Multi-Core Execution of Safety-Critical Component-Based Software

by

Johannes Geismann

Multi-Core Execution of Safety-Critical Component-Based Software

Master's Thesis

Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of

Master of Science

by
Johannes Geismann
Greitelerweg 88
33102 Paderborn

Thesis Supervisor:
Dr. Matthias Meyer
and
Prof. Dr. Wilhelm Schäfer

Paderborn, December 2015

Master's Thesis No. MA0058

Multi-Core Execution of Safety-Critical Component-Based Software

am: 10.12.2015

FRAUNHOFER-INSTITUT FÜR PRODUKTIONSTECHNOLOGIE IPT
Projektgruppe Entwurfstechnik
Mechatronik
Zukunftsmeile 1
D-33102 Paderborn

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date	Signature

Extended Abstract

Modern Cyber-Physical Systems (CPSs) have to interact with other systems and their environment. They use several multi-core ECUs for executing complex software. CPSs have to operate in safety-critical situations correctly and, therefore, have to fulfill hard real-time requirements. Software developers design the software for such systems in a model-driven way. Thereby, they are able to verify hard real-time requirements for these models formally by using model checking. It is important to ensure that all verified requirements are still fulfilled in the executed software. Hence, the developers have to guarantee that the semantics of the verified models are preserved in the execution. Using real-time operating systems for the execution guarantees a deterministic execution order, a so-called schedule, which is mandatory to satisfy real-time requirements. For determining a schedule, the developers have to translate the models into executable software parts. Additionnaly, they have to define properties (like execution time and execution order) and dependencies like access to shared variables. In current approaches, deriving the software parts, their properties and dependencies is done manually by domain experts. This is a complex and error-prone task for large systems with many dependencies that requires domain-knowledge.

This thesis contributes semantic preserving transformations from verified software models to schedulable software parts. We provide an automatic separation of behavioral models in executable software parts for systems with hard real-time requirements. Additionally, we derive their properties and dependencies automatically. We automatically determine a multi-core scheduling for a real-time operating system based on the executable software parts, their properties, and their dependencies. Furthermore, we provide constraints to derive an allocation of software parts to ECUs of a distributed systems automatically. These constraints ensure real-time requirements of the communication in distributed systems. Using our approach, developers have to apply fewer steps manually, which reduces possible errors within the process of deriving a multi-core scheduling. Moreover, by automating steps for deriving software parts and their properties, we reduce the need of domain knowledge. Hence, applying our approach helps increasing trust in the developed software, since we reduce the number of possible errors. We reduce the development effort by providing automatic transformations for complex and error-prone steps of the process. Thus, our approach reduces development costs and increases the efficiency when developing safety-critical software with hard real-time requirements.

For implementing our approach, we use the MechatronicUML method for defining the system specification and the Amalthea tool platform to compute the multi-core scheduling. We evaluate the correctness of our transformations by applying them to an example describing a system of autonomously overtaking cars.

Contents

Abbreviations

ASL	Allocation Specification Language
CIC	Component Instance Configuration
CPS	Cyber-Physical System
ECU	Electronic Control Unit
HPIC	Hardware Platform Instance Configuration
ILP	Integer Linear Program
MDA	Model Driven Architecture
MDSD	Model-Driven Software Development
OMG	Object Management Group
PDM	Platform Description Model
PIM	Platform Independent Model
PSM	Platform Specific Model
QoS	Quality of Service
RTCP	Real-Time Coordination Protocol
RTOS	Real-Time Operating System
RTSC	Real-Time Statechart
WCET	Worst-Case Execution Time

List of Figures

List of Tables

1 Introduction

Nowadays, developing software for Cyber-Physical Systems (CPSs), like cars or robots, is facing challenges. In general, the controlling software in such systems has to interact with electronic devices, like sensors and actuators, which are used to measure data from and interact with the environment [SW07] under hard real-time requirements [Kop11]. Modern CPSs do not work in isolation but have to interact in collaboration with other CPSs. Additionally, many systems have to control and act in safety-critical situations.

For example, software in modern cars is used to control safety-critical systems like airbags, anti-lock braking systems (ABS), or driver assistance systems, like an Adaptive Cruise Control (ACC). Using an ACC, the driver is able to set a speed that has to be maintained by the ACC. If a car with lower speed is detected in front, the ACC adjusts the speed automatically to avoid a crash. Thus, such a system has to fulfill hard real-time requirements, e.g., by reacting on speed changes immediately.

