software Engineering". In: *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on*, pp. 40–47. DOI: `10.1109/STEP.2003.33` (cit. on pp. 6, 57).

219. Segal, Judith, Antony Grinyer, and Helen Sharp (May 2005). "The Type of Evidence Produced by Empirical Software Engineers". In: *SIGSOFT Softw. Eng. Notes* 30.4, pp. 1–4. DOI: `10.1145/1082983.1083176` (cit. on pp. 6, 57).

220. Sha, Lui (July 2001). "Using Simplicity to Control Complexity". In: *Software, IEEE* 18.4, pp. 20–28. DOI: `10.1109/MS.2001.936213` (cit. on pp. 67, 188).

221. Sha, Lui (Oct. 2009). "Resilent Mixed Criticality Systems". In: *Cross Talk The Journal of Defense Software Engineering*, pp. 9–14 (cit. on pp. 18, 19, 21, 68).

222. Simonot-Lion, Françoise (Nov. 2009). "Guest Editorial - Special Section on In-Vehicle Embedded Systems". In: *IEEE Transactions on Industrial Informatics* 5.4, pp. 372–374. DOI: `10.1109/TII.2009.2031747` (cit. on p. 17).

223. Smethurst, Graham (2010). *Changing the In-Vehicle Infotainment Landscape.* Whitepaper. GENIVI Alliance (cit. on pp. 10, 11, 42, 47, 53).

224. Smit, Gerard JM, André BJ Kokkeler, Pascal T Wolkotte, and Marcel D van de Burgwal (2008). “Multi-core architectures and streaming applications”. In: *SLIP ’08: Proceedings of the 2008 international workshop on System level interconnect prediction.* Newcastle, United Kingdom: ACM, pp. 35–42 (cit. on pp. 30, 79).

225. Smith, James E. and Ravi Nair (2005). *Virtual Machines: Versatile Platforms for Systems and Processes.* Elsevier (cit. on p. 90).

226. Soltesz, Stephen, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson (Mar. 2007). “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors”. In: *SIGOPS Oper. Syst. Rev.* 41.3, pp. 275–287. DOI: `10.1145/1272998.1273025` (cit. on p. 91).

227. Sommerville, Ian, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Kwiatkowska, John Mcdermid, and Richard Paige (July 2012). “Large-Scale Complex IT Systems”. In: *Commun. ACM* 55.7, pp. 71–77. DOI: `10.1145/2209249.2209268` (cit. on pp. 10, 153).

228. Staelin, Carl (2005). “lmbench: an extensible micro-benchmark suite”. In: *Software: Practice and Experience* 35.11, pp. 1079–1105. DOI: `10.1002/spe.665`. URL: `http://dx.doi.org/10.1002/spe.665` (cit. on p. 88).

229. Steinmetz, Ralf and Klara Nahrstedt (2004). *Multimedia Systems.* X.media.publishing. Berlin, Heidelberg, New York: Springer (cit. on pp. 9, 33).

230. Stevens, A., A. Quimby, A. Board, T. Kersloot, and P. Burns (Feb. 2002). *Design Guidelines for Safety of In-Vehicle Information Systems.* Tech. rep. Project Report PA3721/01. TRL Limited (cit. on p. 74).

231. Stirgwolt, Philip (Jan. 2013). “Effective Management Of Functional Safety For ISO 26262 Standard”. In: *Reliability and Maintainability Symposium (RAMS), 2013 Proceedings - Annual*, pp. 1–6. DOI: `10.1109/RAMS.2013.6517758` (cit. on p. 23).

232. Strauss, David (Apr. 2013). “The Future Cloud is Container, Not Virtual Machines”. In: *Linux Journal* 2013.228 (cit. on p. 91).

233. Strayer, David L., Jason M. Watson, and Frank A. Drews (2011). “The Psychology of Learning and Motivation”. In: ed. by Brian Ross. Vol. 54. Burlington: Academic Press. Chap. Cognitive Distraction While Multitasking in the Automobile, pp. 29–58 (cit. on p. 19).

234. Stroustrup, Bjarne (2009). *Programming: Principles and Practice Using C++.* Addison-Wesley (cit. on p. 18).

235. Sutter, Herb and James Larus (Sept. 2005). “Software and the Concurrency Revolution”. In: *ACM Queue* 3 (7), pp. 54–62. DOI: `10.1145/1095408.1095421` (cit. on pp. 30, 79).

236. Tam, David, Reza Azimi, and Michael Stumm (2007). "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM New York, NY, USA, pp. 47–58 (cit. on pp. 80, 82, 85).

237. Tan, Keah Choon, Steven B. Lyman, and Joel D. Wisner (2002). "Supply chain management: a strategic perspective". In: *International Journal of Operations & Production Management* 22.6, pp. 614–631. DOI: 10.1108/01443570210427659 (cit. on p. 55).

238. Theis, Andreas (Apr. 2013). "User-Interfaces einzelner Android-Apps auf einem entfernten Wayland-Compositor". Master's thesis. Faculty of Computer Science, h_da Hochschule Darmstadt, University of Applied Sciences (cit. on pp. 137, 138).

239. Thiebaut, Stefaan Sonck (Oct. 2011). "Angriffsschutz für Infotainment-Integration". In: *Automotive*, pp. 42–45 (cit. on p. 47).

240. Tizen Project (2014a). *Dev Guide - Tizen Developers*. URL: https://developer.tizen.org/documentation/dev-guide (visited on Oct. 10, 2014) (cit. on p. 48).

241. Tizen Project (2014b). *Tizen - An open source, standards-base software platform for multiple device categories*. URL: http://www.tizen.org (visited on Oct. 10, 2014) (cit. on p. 48).

242. Tizen Project (2014c). *UX Guide - Tizen Developers*. URL: https://developer.tizen.org/documentation/ux-guide/ (visited on Oct. 10, 2014) (cit. on p. 48).

243. Tonguz, Ozan K. and Mate Boban (2010). "Multiplayer games over Vehicular Ad Hoc Networks: A new application". In: *Ad Hoc Networks* 8.5. Vehicular Networks, pp. 531–543. DOI: 10.1016/j.adhoc.2009.12.009 (cit. on p. 28).

244. Tönnis, Marcus, Verena Broy, and Gudrun Klinker (Mar. 2006). "A Survey of Challenges Related to the Design of 3D User Interfaces for Car Drivers". In: *Proceedings of the 1st IEEE Symposium on 3D User Interfaces (3D UI)* (cit. on p. 20).

245. Torres, Lionel, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy, and Diego Puschini (2011). "An Introduction to Multi-Core System on Chip – Trends and Challenges". English. In: *Multiprocessor System-on-Chip*. Ed. by Michael Hübner and Jürgen Becker. Springer New York, pp. 1–21. DOI: 10.1007/978-1-4419-6460-1_1 (cit. on p. 102).

246. Tsai, Chia-Che, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter (2014). "Cooperation and Security Isolation of Library OSes for Multiprocess Applications". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 9:1–9:14. DOI: 10.1145/2592798.2592812 (cit. on p. 91).

247. Uzcategui, Roberto and Guillermo Acosta-Marum (May 2009). “Wave: A tutorial”. In: *Communications Magazine, IEEE* 47.5, pp. 126–133. DOI: `10.1109/MCOM.2009.4939288` (cit. on p. 28).

248. Vaughan, Tay (2011). *Multimedia: Making It Work*. Eighth. New York: McGraw-Hill (cit. on p. 9).

249. Vergata, Sergio, Andreas Knirsch, and Joachim Wietzke (2012). “Integration zukünftiger In-Car-Multimediasysteme unter Verwendung von Virtualisierung und Multi-Core-Plattformen”. In: *Herausforderungen durch Echtzeitbetrieb - Echtzeit 2011*. Ed. by Wolfgang A. Halang. Informatik aktuell. Springer, pp. 21–28. DOI: `10.1007/978-3-642-24658-6_3` (cit. on pp. 87, 90, 236).

250. Vergata, Sergio, Joachim Wietzke, Alois Schütte, and Paul S. Dowland (Nov. 2010). “System Design for Embedded Automotive Systems”. In: *Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*. University of Plymouth, pp. 53–60 (cit. on pp. 87, 88).

251. Verhulst, Eric and Bernhard H.C. Sputh (Nov. 2013). “ARRL: A criterion for compositional safety and systems engineering: A normative approach to specifying components”. In: *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 37–44. DOI: `10.1109/ISSREW.2013.6688861` (cit. on p. 24).

252. Wagner, Stefan (2013). *Software Product Quality Control*. Springer. DOI: `10.1007/978-3-642-38571-1` (cit. on pp. 31, 38).

253. Waldspurger, Carl and Mendel Rosenblum (Jan. 2012). “I/O Virtualization”. In: *ACM Communications* 55 (1), pp. 66–73. DOI: `10.1145/2063176.2063194` (cit. on p. 123).

254. Walli, Stephen R. (Mar. 1995). “The POSIX Family of Standards”. In: *StandardView* 3 (1), pp. 11–17. DOI: `10.1145/210308.210315` (cit. on p. 40).

255. Wietzke, Joachim (2012). *Embedded Technologies - Vom Treiber bis zur Grafik-Anbindung*. Xpert.press. Springer (cit. on pp. 79, 81, 84, 92, 149).

256. Wietzke, Joachim and Manh Tien Tran (2005). *Automotive Embedded Systeme: Effizientes Framework – Vom Design zur Implementierung*. Xpert.press. Berlin, Heidelberg: Springer. DOI: `10.1007/3-540-28305-6` (cit. on pp. 11, 34, 44, 49, 50, 69, 188, 189).

257. Winner, Hermann (2013). “Challenges of Automotive Systems Engineering for Industry and Academia”. English. In: *Automotive Systems Engineering*. Ed. by Markus Maurer and Hermann Winner. Springer, pp. 3–15. DOI: `10.1007/978-3-642-36455-6_1` (cit. on p. 56).

258. Wünderlich, Martin (2007). *Informatik der h_da baut Kooperation mit Harman/Becker Automotive Systems aus.* Press Release. h_da University of Applied Sciences Darmstadt (cit. on p. 57).

259. Xavier, Miguel G., Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose (Feb. 2013). "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments". In: *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pp. 233–240. DOI: `10.1109/PDP.2013.41` (cit. on p. 48).

260. Yocto Project (2014). *Yocto Project - Open Source embedded Linux build system, package metadata and SDK generator.* URL: `http://https://www.yoctoproject.org` (visited on Sept. 10, 2014) (cit. on p. 146).

261. Zhu, Haibin (July 2005). "Challenges to reusable services". In: *Services Computing, 2005 IEEE International Conference on.* Vol. 2, 243–244 vol.2. DOI: `10.1109/SCC.2005.36` (cit. on p. 85).

