Software and Systems Modeling (2023) 22:1085–1097
https://doi.org/10.1007/s10270-023-01107-3

EXPORT VOICE

From low-level programming to full-fledged industrial model-based development: the story of the Rubus Component Model

Alessio Bucaioni[1] · Federico Ciccozzi[1] · Amleto Di Salle[2] · Mikael Sjödin[1]

Published online: 19 May 2023

Abstract
Developing distributed real-time systems is a complex task that has historically entailed specialized handcraft. In this paper, we propose a retrospective on the (r)evolutionary changes that led to the transition from low-level programming to industrial full-fledged model-based development embodied by the Rubus Component Model and its tool-ecosystem. We focus on the needs, challenges, and solutions of a 15-year-long evolution journey of a software development approach that has gone from low-level and manual programming to a highly automated environment offering modeling, analysis, and development of vehicular software systems with multi-criticality for deployment on single- and multi-core platforms.

Keywords Component model · Model-based development · Vehicular embedded systems real–time systems

1 Introduction

Vehicles have gone from mechanics-intensive to software-intensive in the last two decades. With software replacing mechanical and hydraulic components, both manufacturers and customers started to envision an entirely new set of features that modern vehicles could provide (e.g., autonomous driving). Since then, demands on vehicular embedded systems have constantly increased, leading to a steady growth of vehicular software's complexity. Estimations are pointing out that current low-end vehicles are rapidly nearing 100 Electronic Control Units (ECUs) and 100 million lines of code,[1] which is expected to reach 300 million by 2030.[2]

Given these numbers, it became clear rather early to manufacturers that they needed efficient processes to cope with the size of these newly become software-intensive systems to maximize the throughput of software development in terms of cost and time. In addition, most vehicular embedded systems have extra-functional requirements that must be taken into serious account from the very early stages of development. More specifically, vehicular embedded systems are real-time systems, meaning that they must deliver their functionality within their timing deadlines. Consequently, timing requirements are crucial for these systems. It became soon evident that low-level programming, although accessible and non-disruptive to manufacturers' existing processes, was not going to be a sustainable solution for the engineering of software for vehicular embedded systems (or ECUs).

Instead, component-based software engineering (CBSE) first and model-driven engineering (MDE) later were identified as key methodologies to effectively deal with the increasing complexity of vehicular embedded software for complementary reasons. In 2003, the AUTOSAR partnership was officially presented at the VDI Conference Baden-Baden[3] with the goal of settling an open and standardized software architecture for vehicular ECUs. In 2008, the Rubus Component Model (RCM) [1] was formalized to add a component-based engineering layer for the definition,

Communicated by Bernhard Rumpe.

✉ Alessio Bucaioni
alessio.bucaioni@mdu.se

Federico Ciccozzi
federico.ciccozzi@mdu.se

Amleto Di Salle
amleto.disalle@unier.it

Mikael Sjödin
mikael.sjodin@mdu.se

1 Mälardalen University, Västerås, Sweden

2 European University of Rome, Rome, Italy

[1] https://spectrum.ieee.org/software-eating-car.

[2] https://www.vehicledynamicsinternational.com/.

[3] https://www.autosar.org/about/history.

early analysis, and implementation of software for vehicular ECUs.

In this paper, we present a retrospective on the (r)evoluti onary changes that led to the transition from low-level programming to industrial full-fledged model-based development embodied by the RCM and its underlying tools. Starting from the first formalization of the RCM in 2008, we unwind and discuss the set of needs, challenges, and solutions of a 15-year-long evolution of a development approach that has gone from low-level and manual programming to a highly automated environment offering modeling, analysis, and development of component-based vehicular software with multi-criticality for deployment on distributed single- and multi-core platforms. We achieve this by providing answers to the following research questions (RQs), which contribute to different and unique objectives of this study:

RQ1: *Which are the research publication trends on RCM?*
RQ2a: *Which were the requirements that drove the design and development of RCM?*
RQ2b: *Which were the strategies emerged to fulfill the requirements?*
RQ3a: *Which were the requirements that triggered the evolutions of RCM?*
RQ3b: *Which were the challenges to be tackled for evolving RCM?*

The remainder of this paper is structured as follows. Section 2 describes the Rubus concept as well as RCM and its accompanying tool. Section 3 answers RQ1 by presenting the publication trends over the years and venue types. Section 4 answers RQ2 by eliciting the core requirements that drove the design and development of RCM. In addition, for each of these requirements, it identifies the strategies emerged and used to fulfill the requirements. Section 5 answers RQ3 by eliciting the requirements that triggered the various evolutions of RCM. Additionally, Sect. 5 describes the challenges related to the evolutions of RCM. A discussion on lessons learnt is provided in Sect. 6. Section 7 concludes the paper with final remarks and planned future works.

2 The Rubus concept

Arcticus Systems[4] has been developing Rubus in collaboration with Mälardalen University (MDU) and other academic and industrial partners for over 25 years. The overarching goal of Rubus is to develop predictable and analyzable control functions in resource-constrained embedded systems by being aggressively resource-efficient. The Rubus concept is materialized in terms of the RCM [1], the Rubus Integrated Component development Environment (Rubus-ICE), and its real-time operating system Rubus Kernel Real-Time Operating System (RTOS). Rubus-ICE includes modeling and analysis tools, code generators, and a run-time infrastructure. The RTOS is certified to the highest ASIL level (ASIL-D) according to the ISO 26262 Road vehicles–Functional Safety Standard.[5] Several OEMs and Tier-1 companies in the vehicle industry (e.g., Volvo Construction Equipment, BAE Systems Hägglunds, Hoerbiger and Knorr-Bremse) in South Korea, China, Germany, France, USA, Sweden, use Rubus for the development of safety-critical real-time embedded software systems. Rubus-ICE provides integration of Simulink models in RCM, which facilitates analysis of the real-time behavior of the design including Simulink models.

Alternatives to RCM and Rubus-ICE are Vector's DaVinci, which differently from RCM only focuses on the implementation level (i.e., AUTOSAR), MentorGraphics' VNA, which only focuses on network modeling, Inchron's chronSIM, which focuses on timing analysis only. To summarize, Rubus-ICE is the only model-based environment providing modeling and analysis/simulation of hardware and software, including network, of software-circuit-based distributed systems, and it covers all abstraction levels of automotive systems development—vehicle, analysis, design, implementation.

