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

Johannes Geismann[1], Uwe Pohlmann[2] and David Schmelter[2]
[1]*Software Engineering Research Group, Paderborn University, Zukunftsmeile 1, Paderborn, Germany*
[2]*Software Engineering Research Group, Fraunhofer IEM, Zukunftsmeile 1, Paderborn, Germany*

Keywords: CPS, MDSD, Real-time Scheduling, Synthesis, Model-transformation, Multi-Core, Automotive.

Abstract: Modern Cyber-physical Systems are executed in physical environments and distributed over several Electronic Control Units using multiple cores for execution. These systems perform safety-critical tasks and, therefore, have to fulfill hard real-time requirements. To face these requirements systematically, system engineers develop these systems model-driven and prove the fulfillment of these requirements via model checking. It is important to ensure that the runtime scheduling does not violate the verified requirements by neglecting the model checking assumptions. Currently, there is a gap in the process for model-driven approaches to derive a feasible runtime scheduling that respects these assumptions. In this paper, we present an approach for a semi-automatic synthesis of behavioral models into a deterministic scheduling that respects real-time requirements at runtime. We evaluate our approach using an example of a distributed automotive system with hard real-time requirements specified with the MechatronicUML method.

1 INTRODUCTION

Cyber-physical Systems (CPSs) are executed in physical environments, interact with each other, and are distributed over several Electronic Control Units (ECUs). Often, these systems perform safety-critical tasks under hard real-time requirements. Heterogeneous hardware architectures consisting of interconnected multi-core ECUs are used in order to fulfill the increasing demand for computing power.

Model-driven development methods like MECHATRONICUML (MUML) (Becker et al., 2014) 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 (Alur and Dill, 1994) 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.

In this paper, 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. We embed our approach in the MUML (Becker et al., 2014) and AMALTHEA (Amalthea, 2013) toolchains and evaluate our results with an automotive example. MUML provides a modeling language, a development process, and an Eclipse-based tooling to design software for interconnected CPSs. AMALTHEA focuses on the optimization of timing and scheduling in embedded multi- and many-core systems in the context of AUTOSAR (AUTOSAR, 2014).

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,

Geismann J., Pohlmann U. and Schmelter D.
Towards an Automated Synthesis of a Real-time Scheduling for Cyber-physical Multi-core Systems.
DOI: 10.5220/0006117702850292
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 285-292
ISBN: 978-989-758-210-3

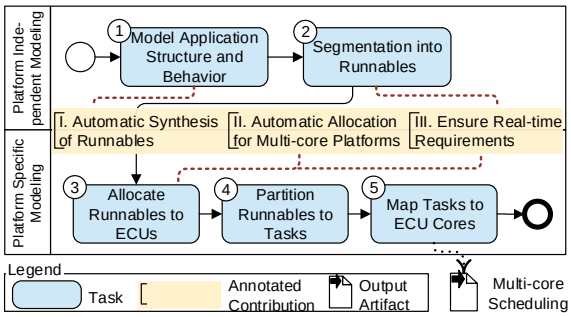

Figure 1: Process Diagram and Contributions.

the statecharts are split into small executable parts that allow parallel execution of the modeled software. Corresponding to the AUTOSAR specification (AUTOSAR, 2014), 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 AMALTHEA's partitioning and mapping algorithms, respectively. The overall result of the presented process is a deterministic scheduling that is suited for multi-core target platforms. In Task 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 will 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 (Gerking et al., 2015).

The remainder of this paper is structured as follows. In the next section, we introduce the MUML 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.

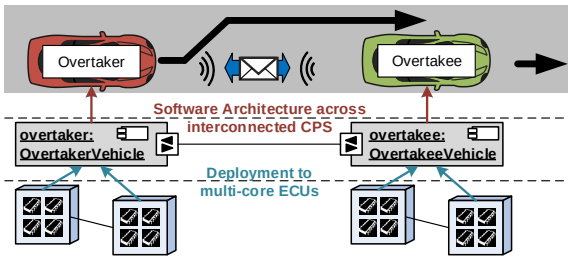

Figure 2: Running Example Autonomous Overtaking.

