Noname manuscript No.
(will be inserted by the editor)

Modelling multi-criticality vehicular software systems: evolution of an industrial component model

Alessio Bucaioni · Saad Mubeen · Federico Ciccozzi · Antonio Cicchetti · Mikael Sjödin

Received: September 2019 / Accepted:

Abstract Software in modern vehicles consists of multi-criticality functions, where a function can be safety-critical with stringent real-time requirements, less critical from the vehicle operation perspective, but still with real-time requirements, or not critical at all. Next-generation autonomous vehicles will require higher computational power to run multi-criticality functions and such a power can only be provided by parallel computing platforms such as multi-core architectures. However, current model-based software development solutions and related modelling languages have not been designed to effectively deal with challenges specific of multi-core, such as core-interdependency and controlled allocation of software to hardware.

In this paper, we report on the evolution of the Rubus Component Model for the modelling, analysis, and development of vehicular software-systems with multi-criticality for deployment on multi-core platforms. Our goal is to provide a lightweight and technology-preserving transition from model-based software development for single-core to multi-core. This is achieved by evolving the Rubus Component Model to capture explicit concepts for multi-core and parallel hardware and for expressing variable criticality of software functions. The paper illustrates these contributions through an industrial application in the vehicular domain.

Keywords Model-based engineering, metamodelling, single-core, multi-core, multi-criticality, vehicular embedded systems, real-time systems.

A. Bucaioni, S. Mubeen, F. Ciccozzi, A. Cicchetti, M. Sjödin
School of Innovation, Design and Engineering
Mälardalen University, Sweden
E-mail: {name.surname}@mdh.se

A. Bucaioni, S. Mubeen
Arcticus Systems AB, Järfälla, Sweden
E-mail: {name.surname}@arcticus-systems.com

1 Introduction

Software has become the heart of modern systems across most domains and is enhancing, sometimes even replacing, an ever larger number of mechanical and electrical parts. The automotive industry, for instance, has experienced a dramatic increment of software in vehicles. A modern car is a software-intensive system, while historically vehicles have been considered to be mechanics-intensive [49]. The same is happening in other domains, such as, for example, aerospace, automation, and robotics.

In our research, we target vehicular applications, where multiple networks, consisting of Electronic Control Units (ECUs), sensors, and actuators whose most value-added functionalities are realised by software. An example is the throttle control system, implemented now in these terms and replacing the old-fashioned mechanical linkage between the gas pedal and the throttle valve. Besides single specific control functionalities, the current trend in automotive is to empower software to steer vehicles, break, and in general to control complex vehicle behaviours. While most current vehicular systems only leverage single-core ECUs, to be able to provide suitable computational power to such a complex software, the trend is to switch to ECUs with multi-core microprocessors.

Shifting to multi-core impacts the way vehicular software is designed, analysed and developed. Current model-based solutions, specifically tailored to single-core, are hardly reusable when dealing with challenges specific to multi-core [33], such as core-interdependency and allocation of parallel software to hardware. The latter is further complicated by the so called *multi-criticality* problem. Vehicular software typically consists of functions with different criticality levels: some are safety-critical with stringent real-time requirements (e.g., airbag deployment), some are not safety-critical although with real-time requirements (e.g., speedometer), others are not critical (e.g., infotainment). In this landscape, a challenge is to design and develop multi-criticality software and allocate it to hardware in a reliable and cost-effective manner.

We have long been working on enhancing Rubus [1], a commercial model-based approach and modelling language for vehicular single-core systems, to address multi-core platforms with the intent of not disrupting the existing vehicular software-assets developed in the vehicular industry. Such core assets include:

- *own legacy*, up to 90% of the software in a new vehicle release can be reused from previous releases [47];
- *supplier legacy* Original Equipment Manufacturers (OEMs) have decennial contracts with Tier-N suppliers and changes to assets shall not affect them, and
- *certified run-time support*[1], important since model-based solutions rely on certified development environments and real-time operating systems

[1] Rubus RTOS is certified according to the ISO 26262 safety standard, whereas the certification of the Rubus-ICE development environment is undergoing.

(RTOS) [2] and, typically, the certification process adds a development cost overhead between 25 and 100% [46].

Our goal was to evolve the Rubus Component Model (RCM) [40], core of the Rubus approach, to support multi-core in modelling multi-criticality vehicular software systems. In this paper, we describe the process that we followed and show the results of our effort. Moreover, we discuss key lessons we learnt from this effort and that could help other technology providers facing similar transitions.

The remainder of the paper is structured as follows. Section 2 describes the specific contributions of this work and put them in relation with the authors' previous works. Section 3 motivates the selection and evolution of RCM, the background as well as the comparison between existing related approaches documented in the literature and our solution. Section 4 describes the evolution of RCM. Section 5 discusses the application of the proposed solution to an industrial application in three different configurations. Section 6 discusses our contributions from a more generic viewpoint while Section 7 concludes the paper with final remarks and intended future work.

2 Contribution

Although there are several modelling languages used in the vehicular domain such as, for example, AUTOSAR [3], ProCom [56], COMDES [41], AADL [48], we focus on RCM and its extension for multi-core due to the following reasons. Thanks to its statically synthesised communication as well as its predictable and fine-grained execution model, RCM allows overcoming the issues related to predictability [53]. Moreover, RCM uses pipe-and-filter[2] communication and distinguishes between the control and data flows among its software components; this communication mechanism resembles the sender receiver communication in the AUTOSAR standard [3]. In [55], we showed how these features contributed to enable early timing verification of the modelled system, e.g., by supporting end-to-end timing analysis [45] of software architectures of the system at early stages during the development. Lastly, the automatically generated footprint of the executable image of the system is consistently smaller than for other modelling languages [55]. Whereas our work focuses on RCM, we believe that the evolution problems that we faced are commonly emerging in the transition of similar languages towards the support of multi-core platforms.

Originally, RCM was aimed at providing support for the development of distributed real-time systems through the specification of re-usable units of software (i.e., software components). However, RCM did not feature any form of automation. In order to achieve a full-fledged model-based approach, in [16] we reverse-engineered the RCM specification in order to express it in a more

[2] The pipe-and-filter communication mechanism is used by many software component models in the vehicular domain [53], e.g., COMDES [41], ProCom [56].

canonical form, i.e. through a metamodel, called RubusMM. RubusMM included concepts for expressing software architectures and concepts for describing timing information of vehicular single-core applications. In [17], we evolved RubusMM to entail concepts *for modelling vehicular applications on multi-core*.

In this work, we report on the evolution of RubusMM, the modelling language core of the Rubus approach, to support multi-core in modelling software with multi-criticality for vehicular embedded systems. This evolution represents a crucial step in transitioning from single-core to multi-core without disrupting the current vehicular software-assets. In particular, the contribution is two-fold.

Contribution 1 (C1): *We revolve the Rubus modelling language so as to enable the prescription of the structure of multi-criticality software systems to be deployed on single- and multi-core as well as on mixed architectures*[3].

Contribution 2 (C2): *We ensure backward compatibility with legacy software systems modelled with RubusMM and do not cause any modification to the Rubus run-time layer, the certified Rubus Kernel.*

While evolving RubusMM, we built upon the previous extensions introduced in [17], focusing on introducing concepts *for modelling multi-criticality vehicular systems* (C1, C2). More specifically, we answered to the following needs:

- **Improve separation of concerns.** Separation of concerns is a long-known practice (first advocated by Dijkstra [31]), which aims at separating different aspects of software for improving separate reasoning and focused specification. Such a practice has been a pivotal aspect in the definition of modern (modelling) languages especially for those supporting the component-based design pattern [37]. Nevertheless, separation of concerns has been sometimes neglected in favour of more pragmatic choices when defining languages. This is especially true for languages having a strong industrial connotation, as in the case of RubusMM, which, prior to this extension, was fairly monolithic, blending software, hardware, and analysis aspects together. Thereby, in order to make the language more flexible to evolutions, such as the one presented in this paper, a first step was to enhance, or rather include, separation of concerns at metamodelling level in RubusMM.
- **Extend timing modelling concepts**. Real-time embedded systems require evidence that their output will be delivered at the time that is suitable for the environment they interact with. Schedulability analysis is an a-priori timing analysis technique which ascertains whether each function in the system is going to meet its timing requirements. The majority of existing timing analyses are used at an abstraction level that is close to the

[3] With respect to C1, the RubusMM presented in [17] did not provide support for multi-criticality software systems.

system implementation (e.g., code). The language needed modelling support that was expressive enough to allow the timing analysis of software architectures at a higher level of abstraction (i.e., models).
- **Extend hardware modelling concepts**. The language only allowed modelling of single-core CPU-based hardware. Therefore, we needed to provide the structural constructs for describing the multi-core, even heterogeneous, hardware architectures. This includes general information about the hardware (for example: partitions, number of cores, etc.) as well as relationships among hardware elements (for example: containment relations, etc.). To ensure compatibility with the previous RubusMM versions and back-compatibility of legacy models, we needed to take specific countermeasures especially when extending hardware modelling concepts.
- **Extend software modelling concepts**. The ISO 26262 [2] safety standard defines the Automotive Safety Integrity Level (ASIL) as a risk classification system for the functional safety of road vehicles. Accordingly, vehicular software functions are categorised with respect to the risks associated with their failure. There are four ASILs identified by ISO 26262: A, B, C, and D, where ASIL A represents the lowest degree (lowest criticality level) and ASIL D represents the highest degree (highest criticality level) of automotive hazard. In order to support the modelling and execution of software functions with different criticality levels on the same core, we use partitions to logically isolate the software functions with respect to their criticality levels. Partitioning is a proven method for implementing the logical isolation of software on the same core. A core can have a minimum of one and a maximum of five partitions, where each partition hosts software with the same criticality level that corresponds to ASIL A, B, C, D or non-critical. Note that there are several ways to implement the partitions, e.g., assigning distinct runtime priorities to each partition or assigning a dedicated execution budget to each partition with respect to its criticality level. In this work, we assume the priority-based implementation.