Glossary

"When I use a word it means just what I choose it to mean neither more nor less." (Humpty Dumpty (Carroll, 1887))

Application *An application provides a set of functional features to its user. It may consist of one or more components, modules or subsystems (ISO 24765, 2010, p17).*

Application Developer *A person or group that develops an application or subsequent components. Preferably an available framework is utilised to improve portability and compatibility to interdependent applications and components.*

Application Domain *An application domain is an application that consists of components.*

Central Processing Unit *Provides the main computational resources of a hardware system.*

Compatibility *Compatibility implies that different components of a system can coexist without adverse side effects (Peter G Neumann, 2004, p3).*

Complexity *Per Leveson (2011, p4), complexity can be defined as intellectual unmanageability. She further subdivides into interactive complexity (interaction among system components), dynamic complexity (changes over time), decompositional complexity (structural decomposition is not consistent with the functional decomposition), and nonlinear complexity (cause and effect are not directly related).*

Component *A component denominates a subset of functional features in comparison to an application. This means an application (or application domain) consists of one or more components.*

Composability *A quality that refers to correctness of a system that integrates different components, whereas correctness includes the fulfilment of both functional and temporal requirements. It subsumes system qualities like compatibility and interoperability.*

Dependability *Dependability of a (computing) system is related to its ability to deliver a predefined service that can justifiably be trusted (Avižienis et al., 2001, p2). It is essentially indistinguishable from trustworthiness (Peter G Neumann, 2004, p2).*

Driver Distraction *An inattention that occurs when drivers divert their attention away from the driving task to focus on another activity. These distractions can be from electronic devices or more conventional distractions such as interacting with passengers and eating (NHTSA, 2012, p5).*

End of Production *The end of production (EoP) describes a certain point in time on which the production line for a particular vehicle or vehicular component is stopped.*

Failure in Time *A quantitative figure to express the failure rate of a component within a predefined time span. ISO 26262 (2011) extensively refers to 'FIT' in relation to functional safety.*

Functional Safety *The part of the overall safety that depends on a system or equipment operating correctly in response to its inputs (IEC, 2014).*

Head-Unit *A hardware platform (i.e., computer) that represents the central computing instance of an ICM system (Wietzke and Tran, 2005).*

Human Machine Interface *cf. User Interface*

In-Car Multimedia *In-Car Multimedia describes a software-intensive system, which consists of components that provide information and entertainment within an automotive context. Related terms are infotainment or In-Vehicle Infotainment (IVI).*

Integrator *A person or team to put parallel and independently developed system parts together to build an integral whole.*

Interoperability *Interoperability implies that different components of a system are able to work constructively with one another (Peter G Neumann, 2004, p4).*

Logical Complexity *Logical complexity is expressed due to the number of cases (or states) that the verification (or testing) process must handle (Sha, 2001).*

Multi-Sourced Software *The development of software is divided into distinct departments that may not belong to a single organisation.*

Multimedia *An interdependent combination of digital manipulated text, graphic, sound, and video, whereas the delivery can optionally be controlled by the user.*

Multimodal *Multiple interaction modes with a system for input (e.g., push button, touch display, speech recognition) and output of data (e.g., audible visual).*

OpenICM *An academic SW framework maintained at the ICM labs at the University of Applied Sciences Darmstadt (Knirsch et al., 2012b).*

Portable Operating System Interface *POSIX defines a standard OS interface and environment, including a command interpreter, and common utility programs to support applications portability at the source code level (ISO/IEC/IEEE 9945, 2008).*

Processing Unit *A hardware device that provides computational resources (i.e. microcontroller).*

Quality Management System *A set of interconnected processes that support an organisation to achieve its objectives (Hoyle, 2005, p685).*

Quality of Service *The QoS describes a requirement for the delivery of information with respect of the end-to-end path of a certain data flow. This can be related to operational deadlines as well to the resolution of video, audio, or other media.*

Safety *Safety is the state of being safe from consequences of events that are considered non-desirable. This implies a protection from events that cause health or economic losses, or at least a control of particular hazards to achieve a defined level of risk.*

Software Framework *A common infrastructure accompanied with guidelines to ease the development and maintenance process in particular for projects with independent and parallel development. It supports both the design phase and the implementation phase by domain specific abstractions of the underlying system (Wietzke and Tran, 2005).*

Start of Production *The start of production (SoP) describes a certain point in time on which the production line for a particular vehicle or vehicular component is started. All development has to be finished before SoP to prevent a delay in production.*

Task *A task represents a single computational context with, e.g., an individual stack and program counter. Within the context of this report, it is used interchangeably with the terms thread and process.*

Trustworthiness *Trustworthiness represents the extent to which a defined set of requirements is likely to be satisfied under specified conditions. According to Peter G*

Neumann (2004, p1), this means 'worthy of being trusted to satisfy the given expectations'.

User Experience *The User Experience subsumes the perceived quality aspects of an interactive system, which includes dimensions like perceived ease of use and perceived visual attractiveness (cf. (Mahlke, 2005)).*

User Interface *An interface that enables information to be passed on between a human user and hard- or software components (i.e., to allow a user to interact) (ISO 24765, 2010, p390).*

Verification *The verification of a system is the process to formally prove its conformance with its intended behaviour (i.e., specification) (Kreiker et al., 2011b, p26).*

Abbreviations

3G	Third Generation of Mobile Telecommunications Technology
AMP	Asymmetric Multi-Processing
APS	Adaptive Partitioning Scheduler
AUTOSAR	Automotive Open System Architecture
CAN	Controller Area Network
CD	Containment Domain
COTS	Contributed Off The Shelf
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DI	Device Instance
DM	Device Manager
DPS	Dynamic Priority Scheduling
E/E	Electrical/Electronic
ECU	Electronic Component Unit
ED	Execution Domain
EoP	End of Production
FCFS	First Come First Serve

FIFO	First In First Out
FIT	Failure in Time
FOSS	Free and Open Source Software
FPS	Fixed Priority Scheduling
GPU	Graphical Processing Unit
HTML	Hyper Text Markup Language
HW	Hardware
I/O	Input/Output
IATF	International Automotive Task Force
ICM	In-Car Multimedia
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IMA	Integrated Modular Avionics
IPC	Inter-Process Communication
IRQ	Interrupt Request
ISO	International Organization for Standardization
KMS	Kernel Mode Setting
KVM	Kernel-based Virtual Machine
LTE	Long Term Evolution
LVDS	Low Voltage Differential Signal
LXC	Linux Containers
M2M	Machine-to-Machine
MC	Multicore
MMU	Memory Management Unit
MOC	Models of Computation
MOST	Media Oriented System Transport

NFR	Non-Functional Requirements
NHTSA	National Highway Traffic Safety Administration
OEM	Original Equipment Manufacturer
OS	Operating System
OSD	Operating System Domain
OSGi	Open Services Gateway Initiative
POSIX	Portable Operating System Interface
PU	Processing Unit
QMS	Quality Management System
QoS	Quality of Service
RT	real-time
RTOS	Real-Time Operating System
SD	Service Driver
SDK	Software Development Kit
SHARB	Shared Resource Arbiter
SMP	Symmetric Multi-Processing
SOA	Service Oriented Architecture
SoC	System-on-Chip
SoP	Start of Production
SQuaRE	Software product Quality Requirements and Evaluation
SW	Software
TDMA	Time Division Multiple Access
UI	User Interface
UK	United Kingdom
UKCRC	UK Computing Research Committee
UMTS	Universal Mobile Telecommunications System

US	United States
UX	User Experience
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMMC	Virtual Machine Multicore
vOSD	virtual Operating System Domain
vPU	virtual PU
WAVE	Wireless Access in Vehicular Environments
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
WHO	World Health Organization
WLAN	Wireless Local Area Network

Index

A

Experimental Results

The main body of this thesis focuses on the discussion on structuring potentially incompatible software components. In the interest of readability, the following experiments are located separately and referenced in the respective sections. However, the intention of this appendix is to provide empiric, quantitative and reproducible results to foster comprehensible evidence for certain isolated aspects. This means the following sections take up on abstract problems with prototype implementations to demonstrate problems and effects of proposed concepts.

A.1 Incompatible scheduling policies

The following experiments exemplify the implications that arise with the coexistence of concurrent scheduling policies in MC environments based on the discussion in Section 4.2.3. Therefore $n$threads ($\mathrm{T_n}$) are created that share a common scheduling priority. They are organised into two components ($\mathrm{C_0}$ and $\mathrm{C_1}$), and while regarding the scheduling policy, the threads of $\mathrm{C_0}$ make use of RR and the threads of $\mathrm{C_1}$ make use of FIFO. All threads compute the same independent workload that is not interrupted (i.e., by synchronisation, cooperative 'yields', or device access) using two PUs and SMP scheduling. This implies threads of $\mathrm{C_1}$ are not pre-empted to grant computational resource to the ones of $\mathrm{C_0}$, while threads of $\mathrm{C_0}$ are pre-empted and do not receive computational resources before the ones of $\mathrm{C_0}$ are finished.

For the experiment, each processing of workload is repeated r times, while for each processing, elapse time is measured. This means for each T a number of r 'elapse times' ($\mathrm{t_r}$) are available. These establish the base to derive temporal characteristics regarding the elapse times in form of sum (t_{sum}), arithmetic mean (t_{mean}), maximum (t_{max}), minimum (t_{min}) and standard deviation (t_{sd}). The first elapse time (t_0) additionally contains the startup time ($t_{startup}$) of T. Figure A.1 illustrates the design of the respective application component with regard to the measuring points of t_r.

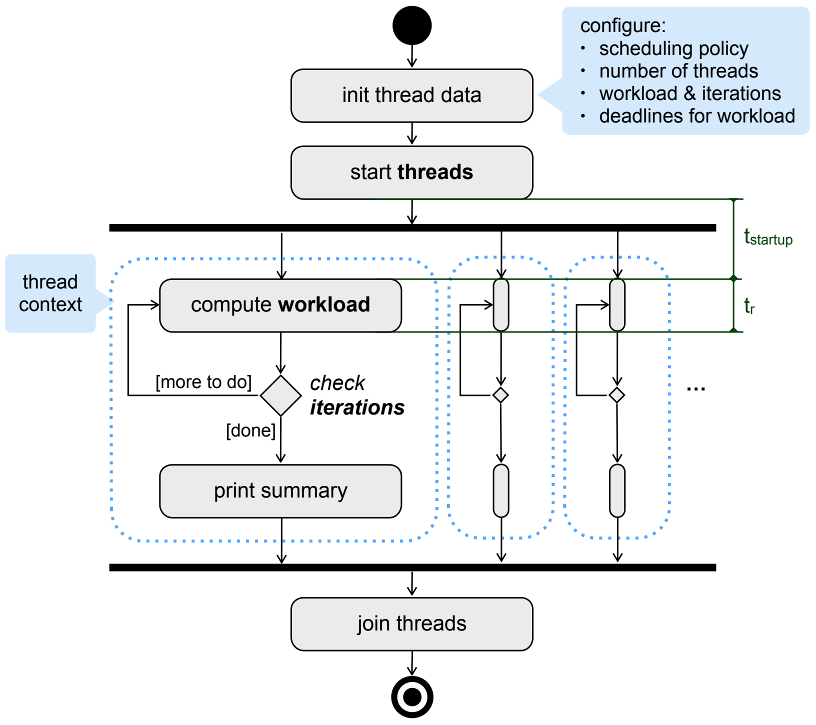

Figure A.1: Multithreaded test component

Based on the proposed concept for EDs using PU-affinity, the experiment is repeated with the same configuration, except that C_0 and C_1 are assigned to different PUs to isolate the scheduling domains. This means the experiment is conducted using two setups (S), as depicted in Figure A.2:

- S_1 (mixed) - an uncoordinated allocation of PUs by the OS's task scheduler (using SMP).
- S_2 (partitioned) - a dedicated allocation of PUs by configuration of the OS's task scheduler (using SMP with PU affinity).