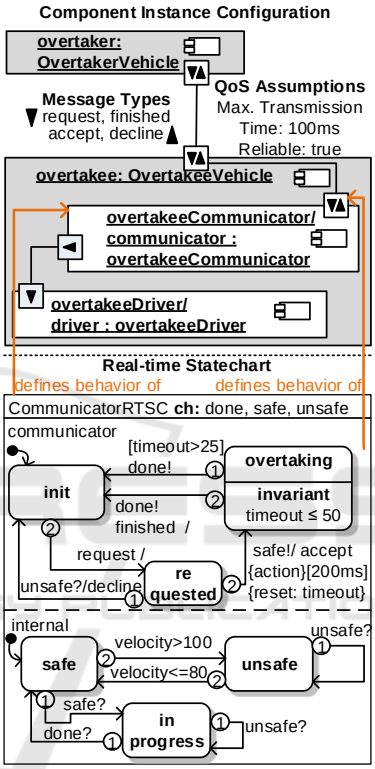

Figure 3: Overview of Software Development Views.

2 MODELING THE APPLICATION

In this section, we give an introduction to the MUML 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 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. Com-

ponent 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 (OMG, 2011) and timed automata (Alur and Dill, 1994). 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 /), 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 SEGMENTATION AND ALLOCATION

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.

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 are respected.

In a first step, MUML 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 soft-

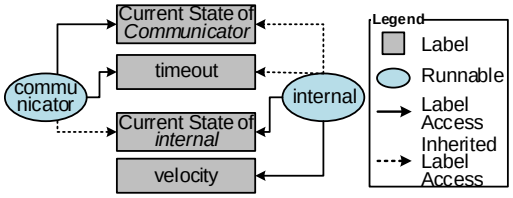

Figure 4: Runnables have to Specify Label Accesses.

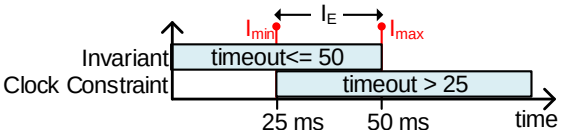

Figure 5: Finding the Enabling Interval of a Transition.

ware 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 will execute 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. 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 (Austin et al., 2002) or aiT (Ferdinand and Heckmann, 2004)) 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 $25ms$. 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 will be executed completely before it is activated again.

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:

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

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 value of the runnable, since the runnable has to be finished before it is activated again.

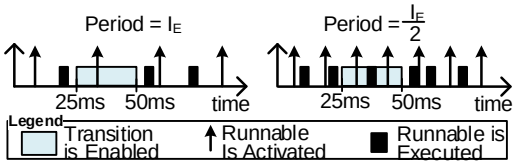

Figure 6: Length of Period Affects the Execution.

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 have 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 will 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.

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

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 MUML, 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

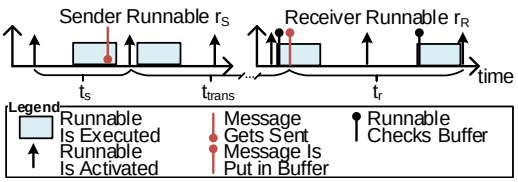

Figure 7: Upper Bound of Time for Sending and Receiving.

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 (Tindell et al., 1995), 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. 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:

$$\pi_s + t_{trans} + 2 * \pi_r \leq T_{ConInst} \quad (3)$$

Both proposed constraints (Equations 2 and 3) are implemented using the allocation approach of MUML (Pohlmann and Hüwe, 2015), 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 (Gill and Grover, 2003). Hence, in this step, we still allocate components to ECUs with respect to the runnable properties.

3.3 Partitioning and Mapping

For each ECU, further actions are needed to refine the models to schedulable software: *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. We utilize concepts and tooling of AMALTHEA for partitioning and mapping and concepts and tooling for code generation of AMALTHEA and MUML. Hence, we provide an automatic model-to-model transformation from MUML to AMALTHEA in order to reuse existing algorithms for partitioning and mapping.

The output of the allocation task describes the allocation of runnables to ECUs. For the execution, these runnables have to be grouped to tasks, which is done in the partitioning. For this, we automatically transform the runnable models from MUML to AMALTHEA by providing a model-to-model transformation. Then, an algorithm to find a feasible partitioning of AMALTHEA (Höttger et al., 2015) can be applied. In this algorithm, runnables are grouped to tasks based on their properties and dependencies.

After that, in the mapping, the newly created tasks are mapped to the cores of the ECU. AMALTHEA provides an algorithm to allocate a set of tasks to the cores of one multi-core ECU regarding specific optimization criteria, like load balancing. The result of this algorithm is a scheduling for each ECU.

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. (Kitchenham et al., 1995) and the Goal-Question-Metric (GQM) method (Van Solingen et al., 1999) 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

Table 1: Tasks Resulting by Partitioning.