3 Background and related work

3.1 The Rubus concept

Rubus is developed by Arcticus Systems[4] in collaboration with Mälardalen University and other academic and industrial partners [51]. The Rubus concept is based around the RCM [40] and its development environment Rubus-ICE (Integrated Component development Environment), which includes modeling tools, code generators, analysis tools and run-time infrastructure. Rubus also provides a real-time operating system which has been certified to the highest ASIL level according to the ISO 26262 safety standard. The overall goal of Rubus is to be aggressively resource-efficient and to provide means for developing predictable and analyzable control functions in resource-constrained em-

[4] `https://www.arcticus-systems.com`

bedded systems. RCM and Rubus tools have been used in the vehicle industry for over 25 years by several OEMs and Tier-1 companies (e.g., Volvo Construction Equipment, BAE Systems Hägglunds, Hoerbiger and Knorr Bremse) for the software development of real-time embedded systems.

3.1.1 The Rubus Component Model (RCM)

The purpose of RCM is to express the infrastructure for software functions. In RCM, a Software Circuit (SWC) is the lowest-level hierarchical element and its purpose is to encapsulate software functions. An SWC can be seen as a type, or a class that can be instantiated an arbitrary number of times. The interaction between the SWCs is clearly separated in terms of data and control flows for facilitating the definition of the control specification, typical of real-time embedded systems, and interactions. Hence, the SWCs interact with each other via data and control ports, separately. Data ports used for modelling data communication while control ports are used for modelling triggering conditions. One important principle in RCM is to separate functional code from the infrastructure implementing the execution model. This allows visualising explicit synchronisation and data access at the modelling level, distinctively. Furthermore, this principle facilitates the analysis and reuse of SWCs in different contexts (an SWC has no knowledge how it connects to other components). The execution semantics of an SWC are as follows:

1. upon receiving a trigger signal on the trigger input port, read data on all data input ports;
2. execute the encapsulated software function;
3. write data to all data output ports; and
4. activate the trigger output port.

An example software architecture of a single-node system modelled with RCM, depicted in Figure 1, shows how components interact with external events and actuators with respect to both data and triggering. The figure also shows the internal structure of one of the SWCs, which consists of one or more behaviour and one interface that contains all the ports. In RCM a Behaviour element acts as a root container for the behavioural model or code. Hence, such an element can contain either software code (e.g. C) or an executable model (e.g. Simulink).

3.1.2 The Rubus analysis framework

RCM allows expressing timing requirements and properties of the software architecture. For example, it is possible to associate real-time requirements from a generated event and an arbitrary output trigger along the chain of SWCs. These requirements are expressed by means of timing constraints. All timing constraints that are part of the timing model in the AUTOSAR standard are included in RCM [55]. Moreover, the designer can express real-time properties of SWCs, such as worst-case, best-case and average-case execution times as well as the stack usage. The scheduler will take these real-time constraints into consideration when producing a schedule. For event-triggered

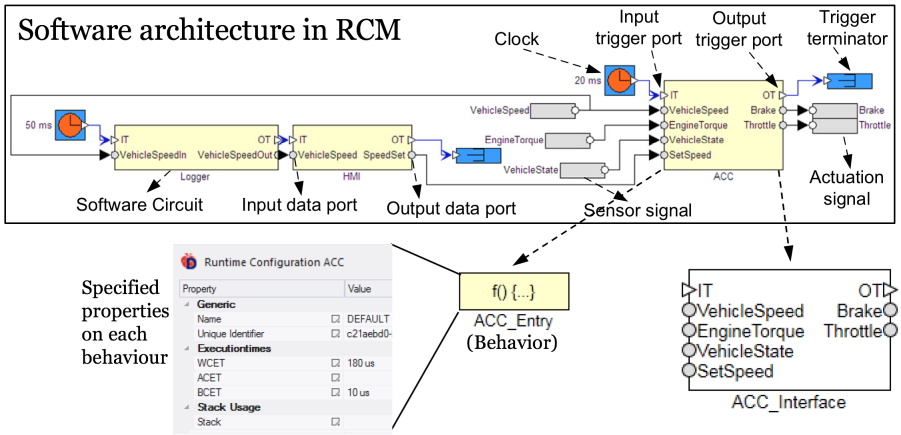

Fig. 1 Example of a software architecture modelled in RCM.

SWCs, response-time calculations are performed and compared to the corresponding timing requirements. The analysis engines support various types of timing analysis that include response-time analysis and end-to-end data-propagation delay analysis [52].

3.1.3 The Rubus run-time framework

The SWCs in the resulting software architecture are mapped to run-time entities called tasks. Each external event trigger defines a task and SWCs connected through the chain of triggered SWCs are allocated to the corresponding task. All SWC chains that are triggered by periodic clocks are allocated to an automatically generated static schedule that fulfils the precedence order and timing requirements. Within these chains, the inter-SWC communication is highly optimised to use the most efficient means of communication possible for each communication link. The mapping of SWCs to tasks and generation of the schedule can be optimised to minimise response times for different types of tasks or memory usage. The run-time system executes all tasks on a shared stack, thus eliminating the need for static allocation of stack memory to each individual task. This optimisation results in a small runtime footprint of the software architecture.

3.1.4 The Rubus multi-core hypervisor

The Rubus multi-core hypervisor uses resource-isolation techniques [35] for arbitration of intra- and inter-core shared resources. These isolation techniques are commonly used in many application domains to simplify and partition the system resources in time and space, e.g., the Avionics Application Standard Software Interface ARINC 653 [4] uses these techniques [60,61]. The Rubus hypervisor implements the Time Division Multiple Access (TDMA) protocol to arbitrate the shared system bus among the cores [62]. Similarly, memory partitioning techniques are used to isolate the shared memories, including L3

cache and RAM among the cores [29]. Isolation techniques enable cores, as well as partitions within those cores, to become virtually independent from other cores and their partitions. In other words, each partition can be seen as a single-core processor equivalent with dedicated system resources (although with reduced capacity which is equivalent to the allotted size of shared memory and bandwidth of the system bus). One notable advantage of the single-core processor equivalent model is that the overall system becomes simple to model, i.e., there is no need to explicitly model memories, I/Os and other shared resources in the software architecture.

3.2 Related work

In this section, we survey the principal strands of research that relate to the work we present in this paper. In Section 4 we use these lines of research for categorising the proposed extensions, and similarly in Section 6 we use them to discuss strengths and limitations of this work.

3.2.1 Enhance separation of concerns

Numerous research efforts[5], AMALTHEA/APP4MC[6], CHESS [22], DREAMS [50], MultiPARTES [57] address the design of multi-criticality embedded real-time systems in a similar way to what is proposed in this article. In particular, they also propose the separation of concerns when dealing with software, hardware, and allocation modelling, that is indeed an implicit consequence of obeying to safety standards. Nonetheless, typically these solutions aim to be very generic in order to support multiple application domains, hence not only automotive but also industrial automation, avionics, and so forth. On the one hand, this generality makes the approaches more malleable, e.g. in case of long-lasting maintenance and evolution of existing applications. On the other hand, the price to pay is the inherent complexity of eliciting and specifying all the details such that the tooling support would work as expected (e.g. validation checks, generation of artefacts, etc.). In this respect, a domain-specific approach like the one we propose embeds most of the automotive systems characteristics in the language and surrounding tooling, relieving the users from the burden of specifying all the details. However, major technical updates (notably the introduction of multi-core platforms) would typically require an update of the language, tooling, and existing models (see for instance the work in [30] about consequences of AUTOSAR metamodel evolutions).

3.2.2 Extend timing modelling

There exists a number of languages that support modelling of timing properties and requirements in vehicular embedded software systems. TIMMO [13] is an industrial initiative to provide AUTOSAR [10] with a timing model and it is based on a language called the Timing Augmented Description Language

[5] `https://af3.fortiss.org`

[6] `http://www.amalthea-project.org`

(TADL) [11] and inspired by MARTE [5]. TADL is redefined and released in the TADL2 [6] specification of the TIMMO-2-USE project [12]. There are several other approaches from academia like COMDES-II [41] and ProCom [56]. Most of these initiatives lack the support for expressing low-level details at the higher levels such as linking information in distributed chains. This information is necessary to extract the end-to-end timing information from the software architecture to perform its end-to-end timing analysis [52]. Moreover, no support is provided for extracting this information from the software architectures of these systems. Most of the above cited languages such as AUTOSAR, ProCom, COMDES and CORBA [39] support modelling of on board real-time network communication. However, most of them allow modelling of only low-bandwidth real-time on-board networks, e.g., CAN, while there are very few works that support modelling of high-bandwidth real-time on-board networks that are based on, e.g., switched Ethernet. The work in [43] is one of such works which provides support for modelling of switched Ethernet on-board networks. AUTOSAR, complemented by the SymTA/S tool, facilitates modelling of Ethernet, but lacks support for modelling Time Sensitive Networking (TSN) network. In [44], the authors present a work in progress for developing an approach to configure TSN network and verify its configuration.