Hard real-time requirements need mechanisms that ensure deterministic and predictable execution of the software. Real-Time Operating Systems (RTOSs) [But11] are designed for executing software with real-time requirements. In RTOSs, the software is usually split into executable units, called tasks. A task has task properties like a priority and a deadline for the execution and it executes parts of the software. Corresponding to the automotive standard AUTOSAR [AUT14b], we call these executable parts *runnables*. Each task can execute several runnables, which corresponds to a call of behavior code. Based on all task properties, runnables properties, and runnable dependencies, we can compute an order of the tasks (called scheduling), which ensures that all deadlines and priorities are respected at runtime. Using an RTOS, the execution behavior of a system is predictable and deterministic [BEP+07]. Thus, it can be proven, if all deadlines are met, when the system is executed and the fulfillment of real-time requirements can be shown.

In modern CPSs, ensuring requirements is getting more complex, since the number of concurrent software systems within a CPS and the amount of requirements that have to be fulfilled are rising. Modern cars, for example, use more than 70 Electronic Control Units (ECUs) "[...] connected by more than 5 different bus systems [...]" [Bro06]. Due to the need for high-performance systems, a modern ECU might provide a multicore processor, i.e., the ECU has more than one processor core, which allow parallel execution of the software. Additionally, the software has to interact with several sensors and actuators, which make the development more complex. [PBKS07].

The software for CPSs is often developed component-based to tackle the complexity, i.e., the structure of the software is described by component models [SGM02]. Each component encapsulates a behavior and defines interfaces for communication with other components. In general, the component model is independent of the target platform. Hence, this model is called a Platform Independent Model (PIM). Using Model-Driven Software Development (MDSD) [SVB$^+$06], the PIM is combined in the second step with a Platform Description Model (PDM), which describes the target platform, to generate a Platform Specific Model (PSM) [Obj03]. The PSM describes the software specialized for a concrete target platform. Moreover, if adequate models are provided, techniques like Model Checking [CGP00] can be used early in the development process to verify specific functional requirements. When deriving the PSM, it is important that the verified properties hold at runtime. Thus, the deployment of component-based software to the concrete target platform is a complex task.

1.1 Problem Domain

There are several model-driven approaches for developing CPSs. One example is the MechatronicUML method. Typically, model-driven approaches for developing CPSs follow the MDA guide [Obj03] by the Object Management Group (OMG) and use a PIM and a PDM to derive platform specific description of the software (the PSM). For CPSs, this includes several sub-steps. Figure 1.1 illustrate these steps for a component-based input model.

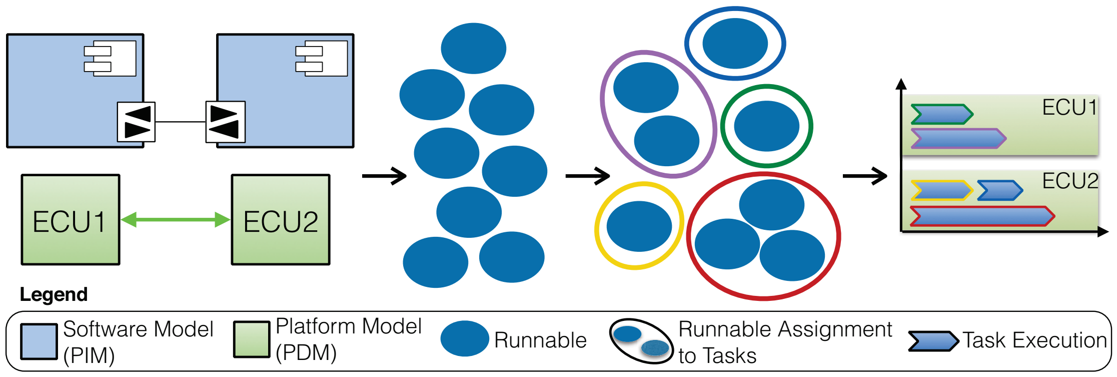


Figure 1.1: Partitioning and Mapping of a Component Model to RTOS Tasks.

First, the PIM (blue boxes) and PDM (green boxes) are described by models, e.g., for specifying the architecture for the system and behavior of the software.

Next, the component models are partitioned into runnables (blue ellipses), which is called *segmentation*. These runnables are assigned to tasks, which is indicated by colored circles, which is called *partitioning*. Using information about runnables, task properties, and the PDM, an allocation to ECU cores and a schedule are created, which is called *mapping*. In existing approaches, we identified a significant problem in these steps.