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

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. Partitioning and mapping of AMALTHEA 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. 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 AMALTHEA 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 MUML and AMALTHEA, but can be adopted to other approaches. We provide an Eclipse bundle that contains our implementation and model files of the running example (Geismann et al., 2016).

5 RELATED WORK

Our approach is related to component-based approaches for CPS and to approaches for scheduling and safe deployment of CPS.

ProCom (Bureš et al., 2008) provides a component model for the development of embedded real-time systems. 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 that aims to preserve the semantics of ProCom at runtime. The code for every component is executed concurrently. In contrast to our approach, the resulting system is mainly event-triggered, which does not allow a static scheduling analysis. MEMCONS (Macher et al., 2015) provides a model-driven framework for embedded systems. It follows the AUTOSAR methodology and 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, the behavior of the software components is not specified model-driven and cannot be used for segmentation. In both approaches the behavior is not specified model-driven. Thus, a segmentation of verified models is not possible.

Another related area is the safe deployment to real-time systems, where approaches focus on the modeling of (real-time) operating system elements to improve the deployment of the software. In (Lelionnais et al., 2012) a DSL is used to describe the behavior of the RTOS in a platform model, i.e., tasks and semaphores. Thus, model checking can be applied which considers both the application behavior and the behavior of the underlying system. In contrast to our approach, distributed systems and multi-core ECUs are not taken into account. (Lukasiewycz et al., 2013) 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.

6 CONCLUSION AND OUTLOOK

In this paper, we presented a systematic approach that enables a step-wise, 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 MUML and AMALTHEA 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 future work, we want to introduce a reachability analysis to cope with the mentioned limitations regarding clocks. Also, we want to address dynamic scheduling in case of event-triggered systems. Finally, we 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.

ACKNOWLEDGEMENTS

We thank Andreas Dann for feedback on drafts of the paper. This work was partially developed in the Leading-Edge Cluster 'Intelligent Technical Systems OstWestfalenLippe' (IT'S OWL) and in the ITEA 2 AMALTHEA4public project (No. 01IS14029I). The IT'S OWL and the AMALTHEA4public projects are funded by the German Federal Ministry of Education and Research.

REFERENCES

Alur, R. and Dill, D. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

Amalthea (2013). Deliverable: D3.1 concept for a partitioning/ mapping/ scheduling/ timing-analysis tool. Technical Report 3.4, Amalthea.

Austin, T., Larson, E., and Ernst, D. (2002). Simplescalar: an infrastructure for computer system modeling.

AUTOSAR (2014). *Release 4.2 Overview and Revision History*.

Becker et al. (2014). The mechatronicuml design method - process and language for platform-independent modeling. Technical Report tr-ri-14-337, Heinz Nixdorf Institute, Paderborn University. Version 0.4.

Bureš et al. (2008). Procom–the progress component model reference manual. *Mälardalen University, Västerås, Sweden*.

Ferdinand, C. and Heckmann, R. (2004). ait: Worst-case execution time prediction by static program analysis. In Jacquart, R., editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 377–383. Springer US.

Geismann et al. (2016). Implementation and example models. https://trac.cs.upb.de/mechatronicuml/wiki/PaperModelsward17.

Gerking et al. (2015). Domain-specific model checking for cyber-physical systems. In *Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation*, volume Vol-1514 of *MoDeVVa '15*.

Gill, N. S. and Grover, P. S. (2003). Component-based measurement: Few useful guidelines. *SIGSOFT Software Engineering Notes*, 28(6):1–6.

Höttger et al. (2015). Model-based automotive partitioning and mapping for embedded multicore systems. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 9(1):268–274.

Kitchenham et al. (1995). Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62.

Lelionnais et al. (2012). Formal Behavioral Modeling of Real-Time Operating Systems. In *Proceedings of the 14th International Conference on Enterprise Information Systems (ICEIS (2) 2012)*, Wroclaw, Poland.

Lukasiewycz et al. (2013). Priority assignment for event-triggered systems using mathematical programming. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 982–987, San Jose, CA, USA. EDA Consortium.

Macher et al. (2015). Filling the gap between automotive systems, safety, and software engineering. *e & i Elektrotechnik und Informationstechnik*, pages 1–7.

OMG (2011). Unified Modeling Language, version 2.4.1. Superstructure Specification.

Pohlmann, U. and Hüwe, M. (2015). Model-driven allocation engineering. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. ACM/IEEE, IEEE.

Tindell et al. (1995). Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171.

Van Solingen et al. (1999). *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill.