Automated Synthesis of a Real-time Scheduling for Cyber-physical Multi-core Systems

Johannes Geismann[1], Robert Höttger[2], Lukas Krawczyk[2], Uwe Pohlmann[3], and David Schmelter[3]

[1] Software Engineering Research Group, Paderborn University
Zukunftsmeile 1, 33102 Paderborn, Germany
johannes.geismann@upb.de
[2] IDiAL Institute, Dortmund University of Applied Sciences and Arts
Otto-Hahn-Str. 23, 44227 Dortmund, Germany
[robert.hoettger|lukas.krawczyk]@fh-dortmund.de
[3] Software Engineering Department, Fraunhofer IEM
Zukunftsmeile 1, 33102 Paderborn, Germany
[uwe.pohlmann|david.schmelter]@iem.fraunhofer.de

Abstract. Cyber-physical Systems are distributed, embedded systems that interact with their physical environment. Typically, these systems consist of several Electronic Control Units using multiple processing cores for the execution. Many systems are applied in safety-critical contexts and have to fulfill hard real-time requirements. The model-driven engineering paradigm enables system developers to consider all requirements in a systematical manner. In the software design phase, they prove the fulfillment of the requirements using model checking. When deploying the software to the executing platform, one important task is to ensure that the runtime scheduling does not violate the verified requirements by neglecting the model checking assumptions. Current model-driven approaches do not consider the problem of deriving feasible execution schedules for embedded multi-core platforms respecting hard real-time requirements. This paper extends the previous work on providing an approach for a semi-automatic synthesis of behavioral models into a deterministic real-time scheduling. We add an approach for the partitioning and mapping development tasks. This extended approach enablse the utilization of parallel resources within a single ECU considering the verification assumptions by extendig the open tool platform App4mc. We evaluate our approach using an example of a distributed automotive system with hard real-time requirements specified with the MechatronicUML method.

Keywords: CPS · MDSD · Real-time Scheduling · Synthesis · Model-transformation · Multi-Core · Automotive · Amalthea · APP4MC

1 Introduction

Cyber-physical Systems (CPSs) are executed in physical environments, interact with each other, and are distributed over several Electronic Control Units (ECUs). Examples of CPSs are modern cars in Car-2-Car and

The final publication is available at Springer via:
https://doi.org/10.1007/978-3-319-94764-8_4

Car-2-X scenarios. Often, these systems perform safety-critical tasks under hard real-time requirements. Heterogeneous hardware architectures consisting of interconnected multi-core ECUs are increasingly used in order to fulfill the increasing demand for computing power.
Model-driven development methods like MechatronicUML [7] are applied to develop the embedded software of interconnected CPSs efficiently, correctly, and to cope with the overall complexity. For this, a Platform Independent Model (PIM) is developed consisting of a component-based software architecture. Formal verification approaches like timed model checking [1] are applied to ensure the functional correctness of the modeled behavior. Afterwards, the PIM is refined to a Platform Specific Model (PSM) in order to map the PIM to the underlying multi-core platform. Especially, a scheduling needs to be derived for utilizing a multi-core platform efficiently. Moreover, the verified safety and real-time requirements need to be preserved in the scheduling. However, a systematic method to derive a feasible multi-core scheduling for interconnected CPSs that preserves verified safety and real-time requirements by design is missing.
This paper is an extended version of [14]. We present an approach that enables a step-wise, semi-automatic synthesis of behavioral models into a deterministic scheduling suited for multi-core target platforms and respects safety and real-time requirements. In addition to [14], we present in this version sophisticated techniques for grouping software parts into executable units (called partitioning) and for assigning these units to the execution cores respecting all real-time requirements by design. We embed our approach in the MechatronicUML [7] and App4mc [2] toolchains and evaluate our results with an automotive example. MechatronicUML provides a modeling language, a development process, and an Eclipse-based tooling to design software for interconnected CPSs. App4mc focuses on the optimization of timing and scheduling in embedded multi- and many-core systems in the context of AUTOSAR [6]. Therefor, App4mc provides and utilizes the Amalthea model.
In Figure 1, we give an overview of our synthesis approach by means of a Business Process Model and Notation (BPMN) diagram. The upper BPMN pool represents the PIM modeling. First, the software architecture of the system is created (BPMN Task 1). Software components with behavior in terms of statecharts are part of this architecture. The resulting architecture is the input of our approach. Task 2 is the first contribution of this paper. Here, the so-called *segmentation* is applied. In the segmentation, the statecharts are split into small executable parts that allow parallel execution of the modeled software. Corresponding to the AUTOSAR specification [6], we call these parts *runnables*. Also, *runnable properties* like a period for periodic execution are determined which are essential to ensure semantically correct execution as we show in this paper. The lower BPMN pool represents the PSM modeling. In Task 3, the generated runnables are automatically allocated to the distributed, interconnected ECUs. This allocation is the second contribution of this paper. In Task 4 and 5, Amalthea tasks are created and mapped to ECU cores by means of App4mc's partitioning and mapping algorithms, respectively. The detailed explanation of partitioning and mapping (cf.

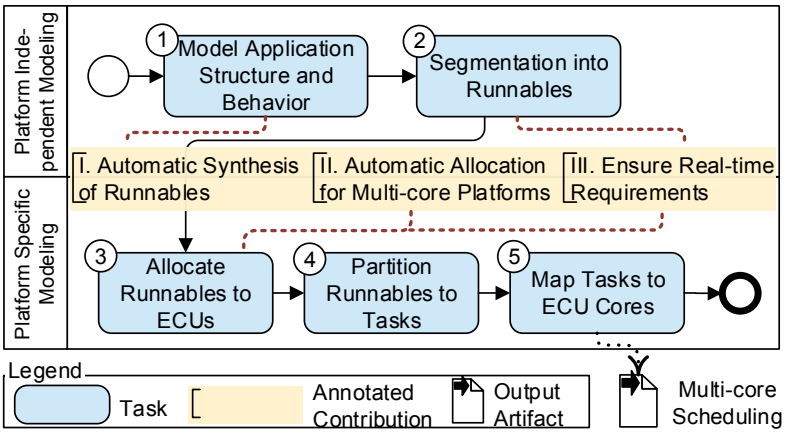

Fig. 1: Process Diagram and Contributions (cf. [14]).

Section 3.3 and 3.4) are the main additional contribution of this long version of the paper. The overall result of the presented process is a deterministic scheduling that is suited for multi-core target platforms. In Tasks 2 and 3, we ensure the execution semantics and real-time requirements of the modeled behavior in the resulting scheduling. This is the third contribution of this paper.

For illustrating our approach, we use the running example shown in Figure 2. The upper part of Figure 2 depicts an autonomous overtaking scenario involving two cars. The cars communicate to coordinate the overtaking maneuver. In our example, the overtaker (red) overtakes the overtakee (green) while the overtakee guarantees that it do not accelerate during the overtaking. This scenario is safety-critical because an error in the communication can result in an unsafe overtaking maneuver. We assume that the correctness of the specified software including its real-time behavior has been formally verified on PIM level by applying model checking [16].