The experiment is targeted for a Linux OS. The results detailed in Table A.1 were collected using kernel release 3.17.3 configured for low-latency scheduling and operated on an Intel Core i7 64bit hardware platform that hosts 4 PUs with simultaneous multi-threading (SMT).

In particular, the reduced t_{sd} for the threads of C_0 within S_2 (partitioned) mean a more deterministic computation of the individual workloads. It appears the temporal characteristics for one of the threads for C_1 decreases. However, this is related to the FIFO

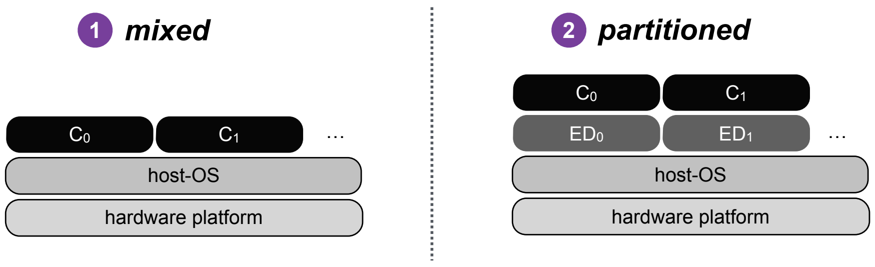

Figure A.2: Setup for demonstrating incompatible scheduling policies

S	C	policy	n	PU	$t_{startup}$	t_{sum}	t_{sd}	t_{min}	t_{max}
mixed	0	RR	0	2	0.000	8.851	1.133	0.229	6.012
			1	0	0.000	8.827	1.132	0.229	6.006
	1	FIFO	2	2	0.199	4.630	0.039	0.229	0.430
			3	0	0.099	4.525	0.020	0.229	0.331
partitioned	0	RR	0	0	0.099	8.844	0.054	0.361	0.494
			1	0	0.000	8.822	0.056	0.361	0.493
	1	FIFO	2	2	4.408	8.806	0.864	0.228	4.638
			3	2	0.000	4.408	0.001	0.229	0.231

configured with n=4 and r=25

Table A.1: Parallel computed components using different scheduling policies

scheduling. Moreover, a parallel computation of workload for a component that is designed to process its tasks sequentially may even cause unexpected behaviour. This implies an increased t_{sd} and $t_{startup}$ is related to FIFO scheduling. In summary, with partitioned scheduling of S_2 compatibility of components that relay on different scheduling policies, determinism is improved.

A.1.1 Listings

The experiment's code listings are detailed in Section A.2.

A.2 Scheduling within virtualised environments

Virtualisation introduces overhead, a result of the abstraction of the native ISA interface to host a virtual guest, as well as due to multiple OS instances. This has an impact on the performance efficiency of applications deployed to a virtual environment. Additionally, the VMM may provide only a subset of computational resources to the VM, which further decrease an application's performance within a VM. Such restriction on provisioned resources reserves computational power for coexisting VMs or applications. PU affinity is an effective means to achieve such limitation or reservation respectively on computational resources. The effect is visualised by an experiment that utilises an application that creates a configurable number tasks T_i using RR scheduling, each processing an independent workload: incrementing a counter from 0 to 109 with five iterations. The arithmetic mean of these iterations is saved before this 'measurement' is repeated for another five times to improve stability of the results. These results are again summed up by use of the arithmetic mean of the iterations, as depicted in Figure A.3.

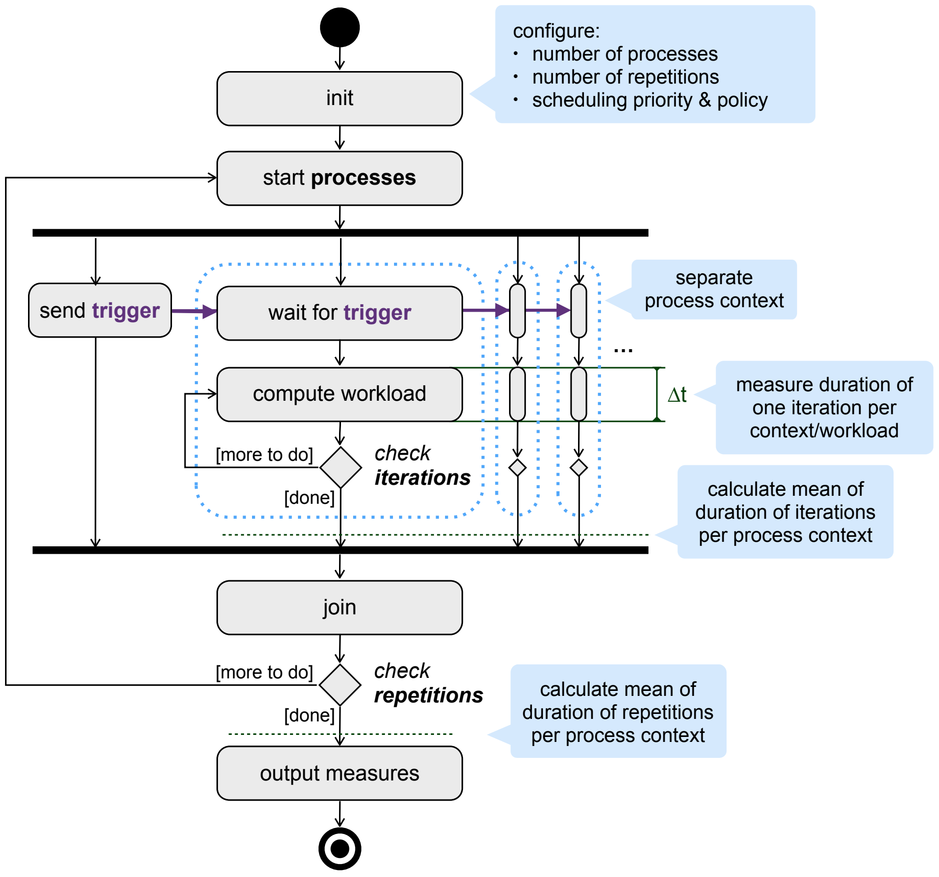

Figure A.3: Test application to apply parallel workload

Figure A.4 depicts the results with four VMMC setups with a varying number of configured vPUs in relation to native execution of the test application using a hardware platform that provides four PUs and eight parallel tasks (i=8). Therefore the test application is re-run with different limitations/reservations enforced with PU affinities while the workload has not been changed. The used PUs are mapped to the figure's horizontal axis. The host OS is Linux, while for the VMM QEMU/KVM is used.

The results show the overhead to the native execution of the test application. Related to the application's actual performance, it is basically insignificant whether the 'limitation' is configured at the level of the host by pinning the VMMC to a set of native hardware PUs, or the guest by pinning the application within the VMMC to a set of vPUs.

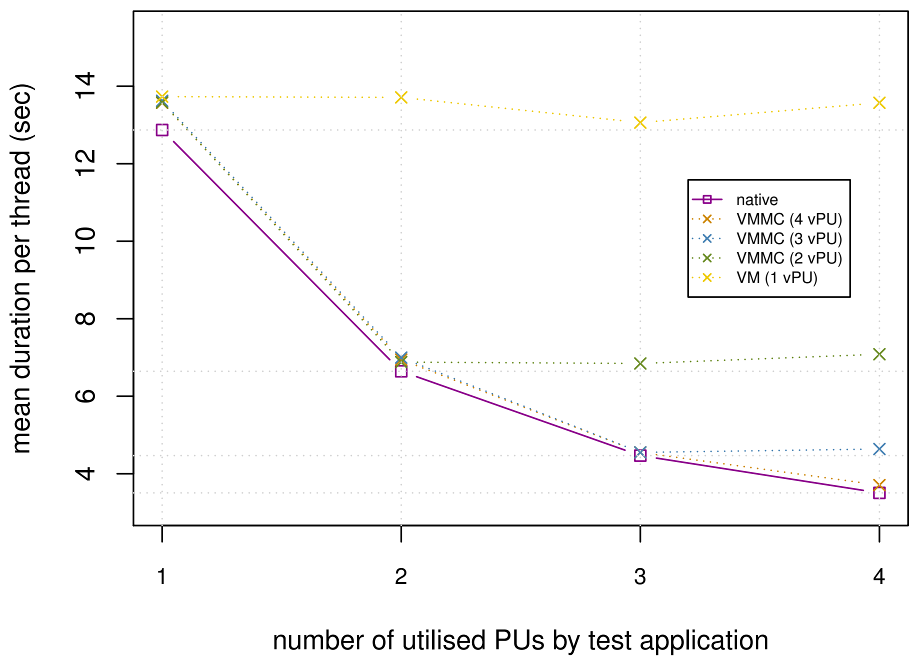

Figure A.4: Parallel workload with PU affinity using different VMMC configurations

To further detail the feature of pinning a VMMC's vPUs to a single native PU, the experiment detailed in Section A.1 is ported to a virtualised environment. The objective is to evaluate the impact of PU-affinity-based EDs using a VMMC architecture as proposed in Section 4.3.3. Therefore the already-available prototype test application is reused without any modification except the underlying OS and hardware platform. The OS is a Linux kernel configured for low-latency with a small-sized RAM file-system, executed on a VMMC that relies on QEMU/KVM. The experiment is therefore extended with two more setups, as depicted in Figure A.5:

- S_3 (vmixed) - an uncoordinated allocation of PUs by the guest-OS's task scheduler.
- S_4 (vpartitioned) - a dedicated allocation of

PUs by configuration of the guest-OS's task scheduler.

Based on the S_3 and S_4 three additional configurations using different numbers of vPUs for the VMMC are tested, as depicted in Table A.2.

Similar to the results for native partitioning, the results for VMMC-based runtime environments also show a significant decrease of the standard deviation of the 'elapse times' (t_{sd}). This difference between the mixed and partitioned setup for the different configuration with regards to t_{sd} and independent of the task's 'overall duration' is depicted in Figure A.6 (which also incorporates the results for native partitioning; cf. Section A.1). The 'overall duration' increases with reduction of available computational resources as the parallelism is substituted with concurrency ('quasi parallel execution').