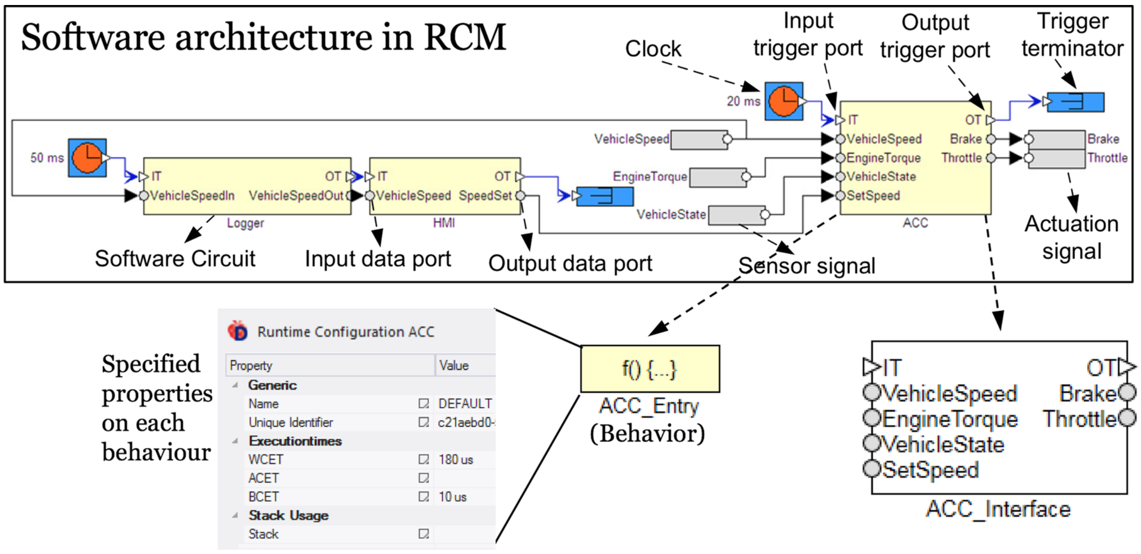
2.1 The Rubus Component Model

RCM [1] is a domain-specific language for developing predictable and analyzable control functions in resource-constrained, real-time embedded systems. An example of RCM model is depicted in Fig. 1. A model is described in block-like composite component diagram where control- and data-flows are explicitly and separately modeled via specific ports. Signals on ports trigger flows and software circuits to start their behavior. Models are mapped to C/C++ code specifically thought for real-time embedded systems. The first public definition of RCM dates back to 2008 when Hänninen et al. introduced a compact component model for the development of distributed embedded systems [1]. Since then, the language has evolved, and currently RCM features four packages, which are hardware [2], software, allocation [2] and timing [3].

The hardware data model provides the description of the hardware platform in terms of its processing units and network busses abstracting from low-level details such as memory hierarchies or the operating system, which are taken care from the accompanying RTOS and development environment. The hardware abstraction is described using `node`, `core`, `partition`, and `network` elements. Processing

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

[5] https://www.iso.org/standard/68383.html.



Runtime Configuration ACC

Property	Value
Generic	
Name	DEFAULT
Unique Identifier	c21aebd0-
Executiontimes	
WCET	180 us
ACET	
BCET	10 us
Stack Usage	
Stack	

Fig. 1 Software architecture example model in RCM

units are represented using `node` elements. Their internal structure is described using `core` and `partition` elements that represent physical cores and logical partitions of cores, respectively. `Network` elements encapsulate network specification details and protocol stack information of different network communication protocols such as the Time Sensitive Networking (TSN) [4] and different CAN protocols [5]. The allocation data model provides for the specification of software-to-hardware allocation.

The timing data model allows expressing of the software architecture's real-time requirements and properties. These are rooted in the timing augmented description language TADL2 [6]. RCM allows defining the timing constraints on events, either for single components or for the data propagation through an event chain. The event chain concept is a direct result of TADL2, where stimuli and a response event define event chain items. A detailed discussion of the representation of TADL2 timing constraints in RCM is provided in [7].

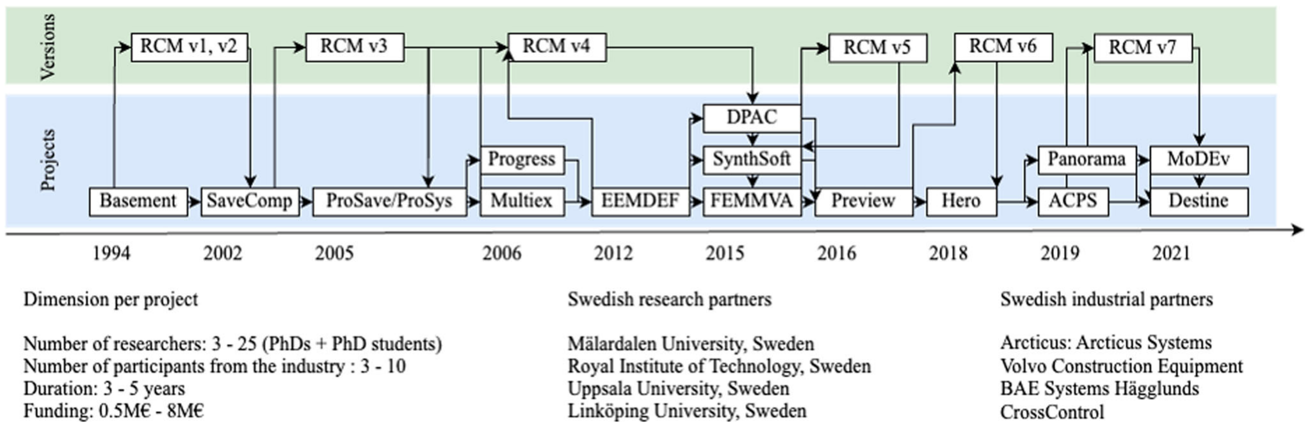
The software data model is used for describing software systems in terms of software functions and connections among them. In RCM, a `Software Circuit` (SWC) is the smallest unit of functionality, similar to a type or class that can be used multiple times. RCM distinguishes between data and control flows when considering interactions between SWCs. This makes it easier to define control specifications and interactions (typical of real-time embedded systems). Interfaces of SWCs contain `data` and `control ports`, with *data ports* representing data communication and `control ports` representing triggering conditions. In RCM, it is important to separate the functional code from the infrastructure that handles execution. This makes it easier to see explicit synchronization and data access at the modeling level and promotes reuse of SWCs in different contexts. Additionally, this principle ensures that a SWC does not need to be aware of how it connects to other components. SWCs have the following semantics:

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

This semantic can often be efficiently implemented without locks or barriers and allow development of very resource efficient systems. The need for locks and/or globally allocated memory can be determined, and automatically implemented, during the compilation stage

2.2 The Rubus analysis framework

RCM allows the specification of timing requirements and properties of the software architecture. These timing constraints can be used to associate real-time requirements with generated events and output triggers along a chain of SWCs. RCM includes all of the timing constraints from the AUTOSAR standard, and allows designers to specify real-time properties such as worst-case, best-case, and average-case execution times, as well as stack usage, for SWCs [7]. The scheduler takes these real-time constraints into account when creating a schedule. For event-triggered SWCs, response times are calculated and compared to the corresponding timing requirements. RCM also supports various types of timing analysis, including response-time analysis and end-to-end data-propagation delay analysis [3].