On the one hand, there are approaches like MechatronicUML that focus on the formal verification of safety requirements of the software behavior. MechatronicUML provides a process and a method for the development, views and viewpoints for PIM and PDM, and allows to derive a PSM automatically. The structural view of the PIM is a hierarchical component model. The behavior of the components is specified using RTSCs, which are based on UML state machines and timed automata. Thus, timing constraints can be modeled in the behavior description to express hard real-time requirements. Furthermore, using the model checker Uppaal [LPY97], MechatronicUML provides formal verification of the behavior on PIM level [DGH15]. A concrete software architecture of a CPS can be modeled by *component instances* that are typed by components. This concrete software architecture is called Component Instance Configuration (CIC). Presently, MechatronicUML has no systematic method for deriving runnables, how to map these runnables to tasks and to derive a scheduling for these tasks. Currently, in MechatronicUML each component instance is mapped to one task and the scheduling is defined manually. Support for multi-core environments is not addressed explicitly during this step. Moreover, timing constraints of the behavior are not considered in these steps. Thus, for each component and its corresponding task, the developer has to review and test if all constraints that were used in the verification step are still fulfilled.

On the other hand, there are approaches that are able to determine valid schedules and allocation of runnables and tasks but do not focus on the verification of safety properties on PIM level explicitly. An example for this is Amalthea [AMA]. Amalthea provides a method and models to develop automotive systems in a model-driven way. Amalthea enables the developers to derive feasible real-time schedules and to analyze the execution in simulation environments or on hardware. For this, Amalthea provides a hardware model to describe multi-core platforms. Furthermore, it provides a component model to describe the structure of the software, which can be transformed into runnables automatically. However, Amalthea does not focus on real-time behavior specification and its formal verification. Amalthea focuses on deriving a real-time schedule for CPS, particularly for automotive systems. For this, the runnable model has to contain information about used variables, shared resources, a period, and a Worst-Case Execution Time (WCET) for each runnable. Amalthea can then determine a partitioning of runnables to tasks and a mapping of tasks to ECU cores. Thus, Amalthea provides a solid method to compute and analyze real-time schedules for a set of runnables.

When specifying software with real-time requirements, like in MechatronicUML, we have to ensure that all assumptions of the verification, like timing constraints of the PIM, are respected in the resulting multi-core scheduling. Currently, there is no method that defines a systematic transition from a PIM to a multi-core scheduling on a concrete platform for software with real-time requirements, such that all assumptions of the verification hold at runtime. This method has to provide a segmentation of the software into runnables, a partitioning of runnables to tasks, and a mapping of tasks to ECU cores. It should support the developer in the steps for segmentation, partitioning, and mapping by automating parts of these steps, e.g., by deriving runnables automatically from the PIM.

1.2 Example

An example scenario for a distributed, cooperative, and safety-critical CPS is the autonomous overtaking of vehicles, e.g., cars. In this thesis, we use such a scenario as a running example for explanation and evaluation of the concepts. We use MechatronicUML models for an overtaking scenario where an approaching vehicle is considered. These models were developed within software engineering research group of the University of Paderborn and bases on the scenario used in [BDG$^+$14b]. In the following, we describe this scenario and the basic models. The scenario consists of *three vehicles* and a *section control*. Figure 1.2 shows a sketch of this scenario.

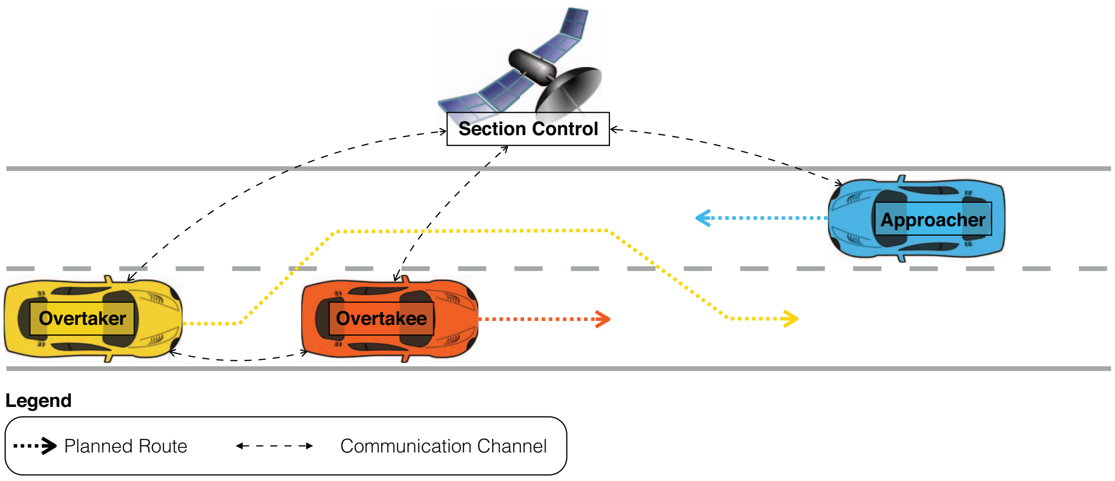


Figure 1.2: Scenario for Overtaking With Approacher.