3.2.3 Extend hardware modelling concepts

There exists a large body of work on multi-criticality systems for single-core systems. The difficulty of realising multi-criticality systems on multi-core is the management of shared resources, which in general makes the timing behaviour of the system very difficult to predict. There exist two different definitions and underlying models of multi-criticality systems in the real-time embedded systems domain. The first model is based on Vestal's work, where different criticality levels are associated to individual software components (at design time) or corresponding tasks (at runtime) [58]. Whereas, the second model associates a unique criticality level to the complete application and not to individual software components or tasks within the application. This latter model complies with various standards such as the functional safety standard for road vehicles ISO 26262 [2] and the aerospace standard DO-178C [7]. These standards also prescribe that the criticality level of an application shall be set as its most critical sub-function, unless time and memory independence between sub-functions is demonstrated. In practice, development efforts tend to take the latter path (i.e., implementing and demonstrating independence), since in general certifying the whole application for the highest criticality would be prohibitively expensive [36]. In this work, we extend RCM with support for multi-criticality systems according to the model in [24]. A complete discussion of the state of the art on multi-criticality systems goes beyond the scope of this article, and the interested reader is referred to [20] [21]. Here it is worth noting that a body of literature has been devoted to scheduling solutions that are able to maximise resources utilisation while preserving the timing correctness of system execution, especially for the Vestal's model of multi-criticality [58].

Vehicular software has to obey safety standards like ISO 26262 [2], therefore the notion of criticality considered in this article conforms to the standards, as clarified above. In particular, for the approach proposed in this article the development process relies on a hypervisor mechanism (e.g., ARINC 653 hypervisor [60]) provided by the underlying runtime environment, i.e. time and memory separation between logical partitions are guaranteed according to a software layer. In this way, software functions can be grouped according to their criticality level and deployed on partitions in a consistent manner, while the runtime guarantees that each partition can be analysed in isolation for certification purposes. Moreover, the correctness of design models with respect to allocation choices is preserved by language constraints, such that any model including heterogeneous allocations, i.e. sub-functions with different criticality levels allocated to the same partition, would make model validation checks to fail.

There exist other hypervisor/virtualisation-based solutions in the literature, like for example the one presented in [57], that is based on the XtratuM virtualisation layer [23]. Other approaches that leverage hardware features to ensure space and time separation required by multi-criticality systems. Notably, in [36] the authors illustrate the safety argumentation for an automotive case study based on the AURIX processor family. Those concerns however fall outside of the scope of this paper, since its primary goal is to spare the need to re-certify the Rubus kernel.

3.2.4 Extend software architectural modelling

Some of the mentioned approaches like CHESS [22], DREAMS [50] and MultiPARTES [57] provide also design-/deployment-space exploration (DSE) features, that is they support the automatic synthesis of allocation and scheduling configurations such that multi-criticality and other safety constraints are obeyed [59][27]. Even though in this article we consider the allocation as a manual activity, the language expressiveness does not prevent the development of DSE mechanisms to automatically derive allocation alternatives. Indeed, in our previous works we have already proposed model transformations able to perform DSE to generate correct timing configurations for the system under development [19].

EAST-ADL [8] is an architecture description language devoted to the specification of automotive electronic systems. To cope with vehicular systems' complexity, EAST-ADL leverages a multi-layer approach, where each layer describes the system at a different abstraction level and from a different perspective. In particular, EAST-ADL defines the vehicle, analysis, design, implementation, and operational layers, and each layer includes engineering information like requirements, vehicle features and functions, variability, software and hardware components, and communication. The implementation layer is usually delegated to domain-specific languages (DSL), notably AUTOSAR, RCM, and so forth. Therefore, the work described in this article can be considered as complementary to EAST-ADL. Indeed, we have already shown in [15,14] how to employ model transformations to close the abstraction gaps between

EAST-ADL and the DSL used at implementation level (namely RCM) and to anticipate timing analysis at the EAST-ADL design level. This work furthers the same front of research, introducing modelling elements for handling multi-core platforms and multi-criticality issues.

Besides automotive-specific modelling technologies, there exists a plethora of general purpose approaches that could be used to complement the implementation level DSLs like AUTOSAR and RCM. In general those approaches use the UML language and its profiles MARTE [5] and/or SysML [32].

AADL [48] was conceived as an architecture description language for the avionics domain, but it is being increasingly used for modelling embedded systems in general. When compared to RCM, AADL also provides multi-core support and a clear separation of concerns between software and hardware elements. However, AADL adopts a lower level of abstraction approach for software architecture modelling, where typical elements are, e.g., *Processes* and *Threads*.

VERTAF/Multi-core is a UML/SysML-based framework for the development of multi-core software [28]. VERTAF/Multi-core proposes UML class diagrams, timed state machines, and sequence diagrams as the modelling instruments to describe multi-core software systems. The viability of the design with respect to schedulability and conformance to the specifications is verified automatically through model transformations. In particular, the transformations generate opportune extensions of the input models to enable the analyses mentioned earlier. The UML profile MARTE provides extensive support for the design of multi-core systems. Notably, in [33] the authors provide a detailed description of the software system architecture to generate code and perform timing verification through simulation. In particular, the software components modelled in UML are complemented with hardware and software to hardware allocation specifications by means of MARTE. Instead, in [26] the authors propose to employ MARTE for system development of component-based systems: MARTE models describing the system are completed with allocation of components information through automated code generation. GASPARD [25] is a MARTE-based framework for the design and implementation of (massive) parallel embedded systems. MARTE design models are exploited as source to automatically generate implementation alternatives, giving place to design space exploration. The framework also supports code generation for formal verification, simulation and hardware synthesis. On a similar line of research, the COMPLEX framework [34] leverages MARTE models, describing embedded systems, to perform design space exploration in order to derive corresponding architectural solutions. All the above mentioned approaches are mainly devoted to optimisation of the designed system architectures, e.g. to maximise parallel execution, while in general safety/certification issues are not taken into account.

4 Evolving the Rubus Component Model

In this section, we describe the evolution of RubusMM (the Rubus metamodel formalising its component model, RCM) for modelling multi-criticality vehicle software for multi-core. The evolution entailed adding, extending and modifying modelling concepts and formalising the evolution by means of metamodelling. We compare the evolved RubusMM with its previous definition, given in [16], highlighting differences and commonalities.

4.1 Improving separation of concerns

We introduced packages for ensuring a better separation of concerns, improving the understandability of the metamodel, and enhancing its extensibility. The RubusMM packages involved in the evolution are *RCM_COMMON*, *RCM_HW* and *RCM_SW*, where *RCM_HW* contains the elements for modelling the hardware platform, *RCM_SW* contains the elements for modelling the software architecture and *RCM_COMMON* contains elements that are common to multiple packages, respectively. Prior to this evolution, RubusMM did not feature any package[7]. We removed the structural containment occurring between hardware and software elements, too. More details on this enhancement come in the following sections.

4.2 Extending timing modelling concepts

System represents the system under development. It inherits from the abstract metaclass *NamedElement*, which provides two attributes: *name* and *ID*. Such an inheritance is common to all the elements in RubusMM. We extended *System* with the reference *timingConstraint*. The addition of the reference *timingConstraint* on the element *System* as well as on other elements enables the specification of various timing requirements on these elements and the verification of the timing requirements on the software architectures using state-of-the-art timing analysis techniques [54][8]. Furthermore, the reference *timingConstraint* is pivotal for the design space generation technique described in [15]. Without this extension, it would not be possible to leverage the aforementioned techniques at system level. We extended RubusMM with *Core* and *Partition* elements (and related references and attributes) so the language could support the design and timing analysis of multi-core applications.

[7] The complete explanation of RubusMM is not in the scope of this work. The interested reader may refer to [16].

[8] *TimingConstraint* and other elements from different RubusMM packages are not part of this extension. However, they are put in relation to this extension as they contribute to a holistic view of the language and its peculiarities.

4.3 Extending hardware modelling concepts

Figure 2 shows a fragment of RubusMM containing elements from *RCM_HW* for modelling the hardware platform. One important principle driving the RubusMM evolution with respect to the hardware platforms modelling is that we aimed at introducing the minimum number of hardware elements that are crucial for the allocation of software to hardware and for modelling and extracting the timing information to support the timing analysis engines.

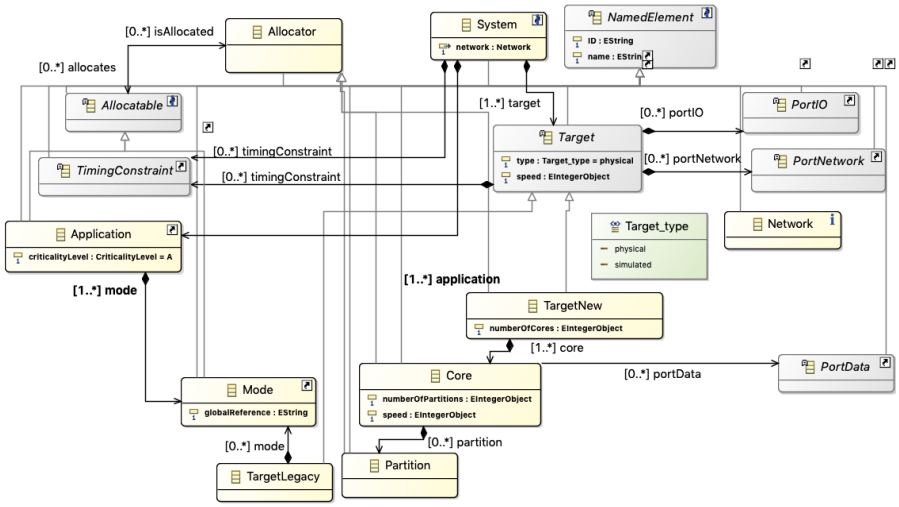

Fig. 2 Fragment of the *RCM_HW* package for modelling the hardware platform.