Dimension per project

Number of researchers: 3 - 25 (PhDs + PhD students)
Number of participants from the industry : 3 - 10
Duration: 3 - 5 years
Funding: 0.5M€ - 8M€

Swedish research partners

Mälardalen University, Sweden
Royal Institute of Technology, Sweden
Uppsala University, Sweden
Linköping University, Sweden

Swedish industrial partners

Arcticus: Arcticus Systems
Volvo Construction Equipment
BAE Systems Hägglunds
CrossControl

Fig. 2 Rubus research collaborations

2.3 The Rubus run-time framework

In RCM, SWCs are mapped to run-time entities called tasks. Each external event trigger defines a task, and the SWCs that are connected through the chain of triggered SWCs are allocated to that task. SWC chains that are triggered by periodic clocks are assigned to an automatically generated static schedule that satisfies precedence order and timing requirements. The inter-SWC communication within these chains is optimized to use the most efficient means of communication for each link. The mapping of SWCs to tasks and the generation of the schedule can be optimized to minimize response times or memory usage. The run-time system executes all tasks on a shared stack, eliminating the need for static allocation of stack memory to each task, which results in a small runtime footprint for the software architecture.

2.4 The Rubus multi-core hypervisor

The Rubus multi-core hypervisor uses resource isolation techniques to manage shared resources within and among cores [8]. These techniques are commonly used in various fields, e.g., avionics [9, 10], to partition system resources in time and space. The Rubus hypervisor implements the time division multiple access (TDMA) [11] protocol to arbitrate the shared system bus among cores and uses memory partitioning techniques to isolate shared memories such as L3 cache and RAM [12]. These isolation techniques allow cores and partitions within cores to be virtually independent from one another, meaning that each partition can be treated like a single-core processor with dedicated resources, although with reduced capacity based on the size of the shared memory and system bus bandwidth allocated to it. One benefit of this model is that the overall system becomes simpler to model, as there is no need to explicitly model shared resources such as memories and I/O in the software architecture.

2.5 The Rubus collaborative research

Rubus has been developed by Arcticus Systems in collaboration with several academic and industrial partners. Figure 2 shows some of these research collaborations in the form of collaborative research projects. The first collaboration dates back to 1994 when the Basement collaborative research project with several Swedish companies and universities was started. Basement introduced a distributed real-time architecture for the automotive industry that was the foundation for RCM, which was presented to the research community later in 2008 [13]. Ever since then, RCM and the whole Rubus approach have undergone through three major improvements that happened within more than ten collaborative national and European research projects.

3 RQ1: publication trends

To answer RQ1, we conducted a lightweight literature review to identify the publication trends of this 20-year-long academia-industry collaboration over time. We collected the data through the web search engine Google Scholar[6] using the following search string:

"rubus AND component AND model"

We ran a full-text search. The RCM was formally defined in 2008 by Hänninen et al. [1], but the first hints on RCM date back to 2003. For this reason, we carried out our search from 2003 to date (December 15, 2022).

The initial search gave 265 results. After removing impurities and duplicates, we applied the following inclusion and exclusion criteria:

- **I1**: Studies written in English
- **I2**: Studies exploiting the Rubus Component Model

[6] https://scholar.google.com.

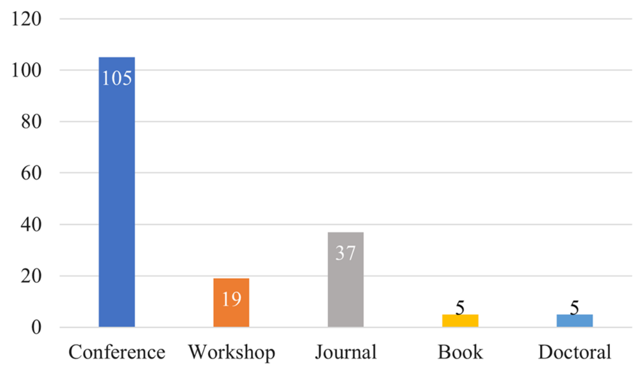


Fig. 3 Targeted venue types

- **E1**: Studies with full-text not available
- **E2**: Studies in the form of tutorial papers, editorials, reports, etc., since they do not carry the type of information that we seek.

Eventually, we obtained a set of 171 primary studies. We provide a publicly available replication package[7] containing the raw data from the initial search and the list of primary studies.

Figure 3 shows the venue types of the studies over the years. The high number of conference articles (105 out of 171) and journal articles (37 out of 171) suggests that RCM has gained interest as a research topic despite being mainly developed in industrial settings. Conversely, the low number of workshop papers (19 out of 171) indicates that researchers tend to focus on journal and conference publications, which are more likely to carry mature valuable scientific results. Finally, the tight and fruitful collaboration between Arcticus and MDU on RCM-related research has produced 5 successful PhDs.

Figure 4 depicts the distribution of publications on RCM over the years. From 2003 to 2008, seventeen articles were published. The main focus was on the development of the commercial product around RCM as well as on its limitations [14]. For example, the Rubus Design Language was implemented in-house with a custom-built modeling language. Additionally, component-level program debugging was not supported. The generation and synchronization of the schedulers could not be explicitly defined but were instead hidden in the tools. From 2008 onward, the number of studies applying or using RCM increased gradually and reached its peak in 2017, but maintaining a steady >10 publications per year. A few publications were produced in 2009 and 2010, the first two years after RCM was formalized, and researchers mainly employed it as related work. However, starting in 2011, the increasing scientific interest in the RCM became more evident. For instance, Mubeen et al. [15] extended the RCM to model different nodes within an embedded system. Finally, from 2013, studies published in journal venues, including the Journal of Systems and Software (JSS), Software and Systems Modeling (SoSyM), and the Journal of Systems architecture (JSA), increased.

In the following, we give some insights into the primary studies that can be considered milestones in the history of Rubus.

In 2008 Hänninen et al. [1] defined the new component model RubusCM v3 to enable the creation of embedded control systems with a combination of hard, soft, and non-real-time requirements. In particular, the authors defined the essential architectural elements such as software circuits (SWCs), input and output ports, assemblies, and composites as building blocks to develop the system through different views, emphasizing different aspects. Additionally, a developer could provide real-time execution properties that allowed triggering executions, execution requirements, and temporal characteristics of SWCs through actions. Each SWC has a run-time profile that details the execution time and memory use across several platforms to enable real-time analysis.

In 2013 Mubeen et al. [3] extended Rubus-ICE to provide a complete timing behavior prediction of multi-rate real-time systems. The authors implemented two plugins for two well-known timing analysis techniques, i.e., the end-to-end response-time named end-to-end delay analysis (E2EDA) and delay analysis named Holistic Response-Time Analysis (HRTA).

In 2014, Bucaioni et al. formalized the first version of the RCM metamodel focusing on the elements describing the software architecture. [16]. In particular, the metamodel allows abstracting hardware and operating system elements [17]. Moreover, the metamodel defines metaclasses for the data and the control flows. In 2020, Bucaioni et al. [2] extended the first version by including elements to manage multi-criticality for deployment on multi-core platforms of vehicular software systems.