Two vehicles, `Overtaker` and `Overtakee` drive on one lane in the same direction. On the other lane, a third vehicle (`Approacher`) is driving in the opposing direction. Each vehicle communicates with the `Section Control` and informs about its current position on the lane. Thus, the `Section Control` knows the position of each vehicle. The vehicles themselves do only know their own position.

The `Overtaker` drives with higher speed than the `Overtakee` and when the `Overtaker` gets to a specific distance to the `Overtakee`, it adapts its speed and starts to communicate with the `Overtakee` and the `Section Control` to initiate an overtaking action. It waits for acknowledgments of both communication partners before starting the overtaking. The `Section Control` has to confirm that the distance to the `Approacher` is large enough. The `Overtakee` can accept the overtaking request and confirms that it will not accelerate during the overtaking.

In this thesis, we focus on the `Overtaker`, because it contains all relevant elements for this thesis. Figure 1.3 shows its CIC. The CIC of the complete scenario is shown in Appendix B.1.1. The CIC of the `Overtaker` consists of seven atomic

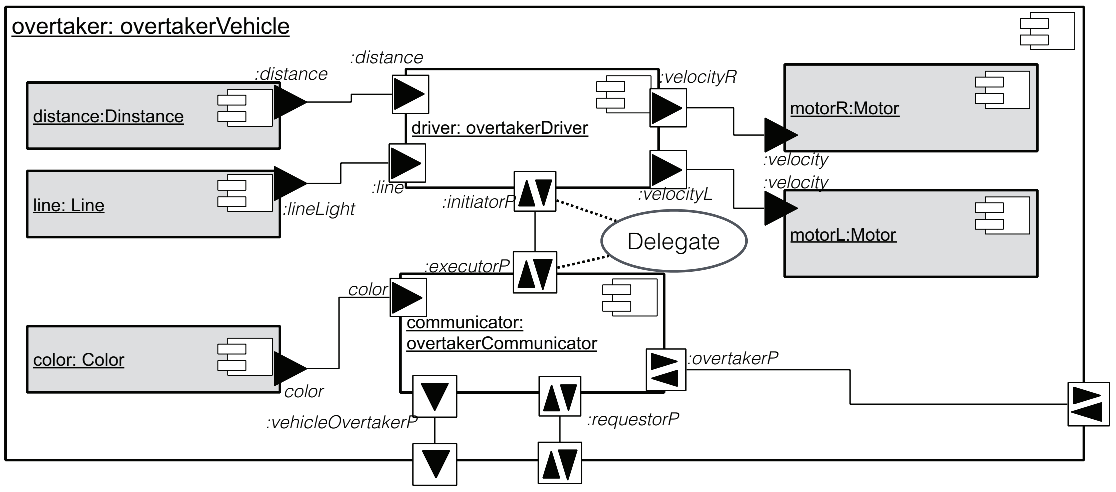


Figure 1.3: Component Instance Configuration of the Overtaker.

components. The components `OvertakerDriver` and `OvertakerCommunicator` define RTSCs as behavior and, therefore, have timing requirements that have to be considered in segmentation, partitioning, and mapping. Additionally, the communication of these components is defined by a so-called Real-Time Coordination Protocol (RTCP) `Delegate`. A RTCP defines Quality of Service (QoS) assumptions that have to be considered when deriving a multi-core scheduling, e.g., an upper bound for the communication time. Furthermore, we assume that the hardware for each vehicle consists of two multi-core ECUs. The CIC, the RTSCs, and the RTCP of the `Overtaker` are described in more detail later on when needed.

1.3 Goals

The overall goal of this thesis is to provide a method to derive a multi-core scheduling for component-based CPSs with respect to hard real-time constraints in the behavior. For this, we have to apply the following tasks:

1. Define a method to determine a segmentation of a component-based description of the PIM into runnables, such that parallel execution is possible.
2. Define a method to derive period, deadline, and WCET for every runnable, such that all real-time requirements are fulfilled at runtime, i.e., timing features of RTSCs and QoS assumptions of the RTCPs have to be considered.
3. Define a partitioning of runnables to tasks and a mapping of these tasks to ECU cores.
4. Integrate the steps for segmentation, partitioning, and mapping into the MechatronicUML development process [BDG+14a].

In this thesis, we use MechatronicUML [BDG+14a] for the specification of the PIM and algorithms of Amalthea [FHKK] to determine partitioning and mapping.