S	C	policy	n	$t_{startup}$	t_{sum}	t_{sd}	t_{min}	t_{max}
mixed	0	RR	0	0.000	8.861	1.133	0.230	6.011
			1	0.000	4.429	0.001	0.230	0.237
	1	FIFO	2	4.422	8.845	0.867	0.229	4.657
			3	0.000	4.422	0.002	0.228	0.235
partitioned	0	RR	0	0.099	8.865	0.053	0.361	0.492
			1	0.000	8.835	0.056	0.361	0.496
	1	FIFO	2	4.423	8.845	0.867	0.230	4.654
			3	0.000	4.423	0.000	0.230	0.232

(a) VMMC with four vCPUs

S	C	policy	n	$t_{startup}$	t_{sum}	t_{sd}	t_{min}	t_{max}
mixed	0	RR	0	0.00	8.81	1.12	0.22	5.98
			1	0.00	4.37	0.00	0.22	0.23
	1	FIFO	2	0.00	8.85	0.00	0.22	0.27
			3	0.00	4.38	0.00	0.22	0.23
partitioned	0	RR	0	0.09	8.74	0.05	0.35	0.49
			1	0.00	8.67	0.05	0.35	0.49
	1	FIFO	2	4.37	8.75	0.85	0.22	4.60
			3	0.00	4.37	0.00	0.22	0.23

(b) VMMC with two vCPUs

S	C	policy	n	$t_{startup}$	t_{sum}	t_{sd}	t_{min}	t_{max}
mixed	0	RR	0	0.00	17.29	2.21	0.44	11.77
			1	0.00	17.24	2.21	0.44	11.73
	1	FIFO	2	0.00	8.63	0.00	0.44	0.47
			3	0.00	8.60	0.00	0.44	0.45
partitioned	0	RR	0	0.11	17.44	0.07	0.83	1.07
			1	0.00	17.29	0.06	0.83	0.98
	1	FIFO	2	8.69	17.47	1.70	0.44	9.16
			3	0.00	8.66	0.00	0.44	0.47

(c) VMMC with one vCPU

Table A.2: Mixed scheduling analysis using different VMMC configurations (a-c)

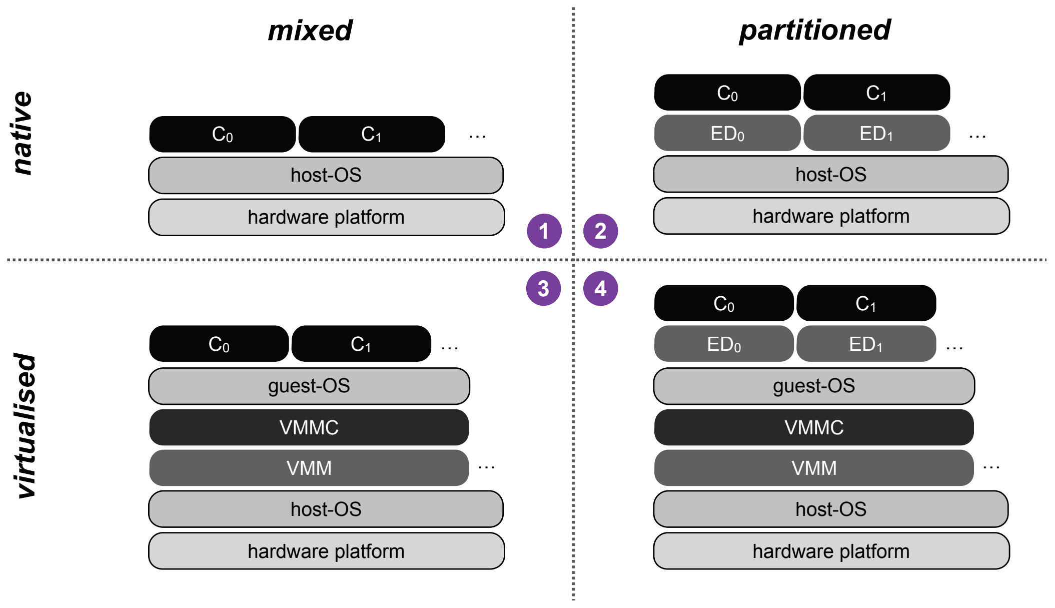

Figure A.5: Counter incompatible scheduling policies with virtualisation

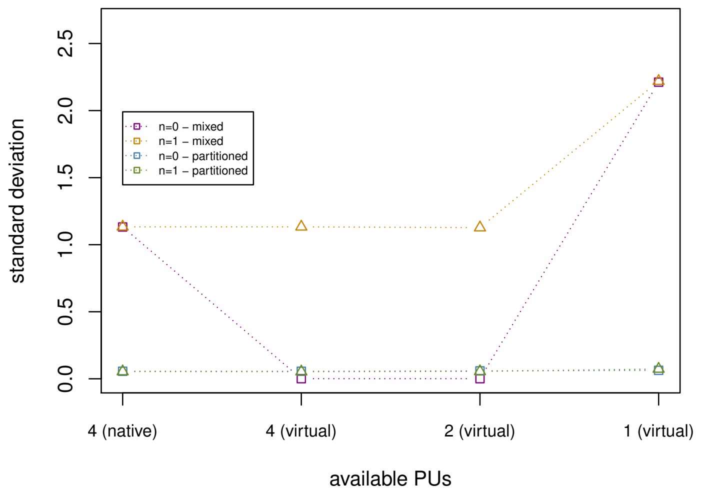

Figure A.6: RR task's standard deviations related to different VMMC configurations

A.2.1 Listings

The following code listings exemplify the realisation of the experiment on applying parallel workload on different environments.

Listing A.1 details the implementation of the experiment as outlined in Figure A.3. The script detailed in Listing A.2 improves reproducibility and usability by repeating the workload using different PU affinities.

Listing A.1

```
#include <iostream>
#include <wait.h>
#include <stdlib.h>
#include <new>
#include "prios_helper.h"
#include "memory.h"

using namespace std;

/* data structure passed to the worker tasks (note: not threadsafe) */
struct measure_component_info {
  unsigned int id;
  pid_t pid;
  double result;
};

/* do the work and measurement */
double measure(measure_component_info & measure_data,
                 unsigned int measure_runs = 5) {
  const unsigned long long loops = 1000000000;
  double *measure_duration = new double[measure_runs];
  double measure_duration_sum = 0.0;

  for (unsigned int run = 0; run < measure_runs; run++) {
    DEBUG_TRACE("[comp #%u] Starting run #%u/%u", measure_data.id,
                run + 1, measure_runs)
      measure_duration[run] = work_additions(loops);
    measure_duration_sum += measure_duration[run];
    DEBUG_TRACE("[comp #%u] Finished run #%u/%u after %f s",
                measure_data.id, run + 1, measure_runs,
                measure_duration[run])
  }
  measure_data.result = measure_duration_sum / measure_runs;

  DEBUG_TRACE("[comp #%u] Measured %f s on average to loop %lld times",
              measure_data.id, measure_data.result, loops)
    delete[] measure_duration;
  return measure_data.result;
}


/* create shm and fork worker tasks */
int parallel_load(const unsigned int test_proc_cnt, const unsigned int provided_cpus)
       {
  const char *shm_name = "TESTPROG_SHM";
  const unsigned int runs_per_test_proc = 5;

  /* init shared memory for data
  exchange
    void *shm_ptr =
  allocateSHM(shm_name,
              MAKE_ALIGNMENT_SIZE(test_proc_cnt *
                                    sizeof(measure_component_info)));
  measure_component_info *data_exchange_loop_test =
  new(shm_ptr) measure_component_info[test_proc_cnt];
  for (unsigned int i = 0; i < test_proc_cnt; i++) {
    data_exchange_loop_test[i].id = i;
    data_exchange_loop_test[i].pid = 0;
    data_exchange_loop_test[i].result = 0.0;
  }

  /* start measuring */
```

Listing A.1

```
  DEBUG_TRACE("Starting measuring components ...")
    for (unsigned int test_proc = 0; test_proc < test_proc_cnt;
         test_proc++) {
    pid_t comp_pid = -666;
    if ((comp_pid = fork()) == 0) {
      measure(data_exchange_loop_test[test_proc], runs_per_test_proc);
      /* abort execution of child process after measuring */
      return 666;
    } else {
      data_exchange_loop_test[test_proc].pid = comp_pid;
      DEBUG_TRACE
        ("Measuring component %u started and makes use of PID %d",
         test_proc, comp_pid)
    }
  }

  DEBUG_TRACE("Waiting for measuring components ...")
    unsigned int proc_finished = 0;
  while (proc_finished < test_proc_cnt) {
    int status = 0;
    pid_t pid = wait(&status);
    if (status != EXIT_SUCCESS) {
      cerr << "Ooops component with pid " << pid <<
        " died with status " << status << endl;
    } else {
      DEBUG_TRACE
        ("Component with pid %d finished successfully", pid)
    }
    ++proc_finished;
  }

  /* print results */
#ifdef READABLE

  for (unsigned int i = 0; i < test_proc_cnt; i++) {
    cout << "Component #" << data_exchange_loop_test[i].
      id << " with pid " << data_exchange_loop_test[i].
      pid << " measured " << data_exchange_loop_test[i].
      result << " s " << endl;
  }

#else

  cout << provided_cpus << ";";
  for (unsigned int i = 0; i < test_proc_cnt; i++) {
    cout << data_exchange_loop_test[i].result;
    if (i < test_proc_cnt - 1) {
      cout << ";";
    } else {
      cout << endl;
    }
  }

#endif


  deallocateSHM(shm_name);

  return EXIT_SUCCESS;
}
```

Listing A.1

```
/* main routine */
int main(int argc, char **argv) {
  unsigned int test_proc_cnt = 4;
  unsigned int test_runs = 1;
  unsigned int provided_cpus = 99;
  int c, success;

  opterr = 0;
  while ((c = getopt(argc, argv, "c:r:p:")) != -1)
    switch (c) {
     case 'c':
      /* number of parallel processes */
      test_proc_cnt = atoi(optarg);
      break;
     case 'r':
      /* number of repetitions of the whole test */
      test_runs = atoi(optarg);
      break;
     case 'p':
      /* first column of csv output(CPUs available) */
      provided_cpus = atoi(optarg);
      break;
     case '?':
      cerr << "Unknown option character " << hex << optopt << endl;
      return 1;
     default:
      abort();
    }

  for (int index = optind; index < argc; index++) {
    cerr << "Non-option argument " << argv[index] << endl;
  }

  for (unsigned int run = 0; run < test_runs; run++) {
    DEBUG_TRACE("Start run %u\n", run);
    success = parallel_load(test_proc_cnt, provided_cpus);

    if (success == 666) {
      return EXIT_SUCCESS;
    }
    assert(success == EXIT_SUCCESS);
  }

  return EXIT_SUCCESS;
}
```

Listing A.1: Generate parallel workload using RR scheduling

Listing A.2

```
#!/bin/sh
# measure duration of concurrent workload on available HW platform

repetitions=5
parallel=8
```

Listing A.2

```
cpus_available=$1
if [ -z "$cpus_available" ] ; then
  cpus_available=`grep -c processor /proc/cpuinfo`
fi

SCRIPTPATH="`pwd -P`/`dirname $0`"
cd $SCRIPTPATH

outputfile=/tmp/parallel_load_`date "+%Y%m%d%H%M%S"`.csv
rm -f $outputfile

echo -n "CPU" > $outputfile
i=1
while [ "$i" -le "$parallel" ] ; do
  echo -n ";T$i" >> $outputfile
  i=$(($i + 1))
done
echo >> $outputfile

allmask="$(((1<<$cpus_available)-1))"
procs=0
cnt=$cpus_available
while [ "$procs" -lt "$cpus_available" ] ; do
        mask="$(($allmask >> $procs))"
  taskset $mask ./parallel_load -c8 -p$cnt -r$repetitions >> $outputfile
  procs=$(($procs + 1))
  cnt=$(($cnt - 1))
done

echo ">>> check $outputfile"
```

Listing A.2: Configure parallel load

Listing A.3 shows the ‘application component starter’ that configures and starts the application components. The latter creates the worker threads for using a given scheduling policy and measure durations, as outlined in Figure A.1 and detailed in Listing A.4.