In 2018, Bucaioni et al. [18] defined a methodology for vehicular embedded systems named MoVES based on model-driven techniques. The aim is to support a system's semi-automatic and guided development following timing properties. In particular, MoVES leverages the collaboration between EAST-ADL and RCM through model transformations to allow timing-aware design and model-based timing analysis of a system.

In 2022, Bucaioni et al. [19] published a study on the interoperability between the two architectural languages, AMALTHEA and RCM, for the design and timing analysis of automotive software systems.

[7] https://github.com/amletodisalle/SosyM-expert-voice-RCM.

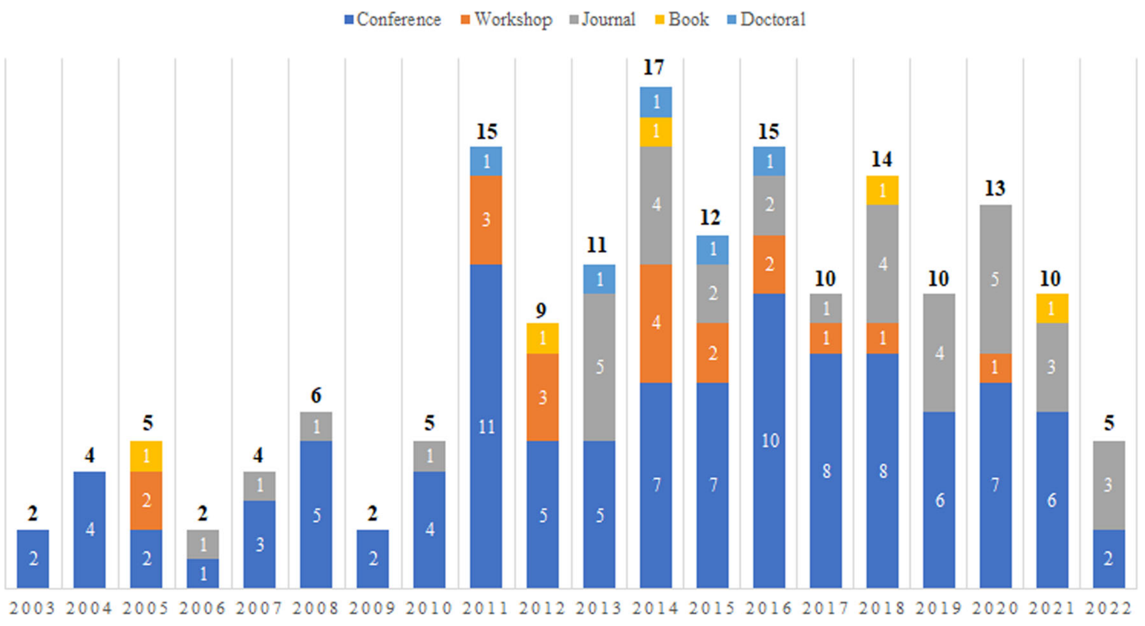


Fig. 4 Publication trend over time

4 RQ2: requirements for the development of RCM and strategies to fulfill them

The idea for the Rubus concept originated in the late 1990 s in the Basement architecture. At that time, the computing power of commonly produced electronic control units was very limited compared to the present day, and the use of 16-bit architecture and operating systems were just starting to become popular. Automotive manufacturers primarily focused on per-unit cost, and development time and time-to-market for software were not yet a significant consideration. Nevertheless, developers were still struggling with issues related to software quality and integration.

At its dawn, the core emphasis of the Rubus concept was on resource efficiency and predictability. Over time, many other aspects of software development have been taken into account, and, currently, the Rubus concept takes a wide range of requirements into consideration. The primary technical requirements (TR) that the Rubus concept aimed at fulfilling are:

TR1: Resource constrained The implemented system should conserve use of memory. Especially expensive and power-hungry RAM should be kept to a minimum.

TR2: Testable and debuggable Components, subsystems, and the complete system should be amenable for testing both on the target and the development platforms.

TR3: Analyzable The design needs to be automatically analyzable both to derive static properties like memory usage and timing delays, and also to allow optimized code generation when building tasks and task-chains during synthesis.

TR4: Portable Components and systems designs should be portable to new platforms with limited manual effort. The portability of Rubus designs is dependent on the availability of compatible compilers and RubusOS for the new platform. Also, architectural features like word-length and compiler features like the size of native types may need manual effort for portation.

TR5: Interoperable Designs should be able to interoperate with selected technologies to allow system integration when parts of the system are developed using other approaches than the Rubus concept. Technologies that are currently supported include execution of Autosar components, development of components using Simulink, architecture generation from EAST-ADL specifications, and integration of legacy ECUs communicating in various CAN protocols.

TR6: Validatable and certifiable The system should be able to validate against top-level requirements and, different parts of the system should be able to be certified to different safety levels. This means that both subsystems and the whole system should be executable and observable to give sufficient evidence that all needed properties are satisfied.

In addition, development requirements (DR) addressed by the Rubus concept include:

DR1: Reusable Components and subsystems should be reusable across product variants and versions. One of the industrial users maintains a product line of over 100 variants of vehicles with many shared physical components. The associated software components need to be able to be integrated into various combinations corresponding to the variations of the products.

DR2: Maintainable A product in the vehicular industry has a planned life span for several decades and thus the software needs to be able to evolve with the maintenance of the product. The tooling environment needs to allow maintenance of software components and keep track of versions and variants and their usage in different products.
DR3: Understandable The component technology needs to be easy to learn and understand for many types of stakeholders. A component developer needs to understand the requirements of a component (usually specified in terms of input and output of data, and limits on the resource consumption of the component), and a system architect needs to be able to view the system in a hierarchical way and abstract away from low-level details, test engineers need to understand the external interfaces and how to insert stimulus and interpret output to asses test conformance. Etc.
DR4: Introducible The component technology needs to fit into the existing development processes and tools. The use of existing programming languages and system development tools needs to be supported rather than hindered.

Some of the strategies derived to address DRs are:
Run-to-completion semantics The execution semantics described in Sect. 2.1 permit a "run-to-completion" approach where a component continues without interruption once it starts, as long as all necessary inputs are present. This simplicity makes it easy to combine components and achieve efficient performance in a minimal runtime environment. Additionally, the code generator can create individual memory spaces for input and output buffers, allowing for lock-free data manipulation. Furthermore, tasks can be preempted and share the same stack, negating the need for allocating memory for specific tasks.

Furthermore, the run-to-completion semantics allow components (and aggregates of components) to be executed in isolation for, e.g., testing and debugging. Unit testing of components can easily be automated and most tests can be done in the development environment without the need to deploy the unit under test to the target hardware.
Source code components The internal function of components are represented in C or some dialect of C such as System C or C++. Basically, any language that can be compiled to stand-alone machine-code for the target platform can be used as long as the components' entry- and exit-points conform to the C calling conventions. The use of source-code components allows components to be generated from other tools such as Matlab/Simulink.