To reach our goals, we organize this thesis in three parts. First, we state new steps for the current development process of MechatronicUML and how they can be integrated into the actual process. Second, we propose concepts for deriving multi-core scheduling for one multi-core ECU. For this, we discuss different strategies for a segmentation and apply partitioning and mapping. After that, we state how these concepts can be extended if the target platform consists of several ECUs and still fulfill hard real-time requirements.

For evaluation, we present a prototypical implementation of our concepts and apply it to the *Overtaking with approacher* scenario.

1.4 Outline

This thesis is structured as follows: Section 2 introduces general foundations that are essential for this thesis, like important concepts for model-driven development, RTOSs, and the used methods MechatronicUML and Amalthea. In Section 3, we define a process for deriving a multi-core schedule for CPS with steps for segmentation, allocation, partitioning, and mapping. Additionally, we state the contributions of the new approach in more detail for each process step. After that, in Section 4, we present an approach for the segmentation of MechatronicUML models into runnables and apply partitioning and mapping for one multi-core ECU. In Section 5, we extend this approach for CPSs that consists of several ECUs by extending the current allocation approach of MechatronicUML. Next, in Section 6, we discuss related work for our approach. After that, in Section 7 we explain our prototypical, Eclipse-based implementation for evaluating the concepts. In Section 8, we discuss the accomplishment of this evaluation and state the results. Finally, Section 9 concludes this thesis and we propose ideas for future work.

2 Fundamentals

In this chapter, we present the basics of the concepts that are relevant for this thesis. For a detailed description of the concepts, we refer to the stated literature.

This chapter is structured as follows. In Section 2.1, we explain the fundamentals for Model-Driven Software Development that is the basic software development paradigm that we use. In Section 2.2 we describe RTOS and the well-known standards for the automotive industry in this area OSEK and AUTOSAR. In Section 2.3, we present the basics for software development for multi-core environments. In Section 2.4, we shortly introduce Integer Linear Programming that is used for solving constraint problems in this thesis. In the last two sections, we introduce two design methods for developing CPS model-driven that are used within this thesis: MechatronicUML (Section 2.5) and Amalthea (Section 2.6).

2.1 Model-Driven Software Development

In a development process for modern software systems, many different problem domains have to be considered. Thus, several experts for different domains are involved in the development. For example, for CPS, experts for developing software, developing hardware, and for the interaction of software and hardware are needed. Often models are used to overcome the complexity and to improve the interaction of all involved developers [BBG05].

In general, according to Stachowiak [Sta73], a model is an abstract representation of real-world entities and their relations with certain pragmatics. Stahl et al. define a model for software systems as "[...] an abstract representation of a system's structure, function or behavior [...]" [SVB$^+$06], which is more suitable for our context. In MDSD, models are not only used to describe the system for the purpose of documentation, but they are used to specify certain aspects of the software system. Model transformations are used to automatically translate one model into another model or into source code for a specific target platform. MDSD helps to handle complex safety requirements and to save costs since errors and risks can be detected early. Thus, MDSD is able to increase the quality of the software and the productivity of the development for embedded systems [BFH$^+$10]. In the following, we explain foundations for modeling languages, which are essential for MDSD, foundations for and different kinds of model transformations, and examples for MDSD approaches.

2.1.1 Modeling Languages

For modeling, a well-defined modeling language is needed. A commonly used and well-known is the Unified Modeling Language (UML) [Obj11b] defined by the Object Management Group (OMG) [Obj]. The UML provides meta-models for software developers, e.g., to describe the structure or behavior of a software system. However, its domain is software in general. Software systems in different domains have different pragmatics and, therefore, need different models, viewpoints, and views. Therefore, a first step for MDSD is to define a Domain-Specific Language (DSL) that fits the problem domain.

For this, a meta-model for the DSL has to be defined first. We use the definition for a meta-model in [Koz08]: "A meta-model is a precise definition of the constructs and rules needed for creating semantic models."[Koz08] For this, it provides an *abstract syntax* and *static* and *dynamic semantics*. The abstract syntax defines all model elements, their structure, and relations. The semantics describe the meaning of the model, where static semantics can be evaluated without execution and dynamic semantics describe the meaning during the execution.

2.1.2 Model Transformations

Model transformations are a key feature of MDSD. They are essential for MDSD since they enable the automatic and deterministic translation between models. Special model transformations languages are used for this purpose. Examples are QVTo and QVT-R [Obj15] for model-to-model transformations or Acceleo [MJL06] for model-to-text transformations.

There are several different kinds of transformations. According to Mens and Van Gorp [MVG06], we distinguish between horizontal and vertical transformations, and endogenous and exogenous transformations. Horizontal transformations are used to translate a model into a model on the same abstraction layer, e.g., apply model normalizations. Vertical transformations describe transformations to another abstraction layer, e.g., by restructuring the model or adding additional information, like generating source code from a formal model. Such transformations are also known as refinement of a model. Endogenous transformations are transformations where the input and output models are instances of the same meta-model. Exogenous transformations, on the other hand, transform a model into a model of another meta-model.