Listing A.3

```
#include "prios_helper.h"

#include <assert.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <wait.h>

// main routine
int main(int argc, char **argv) {

  /* check for SU privileges */
  assert(getuid() == 0);
```

Listing A.3

```
/* default configuration */
int threads = 2;
int prio = 20;
int loops = 25;
int iterations = 20000;
int deadline = 1500;
char method[80];
bool mixedMode = false;

/* init affinity */
cpu_set_t runmask;
CPU_ZERO(&runmask);

/* check arguments */
int c;
opterr = 0;
while ((c = getopt(argc, argv, "t:s:p:l:d:m:i:M")) != -1)
  switch (c) {

    /* loop counter */
   case 'l':
    loops = atoi(optarg);
    break;

    /* iterations - depending on acutal work */
   case 'i':
    iterations = atoi(optarg);
    break;

    /* deadline in msec */
   case 'd':
    deadline = atoi(optarg);
    break;

    /* number of created worker tasks */
   case 't':
    threads = atoi(optarg);
    break;

    /* priority of worker tasks */
   case 'p':
    prio = atoi(optarg);
    break;

    /* workload */
   case 'm':
    strncpy(method, optarg, 80);
    break;

    /* mixed mode */
   case 'M':
    mixedMode = true;
    break;

   case '?':
    fprintf(stderr, "Unknown option character `\\x%x'.\n", optopt);
    return 1;

   default:
    abort();
  }
```

Listing A.3

```
for (int index = optind; index < argc; index++) {
  printf("Non-option argument %s\n", argv[index]);
  return 1;
}

/* set priority (inherited by components - if not explicitly assigned) */
set_priority(prio + 2, pthread_self(), SCHED_RR);

int cmd_argv_len = 8;
int cmd_argv_item_maxlen = 80;
char **cmd_argv = new char *[8];
cmd_argv[0] = (char *)"application_component";
for (int i = 1; i < cmd_argv_len - 1; i++) {
  cmd_argv[i] = new char[cmd_argv_item_maxlen];
}
snprintf(cmd_argv[2], cmd_argv_item_maxlen, "-t%d", threads);
snprintf(cmd_argv[3], cmd_argv_item_maxlen, "-l%d", loops);
snprintf(cmd_argv[4], cmd_argv_item_maxlen, "-i%d", iterations);
snprintf(cmd_argv[5], cmd_argv_item_maxlen, "-d%d", deadline);
snprintf(cmd_argv[6], cmd_argv_item_maxlen, "-m%s", method);
cmd_argv[7] = NULL;


/* create components */
pid_t pid = -666;
if ((pid = fork()) == 0) {
  /* set affinity */
  if (mixedMode) {
    CPU_SET(1, &runmask);
  }
  CPU_SET(0, &runmask);
  set_affinity(runmask);

  snprintf(cmd_argv[1], cmd_argv_item_maxlen, "-s%s", "RR");
  int result = execve(cmd_argv[0], cmd_argv, NULL);
  assert(result == 0);
}
if ((pid = fork()) == 0) {
  /* set affinity */
  CPU_SET(1, &runmask);
  if (mixedMode) {
    CPU_SET(0, &runmask);
  }
  set_affinity(runmask);

  snprintf(cmd_argv[1], cmd_argv_item_maxlen, "-s%s", "FIFO");
  int result = execve(cmd_argv[0], cmd_argv, NULL);
  assert(result == 0);
}
/* wait for components to finish */
int status = 0;

/* ...first component */
pid = wait(&status);
if (status != EXIT_SUCCESS) {
  fprintf(stderr, "Ooops component with pid %d died with status %d \n", pid, status
      );
}
/* ...second component */
pid = wait(&status);
```

Listing A.3

```
  if (status != EXIT_SUCCESS) {
    fprintf(stderr, "Ooops component with pid %d died with status %d \n", pid, status
          );
  }
  /* cleanup */
  for (int i = 1; i < cmd_argv_len - 1; i++) {
    delete[] cmd_argv[i];
  }

  return EXIT_SUCCESS;
}
```

Listing A.3: Application component starter

Listing A.4

```
#include "prios_helper.h"

#include <assert.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>

enum Workload {
  MEASURE_PRIME, MEASURE_ADD
};

enum Policy {
  MIXED = 0, FIFO = SCHED_FIFO, RR = SCHED_RR
};

struct ThreadData {
  int id;
  int policy;
  int prio;
  int loops;
  unsigned int deadline;
  Workload workload;
  unsigned long long iterations;
  timespec start;
};

/* the thread function */
void *threadProc(void *arg) {
  timespec tend, startup;
  ThreadData td = *(reinterpret_cast < ThreadData * >(arg));
  set_priority(td.prio, pthread_self(), td.policy);
  double *durations = new double[td.loops];
  unsigned int *cpus = new unsigned int[td.loops];
  double cpuspeed = rdcpuspeed();

  stopwatch(startup);
  unsigned long long cur_tsc, last_tsc = rdtsc();
  for (int j = 0; j < td.loops; j++) {
```

Listing A.4

```
    switch (td.workload) {
     case MEASURE_PRIME:
      work_prime(td.iterations);
      break;
     case MEASURE_ADD:
      work_additions(td.iterations);
      break;
     default:
      assert(false);
      break;
    }

    cpus[j] = get_cpuid();
    cur_tsc = rdtsc();
    durations[j] = (cur_tsc - last_tsc) / cpuspeed;
    last_tsc = cur_tsc;
  }

  double sum = 0.0, max = 0.0, min = (double)1E9;
  unsigned int missed_deadlines = 0;

  durations[0] += diff2double(td.start, startup);

  for (int j = 0; j < td.loops; j++) {
    sum += durations[j];
    if (max < durations[j]) {
      max = durations[j];
    }
    if (min > durations[j]) {
      min = durations[j];
    }
    if (td.deadline > 0 && durations[j] > (double)(td.deadline) / 1000.0) {
      missed_deadlines++;
    }
  }

  stopwatch(tend);
  printf(
         "%d %d %d - %s on %d start: %6.3f duration: %6.3f [sum=%06.3f mean=%06.3f
               min=%06.3f max=%06.3f sd=%06.3f md=%u/%d durations=",
         td.id, getpid(), gettid(), (td.policy == FIFO) ? "FIFO" : "RR  ", get_cpuid
               (), diff2double(td.start, startup), diff2double(td.start, tend),
         sum, sum / (double)(td.loops), min, max,
         standardDeviation(durations, td.loops), missed_deadlines, td.loops);
  for (int j = 0; j < td.loops; j++) {
    printf("%4.2f ", durations[j]);
  }
  printf(" cpus=");
  for (int j = 0; j < td.loops; j++) {
    printf("%u", cpus[j]);
  }
  printf("]\n");

  delete[] durations;
  delete[] cpus;

  return NULL;
}

/* create a new thread */
```

Listing A.4

```
pthread_t createThread(ThreadData * td) {
  pthread_t tid;
  assert(0 == pthread_create(&tid, NULL, threadProc, td));
  return tid;
}

/* main routine */
int main(int argc, char **argv) {

  /* check for SU privileges */
  assert(getuid() == 0);

  /* default configuration */
  unsigned int number_of_subtasks = 1;

  ThreadData td;
  td.policy = MIXED;
  td.prio = 20;
  td.loops = 10;
  td.deadline = 0;
  td.workload = MEASURE_ADD;
  td.iterations = 20000000;

  /* check arguments */
  int c;
  opterr = 0;
  while ((c = getopt(argc, argv, "t:s:p:l:d:m:i:")) != -1)
    switch (c) {
      /* loop counter */
     case 'l':
      td.loops = atoi(optarg);
      break;

      /* iterations - depending on acutal work(additions or pi iterations) */
     case 'i':
      td.iterations = atoi(optarg);
      break;

      /* deadline in msec */
     case 'd':
      td.deadline = atoi(optarg);
      break;

      /* number of created worker tasks */
     case 't':
      number_of_subtasks = atoi(optarg);
      break;

      /* priority of worker tasks */
     case 'p':
      td.prio = atoi(optarg);
      break;

      /* scheduling policy of worker tasks */
     case 's':
      if (strncmp(optarg, "FIFO", 4) == 0) {
        td.policy = FIFO;
      } else if (strncmp(optarg, "RR", 2) == 0) {
        td.policy = RR;
      } else if (strncmp(optarg, "MIXED", 5) == 0) {
        td.policy = MIXED;
```

Listing A.4

```
      } else {
        fprintf(stderr,
                "Unkown scheduling policy [%s] (only MIXED, FIFO or RR are supported
                      yet)\n", optarg);
        return 1;
      }
      break;

      /* type of workload */
     case 'm':
      if (strncmp(optarg, "ADD", 4) == 0) {
        td.workload = MEASURE_ADD;
      } else if (strncmp(optarg, "PRIME", 2) == 0) {
        td.workload = MEASURE_PRIME;
      } else {
        fprintf(stderr,
                "Unkown workload (only ADD or PRIME are supported yet)\n");
        //return 1;
      }
      break;

     case '?':
      fprintf(stderr, "Unknown option character `\\x%x'.\n", optopt);
      return 1;

     default:
      abort();
    }

  for (int index = optind; index < argc; index++) {
    printf("Non-option argument %s\n", argv[index]);
  }

  /* set priority (inherited by worker tasks) */
  set_priority(td.prio + 1, pthread_self(), SCHED_RR);

  /* create container to store thread ids */
  pthread_t *threads = new pthread_t[number_of_subtasks];
  ThreadData *tds = new ThreadData[number_of_subtasks];

  stopwatch(td.start);

  for (unsigned int i = 0; i < number_of_subtasks; i++) {
    memcpy(&tds[i], &td, sizeof(ThreadData));
    tds[i].id = i;
    if (tds[i].policy == MIXED) {
      if (i % 2) {
        tds[i].policy = SCHED_FIFO;
      } else {
        tds[i].policy = SCHED_RR;
      }
    }
  }

  /* create tasks */
  for (unsigned int i = 0; i < number_of_subtasks; i++) {
    threads[i] = createThread(&tds[i]);
  }

  /* wait for tasks to finish */
  for (unsigned int j = 0; j < number_of_subtasks; j++) {
```

Listing A.4

```
    pthread_join(threads[j], NULL);
  }

  /* tidy up */
  delete[] threads;

  return EXIT_SUCCESS;
}
```

Listing A.4: Multi-threaded Application component

Both the ‘application component starter’ and the ‘application component’ as well as the ‘parallel load’ generator utilise a set of helper functions to abstract the complexity regarding PU affinity, task prioritisation, workload generation, measuring durations and post-processing the results (cf. Listing A.5).