System contains one *Network*, one or more *Target* elements, and one or more *Application* elements. A *Network* element models all the messages exchanged among the *Target* elements and has two attributes, *protocol* and *speed*, which specify the protocol (e.g., Controlled Area Network (CAN) [38]) and the speed of the network in Kbit/s, respectively.

Target is a hardware-specific element which represents a processor architecture. It has been extended with the references *timingConstraint*, *portIO*, and *portNetwork*. As for the *System* element, the reference *timingConstraint* enables the specification of timing constraints, occurrences and events used for timing verification (at *Target* level). *portIO* and *portNetwork* model the peripherals and the inter-target communication, respectively.

In the previous definition of RubusMM, *Target* contained *Mode*, acting as a container for the software application. However, such a containment relation was too restrictive for modelling multi-criticality applications on multi-core. In fact, the containment relation between *Target* and *Mode* prescribed that software elements (represented by *Mode* elements) were structurally contained by hardware (represented by *Target* elements). Although not providing a clear

separation between software and hardware, this structural containment suited the single-core case, since allocation of software to hardware was not variably split across different cores. Modelling for multi-criticality software on multi-core demands more flexibility, since allocation of software to hardware is a variability point that can be affected by the software criticality levels and it can hardly be represented by a structural containment.

4.3.1 Ensuring backward compatibility with legacy models

In order to provide such a flexibility while ensuring backward compatibility with legacy RubusMM models, we have modified the existing hierarchy as follows. We have added the metaclasses *TargetLegacy* and *TargetNew*, both inheriting from the abstract metaclass *Target*. *TargetLegacy* represents a legacy (single-core) processor and it contains one or more *Mode* elements. This containment is specified through the reference *mode*. *TargetNew* represents a single- or multi-core processor and contains one or more *Core* elements, which in turn can contain *Partition* elements. *TargetNew*, *Core* and *Partition* elements inherit from the abstract metaclass *Allocator*, representing hardware elements to which software elements, represented by the metaclass *Allocatable*, can be allocated. The metaclasses *Allocator* and *Allocatable*, together with the reference *isAllocated*, provide the flexible mechanism for the allocation of software to hardware that we needed, without any structural containment.

4.3.2 Providing support for multi-core hardware modelling

The metaclass *Target* provides the following attributes: *speed*, which specifies its speed in MHz, and *type*, which specifies whether it is a *physical* or a *simulated* target. A simulated target represents the simulation of the actual target processor in a host environment such as Windows or Linux. Both *TargetLegacy* and *TargetNew* inherit the *speed* and *type* attributes. Moreover, *TargetNew* provides additional multi-core specific attributes. *numberOfCores* specifies the number of cores composing the *TargetNew* and it is used by the model-based timing analysis and to automatically allocate software to hardware. The reference *core* links *Core* elements to their respective *TargetNew* elements.

Core may contain *Partition* elements. The attribute *numberOfPartitions* specifies the number of partitions within a *Core* and the reference *partition* links them to the *Core* elements. *Partition* elements represent a logical division of a core into multiple sets of resources so that each *Partition* element can operate independently. In other words, the *Partition* element isolates one part of software from the other in both time and space. Isolation in time means that each partition gets a reserved share of the core processing time for the execution of the software allocated to it. Isolation in space means that the memory available to each core is divided among its partitions. Within RubusMM, any inter-partition interference is prevented by using memory protection mechanisms. As discussed above, the isolation in time and space is supported by the Rubus multi-core hypervisor by means of resource-isolation techniques [35].

Target, *TargetLegacy*, *TargetNew*, *Core*, *Partition*, *Allocator*, *Allocatable*, as well as their attributes and related references were not part of the previous RubusMM definition [14]. It is important to note that the modelling elements describing multi-core processors could have been introduced without modifying the original structural containment. However, this choice would have limited the usability and flexibility of RubusMM as the allocation choices could be made early in the development process when relevant information may not be available. Besides, the allocation mechanism introduced in RubusMM is common in many automotive domain-specific modelling languages such as EAST-ADL, AUTOSAR, etc.

4.4 Extending software architectural modelling concepts

Figure 3 shows a fragment of the RubusMM containing elements from the *RCM_SW* and the *RCM_COMMON* packages for modelling the software architecture.

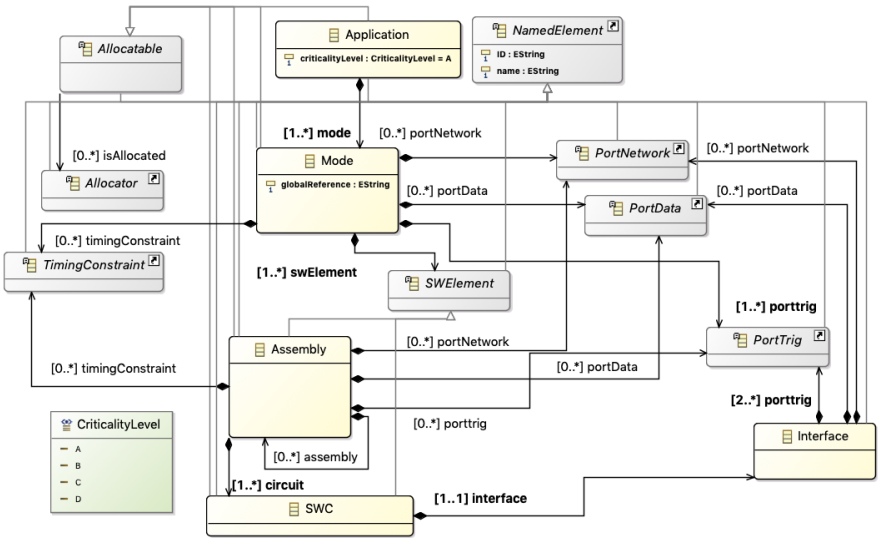

Fig. 3 Fragment of the *RCM_SW* package for modelling the software architecture.

4.4.1 Enforcing correctness of criticality-driven allocation of SW to HW

Within RubusMM, *Application* models a piece of software implementing a dedicated functionality, e.g., brake-by-wire. *Application* has one attribute, *criticalityLevel* and contains one or more *Mode* elements. This containment is specified through the reference *mode*. *criticalityLevel* specifies the level of safety criticality according to the ISO 26262 automotive safety standard. The

ISO 26262 standard has four levels of criticality (A to D) where A is the lowest criticality level, whereas D is the highest criticality level (the Rubus Kernel supports and is certified for all of them). *Application* and *criticalityLevel* along with the aforementioned allocation mechanism are pivotal in modelling multi-criticality software on multi-core, where software applications with different criticality levels cannot be allocated together on the same partition. In order to prevent such a situation, we enriched RubusMM with a structural constraint specified by means of the Object Constraint Language (OCL)[9] as an invariant of *Partition* elements. Listing 1 shows the pseudo-code for such a constraint.

```
1 FOR each application::Application allocated to partition::Partition
2     criticality.add(application.criticalityLevel);
3
4 IF criticality.size()<=1 THEN
5     True
6 ELSE
7     False
8 }
```

Listing 1 Eclipse OCL constraint avoiding that no *Application* elements with different criticality levels are allocated on the same *Partition* element.

For each *Partition* element, the constraint retrieves all its allocated *Application* elements. Their *criticalityLevel* values are collected into a set. If the size of the set is grater than 1, the constraints would return the logical value false which, in turns, will raise a validation error.

4.4.2 Extending allocability of SW to HW

In RubusMM a software circuit is represented by *SWC* and it is the lowest-level hierarchical element that encapsulates basic software functions. A *SWC* contains one *Interface* which groups all its ports. As RubusMM distinguishes between the data and control flows, an *Interface* contains*PortData* and *PortTrig* elements. The *PortData* elements manage the data communication among *SWC* deployed on the same *Target*. The *PortTrig* elements manage the activation of the *SWC* elements. A *PortNetwork* is a port for the data communication of *SWC* elements deployed on different *Target* elements. The *PortData* elements of a *Core* are referenced to the *PortData* elements of the *SWC*s allocated on that *Core*. Similarly, the *PortNetwork* elements of a *Node* are referenced to the *PortNetwork* elements at *SWC* level. An *Assembly* groups *SWC* and *Assembly* elements in a hierarchical fashion. Its reference *timingConstraint* enables the specification of timing constraints, occurrences and events which are used for timing verification. With respect to the previous definition, *SWC* and *Assembly* have been extended with the inheritance relation from the abstract metaclass *Allocatable*. A *Mode* groups *Assembly* and *SWC* elements and it is used for modelling a specific configuration of the software architecture (e.g., start-up or error mode). The attribute *globalReference* serves for creating a reference among all the *Mode* elements contributing to the same mode. With respect to its previous definition, *Mode* has been extended with the inheritance relation from the abstract metaclass *Allocatable*.

[9] https://projects.eclipse.org/projects/modeling.mdt.ocl

The metaclasses *Allocatable* and *Allocator* together with the reference *isAllocated* enable the specification of the allocation of software to hardware. More precisely, an *Allocatable* element can be deployed to an *Allocator* element by setting the *isAllocated* reference. *Application*, *Allocatable*, *Allocator*, and related references were not part of the previous RubusMM definition.