Although there is a risk that component developers may violate the established model by including code that creates unpredictable dependencies on the environment (such as using global variables), this approach is still chosen. The Rubus concept does not provide any means to automatically verify that a component's implementation adheres to the defined semantics.

Static configuration The strong requirements for resource efficiency, predictability, testability, and ultimately certifiability to the highest safety levels have led to the decision to only allow statically configured systems. In Rubus, all tasks must be defined off-line and created by the code-generation tool. This results in a very predictable system with very limited dynamic behavior. The lack of dynamicity makes system-level testing much easier than it is for highly dynamic systems. Certain types of run-time errors are completely eliminated by the static configuration approach. These include failure to create tasks and failure to allocate memory. To provide some level of dynamicity, the Rubus concept allows a system to switch between a set of predefined modes. Each mode defines a set of tasks that are executed in that mode. For an engine-control node, some of the typical modes would be engine start mode, engine run mode, and engine shutdown mode. The Rubus concept guarantees that mode-shifts are done within a predictable time and that all tasks and memory buffers that are needed for the new mode are preallocated in memory.

The need for resource efficiency, predictability, testability, and certifiability to high safety levels has led to the decision to use only statically configured systems in Rubus. All tasks must be defined offline and generated by the code-generation tool, resulting in a highly predictable system with minimal dynamic behavior. This lack of dynamicity makes it easier to test the system. Additionally, certain types of run-time errors are eliminated through the static configuration approach, such as failures to create tasks and allocate memory.

To provide some level of dynamic behavior, Rubus allows for switching between predefined modes, each of which includes a set of tasks to be executed. For example, in an engine control node, there may be modes for engine start, engine run, and engine shutdown. The Rubus concept guarantees that mode shifts are completed within a predictable time frame and that all necessary tasks and memory buffers are preallocated in memory.
Hybrid task-scheduling The Rubus concept employs three levels of scheduling, each utilizing different scheduling mechanisms. At the highest level, tasks are executed by hardware interrupts, which are scheduled by the hardware and may provide priority or first-in-first-out (FIFO) scheduling. This level is typically reserved for lightweight device drivers that have minimal execution time. The second level of priority is a static cyclic schedule where components triggered by periodic clocks are placed by the offline code generator. This level is typically used for functions in higher safety levels and control loops with tight timing requirements. The lowest level of scheduling is for dynamically scheduled tasks triggered by asynchronous events or with very long periods. These tasks are scheduled using fixed-priority scheduling, allowing for analysis of worst-case response time.

As long as the amount of asynchronous event triggering interrupts and/or dynamically scheduled tasks is bound, each of the scheduling levels can be used for hard real-time processing with deterministic upper bounds on their response-times.

5 RQ3: requirements for the evolution of RCM and related challenges

Since the late 2010 s, the automotive industry witnessed a significant shift in how software was designed, developed, and used in vehicles. One of the most notable innovations was the increasing use of multi-core processors as a way of handling the growing amount of data and real-time processes required by advanced driver assistance systems, infotainment, etc. Such a shift made languages and solutions specifically tailored to single-core hardly reusable when dealing with challenges specific to multi-core, such as core inter-dependency and allocation of parallel software to hardware. Hence, evolving RCM was not only desirable but rather needed to maintain its place in the automotive domain. Hereafter, we list and describe the main requirements that drove this evolution (ER).

ER1: Software legacy In the automotive domain, up to 90% of the vehicle software is reused from previous releases [20]. Hence, the evolution of RCM needed to ensure backward compatibility with legacy software systems modeled with older RCM versions and not to cause any modification to the Rubus run-time framework and certified kernel.

ER2: Supplier legacy Original Equipment Manufacturers (OEMs) have decennial contracts with Tier-N suppliers including modeling languages, environments, etc. Changes to assets such as the evolution of RCM should not affect these contracts.

ER3: Certified run-time support Model-based solutions targeting safety-critical sectors such as the automotive domain rely on certified development environments and RTOS [21]. Typically, certification processes add a development cost overhead between 25 and 100% of the development costs [22]. Hence, evolutions of RCM needed to be compliant with the certified development environment and RTOS.

ER4: Interoperability Typically, automotive software is developed through a constellation of languages, tools and frameworks. The RCM evolution needed to disclose the opportunity to easily integrate the Rubus analysis and run-time frameworks within a typical automotive development chain.

Besides the requirements mentioned above, we evolved RCM by providing means for modeling multi-core and multi-criticality vehicular systems. Multi-criticality refers to vehicular software composed of various functions with varying levels of criticality. Some of these functions, e.g., deployment of airbags, are considered safety-critical and have strict real-time requirements. Other functions, e.g., speedometer, have real-time requirements although not being safety-critical. Nevertheless, other functions, e.g., infotainment systems, are not considered critical at all. Vehicular software needs to handle multiple levels of criticality and allocate it to the appropriate hardware reliably and cost-effectively. In doing so, we encountered several challenges that we discuss hereafter.

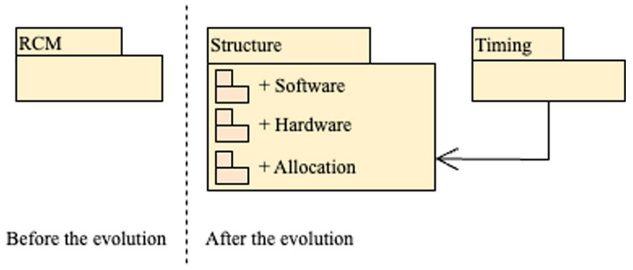


Fig. 5 Simplified representation of the RCM packages before and after the evolution

EC1: Separation of concerns Separation of concerns aims at improving the ability to reason about and specify different aspects of the software. This concept, first advocated by Dijkstra [23], has been a key element in developing modern modeling languages, especially for those supporting the component-based design pattern [24]. However, in the case of languages with a strong industrial focus, such as RCM, the implementation of separation of concerns has been neglected in favor of more practical considerations. Before the evolution, RCM was a fairly monolithic and blended element for modeling, e.g., software, hardware, analysis, and aspects together. Here the challenge was to introduce separation of concerns while ensuring backward compatibility with legacy code.

We dealt with this challenge by introducing packages and removing structural containment occurring between hardware and software elements (more details on this come in the following paragraphs). Prior to the evolution, RCM did not feature any package (see Fig. 5). Although not optimal, this was a pragmatic choice driven by the language only meant to support one single type of target hardware platform. We introduced packages for introducing and ensuring the separation of concerns, improving the understandability of the language, and enhancing its extensibility.