Listing A.5

```
#include "prios_helper.h"
#include <sched.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#include <assert.h>
#include <stdint.h>

/* binds the current process to the given cpu */
void set_affinity(int cpu) {
  cpu_set_t runmask;
  CPU_ZERO(&runmask);
  CPU_SET(cpu, &runmask);
  set_affinity(runmask);
}

/* binds the current process to the given cpu mask */
void set_affinity(cpu_set_t runmask) {
  if (sched_setaffinity(0, sizeof(runmask), &runmask) == -1) {
    printf("failed to bind thread [with pid=%d (%s)]\n", getpid(),
           strerror(errno));
  }
}

/* returns the cpu mask the proc is bound to */
int get_affinity() {
  int result = 0;
  cpu_set_t runmask;
  if (sched_getaffinity(0, sizeof(runmask), &runmask) == -1) {
    printf("failed to get cpu affinity [pid=%d] (%s)\n", getpid(),
           strerror(errno));
  }
```

Listing A.5

```
  for (int i = 0; i < 32; i++) {
    if (CPU_ISSET(i, &runmask)) {
      result += 1 << i;
    }
  }
  return result;
}

/* sets the given priority to current thread and make use of SCHED_RR */
void set_priority(int prio) {
  set_priority(prio, pthread_self(), SCHED_RR);
}

/* sets the given priority to given thread and make use of SCHED_RR */
void set_priority(int prio, pthread_t tid) {
  set_priority(prio, tid, SCHED_RR);
}

/* sets the given priority and policy to given thread */
void set_priority(int prio, pthread_t tid, int policy) {
  struct sched_param threadparam;
  int curpolicy;
  pthread_getschedparam(tid, &curpolicy, &threadparam);
  threadparam.sched_priority = prio;
  int ret = pthread_setschedparam(tid, policy, &threadparam);
  if (ret != 0) {
    fprintf(stderr,
            "failed to set scheduling parameter for pid %d (%s) [prio=%d policy=%d]\n
                  ",
            getpid(), strerror(ret), prio, policy);
  }
}

/* returns the current priortiy */
int get_priority() {
  int policy;
  struct sched_param threadparam;

  pthread_t tid = pthread_self();
  pthread_getschedparam(tid, &policy, &threadparam);
  return threadparam.sched_priority;
}

/* returns the current policy */
int get_policy() {
  int policy;
  struct sched_param threadparam;
  pthread_t tid = pthread_self();
  pthread_getschedparam(tid, &policy, &threadparam);
  return policy;
}

/* get cpu current thread is running on (may not work in virtualised envs) */
unsigned int get_cpuid() {
  unsigned int cpu;
  uint32_t leaf = 0x0B;
  asm volatile ("cpuid":"=d" (cpu):"a"(leaf):"%rbx", "%rcx");
  return cpu;
}

/* get the time stamp counter(TSC) */
```

Listing A.5

```
unsigned long long int rdtsc() {
  unsigned a, d;
  __asm__ volatile ("rdtsc":"=a" (a), "=d"(d));
  return ((unsigned long long)a) | (((unsigned long long)d) << 32);
}

/* get the cpu freq in MHz */
double rdcpuspeed(void) {
#ifndef INTEL
  double result = 2600000000.0;

#else
  char string[65];
  char *pstring = string;
  string[64] = '\0';
  double result = 0.0;

  for (unsigned int i = 0x80000002; i <= 0x80000004; i++) {
    unsigned long eax, ebx, ecx, edx;
    char tmpstring[17];
    tmpstring[16] = '\0';
    int j = 0;

    __asm__ volatile ("cpuid":"=a" (eax), "=b"(ebx), "=c"(ecx), "=d"(edx):"a"(i));

    for (j = 0; j < 4; j++) {
      tmpstring[j] = eax >> (8 * j);
      tmpstring[j + 4] = ebx >> (8 * j);
      tmpstring[j + 8] = ecx >> (8 * j);
      tmpstring[j + 12] = edx >> (8 * j);
    }
    strcpy(pstring, tmpstring);
    pstring += 8;
  }
  pstring = strchr(string, '@');
  if (NULL != pstring && strlen(pstring) > 6) {
    pstring += 2;
    pstring[4] = '\0';
    result = atof(pstring) * 1000000000;
  }
#endif
  return result;
}

/* calc prime numbers using given iterations - just to add workload */
double work_prime(const unsigned long long &iter) {
  timespec begin, end;
  unsigned long long i;

  stopwatch(begin);
  volatile unsigned long long max = 0;
  for (unsigned long long x = 0; x < iter; x++) {
    for (i = 2; i < x; i++) {
      if (x % i == 0)
        break;
    }
    if (i == x) {
      max = x;
    }
  }
  (void)max;
```

Listing A.5

```
  stopwatch(end);

  return diff2double(begin, end);
}

/* measure addition for given iterations - just to add workload */
double work_additions(const unsigned long long &maxloops) {
  timespec begin, end;
  volatile unsigned long long loopcnt = 0;

  assert(maxloops <= ULLONG_MAX);

  stopwatch(begin);
  for (; loopcnt < maxloops; loopcnt++) {
    ;
  }
  stopwatch(end);

  return diff2double(begin, end);
}

/* get current time */
void stopwatch(timespec & curtime) {
  clock_gettime(CLOCK_MONOTONIC, &curtime);
}

/* diff two timespecs */
timespec diff(timespec start, timespec end) {
  timespec temp;
  if ((end.tv_nsec - start.tv_nsec) < 0) {
    temp.tv_sec = end.tv_sec - start.tv_sec - 1;
    temp.tv_nsec = 1000000000 + end.tv_nsec - start.tv_nsec;
  } else {
    temp.tv_sec = end.tv_sec - start.tv_sec;
    temp.tv_nsec = end.tv_nsec - start.tv_nsec;
  }
  return temp;
}

/* convert timespec to double */
double timespec2double(timespec clock) {
  double result;
  result = clock.tv_sec;
  result += (double)clock.tv_nsec / (double)1E9;
  return result;
}

/* diff two timespecs */
double diff2double(timespec start, timespec end) {
  return timespec2double(diff(start, end));
}

/* calculate sd on given array */
double standardDeviation(double data[], int n) {
  double mean = 0.0, deviation = 0.0;
  int i;
  for (i = 0; i < n; ++i) {
    mean += data[i];
  }
  mean = mean / n;
  for (i = 0; i < n; ++i)
```

Listing A.5

```
    deviation += (data[i] - mean) * (data[i] - mean);
  return sqrt(deviation / n);
}
```

Listing A.5: Helper functions

A.3 Evaluate inter-application-container communication

The following experiment evaluates the feasibility of communication and synchronisation between different LXC application containers. Herefore two containers are created, one of which acts as ‘producer’ and one as ‘consumer’. Two corresponding applications are deployed to the respective containers. Both containers share a shared memory region which contains a semaphore, mutex and a data buffer. The producer increments the buffer value and repeatedly triggers the semaphore. The consumer waits for a trigger after reading the current buffer value from the shared memory. The setup is depicted in Figure A.7. The concurrent access to the buffer is synchronised with the mutex located in shared memory.

Due to the possibility of making use of inter-application-container event triggers and data transfer of potentially complex data, application container provides an applicable solution for partition software components while preserving efficient communication facilities.

A.3.1 Listings

The following code listings exemplify the realisation of the experiment on LXC application-container.

Listing A.6 details the configuration and life cycle management for two containers to execute the producer (cf. Listing A.8) and consumer respectively (cf. Listing A.9). These communicate using the structure detailed in Listing A.7.

Listing A.6

```
#!/bin/bash

# desc: setup and start two lxc containers that communicate via shm
# note: ubuntu 12.04 (or newer) recommended

### check host environment
apps_required="lxc-start debootstrap"
apps_missing=""
```

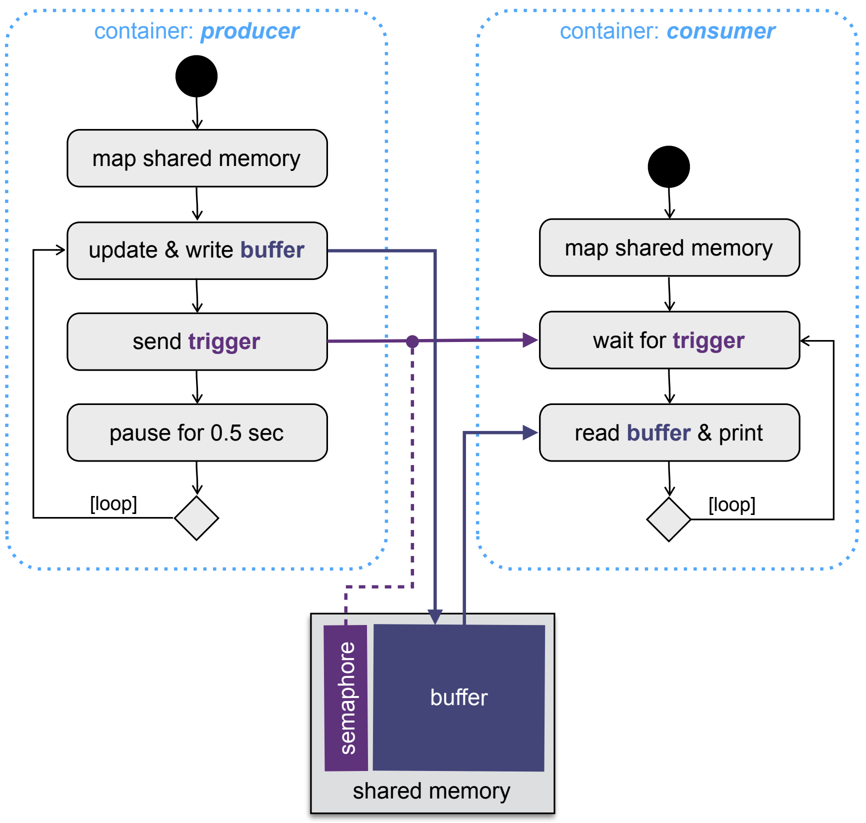

Figure A.7: Producer and consumer with LXC application container

Listing A.6

```
for i in $apps_required ; do
  which $i > /dev/null
  if [ $? -ne 0 ] ; then
    apps_missing="$apps_missing $i"
  fi
done

if [ -n "$apps_missing" ] ; then
  echo ">> ERROR: the follwing applications are missing for the host environment: [
        $apps_missing ] (please install using e.g. apt-get or similar)"
  exit 2
fi


### check privilegues
if [ "$UID" -ne "0" ] ; then
  echo ">> ERROR : su privileges required. abort"
  exit 2
fi

echo ">> create test applications"
make all
```

Listing A.6

```
### create and configure lxc containers and deploy test applications
### note: busybox template is used (cf. /usr/lib/lxc/templates)
### note: the containers are located in /var/lib/lxc
### note: test applications are named as their containers

declare -a CONTAINERS=(producer consumer)    # order is important for startup
declare -a XTERM_WINDOW_LOCATION=(+0+0 -0+0)

echo ">> now setup the containers"
for c in ${CONTAINERS[@]} ; do
  echo ">>> create container [$c]"
  lxc-create -n $c -t busybox

  echo ">> configure shm mount"
  echo "lxc.mount.entry=/dev/shm dev/shm none bind 0 0" >> /var/lib/lxc/$c/config

  echo ">> configure startup behaviour"
  sed -i "s/\/bin\/udhcpc/\/bin\/$c/g" /var/lib/lxc/$c/rootfs/etc/init.d/rcS

  echo ">> deploy app [$c] to container [$c]"
  cp $c /var/lib/lxc/$c/rootfs/bin

  echo
done


echo
echo ">> now start the containers"
cnt=0
for c in ${CONTAINERS[@]} ; do
  echo ">>> start container [$c]"
  xterm -geometry 80x24${XTERM_WINDOW_LOCATION[$cnt]} -e "lxc-start -n $c" &
  sleep 1
  cnt=$(($cnt+1))
done

echo
echo -n ">> now wait a while ... "

sleep 5

echo " done"
echo

echo ">> now stop the containers"
for c in ${CONTAINERS[@]} ; do
  echo ">>> stop container [$c]"
  lxc-stop -n $c
done
echo

echo ">> now cleanup"
for c in ${CONTAINERS[@]} ; do
  echo ">>> destroy container [$c]"
  lxc-destroy -n $c
done

echo ">> done"
```