The remainder of this paper is structured as follows. In the next section, we introduce the MechatronicUML models that are relevant and used for our synthesis approach. In Section 3, we present our segmentation approach. Additionally, we present our allocation approach for interconnected multi-core ECUs. In Section 4, we evaluate our approach. In Section 5, we discuss related work. Finally, we conclude our paper and discuss future work in Section 6.

2 Modeling the Application

In this section, we give an introduction to the MechatronicUML modeling artifacts that we use for the software specification on PIM level. Figure 3 shows an overview of all used modeling views, artifacts, and their relations. The Component Instance Configuration view shows the

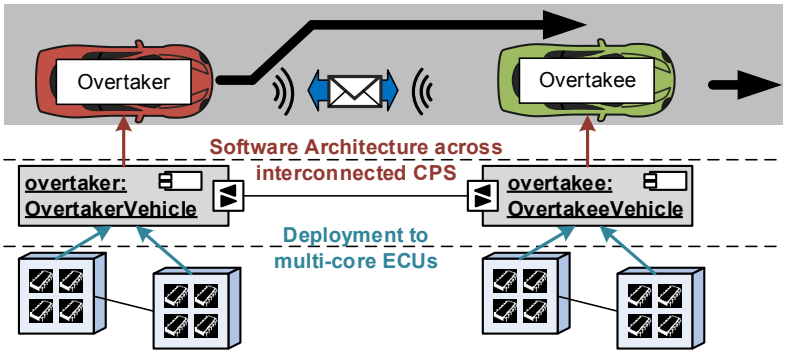

Fig. 2: Running Example Autonomous Overtaking (cf. [14]).

software architecture in terms of a compositional component model. In the top part, Figure 3 shows an excerpt of the software architecture realizing the overtaking scenario. It consists of the component instances overtaker and overtakee. The component instance overtakee is composed of the instances overtakeeCommunicator and overtakeeDriver. Component instances have ports that can send and receive typed messages. Connector instances connect ports and have Quality of Service (QoS) assumptions like a maximum transmission time. For example, the overtaker sends the messages request and finished to the overtakee and can receive the messages accept or decline from the overtakee. Based on the QoS assumptions, the model checking assumes that messages are transmitted within 100ms. Furthermore, component instances can be connected to continuous component instances that represent sensors and actuators of the CPS. For the reason of comprehensibility, we omit these components in the diagram.

The component's behavior is specified in terms of Real-time Statecharts (RTSCs) which combine UML state machines [27] and timed automata [1]. Figure 3 shows the behavior of component instance overtakee. RTSCs can be composed of so-called regions that again contain state machines. For instance, CommunicatorRTSC is composed of the regions communicator and internal. The region communicator represents the behavior of the communication with the overtaker and is composed of the states init, overtaking, and requested. The region internal represents the internal behavior of the component instance that takes the decision whether the overtaking is safe or not and is composed of the states safe, unsafe, and in progress. RTSCs may share variables (e.g., velocity in region internal) and have *clocks* that measure the time and can be *reset* to zero within the statechart, e.g., *timeout* in the region communicator. Furthermore, each RTSC has exactly one currently active state. A state may contain an invariant as a real-time property, which restricts the value of the clock when the state is active. It must be guaranteed during runtime that an invariant is never violated, e.g., the state overtaking has to be left before the clock timeout reaches 50ms. A transition may have a guard ([velocity $>$ 100]) , time constraints ([timeout $>$ 25]), a trigger message (trigger /),

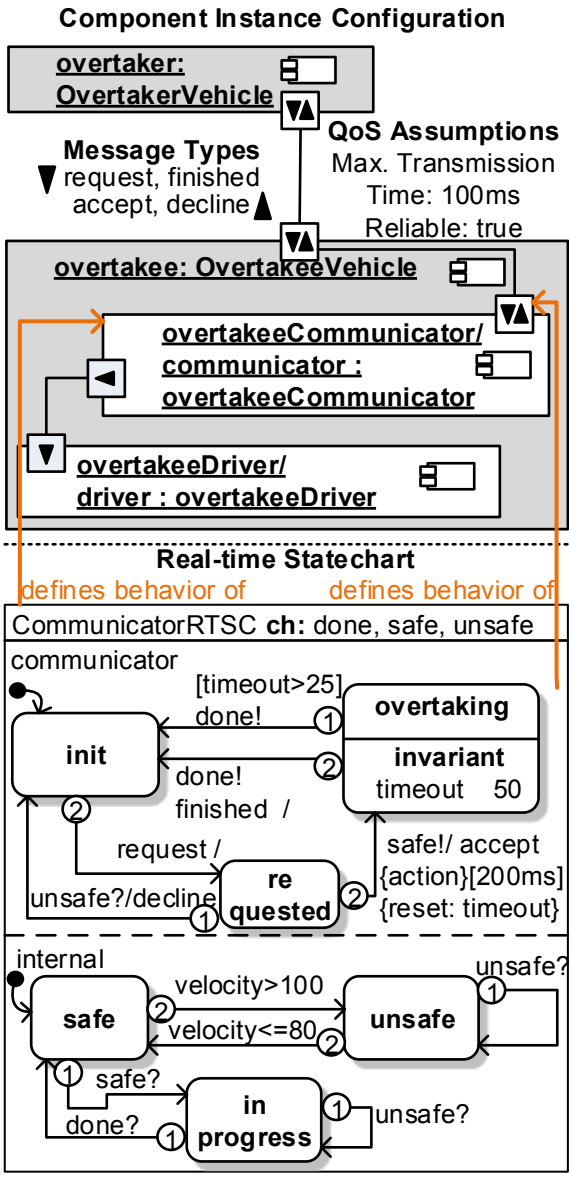

Fig. 3: Overview of Software Development Views (cf. [14]).

and a synchronization channel that restricts the firing (sender channel! /, receiver channel? /). It is *enabled*, i.e., it is able to fire, if its source state is active, its guard evaluates to true, its time constraint evaluates to true, and its trigger message is stored within the buffer. Furthermore, some transitions are connected with each other via synchronization channels; the transition from the state requested to the state overtaking in region communicator is synchronized with the transition from state safe to overtaking in region internal via the synchronization channel safe. Thus, these transitions may only fire jointly.
We assume that RTSCs are executed step-wise, i.e., in each step the outgoing transitions of the currently active state (and all synchronized transitions) are evaluated. If a transition is enabled, the transition with the highest priority fires and the currently activate state gets updated.