5 Modelling the brake-by-wire application

In this section, we leverage the evolved RubusMM for modelling the Brake-by-wire (BBW) vehicular application, which is an innovative stand-alone braking system currently deployed in premium electric and hybrid vehicles [9]. In a nutshell, the BBW application replaces the old-fashioned mechanical linkages and allows controlling the brakes through electronic means. Figure 4 depicts a simplified electrical/electronic (E/E) architecture of the BBW application featuring an anti-lock braking (ABS) function. A sensor attached to the brake

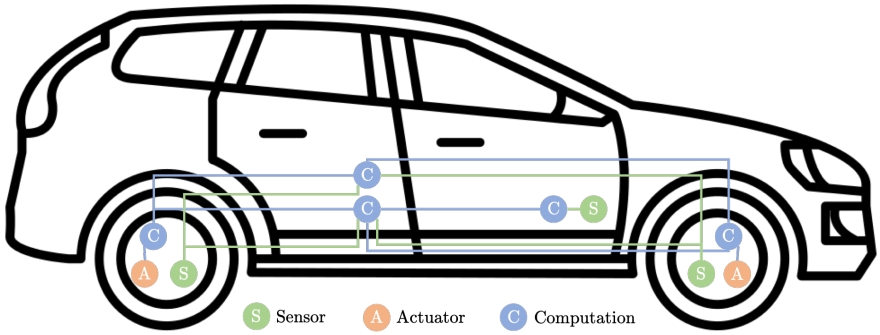

Fig. 4 Simplified E/E architecture of the BBW application.

pedal acquires the signal that corresponds to the position of the pedal. This signal is sent to a computational unit which translates it into a brake torque. For each wheel, a sensor acquires the signal that corresponds to the speed of the wheel. The sensed speed is sent to two computational units. The first unit calculates the individual brake torque for each wheel using the sensed speed and the computed brake torque. The second unit calculates the speed of the vehicle using the speed of each wheel. The ABS units use the speed of the vehicle and sensed wheels brake torques for calculating the optimal brake torque for each wheel. The actuators on the wheels release the optimal brake torques avoiding the brakes to lock. Figure 5 shows a RubusMM model depicting the software architecture of the BBW application. The model consist of 16 software circuits where i) *Brake_Pedal* models the software operating the sensor on the brake pedal, ii) *Speed_FR*, *Speed_FL*, *Speed_RR*, and *Speed_RL* model the software operating the speed sensors on the wheels, iii) *Brake_Torque*, *Brake_Controller*,

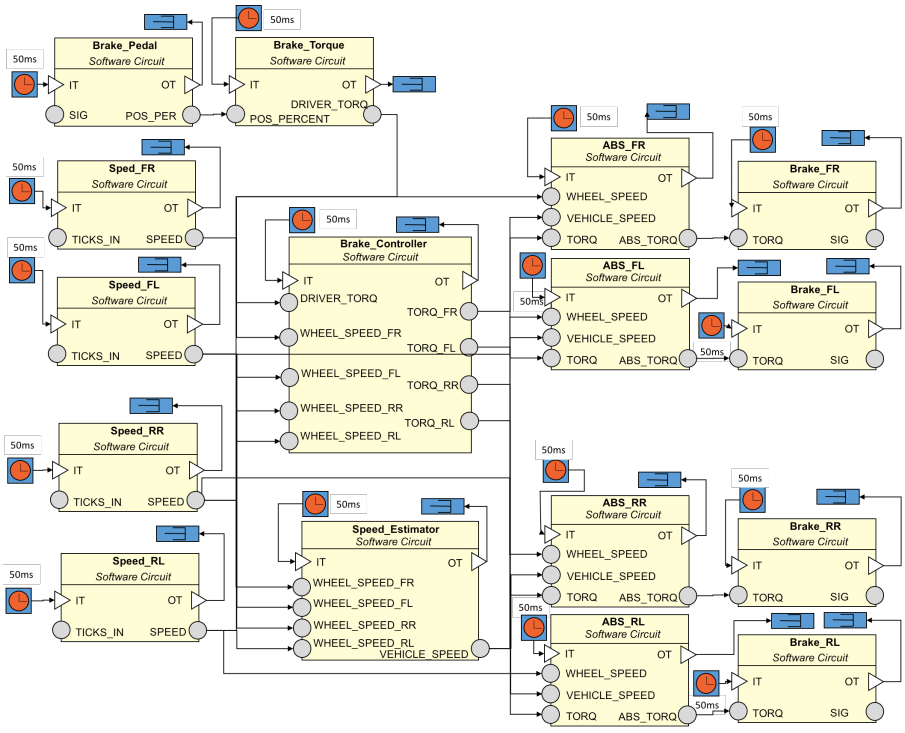

Fig. 5 RubusMM model representing the software architecture of the BBW application.

Speed_Estimator, *ABS_FR*, *ABS_FL*, *ABS_RR*, and *ABS_RL* model the software on the computational units and iv) *Brake_FR*, *Brake_FL*, *Brake_RR*, and *Brake_RL* model the software operating the actuators on the wheels. In order to show how the evolved RubusMM supports the modelling of multi-criticality software systems on multi-core (H1), while ensuring backward compatibility with legacy single-core ones (H2), we describe three different deployment configurations for the BBW application.

5.1 First configuration: legacy single-core platform

In the first configuration, the BBW application is modelled as a legacy single-core software system and deployed to a MPC5744P microcontroller[10], which is a 32-bit unicore microcontroller designed for vehicular applications. Figure 6 shows the Ecore serialisation of such a configuration.
As the BBW application is modelled as a legacy single-core system, deployment on the unicore microcontroller is modelled by the *mode* containment relation between *TargetLegacy* element *MPC574xP* and *Mode* element *Operational*. Although not providing a clear separation between software and hardware,

[10] `http://www.nxp.com/assets/documents/data/en/data-sheets/MPC5744P.pdf`

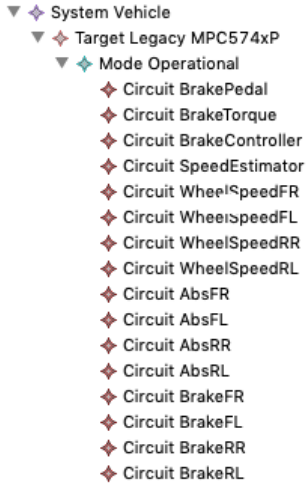

Fig. 6 Serialisation of the BBW application deployed to a unicore microcontroller.

this configuration suits the legacy single-core case as the software cannot be allocated across different cores (Section 4).

5.2 Second configuration: multi-core platform

In the second configuration, the BBW application is modelled as a multi-core software system and deployed to an Infineon SAK-TC299TP-128F300S BB microcontroller[11], which is a tri-core microcontroller developed for applications with high demands of performance and safety. Figure 7 shows an Ecore serialisation of this configuration. As the reader can notice, in this configuration, there is a clear separation between the hardware and the software elements of the BBW application. The allocation among these elements is modelled by means of the *isAllocated* reference. We decided to split the allocation of the *BBW Application* on the three available cores as follows[12] The software circuits modelling the sensors, the computation units and the actuators of the two front wheels (*WheelSpeed_FR*, *WheelSpeed_FR*, *Abs_FR*, *Abs_FL*, *Brake_FR*, *Brake_FL*) are allocated to *Core 1* of the *SAK-TC299TP-128F300S BB* target, as shown by the arrow in the top-left corner of Figure 7. Similarly, the SWCs modelling the sensors, the computation units and the actuators of the two rear wheels (*WheelSpeed_RR*, *WheelSpeed_RR*, *Abs_RR*, *Abs_RL*, *Brake_RR*, *Brake_RL*) are allocated to *Core 2* of the *SAK-TC299TP-128F300S BB* target. The remaining SWCs modelling the computational units

[11] `http://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/channel.html?channel=db3a30433727a44301372b2eefbb48d9`

[12] It is important to note that this allocation is arbitrary hence other, still valid, allocations are possible.

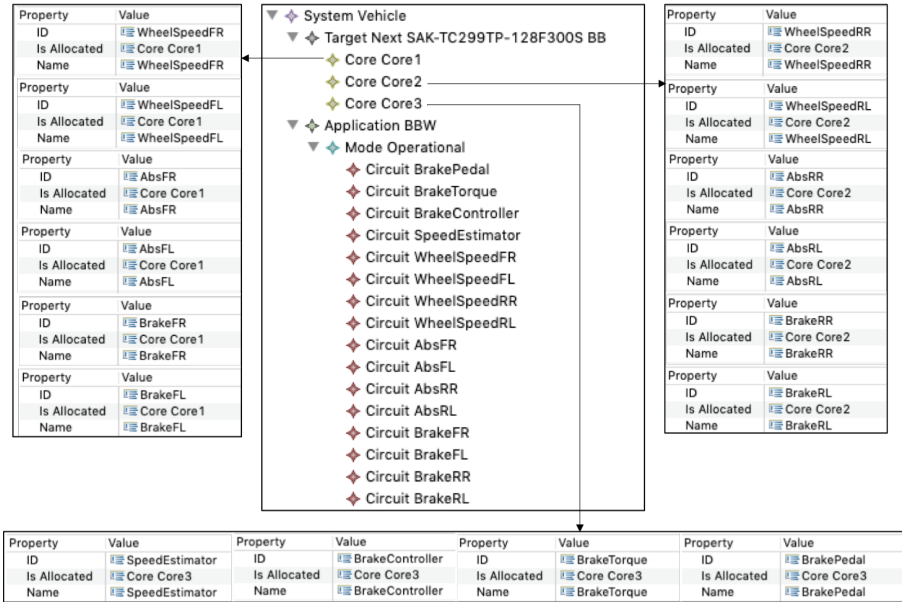

Fig. 7 Serialisation of the BBW application deployed to a tri-core microcontroller.

are allocated to *Core 3* of the *SAK-TC299TP-128F300S BB* target. As discussed in Section 4, the evolved RubusMM leverages a clearer separation of concerns as well as an explicit and more flexible allocation mechanism. For instance, let us suppose that the allocation specified in Figure 7 does not satisfy a given set of fault-tolerance requirements. One way of addressing this issue would be to model a lockstep configuration of the BBW application where each core runs a copy of the complete software in parallel. In order to model a lockstep configuration with the evolved RubusMM it would be sufficient to change the *isAllocated* reference of the *BBW Application*, only. In particular, the *isAllocated* reference of the *BBW Application* should point to *Core1*, *Core2* and *Core3*[13].