Listing A.6: LXC setup and startup

Listing A.6

Listing A.7

```c
#ifndef GLOBAL_DEF_H_
#define GLOBAL_DEF_H_

#include <sys/mman.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <semaphore.h>
#include <errno.h>
#include <pthread.h>

#define SHM_ID "/lxc-consumer-producer"

/* shared memory structure */
typedef struct {
  int buffer;
  sem_t semaphore;
  pthread_mutex_t mutex;
} DATA;


#endif
```

Listing A.7: Data structure within shared memory

Listing A.8

```c
/* producer.c -  maps SHM and modify buffer before trigger event */

#include "global_def.h"

int main() {
  DATA *datum;
  pthread_mutexattr_t attribut;

  /* connect to shared memory */
  int fd = shm_open(SHM_ID, O_RDWR | O_CREAT, 0600);
  if (fd == -1) {
    perror("failed to open SHM");
    return 1;
  }
  if (ftruncate(fd, sizeof(DATA)) != 0) {
    perror("failed to ftruncate SHM");
    return 1;
  }
  datum = (DATA *) mmap(0, sizeof(DATA), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

  /* init semaphore */
  assert(sem_init(&datum->semaphore, 1, 1) == 0);
```

Listing A.8

```
  datum->buffer = 0;

  /* init mutex */
  assert(pthread_mutexattr_init(&attribut) == 0);
  assert(pthread_mutexattr_settype(&attribut, PTHREAD_MUTEX_RECURSIVE) == 0);
  assert(pthread_mutexattr_setpshared(&attribut, PTHREAD_PROCESS_SHARED) == 0);
  assert(pthread_mutex_init(&(datum->mutex), &attribut) == 0);
  pthread_mutexattr_destroy(&attribut);

  printf("producer: start main loop (incr buffer -> trigger event -> wait ->
        startover)\n");
  while (1) {
    pthread_mutex_lock(&(datum->mutex));
    datum->buffer++;
    printf("%i\n", datum->buffer);
    pthread_mutex_unlock(&(datum->mutex));
    sem_post(&datum->semaphore);

    usleep(500000);
  }

  /* unreachable, for the sake of completeness */
  sem_destroy(&datum->semaphore);

  return 0;
}
```

Listing A.8: Producer

Listing A.9

```
/* consumer.c - maps SHM and reads buffer on trigger event */

#include "global_def.h"

int main() {
  DATA *datum;

  /* connect to shared memory */
  int fd = shm_open(SHM_ID, O_RDWR | O_CREAT, 0600);
  if (fd == -1) {
    perror("failed to open SHM");
    return 1;
  }
  datum = (DATA *) mmap(0, sizeof(DATA), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

  printf("consumer: start main loop (wait for trigger -> print buffer -> startover)\n
        ");
  while (1) {
    sem_wait(&datum->semaphore);
    pthread_mutex_lock(&(datum->mutex));
    printf("%i\n", datum->buffer);
    pthread_mutex_unlock(&(datum->mutex));
  }

  /* unreachable, for the sake of completeness */
```

Listing A.9

```
    sem_destroy(&datum->semaphore);

    return 0;
}
```

Listing A.9: Consumer

The detailed implementation is limited to very basic communication and synchronisation mechanisms. However, those primitives can be used to set up the base for more complex scenarios. For example, event-based communication of OpenICM relies on mutexes, semaphores and shared memory.

A.4 Forwarding display content using QNX Screen

The following experiment evaluates the capturing and forwarding of a partition's virtual display content to another partition following the architecture proposed in Chapter 6.

Herefore, two OS domains are employed. One relies on QNX and represents the graphics source, whereas the receiver (i.e., compositing instance) relies on Linux, as depicted in Figure A.8.

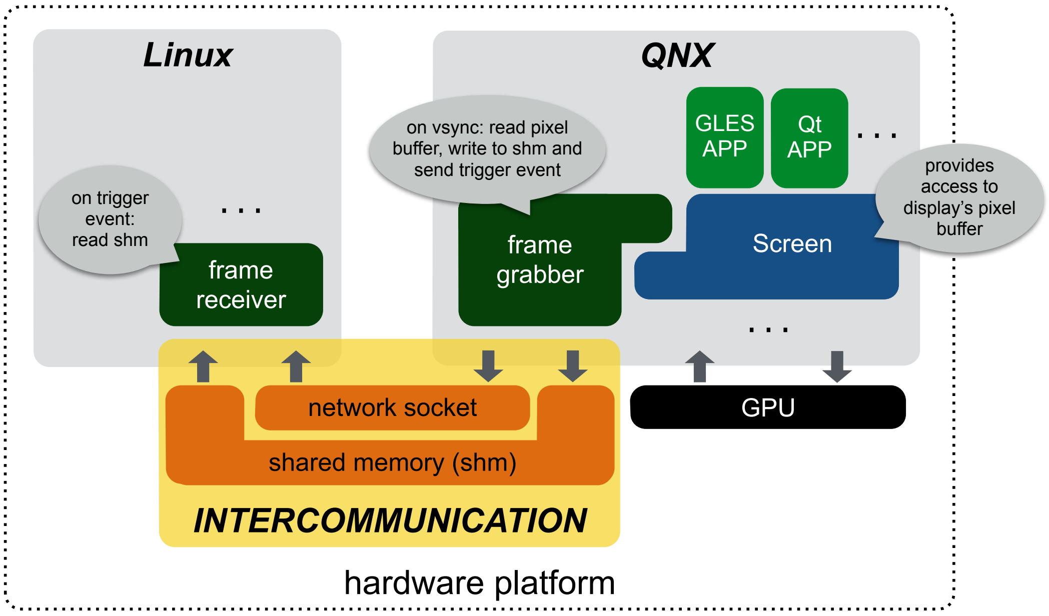

Figure A.8: Frame-grabber/-receiver

The 'frame grabber' connects the pixel buffer of QNX Screen and repeatedly copies the content to a mapped shared memory region for each display synchronisation. On each copy, a network datagram is sent to synchronise with the remote frame receiver, containing information on the graphics format and memory offset for double buffering. The receiver reads and forwards the content of the mapped shared memory on each received network datagram. The network datagram represents the event trigger to synchronize the two OS domains. As it is broadcasted using the virtual network connection, a single 'frame grabber' can be connected to multiple 'frame receivers'. However, the synchronisation for this experiment is unidirectional and hence inhibits UI- and user events addressed to the 'frame grabber'. Nevertheless, this can be implemented by use of an additional control channel or using legacy (virtual) fieldbus communications such as a CAN based connection.

The setup of the experiment corresponds with a use-case, where a formerly isolated assembly is integrated to the head-unit. Within that context the assembly is basically displaying content, such as the instrument cluster. It is not necessary to refactor or adapt the software system of that integrated assembly except for the deployment of the 'frame grabber'. The latter is independent of the software system's semantics, as it merely forwards pixel buffer information and feature inter-partition synchronisation.

For evaluation of the 'frame grabber' a GLES application renders multiple rotating objects with 30 frames per second using a graphics resolution of 640 x 480 pixels and 32 bits depth. The receiver on the Linux based OS instance were also rendered with 30 frames per second, whereas a redraw where only issued on received synchronisation event via the network socket. A degradation of the graphics quality where not observed.

A.4.1 Listings

The following code listings exemplify the implementation of a frame-grabber for QNX Screen, connected to a shared memory and synchronised by use of a socket connection, as detailed with Listing A.10. The access to the QNX Screen framework is implemented by the class 'CGraphics' which features the connection to memory region of the display's pixel-buffer (cf. Listing A.11 and Listing A.12).

Listing A.10

```
#include "CGraphics.h"
#include "CSync.h"
#include "CIVMmemory.h"
#include "CTimer.h"

int main(int argc, char *argv[]) {
  const unsigned int width = 640;
  const unsigned int height = 480;
  const unsigned int depth = 4;
  const unsigned int memsize = width * height * depth;
  const unsigned int numberOfBuffers = 2;

  // setup the event trigger
  CSync signalling(55001);
  if (0 != signalling.init()) {
    return EXIT_FAILURE;
```

Listing A.10

```
  }

  // connect to shared memory
  CIVMmemory shm;
  if (-1 == shm.getFD()) {
    return EXIT_FAILURE;
  }

  // connect to QNX screen
  CGraphics graphics(width, height, depth);
  if (0 != graphics.init()) {
    return EXIT_FAILURE;
  }

  // setup timer to determine fps
  CTimer timer(2);
  timer.init();

  // read and display content and trigger receiver
  unsigned int id = 0;
  unsigned int buffer = 0;
  while (1) {
    graphics.writeToFD(shm.getFD(), buffer * memsize);
    signalling.signal(id, buffer * memsize, 640, 480, 4);
    buffer++;
    if (buffer >= numberOfBuffers) {
      buffer = 0;
    }
    id++;
    timer.incrFPS();
  }

  return EXIT_SUCCESS;  // never reached
}
```

Listing A.10: Main routine of the frame-grabber application

Listing A.11

```
#ifndef CGRAPHICS_H_
#define CGRAPHICS_H_

#include <screen/screen.h>

class CGraphics {
public:
  CGraphics(int width, int height, int depth);
  virtual ~CGraphics();

  /** initialize screen environment */
  int init();

  /** write screen content to fd on vsync */
  int writeToFD(int fd, unsigned int offset = 0, unsigned int destSize = 0);

  /** returns size for a screenshot including depth */
```

Listing A.11

```
  unsigned int getSize() const;

  static const int FORMAT;
  static const int USAGE;

  enum ErrorCode {
    SUCCESS = 0,
    FAILED_GET_CONTEXT,
    FAILED_GET_DISPLAY_COUNT,
    FAILED_ATTACH_DISPLAYS,
    FAILED_CONNECT_DISPLAY,
    FAILED_CREATE_PIXMAP,
    FAILED_SET_PIXMAP_USAGE,
    FAILED_SET_PIXMAP_FORMAT,
    FAILED_SET_PIXMAP_SIZE,
    FAILED_CREATE_PIXMAP_BUFFER,
    FAILED_GET_PIXMAP_RENDER_BUFFER,
    FAILED_GET_BUFFER_POINTER,
    FAILED_WAIT_VSYNC,
    FAILED_READ_DISPLAY,
    FAILED_SET_OFFSET,
    FAILED_WRITE_FD,
  };

private:
  void cleanup();
  int handleError(const char* msg, int exitcode = -1);

  int mDepth;
  int mSize[2];

  screen_context_t mScreenContext;
  screen_display_t* mScreenDisplays;
  screen_display_t mScreenDisplay;
  screen_pixmap_t mScreenPixmap;
  screen_buffer_t mScreenBuffer;
  void* mScreenBufferPtr;

};

#endif /* CGRAPHICS_H_ */
```