3 Software Distribution and Parallelization

In this section, we explain our proposed approach for segmentation and allocation in more detail. We assume that models for the PIM are already created and requirements are verified using model checking (cf. BPMN Task 1, Figure 1). The remainder of this section is structured by following the development process as shown in Figure 1. Afterwards, the *Partitioning* (Section 3.3) and *Mapping* (Section 3.4) approaches are outlined that are used to find a feasible scheduling for all runnables allocated to an ECU under consideration of diverse constraints mentioned accordingly.

3.1 Segmentation into Runnables

The segmentation defines which part of the software models are mapped to a runnable. Runnables are the smallest unit that can be executed by the system and, therefore, segmenting the PIM into runnables affects the behavior execution on the target platform directly. Additionally, WCET, period, and deadline are defined for each runnable. This step is crucial for semantically correct execution because an invariant might be violated if a runnable is executed too late. Thus, the segmentation has to fulfill the following requirements. **R1:** The segmentation has to allow parallel execution. Multi-core environments increase the performance of a system by using parallelization. Therefore, software has to be separated into runnables that can be executed in parallel. **R2:** We aim to generate as few runnables as possible without degrading the possibility of parallel execution because with an increasing number of runnables, the complexity of the partitioning step also increases, which makes it more difficult to find a feasible scheduling and may lead to a decrease in the performance of the system. **R3:** Real-time requirements must be fulfilled at runtime. On PIM level, model checking techniques are used to ensure the fulfillment of these requirements at design time. Executing the software on a platform adds further parameters that have not been considered during the verification step on PIM level, e.g., the activation due to the concrete

scheduling. Thus, a requirement for the resulting scheduling is to ensure that the semantics of the PIM is respected.

In a first step, MechatronicUML software models have to be split into runnables. RTSCs of the software architecture are the starting point for the segmentation. The segmentation directly adresses the first and second requirement because it defines which parts of the software can be executed in parallel. We propose to generate one runnable per region of every RTSC because it allows parallel execution of component behavior without increasing the number of runnalbes significantly. Furthermore, this segmentation is reasonable because each port behavior is described in exactly one region. Hence, we generate one runnable per port behavior and, therefore, the different communication protocols of a component can be executed in parallel. In addition, we generate one runnable per continuous component that is used to read sensor values periodically. Executing the runnable for a region executes one step of the corresponding RTSC, i.e., evaluating and possibly firing outgoing transitions of the currently active state.

The resulting runnables may have dependencies since they may share RTSC variables. These dependencies are important for partitioning and mapping because runnables accessing the same variable are not suitable to be executed in parallel. Corresponding to AUTOSAR, we call such variables *labels*. At first, we define labels and label-accesses of runnables. Furthermore, RTSCs may use shared variables and real-time clocks, for which labels are generated also. These label-accesses are specified for every runnable. Figure 4 shows the label accesses for the example RTSC in Figure 3.

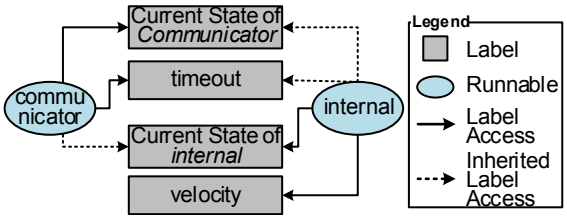

Fig. 4: Runnables have to Specify Label Accesses (cf. [14]).

Both runnables define a label access to their current state label. The runnable for region communicator defines a label access to the label for the clock timeout. The runnable for region internal defines a label access to the variable velocity.

Additionally, both runnables specify inherited label accesses, which are needed, if synchronization channels are used. Since two transitions have to be fired jointly, we propose to extend the models and implementation for runnables by the possibility to evaluate and fire all synchronized transitions. In Figure 3, the transition from state overtaking to init in region communicator are synchronized with the transition from state in

progress to safe via the synchronization channel done. Hence, both runnables inherit the label accesses from the other runnable.
In a second step, we derive runnable properties. Since these properties directly affect the scheduling, their correct determination is crucial for preserving model checking results at runtime. Every runnable has to provide a period, a deadline, and a WCET that are used for partitioning, mapping, and further analyses. Our approach provides an automatic technique to determine a period and deadline for each runnable. Determining a platform-specific WCET is a complex topic and out of scope of this paper. In our approach, we assume that the WCET for each runnable is determined by an appropriate method (e.g., Simple Scalar [5] or aiT [13]) and provided as an annotation for each runnable.
The period describes how frequently a runnable is executed. We provide an automatic technique to determine a period, such that all real-time requirements are fulfilled at runtime without increasing the processor utilization unnecessarily. Determining the period has to respect the semantics of the transition conditions, i.e., guards, deadlines, clock constraints, and invariants. Since a runnable is executed periodically, we have to guarantee that it is executed whenever a transition is enabled.
Based on the transition conditions, we can determine an *enabling interval* I_E which describes the time span when a transition is enabled. We determined a computation rule how I_E can be computed for all combinations of transition conditions. In general, we define $I_E = I_{max} - I_{min}$, where I_{min} is the first point in time and I_{max} is the last point in time when all transition conditions validate to true. As an example, consider the combination of a clock constraint and a state invariant, e.g., the transition from state overtaking to init in region communicator with priority 1 in Figure 3. The transition has a clock constraint that is enabled when the clock timeout is greater than 25ms. Additionally, the state overtaking has an invariant that is valid when the clock timeout is less or equal 50ms. Figure 5 shows the time frames when each constraint validates to true. Hence, I_{min} is at $25ms$ and I_{max} is at $50ms$. Thus, the valid enabling interval I_E has a length of $25ms$.

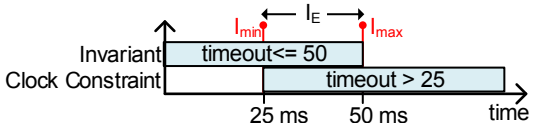

Fig. 5: Finding the Enabling Interval of a Transition (cf. [14]).

If several clock constraints are used, we can generalize I_{min} to the infimum of all *greater-or-equal* constraints and I_{max} to the supremum of all *less-or-equal* constraints. Similar to this, we defined for all other transition conditions a similar computation. Since guards can depend on sensor values, guards also depend on the period of the runnable of the corresponding continuous component. Thus, guards have to be considered in the computation of I_{min} and I_{max}.

It is crucial that the runnable is executed during I_E for each transition because an enabled transition might become disabled again before firing. Otherwise, the assumptions used during model checking would be neglected. Thus, based on I_E we determine a period for the runnable. For this, we set the period to half of the length of the shortest enabling interval I_E. Figure 6 illustrates that a well-chosen period is essential to guarantee the firing of an enabled transition. It shows two different cases of the execution for the runnable that handles the transition of the example above. Each case shows the enabling interval of the transition, the periodic activation times of the runnable, and the concrete execution of the runnable. On the left, the period is set to I_E. Here, the enabling interval of the transition is missed because the transition is evaluated too late. Therefore, the invariant of the state gets violated. On the right, the period is set to $\frac{I_E}{2}$ which ensures that the runnable is executed at least once during the enabling interval because a runnable is executed completely before it is activated again.