2.1.3 Model-Driven Approaches

The OMG provides a standard for model-driven approaches, the Model Driven Architecture (MDA) [Obj03], which provides a general guideline for developing model-driven approaches. It proposes a process of stepwise refinement of models from an abstract description to a platform-specific model. For this, it defines four model layers: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM), and Implementation Specific Model (ISM). Transformations (automatic and semi-automatic) are used to stepwise refine the model from CIM to ISM. In this thesis, we mainly focus on the PIM and PSM. Since the CIM focuses on non-technical issues of the system, e.g., use case diagrams, the CIM is not in the scope of this thesis. Thus, we do not consider any instances of the CIM. Since the goal is generated source code for a specific target platform, this platform specific source code can be seen as the ISM.

The transformation from PIM to PSM needs additional information to enrich the PIM by platform-specific features. This can be done manually or by providing a Platform Description Model (PDM) and using model transformations to automatically derive a PSM from PIM and PDM. Often, an additional model is used that describes the relation of elements of the PIM to elements of the PDM. Typically a model-to-text transformation is used to transform the resulting PSM into the source code for the target platform.

2.2 Real-Time Operating Systems

Following the ISO 9126 standard [ISO03], correctness (or accuracy) of software can be achieved if the result of the computation is equivalent to the expected result. This definition, however, is not sufficient for systems as CPS that have to react on changes in the environment in a certain time frame. A controller for an airbag, for example, has to activate the airbag in milliseconds when an impending crash is detected. Thus, the correctness of a computation in such systems does not only depend on the result of the computation but also on the point in time when the result is available [SR90].

For this purpose, Real-Time Operating System (RTOS) [Kop11] are used, where the term *time* refers to the fact that the correctness does also depend on the time when a result is produced and the term *real* to the fact that not only internal time but the same time scale as for the environment is used [But11]. We distinguish between *hard*, *firm*, and *soft* RTOS as defined in [But11]:

Hard: Hard real-time means that "[...] producing the results after its deadline may cause catastrophic consequences on the system under control" [But11].

Firm: Firm real-time means that if "[...] producing the results after its deadline is useless for the system, but does not cause any damage" [But11].

Soft: Soft real-time means that if "[...] producing the results after its deadline has still some utility for the system, although causing performance degradation" [But11].

Consequently, safety-critical CPSs as considered in this thesis have to handle hard real-time requirements and, therefore, use a hard RTOS. Hence, in this thesis, we focus on hard RTOSs and do not consider firm or soft real-time requirements.

Similar to other operating systems, in RTOSs the software is executed by dedicated system processes. These processes are called *tasks* and have additional parameters (WCET, deadline, and an activation time, e.g., a period) that are needed to ensure real-time requirements explicitly. Two tasks are called *concurrent*, if they are ready to be executed at the same time (or at least partially). If a set of concurrent tasks has to be executed on one processor, an execution order for these tasks has to be defined following a dedicated strategy. According to the literature, we call this order a *scheduling*. At runtime, the RTOS uses a scheduler with a defined scheduling strategy to order the tasks, such that all real-time requirements are fulfilled. The dispatcher of the system can then execute the tasks in the given order.

In the following sections, we give foundations for the concepts for tasks and scheduling, which are relevant for this thesis. In Section 2.2.1, we present basics for tasks, their properties, and their execution. After that, in Section 2.2.2, we introduce terms and concepts for scheduling in RTOSs. At last, in Section 2.2.3, we present the most important standards for RTOSs in the automotive domain: OSEK/VDX [OSE05] and AUTOSAR [AUT].

2.2.1 Task State and Task Properties

Executing a task is handled by the RTOS. During runtime, a task can be in different states. Figure 2.1 shows these states. After activating the task, the task is in the `ready` state. Hence, it is ready to be executed but has to wait until it can be executed. The dispatcher of the RTOS can assign the task to the processor based on a defined scheduling. Hence, the task changes its state to the `run` state, i.e., it is executing. If a task finished its execution, it *terminates*. In this case, the preempted task becomes `ready` again. A task may stop its execution if it waits for a system resource that is needed for further execution. Then, the task releases the processor and changes into the state `waiting`. The task becomes ready again when the needed resource signals that it is free again.