EC2: Extending hardware modeling capabilities – Before the evolution, RCM only allowed the modeling of single-core CPU-based hardware. We needed to evolve RCM to allow the modeling of multi-core, even heterogeneous, hardware architectures. This included general information about the hardware (e.g., partitions, number of cores) and relationships among hardware elements. Here the challenge was two-folded. On the one hand, the evolution should have intro-

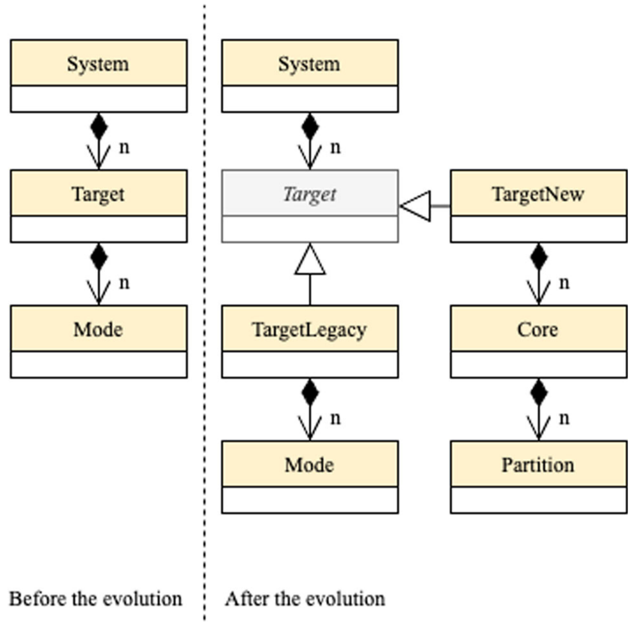


Fig. 6 Simplified representation of the elements in the hardware package before and after the evolution

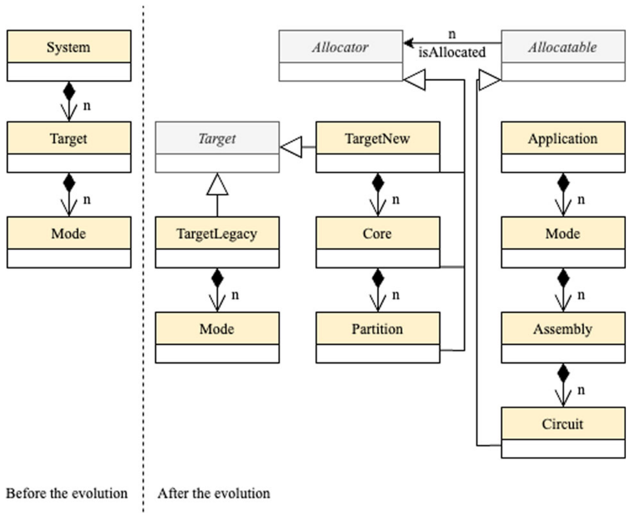


Fig. 7 Simplified representation of the elements providing for the allocation of software to hardware before and after the evolution

duced the minimum number of hardware elements crucial for the software to hardware allocation and for modeling and extracting the timing information to support the timing analysis engines. On the other hand, the evolution should not have invalidated previous hierarchies of hardware elements.

First, we evolved RCM with elements for modeling cores and partitions as depicted in Fig. 6. In its previous definition, Target contained Mode elements, whereas Mode acted as a container for the software application. However, such a containment relation was too restrictive for modeling multi-criticality software on multi-core as it prescribed that software elements (represented by Mode elements) were structurally contained by hardware elements (represented by Target elements). This pragmatic choice suited the modeling of single-core CPU-based hardware since the allocation of software to hardware cannot variably split across cores.

Nevertheless, multi-criticality software on multi-core CPU-based hardware demanded more flexibility since the allocation of software to hardware is a variability point that can be affected by the criticality levels, and a structural containment can hardly represent that. To provide such flexibility while ensuring backward compatibility with legacy RCM models, we modified the existing hierarchy adding the metaclasses TargetLegacy and TargetNew, both inheriting from the abstract metaclass Target. TargetLegacy represents a legacy single-core processor. Hence, we did not modify its structural containment with Mode elements. TargetNew can represent either a single- or multi-core processor. TargetNew contains one or more core elements, which in turn can contain partition elements. The software to hardware allocation information for the elements in the new hierarchy is entrusted by a many-to-many association between allocatable and allocator elements (more details on this come in the following paragraphs).

EC3: Allocation of software to hardware As discussed above, the allocation of multi-criticality software to multi-core CPU-based hardware is a variability point that demands greater flexibility than the allocation of software to single-core CPU-based hardware. We addressed this challenge by acting in two directions. First, we introduced new modeling elements: the metaclasses application, allocatable and allocator (along with their attributes) and the isAllocated association between allocatable and allocator (see Fig. 7). In addition, we enriched RCM with a structural constraint specified through the object constraint language (OCL) as an invariant of partition elements to prevent software with different criticality levels from being allocated together on the same partition.

Application elements model pieces of software implementing a dedicated functionality. They have an attribute criticalityLevel that specifies the level of safety criticality according to the ISO 26262 automotive safety standard. The standard has four levels of criticality (A to D), where A is the lowest criticality level, whereas D is the highest criticality level. To prevent software applications with different criticality levels from being allocated together on the same partition, we evolved RCM with the structural OCL constraint specified in Listing 1.

```
FOR each application::Application allocated to
    partition::Partition
    criticality.add(application.
        criticalityLevel);

IF criticality.size()<=1 THEN
    True
ELSE
```

```
    False
}
```

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

EC4: Run-time support Functional safety is paramount in the vehicular domain. To earn acceptance, RCM needed to provide modeling capabilities for capturing all the characteristics of vehicular applications along with certified run-time support. Here, the challenge was to identify a design option that would enable the reuse of the certified Rubus Kernel. To address this challenge, we used resource-isolation techniques for the arbitration of intra- and inter-core shared resources. Such techniques are used in many sectors (e.g., avionics [9, 10, 25]) to ease and partition the system resources in time and space. The Rubus hypervisor implements the time division multiple access (TDMA) protocol to arbitrate the shared system bus among the cores [11]. Similarly, memory partitioning techniques are used to isolate the shared memories, including L3 cache and RAM among the cores [12]. Isolation techniques enable cores and partitions within those cores to become virtually independent from other cores or partitions. Simply put, each partition can be seen as a single-core processor equivalent with dedicated system resources. The single-core processor equivalent model simplifies the overall system model as there is no need to model memories explicitly, I/Os, and other shared resources in the software architecture. This is reflected in the evolution of RCM, which did not entail any modeling elements for these concerns. One drawback is that the evolved RCM is not suitable for approaches where explicit modeling of the memory is required. Moreover, the virtualized design option increases the overall footprint of the developed vehicular application since each core or partition can host a separate instance of the kernel. In our case, this was not a primary concern as the footprint of the Rubus Kernel is reasonably low.

6 Discussion

Although introduced in the 1990s, the Rubus concept has succeeded in keeping its place as a favorite development approach for market leaders in the automotive, aerospace, and defense sectors throughout these many years. This success can be attributed to a combination of well-tailored initial development requirements (Sect. 4) and thoughtful evolutions over time (Sect. 5). Rubus' ability to meet the demands of end-users while staying at the forefront of technological advancements in multiple IT fields has been a crucial element in its success over time. From component-based and model-driven software architecture, design, and engineering, to safety-critical hard real-time scheduling, the concept has been able to remain relevant by incorporating the latest scientific advancements. The thoughtful evolution of the language improved the decoupling of the modeling concepts from the Rubus ICE. In turn, this allowed for steady progress that did not disrupt the, e.g., tools' compliance with the ISO-26262 standard for functional safety in road vehicles.