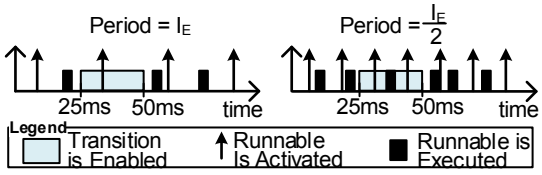

Fig. 6: Length of Period Affects the Execution (cf. [14]).

Since the period π_r has to respect all transitions of the runnable, the period of a runnable r is defined as the minimum of all period values:

$$\pi_r = min\left\{\left\lceil\frac{min(I_E)}{2}\right\rceil \mid \forall\ I_E \in runnable\right\}, \tag{1}$$

The current approach is limited to local (within one region) clocks and to clocks that get reset when entering the state. Otherwise, the enabling interval cannot be determined precisely. If global clocks should be supported in the future, a solution could be to apply a reachability analysis to find all possible clock zones.

Every runnable defines a deadline. Similar to the period of a runnable, the deadline depends on the execution of each transition of an RTSC since every transition can define a dedicated deadline. Consequently, the runnable has to be finished before the deadline of the firing transition expires. Thus, the deadline of a runnable is defined as the minimum deadline of all transitions that are evaluated by this runnable. If no deadline is specified, we set the deadline to the period value of the runnable, since the runnable has to be finished before it is activated again.

3.2 Allocate Runnables to ECUs

After the segmentation, we have to define which runnable is executed on which ECU (cf. BPMN Task 3, Figure 1). Furthermore, hard real-time requirements of the communication have to be respected.
In the following, we derive two constraints that an allocation of runnables to ECUs has to fulfill: 1. A constraint regarding a necessary condition for schedulability. 2. A constraint that ensures the maximum time for communication at runtime. Based on runnable properties, the constraints are used to guarantee the maximum transmission time and schedulability of the system with regard to the real-time requirements during the allocation.
When allocating runnables to ECUs, it is required that all ECUs have enough processing capacity to execute all allocated runnables. The runnables for each allocated component decrease the available processing capacity of the ECU. We restrict the allocation regarding a necessary condition for schedulability: The amount of computing time of the executed software must not exceed the processing capacity of the ECU. We define the processing capacity of each ECU core as 1. For simplicity, we assume that all ECUs use homogeneous cores. Thus, all cores have the same processing capacity and, consequently, the processing capacity of each ECU is defined as $C_{ECU} = |ECUCores|$. The utilization factor of a runnable U_r describes how much percentage of C_{ECU} is needed to execute this runnable. We define U_r of runnable r for a specific ECU as $U_r = \frac{WCET_{r,e}}{\pi_r}$, where $WCET_{r,e}$ is the upper bound of the execution time of runnable r on ECU e and π_r is the period of runnable r. If the sum of the utilization factors of all runnables exceeds the processing capacity of the ECU, it is impossible to find a valid scheduling for a given set of runnables. Hence, this sum has to be *less* than the processing capacity of the ECU.

$$\sum\nolimits_{r \in Runnables(ECU)} U_r < k * C_{ECU}, k \in [0;1] \tag{2}$$

k is a constant factor that can be defined by the developer to adjust this constraint for her needs, e.g., to restrict the maximal processor utilization.
Another crucial aspect is the communication time between two components. The allocation affects the communicating time that is needed for communication. In MechatronicUML, the maximum transmission time is constrained by the QoS of a connector instance, denoted by $T_{ConInst}$, e.g., 100ms for the communication between component instances overtaker and overtakee in Figure 3. For the communication, we assume that each components port behavior (one region of the RTSC) is executed by one runnable: a sender runnable r_S that sends the message and a receiver runnable r_R that receives and processes the message. Additionally, we assume that a lower layer is used to handle the transmission of the message from r_S to r_R, e.g., a middleware. Based on [30], we define that delivering a message relies on time for generating and sending the message t_s, transmitting it from sender to receiver t_{trans}, and queuing it until the receiving process recognizes the message t_r. Figure 7

illustrates the derivation of t_s, t_{trans}, and t_r. When a message is sent by r_S, we assume that the middleware sends the message directly after a task has finished. Thus, the message is processed by the middleware at least before the runnable is executed again. Hence, t_s can be estimated by the period of the runnable π_s.

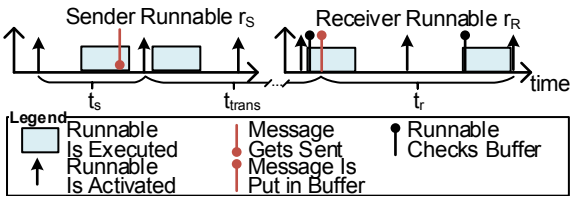

Fig. 7: Upper Bound of Time for Sending and Receiving (cf. [14]).

t_{trans} is based on the used middleware and the underlying communication protocol. We assume that an upper bound constant can be statically determined for each communication channel and used middleware. t_r describes the time it takes from the point in time when the message is put into the message buffer until runnable r_R recognizes the message. Let us assume that the message is put into the buffer immediately after r_R checked the buffer as depicted in the right part of Figure 7. Hence, in this execution, the message is not received by the runnable. Since r_R is activated periodically, it has to be finished completely within the next period interval. Consequently, the time until the message buffer is checked again by the runnable is smaller than $2 * \pi_{receiverrunnable}$. Hence, we use this time as an upper bound for t_r and state the constraint:

$$\pi_s + t_{trans} + 2 * \pi_r \leq T_{ConInst} \tag{3}$$

Both proposed constraints (Equations 2 and 3) are implemented using the allocation approach of MechatronicUML [29], which allows specifying allocation constraints for components, e.g., which components have to be allocated to the same ECU. Thereby, we introduce additional allocation constraints in order to realize an automatic allocation of runnables. We use the heuristic that runnables that belong to the same component instance have to be allocated to the same ECU because a software component instance has a strong coherence [17]. Hence, in this step, we still allocate components to ECUs with respect to the runnable properties.

For each ECU, further actions are needed to refine the models to schedulable software: A *Partitioning* of runnables to tasks and *Mapping* these tasks to ECU cores such that all constraints are fulfilled (cf. BPMN Task 4 and Task 5, Figure 1). Finally, the *deployment* of the software takes place which includes the generation of source code for a given multi-core scheduling.