The scheduler of the RTOS handles all ready tasks and sorts them by their priorities. Based on this order, the dispatcher of the RTOS assigns tasks to free

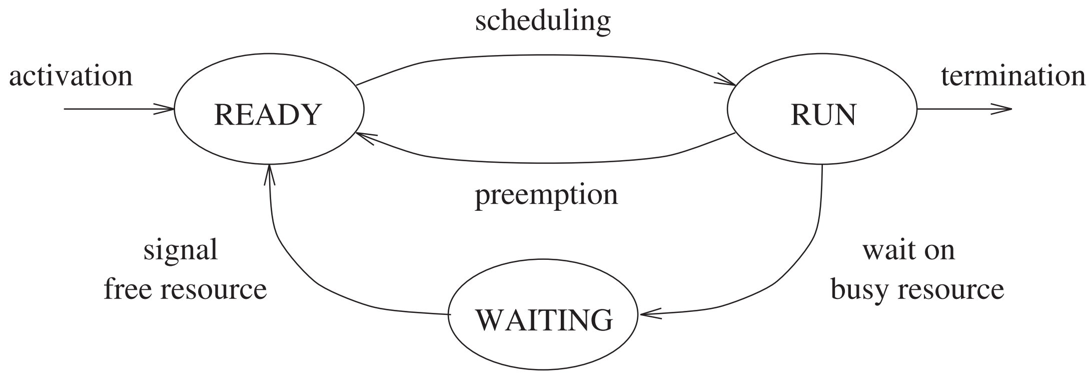


Figure 2.1: Execution States of Tasks (Source: [But11]).

processors. Ready tasks are stored by the system in the *ready queue*. The order of the queue is specified by the scheduling strategy. If a processor is ready to execute a task, the dispatcher chooses the next task on the ready queue. This task is executed until it terminates or it gets preempted by another task with higher priority (decided by the dispatcher). Figure 2.2 shows the execution cycle for the system. In systems that allow different types of tasks, several ready queues can be used.

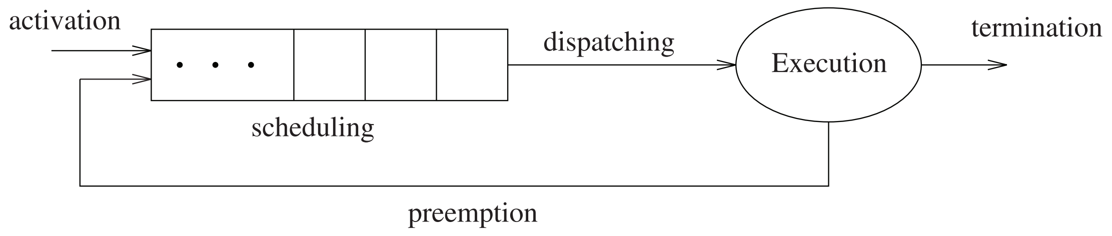


Figure 2.2: Ready Queue and Task Execution Cycle (Source: [But11]).

The scheduler needs several pieces of information about the tasks to compute a feasible scheduling. Buttazzo distinguishes between three different categories that use different information that are important for the scheduler: Time, Precedence, and Resource Accesses. In the following, we present the most important for each category:

Time A task has several important points in time that are needed by the scheduler. Figure 2.3 shows the most important values from becoming ready to termination of the task. Buttazzo defines 12 different values. In the following we give the definition for the six most relevant ones:

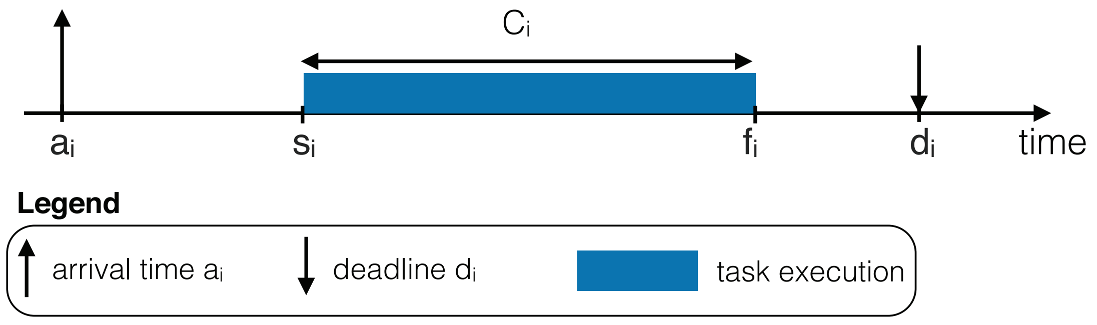


Figure 2.3: Important Points in Time for Tasks (cf. [But11]).

Arrival time a_i (or release time r_i) describes the point in time, when the task becomes ready for execution.

Computation time C_i is the time that is needed to execute the task. The upper bound for this time is the WCET of the task.