From a technical perspective, the journey of defining and evolving RCM has been very enlightening. For example, the diversification of use and details contained in source code components and model components has taught us valuable lessons. By maintaining the existing source code components "as-is" before evolving RCM, we were able to capitalize on existing compilers and third-party tools to generate source code components. This allowed us to purposefully characterize functionality at a different level of detail than model components, which is not only useful but in most cases necessary for device drivers. Additionally, this mechanism can also be used to efficiently include legacy functions that do not conform to RCM metamodel, as well as to include COTS components, provided they are certified. While this flexibility in the definition of source code components is a vital feature, it must be handled with care by the engineer. For example, in early usage, engineers tended to rely on global variables for data structures that were too large to transit through ports, which can lead to problems if not handled properly.

In terms of scheduling, the Rubus concept enforces a strict synchronization model that is based on run-to-completion and read-execute-write semantics. This model has proven to be highly attractive for the embedded systems community, and as a result, major standard safety-critical architectures such as Autosar have adopted it. However, it is important to note that while more flexible models, such as those that allow for synchronization within components or flexible synchronization patterns in component connectors, may seem appealing, they tend to result in systems that are less predictable, less analyzable, and less optimizable. To maintain the aforementioned model, we evolved Rubus according to the virtualization design option. This approach involves using a single-core processor equivalent model for multi-core systems. This simplified the overall system model as there is no need to explicitly model memories, I/Os and other shared resources in the software architecture. However, it should be noted that this approach made the evolved RCM less suitable for situations where explicit memory modeling is required. Additionally, the virtualization design option increases the overall footprint of the developed application as each core or partition can host a separate instance of the kernel.

The practical deployment of software in industrial settings required a high level of verifiability, including analyzability, debuggability, and testability [26]. To achieve this, automation and the ability to perform these activities within the development environment were crucial [27]. The evolution of Rubus, specifically the implementation of a metamodel definition, facilitated the definition of seamless and roundtrip verification processes [28]. This is exemplified by the integration of Rubus with other industrial automotive languages such as EAST-ADL and AMALTHEA, as described in Sect. 2.

Standards are of crucial importance in software engineering, especially in application domains with specific criticality issues. Standards, both de-jure and de-facto, are hard to establish and maintain. RCM and the tooling around it can be considered a de-facto standard in certain safety-critical real-time domains in defence and construction vehicles. Certification processes of both the design tool and the runtime (RTOS) are very expensive activities, both in terms of time and cost. It is then crucial, in similar settings, to build such an ecosystem in a component-based manner, where the various parts can be certified by themselves and the interaction between parts strictly follow a formalized specification. Doing so, changes to one part do not trigger the need to re-certify the entire ecosystem, but rather focus on the changed portion.

7 Outlook

The use of component-based and model-driven software engineering methodologies has proven key to effectively dealing with the great complexity of vehicular embedded software and the challenges posed by its development. Our work in improving the development of such systems through Rubus continues.

Some of the ongoing works focus on addressing the challenges related to providing Rubus ICE with agile features as well as support for blended modeling [29] notations such as textual and graphical. Another research line focuses on providing Rubus with more effective ways to analyze and optimize software product lines. Eventually, we are also extending the language with support for new network protocols such as Ethernet or wireless protocols.

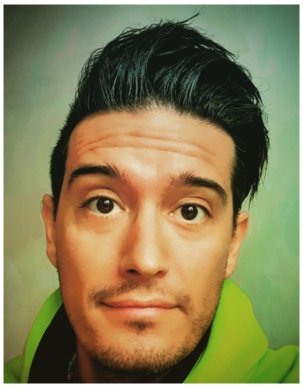
Funding Open access funding provided by Mälardalen University.

References

1. Hänninen, K., Mäki-Turja, J., Nolin, M., Lindberg, M., Lundbäck, J., Lundbäck, K.: The Rubus component model for resource constrained real-time systems. In: IEEE third international symposium on industrial embedded systems, SIES 2008, Montpellier / La Grande Motte, France, June 11-13, IEEE; 2008. pp. 177–183 (2008)
2. Bucaioni, A., Mubeen, S., Ciccozzi, F., Cicchetti, A., Sjödin, M.: Modelling multi-criticality vehicular software systems: evolution of an industrial component model. Softw. Syst. Model. **19**(5), 1283–1302 (2020). https://doi.org/10.1007/s10270-020-00795-5
3. Mubeen, S., Mäki-Turja, J., Sjödin, M.: Support for end-to-end response-time and delay analysis in the industrial tool suite: issues, experiences and a case study. Comp. Sci. Inf. Syst. **10**(1), 453–482 (2013). https://doi.org/10.2298/CSIS120614011M
4. Mubeen, S., Ashjaei, M., Sjödin, M.: Holistic Modeling of Time Sensitive Networking in Component-Based Vehicular Embedded Systems. In: Staron, M., Capilla, R., Skavhaug, A. (eds.) 45th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2019, Kallithea-Chalkidiki, Greece, August 28–30, 2019, pp. 131–139. IEEE (2019)
5. Mubeen, S., Mäki-Turja, J., Sjödin, M.: Communications-oriented development of component-based vehicular distributed real-time embedded systems. J. Syst. Archit. **60**(2), 207–220 (2014). https://doi.org/10.1016/j.sysarc.2013.10.008
6. :Timing Augmented Description Language (TADL2) syntax, semantics, metamodel Ver. 2, Deliverable 11, Aug (2012)
7. Mubeen, S., Nolte, T., Sjödin, M., Lundbäck, J., Lundbäck, K.: Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints. Softw. Syst. Model. **18**(1), 39–69 (2019). https://doi.org/10.1007/s10270-017-0579-8
8. Fernandez, G., Abella, J., Quiñones, E., Rochange, C., Vardanega, T., Cazorla, F.J.: Contention in multicore hardware shared resources: understanding of the state of the Art. In: Falk H, editor. 14th international workshop on worst-case execution time analysis, WCET 2014, July 8, 2014, Ulm, Germany. vol. 39 of OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik; 2014. p. 31–42
9. VanderLeest, S.H.: ARINC 653 hypervisor. In: 29th digital avionics systems conference; p. 5.E.2–1–5.E.2–20 (2010)
10. Gaska, T., Werner, B., Flagg, D.: Applying virtualization to avionics systems - The integration challenges. In: 29th Digital avionics systems conference, pp. 5.E.1–1–5.E.1–19 (2010)
11. Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., Roychoudhury, A.: Static analysis of multi-core TDMA resource arbitration delays. Real Time Syst. **50**(2), 185–229 (2014). https://doi.org/10.1007/s11241-013-9189-x
12. Dasari, D., Nélis, V., Akesson, B.: A framework for memory contention analysis in multi-core platforms. Real Time Syst. **52**(3), 272–322 (2016). https://doi.org/10.1007/s11241-015-9229-9
13. Hansson, H., Lawson, H., Stromberg, M., Larsson, S.: BASEMENT: a distributed real-time architecture for vehicle applications. In: Proceedings of the real-time technology and applications symposium, p. 220 (1995)
14. Möller, A., Åkerholm, M., Fredriksson, J., Nolin, M.: Evaluation of component technologies with respect to industrial requirements. In: 30th EUROMICRO conference 2004, 31 August - 3 September 2004, Rennes, France. IEEE Computer Society; pp. 56–63 (2004)
15. Mubeen, S., Mäki-Turja, J., Sjödin, M., Carlson, J.: Analyzable modeling of legacy communication in component-based distributed embedded systems. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, IEEE Computer Society; 2011. pp. 229–238 (2011)