5.3 Third configuration: multi-criticality on multi-core platform

In the third configuration, we consider the Proactive Wiper (PW) vehicular application in addition to the BBW one. We model these two applications as a multi-core software system with different criticality levels and deploy them into the aforementioned tri-core microcontroller. The PW is a stand-alone system which uses the information from the vehicle's front camera, radar and rain sensor for preventing sudden water splashes (caused by large vehicles) to give

[13] It should be noted that a redundant configuration as the one described above would require a voting software mechanism to be defined and allocated. While this is a valid concern, for the sake of verbosity, we focus on the BBW and its allocation, only.

the driver an unclear view[14]. Figure 8 depicts a simplified E/E architecture of the PW application. A radar positioned on the front of the vehicle acquires

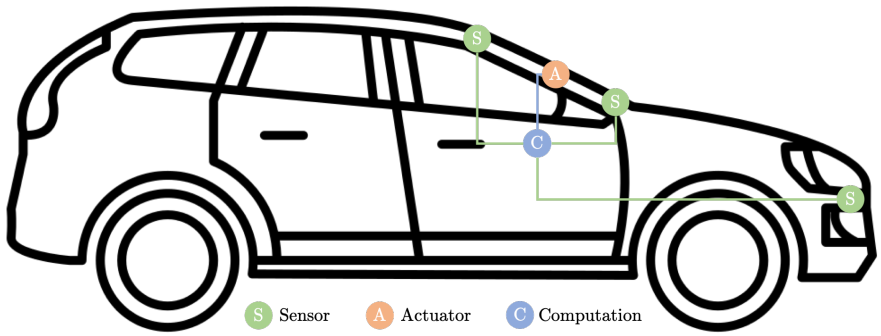

Fig. 8 Simplified E/E architecture of the PW application.

the signal that corresponds to the presence of objects ahead. Such a signal is used in conjunction with the signal from the camera for establishing whether these object could be large vehicles approaching from the opposite direction. A rain sensor attached to windscreen of the vehicle acquires the signal that corresponds to the quantity of water that is raining. This signal, together with the information on the large vehicles, is sent to a computational unit which decides if it is necessary to activate the windscreen wipers. Eventually, the actuators on the wipers will give the driver a clear view.

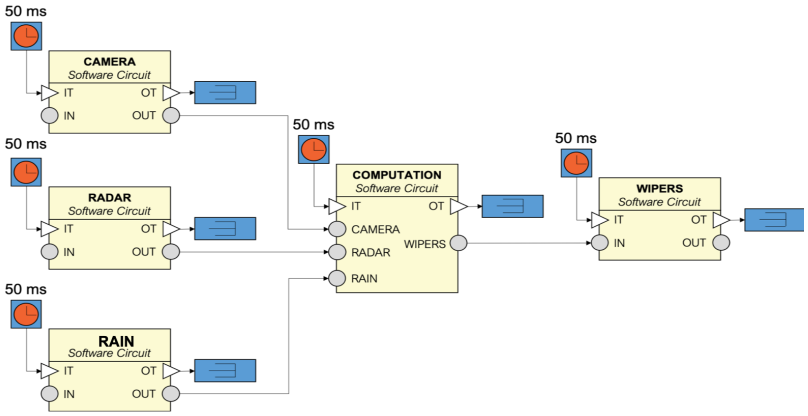

Fig. 9 RubusMM model representing the software architecture of the PW application.

Figure 9 shows the RubusMM model describing the software architecture of the PW application. The model consist of 5 software circuits where i) *Rain*

[14] `https://semcon.com/paw/`

models the software operating the rain sensor, ii) *Radar* and *Camera* model the software operating the front radar and camera, respectively, *Computation* models the software on the computational units and iv) *Wipers* models the software operating the actuator on the windscreen wipers. Figure 10 shows the Ecore serialisation of the model depicting the third configuration. In this configuration, the *System Vehicle* is composed of a *Target* element (modelling the tri-core microcontroller) and two *Application* elements (modelling the BBW and PW applications). According to the ISO 26262 [2] standard, the *BBW* and *PW Application* elements have different criticality levels (as shown in Figure 10). In particular, we assigned the criticality level *D* to the *BBW Application* element whereas the criticality level *A* to the *PW Application* element.

As applications with different criticality levels cannot be allocated to the same partition, the *Core1* element is partitioned into two *Partition* elements (*Partition1* and *Partition2*) and the *BBW* and *PW Application* elements are allocated as follows. For the *BBW Application*, the *WheelSpeed_FR*, *WheelSpeed_FL*, *Abs_FR*, *Abs_FL*, *Brake_FR*, *Brake_FL* software circuits are allocated to *Partition 1*; the (*WheelSpeed_RR*, *WheelSpeed_RL*, *Abs_RR*, *Abs_RL*, *Brake_RR*, *Brake_RL*) software circuits are allocated to *Core 2* while the remaining software circuits are allocated to *Core 3*. The *PW Application* is allocated to *Partition 2* a shown by the arrow. If we would have attempted to

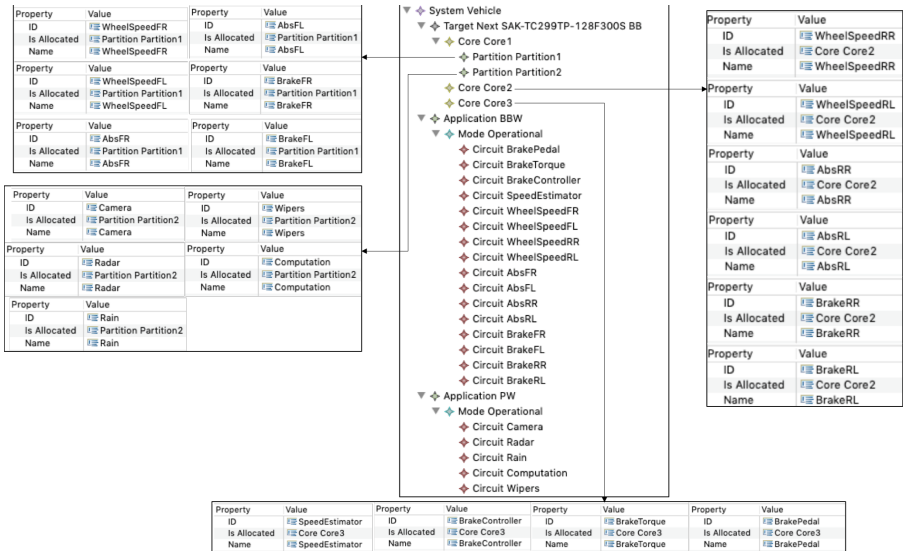

Fig. 10 Serialisation of the BBW and PW applications deployed to a tri-core microcontroller as multi-criticality system.

allocate the *PW Application* to *Partition 1*, the OCL contraint described in

Sec. 4 would have generated a validation error and prevented us from allocating applications with different criticality levels on the same partition.

In this section, we use-cased the BBW application in three different deployment configurations for describing the applicability of the evolved RubusMM. In the first configuration, we deployed the BBW application into a single-core microcontroller for demonstrating the suitability of the evolved RubusMM for describing legacy vehicular software systems. In the second configuration, the BBW application was deployed into a tri-core microcontroller to show how the evolved RubusMM could support the development of upcoming vehicular software systems on multi-core. Finally, in the third configuration we considered the PW application in addition to the BBW one for demonstrating the applicability of the evolved RubusMM in modelling multi-criticality vehicular software.

We model these two applications as a multi-core software system with different criticality levels and deploy them into the aforementioned tri-core microcontroller.

6 Discussion

In this work, we proposed an evolution of the Rubus Component Model (RCM), in terms of its metamodel RubusMM, for developing multi-criticality software systems to be deployed on single- and multi-core as well as on mixed architectures (C1). The proposed evolution ensures backward compatibility with legacy software systems modelled with RubusMM and does not cause any modification to the Rubus run-time layer and certified Rubus Kernel (C2). Both of the previous challenges can be seen as critical preconditions of any complex software in general: it is, at least economically, not affordable to throw away the past history of development efforts due to compatibility problems; if (parts of) the designed product undergoes certification processes, it is desirable to avoid expensive and long-lasting tool re-certification issues due to refinements. Since Rubus targets the vehicular domain, we devote the remainder of this section to reason about backward compatibility and avoidance of re-certification due to the needs conveyed by modelling multi-criticality systems and corresponding introduced features.