Listing A.11: Definition of the Screen access (CGraphics)

Listing A.12

```
#include "CGraphics.h"

#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <unistd.h>

const int CGraphics::FORMAT = SCREEN_FORMAT_RGBX8888;
```

Listing A.12

```
const int CGraphics::USAGE = SCREEN_USAGE_READ | SCREEN_USAGE_NATIVE;

CGraphics::CGraphics(int width, int height, int depth) :
    mDepth(depth) {
  mSize[0] = width;
  mSize[1] = height;

  mScreenContext = NULL;
  mScreenDisplays = NULL;
  mScreenDisplay = NULL;
  mScreenPixmap = NULL;
  mScreenBuffer = NULL;
  mScreenBufferPtr = NULL;

  /* choose grahics driver root */
  setenv("GRAPHICS_ROOT", "/fs/graphics/vmware", 1);
}

CGraphics::~CGraphics() {
  cleanup();
}

unsigned int CGraphics::getSize() const {
  return mSize[0] * mSize[1] * mDepth;
}

int CGraphics::init() {
  int rc = -1;
  int displayCount = 0;
  char *disp = NULL;
  int displayID = 0;
  int displayType = 0;
  int val = 0;

  /* create screen context */
  rc = screen_create_context(&mScreenContext, SCREEN_DISPLAY_MANAGER_CONTEXT);
  if (0 != rc) {
    return handleError("create context failed\n", FAILED_GET_CONTEXT);
  }
  printf("create screen context successfully\n");

  /*
   * Retrieve the current value of the specified context property of type integer:
   * store number of current displays in param count
   */
  rc = screen_get_context_property_iv(mScreenContext,
      SCREEN_PROPERTY_DISPLAY_COUNT, &displayCount);
  if ((0 != rc) || (0 == displayCount)) {
    return handleError("get number of current displays failed\n",
        FAILED_GET_DISPLAY_COUNT);
  }
  printf("nb of current displays: %d\n", displayCount);

  /*
   * allocate heap mem for these diplays and attach to these
   */
  mScreenDisplays = (screen_display_t*) calloc(displayCount,
      sizeof(screen_display_t));
  rc = screen_get_context_property_pv(mScreenContext,
      SCREEN_PROPERTY_DISPLAYS, (void **) mScreenDisplays);
  if (0 != rc) {
```

Listing A.12

```
    return handleError("attach to current displays failed\n",
        FAILED_ATTACH_DISPLAYS);
  }
  printf("attach to current displays successfully\n");

  /*
   * select display wants to be done a screenshot
   */

  // if no display was specified, use the first supported display available for this
      context.
  if ((1 == displayCount) || !disp) {
    mScreenDisplay = mScreenDisplays[0];
  }
  // Otherwise, determine which display has been requested for the screen shot.
  else {
    if (isdigit(*disp)) {
      displayID = strtoul(disp, NULL, 0);
      for (int i = 0; i < displayCount; i++) {
        screen_get_display_property_iv(mScreenDisplays[i],
            SCREEN_PROPERTY_ID, &val);
        if (val == displayID) {
          mScreenDisplay = mScreenDisplays[i];
          break;
        }
      }
    } else {
      if (!strcmp(disp, "internal")) {
        displayType = SCREEN_DISPLAY_TYPE_INTERNAL;
      } else if (!strcmp(disp, "rgb")) {
        displayType = SCREEN_DISPLAY_TYPE_COMPONENT_RGB;
      } else if (!strcmp(disp, "dvi")) {
        displayType = SCREEN_DISPLAY_TYPE_DVI;
      } else if (!strcmp(disp, "hdmi")) {
        displayType = SCREEN_DISPLAY_TYPE_HDMI;
      } else {
        displayType = SCREEN_DISPLAY_TYPE_OTHER;
      }
      for (int i = 0; i < displayCount; i++) {
        screen_get_display_property_iv(mScreenDisplays[i],
            SCREEN_PROPERTY_TYPE, &val);
        if (val == displayType) {
          mScreenDisplay = mScreenDisplays[i];
          break;
        }
      }
    }
  }

  // check if any display could be selected
  if (!mScreenDisplay) {
    return handleError("could not select any display\n",
        FAILED_CONNECT_DISPLAY);
  }
  printf("select display successfully\n");

  // create a pixmap that can be used to do off-screen rendering.
  rc = screen_create_pixmap(&mScreenPixmap, mScreenContext);
  if (0 != rc) {
    return handleError("could not create pixmap from screen context\n",
        FAILED_CREATE_PIXMAP);
```

Listing A.12

```
  }
  printf("create pixmap successfully\n");

  // set usage for pixmap
  rc = screen_set_pixmap_property_iv(mScreenPixmap, SCREEN_PROPERTY_USAGE,
      &(CGraphics::USAGE));
  if (0 != rc) {
    return handleError("could not set usage for pixmap\n",
        FAILED_SET_PIXMAP_USAGE);
  }

  // set format for pixmap
  rc = screen_set_pixmap_property_iv(mScreenPixmap, SCREEN_PROPERTY_FORMAT,
      &CGraphics::FORMAT);
  if (0 != rc) {
    return handleError("could not set format for pixmap\n",
        FAILED_SET_PIXMAP_FORMAT);
  }

  // set size for pixmap
  rc = screen_set_pixmap_property_iv(mScreenPixmap,
      SCREEN_PROPERTY_BUFFER_SIZE, mSize);
  if (0 != rc) {
    return handleError("could not set size for pixmap\n",
        FAILED_SET_PIXMAP_SIZE);
  }
  printf("set properties for pixmap successfully\n");

  // create pixmap buffer (a pixmap could only have one buffer)
  rc = screen_create_pixmap_buffer(mScreenPixmap);
  if (0 != rc) {
    return handleError("could not create pixmap buffer\n",
        FAILED_CREATE_PIXMAP_BUFFER);
  }
  printf("create pixmap buffer successfully\n");

  // get screen buffer
  rc = screen_get_pixmap_property_pv(mScreenPixmap,
      SCREEN_PROPERTY_RENDER_BUFFERS, (void**) &mScreenBuffer);
  if (0 != rc) {
    return handleError("could not get screen buffers\n",
        FAILED_GET_PIXMAP_RENDER_BUFFER);
  }

  // get screen buffer property pointer
  rc = screen_get_buffer_property_pv(mScreenBuffer, SCREEN_PROPERTY_POINTER,
      &mScreenBufferPtr);
  if (0 != rc) {
    return handleError("could not get pointer of screen buffers\n",
        FAILED_GET_BUFFER_POINTER);
  }

  return SUCCESS;
}

int CGraphics::writeToFD(int fd, unsigned int offset, unsigned int destSize) {
  int rc;
  unsigned int wroteBytes;

  if (destSize == 0) {
    destSize = getSize();
```

Listing A.12

```
  }

  /* block until the next vsync on the specified display. */
  rc = screen_wait_vsync(mScreenDisplay);
  if (0 != rc) {
    return handleError("wait for vsync failed\n", FAILED_WAIT_VSYNC);
  }

  /*
   * take screenshot of the display and store the resulting image buffer
   */
  rc = screen_read_display(mScreenDisplay, mScreenBuffer, 0, NULL, 0);
  if (0 != rc) {
    return handleError("could not stride of screen buffers\n",
        FAILED_READ_DISPLAY);
  }

  /* write into fd */
  rc = lseek(fd, offset, SEEK_SET);
  if ((int)offset != rc) {
    return handleError("could not set offset using lseek\n",
        FAILED_SET_OFFSET);
  }
  wroteBytes = write(fd, mScreenBufferPtr, destSize);
  if (wroteBytes < destSize) {
    return handleError("could not write to fd\n", FAILED_WRITE_FD);
  }

  return SUCCESS;
}

/* print message, cleanup and return errorcode */
int CGraphics::handleError(const char* msg, int exitcode) {
  perror(msg);
  cleanup();
  return exitcode;
}

/* free data */
void CGraphics::cleanup() {
  if (mScreenPixmap != NULL) {
    screen_destroy_pixmap_buffer(mScreenPixmap);
    screen_destroy_pixmap(mScreenPixmap);
  }
  if (mScreenDisplays != NULL) {
    free(mScreenDisplays);
  }
  if (mScreenContext != NULL) {
    screen_destroy_context(mScreenContext);
  }
}
```

Listing A.12: Implementation of the Screen access (CGraphics)

B

Published papers

At the time of writing, eight papers have been published that cover significant parts of this research and included in the following pages.

(Knirsch et al., 2014) Knirsch A, Wietzke J, Moore R, Dowland PS: *Connected In-Car Multimedia: Qualities affecting Composability of Dynamic Functionality.* Proceedings of the 10th International Network Conference (INC 2014), Plymouth, UK, 2014.

(Knirsch et al., 2013b) Knirsch, A, Theis, A, Wietzke, J, Moore, R: *Compositing User Interfaces in Partitioned In-Vehicle Infotainment.* Mensch & Computer 2013, Oldenbourg, pp63–70, 2013.

(Knirsch et al., 2013a) Knirsch, A, Schnarz, P, Wietzke, J: *SHARB: Shared Resource Arbitration in Partitioned Multicore Systems via Library Interposition.* International Journal of Design, Analysis and Tools for Integrated Circuits and Systems (IJDATICS), Vol. 4, No. 2, pp18–27, 2013.

(Knirsch et al., 2012a) Knirsch, A, Schnarz, P, Wietzke, J: *Prioritized Access Arbitration to Shared Resources on Integrated Software Systems in Multicore Environments.* 3rd IEEE International Conference on Networked Embedded Systems for Every Application (NESEA), Liverpool, UK, pp1–8, 2012.

(Knirsch et al., 2012b) Knirsch, A, Vergata, S, Wietzke, J: *Strukturierung von Multimediasystemen für Fahrzeuge. Kommunikation unter Echtzeitbedingungen.* Echtzeit 2012, Informatik Aktuell, Springer, pp69–78, ISBN 978-3-642-33706-2, 2013.

(Vergata et al., 2012) Vergata S, Knirsch A, Wietzke J: *Integration zukünftiger In-Car-Multimediasysteme unter Verwendung von Virtualisierung und Multi-Core-Plattformen.* Herausforderungen durch Echtzeitbetrieb, Echtzeit 2011, Informatik aktuell, Springer, pp21-28, ISBN 978-3-642-24658-6, 2012

(Knirsch et al., 2011) Knirsch A, Wietzke J, Moore R, Dowland PS: *Resource Management for Multicore Aware Software Architectures of In-Car Multimedia Systems.* Lecture Notes in Informatics (LNI) - Proceedings, Series of the Gesellschaft für Informatik (GI), Volume P-192, 9th Workshop Automotive Software Engineering, INFORMATIK 2011: Informatik schafft Communities, Berlin, p216 (full text is available online), ISBN 978-3-88579-286-4, 2011

(Knirsch et al., 2010) Knirsch A, Wietzke J, Moore R, Dowland PS: *An Approach for Structuring Heterogeneous Automotive Software Systems by use of Multicore Architectures.* Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010), Plymouth, UK, pp19–30, ISBN 978-1-84102-269-7, 2010