16. Bucaioni, A., Cicchetti, A., Sjödin, M.: Towards a metamodel for the rubus component model. In: Ciccozzi F, Tivoli M, Carlson J, editors. In: Proceedings of the 1st international workshop on model-driven engineering for component-based software systems co-located with ACM/IEEE 17th international conference onmodel driven engineering languages & systems (MoDELS 2014), Valencia, Spain, September 29, vol. 1281 of CEUR Workshop Proceedings. 2014. pp. 46–56. Available from: http://ceur-ws.org/Vol-1281/5.pdf (2014)
17. Bucaioni, A., Cicchetti, A., Ciccozzi, F., Mubeen, S., Sjödin, M.: A metamodel for the Rubus component model: extensions for timing and model transformation from EAST-ADL. IEEE Access **5**, 9005–9020 (2016)
18. Bucaioni, A., Addazi, L., Cicchetti, A., Ciccozzi, F., Eramo, R., Mubeen, S., et al.: MoVES: a model-driven methodology for vehicular embedded systems. IEEE Access **6**, 6424–6445 (2018). https://doi.org/10.1109/ACCESS.2018.2789400
19. Bucaioni, A., Becker, M.: Enabling automated integration of architectural languages: an experience report from the automotive domain. J. Syst. Softw. **184**, 111106 (2022). https://www.es.mdu.se/staff/2662-Alessio_Bucaioni
20. Thorngren, P.: Keynote talk: experiences from east-adl use. In: EAST-ADL Open Workshop, Gothenberg (2013)
21. : ISO 26262-1:2011: road vehicles in functional safety, http://www.iso.org, Accessed: September (2019)
22. Pop, P., Scholle, D., Hansson, H., Widforss, G., Rosqvist, M.: The SafeCOP ECSEL Project: Safe Cooperating Cyber-Physical Systems Using Wireless Communication. In: Kitsos, P. (ed.) 2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016, pp. 532–538. IEEE Computer Society (2016)
23. Dijkstra, EW.: On the role of scientific thought. In: Selected writings on computing: a personal perspective. Springer, Berlin pp. 60–66 (1982)
24. Crnkovic, I., Larsson, M.P.H.: Building reliable component-based software systems. Artech House, Inc., Norwood (2002)
25. : ARINC Specification 653P1-2, Avionics application software standard interface Part 1 - required services, http://www.arinc.com, Accessed: September (2019)
26. Bucaioni, A., Mubeen, S., Lundbäck, J., Lundbäck, KL., Mäki-Turja, J., Sjödin, M.: From modeling to deployment of component-based vehicular distributed real-time systems. In: 2014 11th international conference on information technology: new generations. IEEE, pp 649–654 (2014)
27. Bucaioni, A., Mubeen, S., Cicchetti, A., Sjödin, M.: Exploring timing model extractions at EAST-ADL design-level using model transformations. In: 2015 12th international conference on information technology - new generations; pp. 595–600 (2015)
28. Bucaioni, A., Mubeen, S., Lundbäck, J., Lundbäck, KL., Mäki-Turja, J., Sjödin, M.: From modeling to deployment of component-based vehicular distributed real-time systems. In: 2014 11th International conference on information technology: new generations; pp. 649–654 (2014)
29. Ciccozzi, F., Tichy, M., Vangheluwe, H., Weyns, D.: Blended modelling – What, why and how. In: Procs of MPM4CPS workshop (2019)

Alessio Bucaioni is a software engineer currently working as an assistant professor in computer science at Mälardalen University. His research focuses on several aspects of the development of complex software-intensive systems ranging from software architecture to model-driven engineering. He rece ived his PhD degree from Mälar dalen University in 2018. Thereafter, he worked as software engineer and embedded software consultant. During his doctorate and in his research activity, Alessio collaborated with several international companies. Alessio is involved in the organization and program committees for several conferences and is a reviewer for journals in the software engineering domain. Alessio is active in European and national research projects. More information is available at http://www.es.mdh.se/staff/.



Federico Ciccozzi is an Associate Professor and Head of Research Education in Computer Science and in Electronics at Mälardalen University (Sweden). His research specializes in: definition of DS MLs, model transformations, system properties preservation, multiparadigm modelling, model versioning, combination of MDE and CBSE for complex systems, blended modelling, language and compiler engineering. Federico has organized over 50 conference tracks, sessions, workshops and journal special issues. He has been a program committee member of over 40 scientific events in the last year. He is associate editor for IET Software, and 10 times guest editor of SoSyM, JISA, and COLA. He has (co-)authored over 120 peer-reviewed publications. More info at: http://www.es.mdu.se/staff/266-Federico_Ciccozzi.

Amleto Di Salle is currently an assistant professor at the Department of Human Sciences at the European University of Rome. In 2015, he received a Ph.D. in computer science from the University of L'Aquila. His main research activities are related to several aspects of Software Engineering, particularly in distributed systems composition, software architecture, model-based software engineering, and software systems evolution, focusing on technical debt. He has worked on several European and national research projects, such as CHOReOS, CHOReVOLUTION, INCIPICT, Territori Aperti, and Banca Dati Emergenze. He is currently a member of the Editorial Board of the Journal of Universal Computer Science and the special guest editor of the special issue "Foundations and Practice of Visual Modeling (FPVM)" of the Journal of Computer Languages. He has been involved in the program committee conferences and workshops and organized several workshops, such as MDE4SA@ICSA and FPVM@STAF. More information is available at https://amletodisalle.github.io/.

Mikael Sjödin is focusing his research on new methods to construct software for embedded control systems in the vehicular and telecom industry. The current re search goal is to find methods that will make software development cheaper, faster and yield software with higher quality. Concurrently, Mikael is also been pursuing research in analysis of real-time systems, where the goal is to find theoretical models for real-time systems that will allow their timing behavior and memory consumption to be calculated. Mikael received his PhD in computer systems 2000 from Uppsala University (Sweden). Since then he has been working in both academia and in industry with embedded systems, real-time systems, and embedded communications. Previous affiliations include Newline Information, Melody Interactive Solutions and CC Systems. In 2006 he joined MDU faculty as a full professor with specialty in real-time systems and vehicular software-systems.