Functional safety is paramount in the vehicular domain. To earn acceptance, solutions like Rubus must provide certified run-time support, e.g., real-time operating system, along with modelling languages able to capture all the characteristics of vehicular applications. In this respect, we extended RubusMM (the Rubus metamodel formalising its component model) according to the virtualisation design option, as described in [42], which enables the reuse of the certified Rubus Kernel. The Rubus Kernel is certified according to the ISO 26262 standard ASIL D while Rubus ICE (i.e., the development environment supporting Rubus) is undergoing the same certification. On the one hand, the reuse of the Rubus Kernel makes unnecessary the explicit modelling of the memory since the mapping of data ports to physical memory is handled by the

Need	Feature(s)	Extension
Enhance separation of concerns	We introduced packages and removed structural containment relations	See Section 4.1
Extend timing modelling	We introduced *timingConstraint* reference for several elements in RubusMM. We introduced *Core* and *Partition* elements (and related references and attributes)	See Section 4.2
Extend hardware modelling concepts	We introduced *Target*, *TargetNew*, *TargetLegacy*, *Core*, *Partition*, *Allocator* elements. We introduced *isAllocated* reference and the *speed*, *type* and *numberOfCores* attributes	See Section 4.3
Extend software architectural modelling	We introduced *Application* and *Allocatable*, elements. We introduced the *criticalityLevel* attribute and the OCL constraint	See Section 4.4

Table 1 Summary of features in relation to needs

kernel itself. On the other hand, it makes the current definition of RubusMM not suited for approaches where explicit modelling of the memory is required. Moreover, despite the Rubus Kernel footprint is significantly small, the virtualised design option increases the overall footprint of the developed vehicular application since each core or partition can host a separate instance of the Rubus Kernel.

6.1 Evolution of RubusMM

In the following subsections we discuss our contributions in evolving RubusMM in terms of the extensions described in Section 4, and summarised against needs in Table 1, also from a cross-language perspective, in order to potentially support similar evolution processes of other languages.

Enhance separation of concerns
Separation of concerns is a key aspects of (modelling) languages, especially those supporting the component-based design pattern [37]. Before this extension, and due to the fact that the language was meant to only support one single type of target hardware platform, RubusMM was fairly monolithic, blending software, hardware, and analysis aspects together. This made the RubusMM definition not easy to understand and thereby hard to extend. As first step of evolving it, we enforced separation of concerns by separating metamodelling concepts in three packages. Before to opt for this separation, we attempted to achieve our evolution goals by re-arranging metaconcepts using the existing flat structure, but we could not succeed due to potential circular containments and multiple (invalid) specialisations of metaconcepts. Without enforcement of separation of concerns of the original RubusMM modelling con-

cepts, we could have not been able to provide the subsequent extensions nor open up for possible further extensions (see Section 7).

Extend timing modelling concepts
In Section 2, we pointed out high-precision timing analysis as one of the main reasons that made RubusMM appreciated in the vehicular domain and its evolution for multi-core and multi-criticality compelling. In this work, we have explicitly addressed the modelling of timing information, i.e., timing constraints, occurrences and events at several levels of the structural hierarchy by means of the reference *timingConstraint*. This information is required to perform the timing analysis of the software architectures of vehicular systems. Besides ensuring full compatibility with the existing model-based timing analysis provided by Rubus, this enables the use of the most recent timing analysis for multi-criticality software systems on multi-core [20] [21]. Without the evolution provided in this paper, the timing analysis for multi-criticality software systems on multi-core would not have been possible at different levels of the structural hierarchy due to the missing structural and timing information.

Extend hardware modelling concepts
The addition of modelling of multi-core hardware architectures was one of the core extensions being part of the RubusMM evolution. For providing that, we added the needed metaconcepts without modifying any existing hierarchical structures (as in the case of, e.g., *Target* and *Partition*) and introduced the new metaconcepts as leaves in the metamodel hierarchy (as in the case of, e.g., *Core* and *Partition*). Evolution of a language and the framework supporting it cannot overlook compatibility with previous versions of the language and the existing artefacts conforming to it. The evolution of RubusMM was not an exception, especially since Rubus is an industrial framework, and thereby the ability to keep compatibility with legacy models (and related artefacts) was pivotal. In the case of RubusMM, compatibility issues could arise concerning legacy models starring the old hardware modelling concept of single-core processors only. As part of the language evolution, we decided to go for an additive change, by keeping the old hardware modelling structure and adding the new concepts in a brand new set of metaconcepts. Both are valid (mutually exclusive) options for hardware modelling in the evolved RubusMM. While we were able to approach back-compatibility in this non-disruptive way due to the rather simplicity of the involved evolution, in other cases it may not be possible to do so. In such cases, a valid alternative could be a semi-automated patching mechanism for guiding the modeller in importing (and co-evolving) legacy models to conform to the evolved language.

Extend software modelling concepts
Regarding evolution of software modelling concepts, we introduced a competent support for modelling software applications with multi-criticality as well as the possibility to flexibly allocate them to specific hardware components. The previous RubusMM version, especially through the hindering structural

containment between *Target* and *Mode*, did not allow allocation of different software components to different cores according to their criticality level. We extended RubusMM with a set of allocation metaconcepts that is more flexible and provides multiple allocation strategies to the engineer. The introduced allocation metaconcepts, with embedded variability modelling capabilities, can also be leveraged by model transformations for automatically generating sets of potential allocation candidates, all in a single model with variability points discerning different candidates, as we have shown in a spin-off of this work [18]. It is important to note that, besides allowing any combination of software applications with specific criticality levels to be allocated to any core, we provide an automated validation mechanism to enforce criticality-safe grouping according to the ISO 26262 automotive safety standard. More specifically, we do not allow software applications with different criticality levels to be allocated together on the same partition.

6.2 Applicability and outlook

We showed the applicability of the extended RubusMM through the Brake-by-wire application. In particular, we modelled the functional software architecture of the BBW application and provided three different deployment configurations of varying complexity. In this respect, the functional software architecture in Figure 5 describes the BBW application coming from our industrial partner, Volvo Group. In the first configuration, we used the extended RubusMM for modelling a legacy RCM application. In the second configuration, we showed the applicability of the evolved RCM in modelling vehicular software applications on multi-core. It is worth noting that, despite several allocations of the BBW software components to the tri-core micro-controller could have been possible, we showed one of them (Figure 7) and discussed the realisation of a lockstep allocation configuration, only. In fact, our focus was to demonstrate the flexibility of the evolved RubusMM with respect to allocation strategies rather than discuss the generation of the whole space of possible allocations.

The RCM evolution described in this work has paved the way to the latest release of the Rubus Component Model, namely RCM V.5, currently used within the commercial integrated development environment Rubus ICE. What is more, the challenges and opportunities that arose during this research work have contributed to the definition of several collaborative research projects between academia and industry such as Automation in High-performance Cyber Physical Systems Development (A-CPS)[15] and Heterogeneous systems and software-hardware integration (HERO)[16].

[15] `http://www.es.mdh.se/projects/520-Automation_in_High_performance_Cyber_Physical_Systems_Development`

[16] `http://www.es.mdh.se/projects/511-HERO`

7 Conclusion and Future Work

In this work, we discussed the evolution of the Rubus Component Model (RCM), in terms of its metamodel RubusMM, for modelling software systems to be deployed on single-, multi-core as well as on mixed architectures while ensuring backward compatibility with legacy RCM software systems and compliance to the Rubus run-time layer and certified Rubus Kernel. The proposed evolution enables support for modelling multi-criticality of software components and validity checks for their allocation to cores. We considered an industrial vehicular application to assess the applicability of the evolved RCM, also in terms of backward compatibility.

One line of future work will investigate how to support the analysis and verification of vehicular embedded systems with multi-criticality levels on multi-core with respect to predictable timing behaviour. To this end, we will investigate how to adapt the certified Rubus Kernel for providing run-time support to these systems on multi-core.

In [19], we investigated how to provide automatic support for the allocation of software to hardware. In particular, we developed model transformations that, starting from a model with no modelled allocations and a set of timing constraints, produced a set of models featuring the set of different allocations of software to hardware optimised for satisfying the set of timing constraints. Despite the generation of models is transparent to the engineer and it can be guided through logic constraints, issues about scalability and performance may remain open. In this respect, another line of future investigation encompasses the study of a smarter generation process which could i) narrow and cluster the space of the generated models and ii) use different non functional properties for pruning the set of the generated models. In addition, we are planning to represent the set of generated models by means of the compact notation presented in [18]. Such a notation uses modelling with variability for representing a multitude of models with one single model with variability points.

Due to the terrific data-throughput induced by vehicular software functions, domain experts are investigating the use of heterogeneous platforms built of several different computational units (multi-core central processing units, graphic processing units, etc.) on a single board. In this respect, a future evolution of RCM will provide support for modelling and analysis software for these platforms.

Acknowledgments

The work in this paper has been supported by the Swedish Knowledge Foundation (KKS) through the MINEStrA, HERO and DPAC projects and by the Swedish Governmental Agency for Innovation Systems (VINNOVA) through the DESTINE project. We thank our industrial partners Arcticus Systems and Volvo Construction Equipment, Sweden.

References

1. Rubus ICE-Integrated Development Environment, `http://www.arcticus-systems.com`, Accessed: September, 2019.
2. ISO 26262-1:2011: Road Vehicles in Functional Safety, `http://www.iso.org`, Accessed: September, 2019.
3. The AUTOSAR Consortium, AUTOSAR Techincal Overview, Version 4.3., `http://autosar.org`, Accessed: September, 2019.
4. ARINC Specification 653P1-2, Avionics Application Software Standard Interface Part 1 Required Services, `http://www.arinc.com`, Accessed: September, 2019.
5. The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems, `https://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf`, Accessed: September, 2019.
6. Timing Augmented Description Language (TADL2) syntax, semantics, metamodel Ver. 2, Deliverable 11, Aug. 2012.
7. DO-178C Software considerations in airborne systems and equipment certification, `https://www.rtca.org/content/publications`, Accessed: September, 2019.
8. EAST-ADL Domain Model Specification, Deliverable D4.1.1., `http://www.atesst.org/home/liblocal/docs/ATESST2\_D4.1.1\_EAST-ADL2-Specification\_2010-06-02.pdf`, Accessed: September, 2019.
9. Why Brake-By-Wire Is Coming To Your Car, `https://www.popularmechanics.com/cars/car-technology/a22126727/brake-by-wire/`, Accessed: September, 2019.
10. SymTA/S for Migration from Single-core to Multi-core ECU-Software on Infineon Microcontrollers, Electronic Engineering Journal, December, 2010.
11. TADL: Timing Augmented Description Language, Version 2, Deliverable 6, October 2009. The TIMMO Consortium.
12. TIMMO-2-USE. https://itea3.org/project/timmo-2-use.html.
13. TIMMO Methodology, Version 2, Deliverable 7, Oct. 2009.
14. Bucaioni A., Cicchetti A., Ciccozzi F., Eramo R., Mubeen S., and Sjödin M. Exploring timing model extractions at EAST-ADL design-level using model transformations. In *12th International Conference on Information Technology : New Generations*.
15. Bucaioni A., Cicchetti A., Ciccozzi F., Eramo R., Mubeen S., and Sjödin M. Anticipating Implementation-Level Timing Analysis for Driving Design-Level Decisions in EAST-ADL. In *International Workshop on Modelling in Automotive Software Engineering*, 2015.
16. Bucaioni A., Cicchetti A., Ciccozzi F., Mubeen S., and Sjödin M. A Metamodel for the Rubus Component Model: Extensions for Timing and Model Transformation from EAST-ADL. *Journal of IEEE Access*, 5, 2016.
17. Bucaioni A., Cicchetti A., Ciccozzi F., Mubeen S., and Sjödin M. Technology-preserving transition from single-core to multi-core in modelling vehicular systems. In Springer, editor, *13th European Conference on Modelling Foundations and Applications.*, 2017.
18. Bucaioni A., Cicchetti A., Ciccozzi F., Mubeen S., Sjödin M., and Pierantonio A. Handling Uncertainty in Automatically Generated Implementation Models in the Automotive Domain. In *42nd Euromicro Conference series on Software Engineering and Advanced Applications*, 2016.
19. Bucaioni A., Addazi L., Cicchetti A., Ciccozzi F., Eramo R., Mubeen S., and Sjödin M. Moves: A model-driven methodology for vehicular embedded systems. *Journal of IEEE Access*, 6:6424–6445, 2018.
20. Burns A. and Davis R. Mixed Criticality Systems - A Review, eighth edition. Technical report, 2013.
21. Burns A. and Davis Robert I. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, 2017.
22. Cicchetti A., Ciccozzi F., Mazzini S., Puri S., Panunzio M., Zovi A., and Vardanega T. Chess: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2012.
23. Crespo A., Ripoll I., and Masmano M. Partitioned embedded architecture based on hypervisor: The xtratum approach. In *Proceedings of the 2010 European Dependable Computing Conference*, EDCC '10. IEEE Computer Society, 2010.

24. Esper A., and Nélis G., Nelissen V., and Tovar E. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. ACM, 2015.
25. Gamatié A., Le Beux S., É.Piel, Ben Atitallah R., Etien A., Marquet P., and J.Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems (TECS).*, 10(4):39, 2011.
26. Nicolas A., Posadas H., Peñil P., and Villar E. Automatic deployment of component-based embedded systems from UML/MARTE models using MCAPI. In *Conference on Design of Circuits and Integrated Circuits (DCIS)*, 2014.
27. Schätz B., Voss S., and Zverlov S. Automating design-space exploration: Optimal deployment of automotive sw-components in an iso26262 context. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.
28. Koong C., Yang C., and Chu W. Model-driven multi-core embedded software design. *Multicore Hardware-software Design and Verification Techniques*, 2011.
29. Dasari D., Nelis V., and Akesson B. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, 52, 2016.
30. Durisic D., Staron M., Tichy M., and Hansson J. Evolution of long-term industrial meta-models – an automotive case study of autosar. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014.
31. Edsger W Dijkstra. On the role of scientific thought. In *Selected writings on computing: a personal perspective*, pages 60–66. Springer, 1982.
32. Andrianarison E. and Piques J.D. Sysml for embedded automotive systems: a practical approach. In *Conference on Embedded Real Time Software and Systems*. IEEE, 2010.
33. Ciccozzi F., Seceleanu T., Corcoran D., and Scholle D. UML-Based Development of Embedded Real-Time Software on Multi-Core in Practice: Lessons Learned and Future Perspectives. *IEEE Access*, 4, 2016.
34. Herrera F., Posadas H., Peñil P., Villar E., Ferrero F., Valencia R., and Palermo G. The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems. *Journal of Systems Architecture*, 60, 2014.
35. Fernandez G., Abella J., Quiñones E., Rochange C., Vardanega T., and Cazorla F. J. Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39, 2014.
36. Agirre I., Azkarate-askasua M., Larrucea A., Pérez J., F.J. T., Vardanega, and F.J. Cazorla. Automotive safety concept definition for mixed-criticality integration on a cots multicore. In *Computer Safety, Reliability, and Security*, 2016.
37. Crnkovic I. and Larsson M. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.
38. ISO 11898-1. Road Vehicles Interchange of Digital Information Controller Area Network (CAN) for high-speed communication.
39. Pavon J., Tomas J., Bardout Y., and Hauw L-H. Corba for network and service management in the tina framework. *IEEE Communications Magazine*, 36(3):72–79, 1998.
40. Hänninen K., Mäki-Turja J., Sjödin M., Lindberg M., Lundbäck J., and Lundbäck K. The Rubus Component Model for Resource Constrained Real-Time Systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, 2008.
41. Xu K., Sierszecki K., and Angelov C. COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.
42. Becker M., Dasari D., Nélis V., Behnam M., Luís Miguel P., and Nolte T. Investigation on AUTOSAR-Compliant Solutions for Many-Core Architectures. In *18th Euromicro Conference on Digital System Design*, 2015.
43. Bucaioni M., Mubeen S., Lundbäck J., Gålnander M., Lundbäck K.-L., and Nolte T. Modeling and timing analysis of vehicle functions distributed over switched ethernet. In *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 8419–8424. IEEE, 2017.
44. Farzaneh M.H., Shafaei S., and Knoll A. Formally verifiable modeling of in-vehicle time-sensitive networks (tsn) based on logic programming. In *2016 IEEE Vehicular Networking Conference (VNC)*, pages 1–4. IEEE, 2016.

45. Feiertag N., Richter K., Nordlander J., and Jonsson J. A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics. In *Proceedings of the IEEE Real-Time System Symposium, Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2009.
46. Pop P., Scholle D., Hansson H., Widforss G., and Rosqvist M. The SafeCOP ECSEL Project: Safe Cooperating Cyber-Physical Systems Using Wireless Communication. In *Euromicro Conference on Digital System Design (DSD)*. IEEE, 2016.
47. Thorngren P. Keynote talk: Experiences from east-adl use. In *EAST-ADL Open Workshop, Gothenberg*, 2013.
48. Feiler P.H., Gluch D.P., and Hudak J.J. The architecture analysis & design language (AADL): An introduction. Technical report, 2006.
49. Charette R.N. This car runs on code. *IEEE Spectrum*, 46, 2009.
50. Barner S., Diewald A., Migge J., Syed A., Fohler G., Faugère M., and Pérez D. G. Dreams toolchain: Model-driven engineering of mixed-criticality systems. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2017.
51. Mubeen S., Lawson H., Lundbäck J., Gälnander M., and Lundbäck K. Provisioning of predictable embedded software in the vehicle industry: The rubus approach. In *IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, 2017.
52. Mubeen S., Mäki-Turja J., and Sjödin M. Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study. In *Computer Science and Information Systems, vol. 10, no. 1, pp 453-482, January 2013.*
53. Mubeen S., Mäki-Turja J., and Sjödin M. Communications-Oriented Development of Component- Based Vehicular Distributed Real-Time Embedded Systems. *Journal of Systems Architecture*, 60, 2014.
54. Mubeen S., Nolte T., Lundbäck J., Gålnander M., and Lundbäck K. Refining timing requirements in extended models of legacy vehicular embedded systems using early end-to-end timing analysis. In *13th International Conference on Information Technology : New Generations*, 2016.
55. Mubeen S., Nolte T., Sjödin M., Lundbäck J., and Lundbäck K. Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints. *Software & Systems Modeling*, 2019.
56. Séverine S., Vulgarakis A., Bures T., Carlson J., and Crnkovic I. A Component Model for Control-Intensive Distributed Embedded Systems. In *11th International Symposium on Component Based Software Engineering (CBSE)*. Springer, 2008.
57. Trujillo S., Crespo A., Alonso A., and Pérez J. MultiPARTES: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. *Microprocessors and Microsystems*, 38, 2014.
58. Vestal S. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium*. IEEE, 2007.
59. Voss S. and Schätz B. Deployment and scheduling synthesis for mixed-critical shared-memory applications. In *20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, 2013.
60. VanderLeest S.H. Arinc 653 hypervisor. In *29th Digital Avionics Systems Conference*. IEEE, 2010.
61. Gaska T., Werner B., and Flagg D. Applying virtualization to avionics systems the integration challenges. In *29th Digital Avionics Systems Conference*, 2010.
62. Kelter T., Falk H., Marwedel P., Chattopadhyay S., and Roychoudhury A. Static analysis of multi-core tdma resource arbitration delays. *Real-Time Systems*, 50, 2014.