3.3 Partitioning to Tasks

Partitioning in terms of App4mc focuses on identifying software tasks that can potentially run at different processing cores. Therefore, runnables' activation parameters, instructions, and dependencies are considered. Publication [19] describes the corresponding algorithms. Our experience is that causality, i.e., the runnable order, is the most influencing criterion for the partitioning process. We represent the causality by modeling runnable order using directed acyclic graphs. Due to the specific demands of automotive software the used graph algorithms are extended. Such demands emerge from either communication technologies, advanced driver assistant systems, safety and security concepts, architectural approaches, or diverse design decisions and can often be reflected in specifying and considering constraints. Example for constraints are, among others, core affinities, ASIL level references, software component tags, runnable pairings or separations, or timing constraints. Considering these constraints during the software parallelization is a further contribution of this paper.

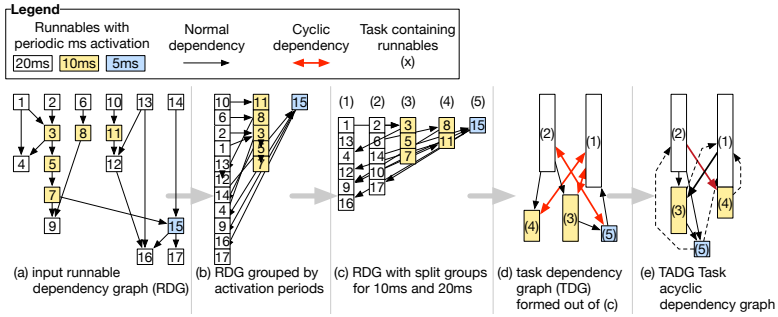

Fig. 8: Example Partitioning of a Runnable Dependency Graph (RDG) to a Task Acyclic Dependency Graph (TADG)

Figure 8 (a) shows a typical graph structure of runnables as rectangles and dependencies as arrows. Figure 8 (b) depicts the same Runnable Dependency Graph (RDG) ordered by the runnables' periods and Figure 8 (c) shows the same graph whereas runnables for 10ms and 20ms are each split once. This partitioning can be configured in different ways. Here shown is a strategy to reduce cross partition dependencies and consider vertical sequences. To maximize parallelism such that runnables of the same activation rate can be computed on different cores concurrently, the vertical partitioning that considers sequences and cross partition dependencies is the prior choice. Another configuration could handle the topological level of runnables, i.e., horizontal partitioning. This latter case is preferred if the partitions are assumed to be scheduled sequentially, e.g., due to the availability of just a few cores. The two different schemes can be configured in App4mc and have to be distinguished carefully to avoid unnecessary inter-communication overheads. Finally, (d)

outlines a Task Dependency Graph (TDG) that contains the runnables from (c) with merged dependencies and (e) depicts the same graph as (d) but without any cycles denoted as Task Acyclic Dependency Graph (TADG). The mechanism to resolve cycles is described in [3]. It is important to note here, that B rectangles outline blocking periods due to shared resources, i.e., labels across cores are already in use by another running runnable on a different core. (a) further assumes having three cores dedicated for runnables with a specific period, i.e., core C0 for 20ms, C1 for 10ms, and C2 for 5ms. However, this model would also be schedulable for 2 cores as shown in Figure 9 (b).

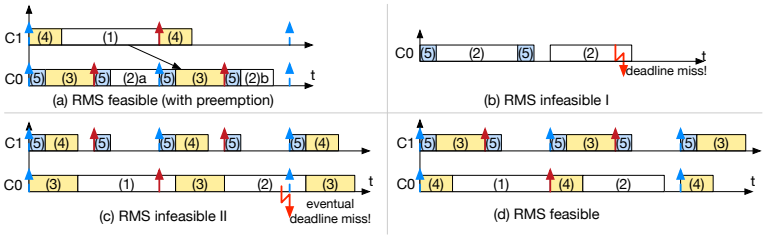

Fig. 9: Gantt Charts of Scheduling for the Example that is Shown in Figure 8 (e)

In order to have more flexibility in terms of software distribution, partitions shown in Figure 8 (c) are formed and transformed to task graphs as (e). Figure 9 provides four different task distribution scenarios (a)-(d) whereas only (d) provides a solution with no preemption and (a) is only feasible with preemption. If either task 5 is combined with task 2 or task 3 is combined with task 1, no feasible schedule can be found. The dashed vertical arrows pointing upwards outline the release of two or more tasks. The scheduling applied to the shown executions is rate monotonic scheduling (RMS). Since the periods are harmonic, the schedulability test

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{4}$$

is sufficient to form schedulable partitions (with C_i denoting the runnable's instructions, T_i denoting the runnable's activation rate, and n representing the number of runnables). The schedulability test during the partitioning (without consideration of the hardware topology) prior to the mapping (including consideration of the hardware topology) ensures valid and coherent solutions in oder to identify the most effective software distribution scenarios.

The challenge in forming partitions like in Figure 8 (c) is not only considering causality, instructions, and activations, but also the above mentioned constraints. If, e.g., runnable 15 is paired with runnable 2 due to, e.g., tight functional relation within the braking system that is not represented by a dependency, partition (2) would have to be composed

differently in order to generate a feasible schedule. Figure 10 outlines the consideration of runnable pairing constraints via runnable cumulation. Any runnable pairing constraint merges the corresponding runnables for

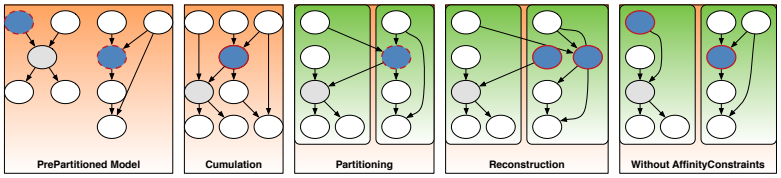

Fig. 10: Runnable Cumulation Mechanism to Consider Runnable Pairing Constraints

the graph algorithms (cumulation) and decomposes (reconstruction) to the original structure after partitions have been formed. Consequently, causality, i.e., the runnable pairing positions and sequences within partitions are considered.
Other than that, if runnables 12, 9, and 16 were safety relevant, e.g., implement a braking system, and reference an according ASIL level, they would have to be separated into an independent partition in order to guarantee freedom from interference, e.g., resource blocking. Therefore, the dependencies must be carefully analyzed and possible blocking situations should be identified so that execution times can be reasoned precisely.
When taking tags, e.g., for software component instances, into account, it is desired to predefine whether and if yes how many component instances can be combined within partitions. Tags are an abstract AMALTHEA model element that can be referenced by runnables or tasks in order to group them according to the diverse users' needs.

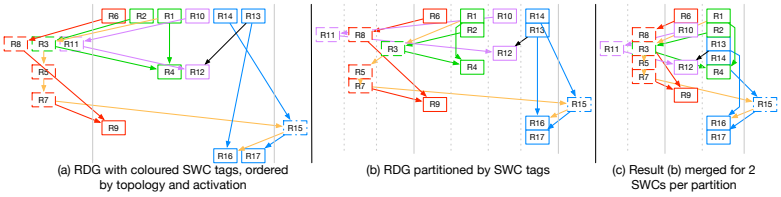

Fig. 11: Partitioning with Tag Consideration for Software Component Instances (SWC)

Figure 11 (a) shows the graph from Figure 8 further extended by colors indicating a specific software component. The runnables are further topologically ordered. (a) is transformed to (b) in order to group component instance-related runnables for each activation. Each column represents

a group respectively a partition. Finally, (c) shows a possible configuration for two component instances per partition. This merging process preserves causal relations within a group and aims at balancing the load across partitions. Obviously, the outcome is quite different from the result shown in Figure 8 due to the consideration of tags.
ASIL levels and tags (e.g., for software components) are equally considered such that separate groups are formed prior to the partitioning process, which always splits the most instructions consuming partition first. Consequently, generated partitions are aligned regarding their instruction sums as much as possible. In other words, all runnables referencing the same tag are grouped into a partition that are may split additionally if the sum of all runnables contained in this group is higher than other the instruction sums at other partitions. An important aspect of this mechanism is its influence of the overall software distribution. In order to keep the amount of generated AccessPrecedencs (i.e., a dissolution of a direct cause and effect relation of two runnables) low, i.e., to keep the program's causality at a high level, existing groups can me merged with other groups or partitions.
When being scheduled, runnables are called directly after each other if no delays are implemented between them. This may eventually result in certain runnables being executed prior to their predecessors. Such behavior can be accepted if these situations were analyzed and verified accordingly, resulting in AccessPrecedence model elements, allowing according runnables to execute with *older* values provided by their predecessors. If such AccessPrecedence is not present, system integrators must assure that runnables wait for label updates provided by their predecessors via events, interrupts, delays, or similar mechanisms.
The difference between runnable pairing and, for instance, tag groups, is that a runnable pairing also influences the position of the corresponding runnables within a task whereas groups have no direct sequencing influence.
As soon as the partitions are formed that consider all the above listed constraints, partitions are transformed to tasks and their possible mappings to processing cores are investigated as described in the following Section 3.4.

3.4 Mapping Tasks to Cores

Mapping in the context of App4mc describes the process of finding a *valid* and *efficient* allocation from software elements to hardware components, i.e., of executable software (runnables or tasks) to cores, data to (distributed) memories, and communications to underlying inter-core networks. These allocations, or mappings, are considered valid if they fulfill all specified constraints, such as meeting an executables' deadlines, providing inter-core communication channels between mapped executables, or being executed on certified hardware. Efficient mappings are achieved by optimizing the distribution w.r.t. one ore more so called quality attributes, e.g., by minimizing the overall runtime, the total energy consumption, or maximizing the reliability of a system. The App4mc OpenMapping plugin implements this functionality and provides several

mapping approaches that are based on various optimization techniques and feature multiple quality attributes. A brief description of these approaches can be found in [21].

In Section 3.2, we described the process of allocating runnables to ECUs. Hence, it is necessary to refine the deployment in order to further distribute the generated tasks from Section 3.3 onto the hardware resources of the corresponding ECU. Similar to the allocation process of runnables to ECUs, the mapping phase has to consider, among others, the execution time (or *response time*) of tasks in order to ensure that deadlines are met. Due to the heterogeneous nature of embedded systems, the response time of a task mainly depends on the core it is mapped to. APP4MC allows specifying the number of instructions for executing tasks on a per core basis, i.e., it is necessary to determine the WCET of a task for all valid mapping targets beforehand, e.g., by means of profiling or appropriate analysis tooling. Once this information is available, the execution time $et_{e,c}$ for executing a task e on core c can be calculated as stated in Eq. 5, with $INS_{e,c}$ being the number of instructions for this concrete mapping, and IPS_c being the number of executable instructions per second. The latter is derived from the AMALTHEA HW Model using Eq. 6, with IPC_c being the executed instructions per cycle, PS_c the Prescaler (frequency scale or divider), and f_c the frequency the core operates at.

$$et_{e,c} = \frac{INS_{e,c}}{IPS_c} \quad \forall \quad e \in Tasks,\ c \in Cores \tag{5}$$

$$IPS_c = IPC_c \times PS_c \times f_c \tag{6}$$

For determining valid mapping targets we consider architectural constraints. Pairing- and separation constraints are treated similarly as in Section 3.3, enforcing or prohibiting the co-existence of a task on the same core. Architectural constraints allow annotating e.g., the required ASIL for a target core, requirements on hardware accelerators, or lock-step modes. Each task can be annotated with features that either are required (enabled) or prohibited (disabled). Accordingly, the final set of valid mapping targets is $Cores \setminus D$ with $E = \emptyset$ and $E \setminus D$ otherwise, with $Cores$ being the set of all available cores, E the set of cores with the required features, and D the set of cores with prohibited features.

Once the solution space is restricted, a mathematical model describing the mapping problem is automatically generated based on the selected approaches optimization technique. In addition to the strategies presented in [21], we have extended this model in order to support communication costs as well as penalty based constraints. Communication costs are an important aspect in distributing tasks among cores, since slow interconnections between cores and a high fragmentation of tightly coupled tasks fosters high execution times. The communication cost is extracted from either the network description or the *AccessPath*s within the ECUs AMALTHEA Hardware Model. *AccessPath*s represent communication channels between, e.g., cores and memories along with their latency. In case *AccessPath*s are not present within the model, the latency is determined by analyzing the ECUs internal network structure,

i.e., by identifying all participants within the internal network, determining all possible paths between them, and evaluating their connection in terms of latency and bit width among each other. While the latter is more complex to be solved due to the exponentially rising number of paths, it provides more flexibility in finding alternative routes on, e.g., NoC architectures.
The extracted communication costs are stored in a $m \times m$ communication matrix T with m being the number of available cores, and T_{ij} the communication cost for transferring information from core i to core j. Having the software in terms of a directed acyclic graph (DAG) $G(V, E)$ with interconnected Tasks V, and E being a set of edges $e(t', t)$ with t' being the source task and t the target task, the matrix is used as lookup-table for determining the execution time on a core. For a simple load balancing approach [12], this is done by adding the resp. communication overheads whenever a task communicates with another over core boundaries. This overhead $comm_{t,c}$ can be determined as shown in Eq. 7

$$comm_{t,c} \;=\; Max\left(\sum_{c'=1}^{m} x_{t',c'} T_{c',c} : t' \in preds(t)\right) \tag{7}$$

The variable $x_{t',c'}$ is set to 1 iff a task t' is mapped to core c', with t' being the predecessor of task t, m the total number of cores, and $preds(t)$ a function for determining the predecessors from task t. Since every task can only be mapped to one core at a time, the sum of the communication overheads always results in the overhead caused by the predecessors mapping. In case of multiple predecessors t', getting the max value ensures that only the highest delay is considered.

4 Evaluation

We conducted a case study to evaluate our approach using the overtaking example. In our case study, we focused on the correctness of the synthesis. We assume the synthesis to be correct if all relevant elements are considered in the applied transformations and all computed values are correct. We based our case study on guidelines by Kitchenham et al. [20] and the Goal-Question-Metric (GQM) method [31] for the structured definition of quality metrics. We state two hypotheses to be validated by the case study. **H1:** We expect, that for the segmentation approach a feasible multi-core scheduling can be found. **H2:** We expect that applying the allocation approach, the result is a correct allocation that respects both stated constraints (cf. Equations 2 and 3), if such an allocation exists. We evaluated schedules for different platforms. In the following, we show the resulting tasks for one multi-core ECU of the overtaker software component instance of the running example. The segmentation of the overtaker components results in 11 runnables, 37 labels, and 39 label accesses.
We applied the segmentation to several additional component models and compared them to manually created reference models. For each model, the segmentation resulted in the expected number of runnables,

labels, and label accesses. Additionally, the generated runnable properties were correct and due to the construction of period and deadline all real-time assumptions hold at runtime. Executing partitioning and mapping of APP4MC resulted in a feasible scheduling with 7 tasks. 5 tasks are mapped to one core and 2 tasks to the other. Table 1 shows the resulting tasks, their properties, and the executing ECU core. Both cores execute runnables of the component instance overtakeeCommunicator and overtakeeDriver. Hence, the execution of the software uses the benefits of parallel execution, which reduced the response time of the system. Overall, we argue that **H1** is fulfilled.

Table 1: Tasks Resulting from Partitioning [14].

Core	**Task**	**Component**	**Period (ms)**
Core 1	T3	Communicator	500
	T6	Driver	500
Core 2	T0	Driver	25
	T1	Communicator	25
	T2	Driver	12
	T4	Communicator	500
	T5	Communicator	500

For evaluating the allocation approach, we considered QoS assumptions of connectors. For each connector, the expected constraints were generated. Additionally, we used different values for the periods of the sender runnable and receiver runnable, as well as for the underlying platform model to test the cases that (A) a valid allocation with two ECUs is found, (B) a valid allocation with only one ECU is found, and (C) no valid allocation is found. For each value combination, the results are as expected. Thus, we state that **H2** is fulfilled. The case study shows that our concepts and the implementation work as expected. Due to the higher degree of automation in the whole development process, there are less manual steps in comparison to state of the art approaches. Additionally, the systems engineer needs less domain knowledge for embedded systems and scheduling. The main threats to validity are: 1. We applied our approach to a small example. 2. We assume that the partitioning and mapping of APP4MC consider all specified constraints correctly, and 3. We assume that the code generation is correct. Overall, we argue that our approach helps to increase the automation of finding a feasible scheduling for software with real-time requirements for multi-core platforms. The concepts are evaluated using MECHATRONICUML and APP4MC, but can be adopted to other approaches. We provide an Eclipse bundle that contains our implementation and model files of the running example [15].

5 Related Work

Our approach is related to component-based approaches for CPS and to approaches for scheduling and safe deployment of CPS. [23] and [11] survey component models in general, whereas [18] survey component models for CPS. Based on that, we state similarities and differences of approaches that consider at least partially concepts for partitioning, mapping, or deployment.

ProCom [10] provides a component model for the development of real-time systems in the automotive and telecommunication domains. ProCom provides a modeling language that is based on Final State Machines enriched by features of Timed Automata to compute (real-time related) dependencies of the model that can affect the scheduling. Additionally, ProCom provides a code synthesis [8] that aims to preserve the semantics of ProCom at runtime. The code for every compenent is executed concurrently. In contrast to our approach, the resulting system is mainly event-triggered, which does not allow a static timing analysis like our approach. Since the component behavior is implemented directly in C, model checking and a model-driven segmentation is not possible. Nevertheless, in [8] a formalization of the generated code is provided.

MEMCONS (Model-based EMbedded CONtrol Systems) [26] provides a model-driven framework for embedded systems and supports the interoperation with AUTOSAR and OSEK models. Since it follows the AUTOSAR methodology, it provides platform independent, component-based development of the system. It also provides an automatic approach for mapping tasks to multi-core ECUs. Furthermore, an analysis of timing constraints can be applied to the deployed system. In contrast to our approach, MEMCONS does not focus on verification of the PIM. Furthermore, the behavior of the software components is not specified model-driven and cannot be used for segmentation.

Further approaches focus on the modeling of (real-time) operating systems elements to improve the deployment of the software. In [24] they extend the DSL RTEPML (Real-time Embedded Platform Modeling Language) [9] to describe the behavior of the RTOS in a platform model, i.e., tasks and semaphores. Using this model for the refinement from PIM to PSM, model checking can be applied, which considers both the application behavior and the behavior of the underlying system. In contrast to our approach, concrete platform properties like the maximum transmission delay are not considered. Furthermore, distributed systems and multi-core ECUs are not taken into account. However, extending this approach to resource management on multi-core environments might be useful to improve our allocation approach.

Lukasiewycz et al. [25] present an approach to derive task priorities in event-triggered systems. The input for the algorithm is a task graph and a mapping. The task graph describes all tasks of the system and their communication. The mapping describes the assignment of tasks and messages to resources, e.g., ECUs or busses. The authors provide an algorithm to find optimal priorities for tasks in event-triggered systems. In contrast, we focus on time-triggered systems and do not consider

priorities of tasks in our approach explicitly. Hence, this approach seems to be interesting to improve the task priorities in our approach.
There are also approaches regarding the semantic-preserving generation of source code for systems with real-time requirements, i.e., approaches for timed automata. In [4], code is generated for timed automata. The authors state that the code generation is platform independent since it also generates a runtime-system that handles task activation and system events. In contrast to our approach, the behavior of the tasks is not generated but implemented manually. Furthermore, the approach does not consider concepts for segmentation, partitioning, and mapping and, therefore, is not applicable for multi-core sytems. In [28], the authors restrict the timed automata to deterministic features. Hence, invariants are not supported in this approach. In [22] on the other hand, they present an approach, where invariants are allowed in the specification. They do not analyze if it all invariants can be guaranteed at runtime. In contrast to our approach, in both approaches properties of the target platform are not considered. Furthermore, both approaches do not consider distributed systems.

6 Conclusion and Outlook

In this paper, we presented a systematic approach that enables a stepwise, semi-automatic synthesis of behavioral models into a deterministic scheduling suitable for multi-core target platforms. We illustrated our approach based on an automotive, autonomous overtaking example and evaluated it based on the MechatronicUML and App4mc platforms.
Firstly, we showed how runnables, runnable properties, and runnable dependencies are synthesized from RTSCs to derive a segmentation that allows parallel execution of software components. We identified limitations in our approach when using clocks across multiple states. Secondly, we introduced an approach for the allocation of runnables to interconnected multi-core ECUs. Especially, we identified and automatically derived necessary conditions an allocation has to fulfill in order to guarantee a valid scheduling. Thirdly, we introduced an approach that preserves verified real-time requirements on PIM level during the synthesis and in the resulting scheduling. In addition to [14], we presented advanced partitioning and mapping approaches considering all real-time constraints derived from former development steps. We used the App4mc open tool platform to validate the correctness of the generated results.
In future work, we want to introduce a reachability analysis to cope with the mentioned limitations regarding clocks. Furthermore, we want to address dynamic scheduling in case of event-triggered systems. We also plan to extend the allocation constraints for ECUs that use cores with different processing capacities and by estimating the transmission time dynamically during the allocation. Finally, our goal is to combine all presented distribution and parallelization technologies along with a single example case study that provides the necessary constraints and reflects industrial needs.

Acknowledgment

This work was partially developed in the Leading-Edge Cluster 'Intelligent Technical Systems OstWestfalenLippe' (it's OWL) and in the ITEA 2 AMALTHEA4public project (Nos. 01IS14029I and 01IS14029K). The IT'S OWL and the AMALTHEA4public projects are funded by the German Federal Ministry of Education and Research.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical computer science 126(2), 183–235 (1994)
2. Amalthea: Deliverable: D3.1 concept for a partitioning/ mapping/ scheduling/ timing-analysis tool. Tech. Rep. 3.4, Amalthea (January 2013)
3. AMALTHEA4public consortium: APP4MC Help Documentation (2017), https://www.eclipse.org/app4mc/help/app4mc-0.8.0/index.html#section4.5.2.3
4. Amnell, T., Fersman, E., Pettersson, P., Yi, W., Sun, H.: Code synthesis for timed automata. Nordic J. of Computing 9(4), 269–300 (Dec 2002), http://dl.acm.org/citation.cfm?id=779110.779112
5. Austin, T., Larson, E., Ernst, D.: Simplescalar: an infrastructure for computer system modeling. Computer 35(2), 59–67 (Feb 2002)
6. AUTOSAR: Release 4.2 Overview and Revision History (2014), http://www.autosar.org/specifications/release-42/
7. Becker et al.: The mechatronicuml design method - process and language for platform-independent modeling. Tech. Rep. tr-ri-14-337, Heinz Nixdorf Institute, Paderborn University (Mar 2014), version 0.4
8. Borde, E., Carlson, J.: Towards verified synthesis of procom, a component model for real-time embedded systems. In: Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering. pp. 129–138. CBSE '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2000229.2000248
9. Brun, M., Delatour, J.: Contribution to the software execution platform integration during an application deployment process. Ph.D. thesis, Ph. D. dissertation, École Centrale de Nantes, Nantes, France (2010)
10. Bureš et al.: Procom–the progress component model reference manual. Mälardalen University, Västerås, Sweden (2008)
11. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.: A classification framework for software component models. IEEE Transactions on Software Engineering 37(5), 593–615 (2011)
12. Drozdowski, M.: Scheduling for Parallel Processing. Computer Communications and Networks, Springer (2009)
13. Ferdinand, C., Heckmann, R.: ait: Worst-case execution time prediction by static program analysis. In: Jacquart, R. (ed.) Building the Information Society, IFIP International Federation for Information Processing, vol. 156, pp. 377–383. Springer US (2004), http://dx.doi.org/10.1007/978-1-4020-8157-6_29

14. Geismann, J., Pohlmann, U., Schmelter, D.: Towards an automated synthesis of a real-time scheduling for cyber-physical multi-core systems. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,. pp. 285–292. INSTICC, ScitePress (2017)
15. Geismann et al.: Implementation and example models. https://trac.cs.upb.de/mechatronicuml/wiki/PaperModelsward17 (2016), http://workupload.com/file/rMP2kVG
16. Gerking et al.: Domain-specific model checking for cyber-physical systems. In: Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation. MoDeVVa '15, vol. Vol-1514 (2015), http://ceur-ws.org/Vol-1514/
17. Gill, N.S., Grover, P.S.: Component-based measurement: Few useful guidelines. SIGSOFT Software Engineering Notes 28(6), 1–6 (Nov 2003)
18. Hošek, P., Pop, T., Bureš, T., Hnětynka, P., Malohlava, M.: Comparison of component frameworks for real-time embedded systems. In: Proccedings of 13th International Symposium on Component-Based Software Engineering, CBSE 2010. pp. 21–36. Springer, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13238-4_2
19. Höttger, R., Krawczyk, L., Igel, B.: Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems. In: International Conference on Parallel, Distributed Systems and Software Engineering. ICPDSSE'15, vol. 2, pp. 2643–2649. World Academy of Science, Engineering and Technology (2015)
20. Kitchenham et al.: Case studies for method and tool evaluation. IEEE Software 12(4), 52–62 (Jul 1995)
21. Krawczyk, L., Wolff, C., Fruhner, D.: Automated Distribution of Software to Multi-core Hardware in Model Based Embedded Systems Development, pp. 320–329. Springer International Publishing, Cham (2015), https://doi.org/10.1007/978-3-319-24770-0_28
22. Kristensen, J., Mejholm, A., Pedersen, S.: Automatic translation from uppaal to c. Tech. rep., Department of Computer Science, Aalborg University (2004)
23. Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering 33(10), 709–724 (2007)
24. Lelionnais et al.: Formal Behavioral Modeling of Real-Time Operating Systems. In: Proceedings of the 14th International Conference on Enterprise Information Systems (ICEIS (2) 2012). Wroclaw, Poland (Jun 2012), https://hal.archives-ouvertes.fr/hal-01093794
25. Lukasiewycz et al.: Priority assignment for event-triggered systems using mathematical programming. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 982–987. DATE '13, EDA Consortium, San Jose, CA, USA (2013), http://dl.acm.org/citation.cfm?id=2485288.2485524
26. Macher et al.: Filling the gap between automotive systems, safety, and software engineering. e & i Elektrotechnik und Informationstechnik pp. 1–7 (2015), http://dx.doi.org/10.1007/s00502-015-0301-x

27. OMG: Unified Modeling Language, version 2.4.1. Superstructure Specification (2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/
28. Opp, D., Caspar, M., Hardt, W.: Code generation for timed automata system specifications considering target platform resource-restrictions. In: Proceedings of the 7th International Conference on Computing and Information Technology 2011. pp. 144–149 (2011)
29. Pohlmann, U., Hüwe, M.: Model-driven allocation engineering. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015). ACM/IEEE, IEEE (Nov 2015)
30. Tindell et al.: Analysis of hard real-time communications. Real-Time Systems 9(2), 147–171 (1995), http://dx.doi.org/10.1007/BF01088855
31. Van Solingen et al.: The Goal/Question/Metric Method: a practical guide for quality improvement of software development. McGraw-Hill (1999)