Absolute Deadline d_i describes the absolute point in time until the task has to be finished.

Relative Deadline D_i describes the point in time until the task has to be finished in relation to a_i, i.e., it is the difference between d_i and a_i

Start time s_i describes the point in time, when the task gets assigned to the processor the first time. Per definition, this time will be greater than a_i and less than $d_i - C_i$ for each feasible scheduling.

Finishing time f_i describes the point in time, when the execution of the task is finished. In a feasible schedule, it has to hold $f_i \leq d_i$

Additionally, a task can be characterized by the fact, if it is activated periodically or aperiodically. Periodic means that the task is activated regularly at a constant time value. This time value is called *period* and denoted by π. The point in time of the first activation is called *phase* ϕ. Each activation of the task is called *instance* or job and is identical to all other instances of the same task. Figure 2.4 shows several instances of a task and its phase, period, and the corresponding time properties.

A periodic task can be completely characterized by phase, computation time, period and relative deadline [But11]. In contrast to that, instances of aperiodic tasks are activated in non-regular time intervals.

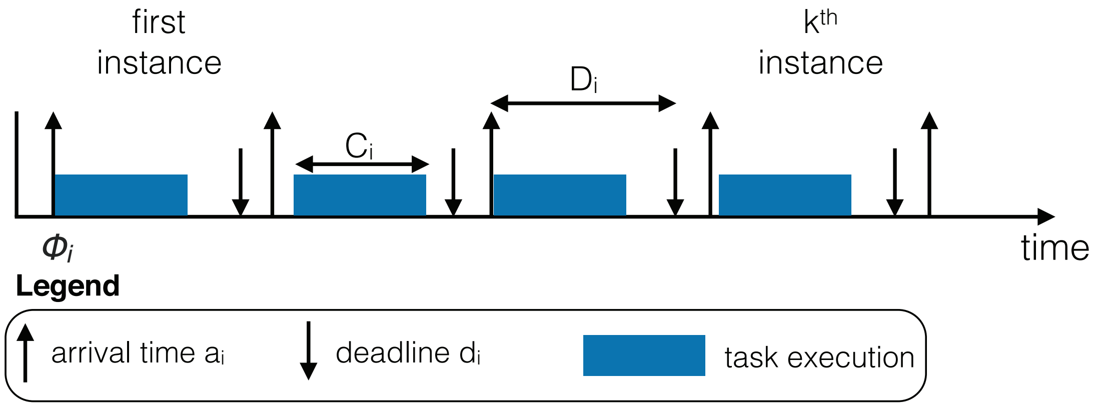


Figure 2.4: Time Properties of Periodic Tasks (cf. [But11]).

Resource Accesses A task can access different resources, like global variables or communication channels. It is important that once a task began accessing a resource, it cannot be preempted until it finishes the resource access. For this, so-called *critical sections* are used, which are protected by a semaphore [And99], such that only one task can access the resource. Thus, the access to shared resources is protected by system mechanisms. Which task accesses which resource is important to the scheduler, because it may influence the scheduling. Specific scheduling strategies (e.g., the *Priority Ceiling Protocol* [But11]) ensure that no deadlocks can occur and that tasks get not preempted by other tasks that also try to access a specific resource.

Precedence Many situations need a specific execution order of tasks, i.e., task T1 must not be executed before task T2 is terminated. Such constraints are called *precedence constraints* and have to be considered by the scheduler. Most scheduling strategies for precedence constraints have to be determined before runtime (called offline, cf. Section 2.2.2). In this thesis, we do not use such constraints explicitly but apply algorithms that already consider such constraints.

2.2.2 Scheduling

A schedule is an assignment of tasks to a processor, such that the processor executes at most one task at any point in time, such that each task is executed until its termination [But11]. If the processor executes no task, the processor is called *idle*. A *feasible* schedule is a schedule, where each task instance meets its deadline.

The scheduler uses all information of the tasks to determine an execution order. Besides this information, additional parameters can be used to classify the chosen scheduling strategy [But11]:

Preemption: If preemption is allowed, a currently running task can be preempted by a task with higher priority. The preempted task can only resume when the higher prioritized task has terminated. If preemption is not allowed, higher prioritized tasks have to wait until the currently running task has terminated.

Static vs. Dynamic: In static scheduling strategies all information have fixed values, i.e., the values will not change at runtime. In dynamic scheduling strategies the values, which are used for the scheduling, can change during execution.

Offline vs. Online: In offline algorithms, all task instances and their properties are specified before the system starts. In online algorithms, the scheduling-decisions have to be performed at runtime.

In the next section, we describe a standard for RTOSs that is used within this thesis: OSEK/VDX [OSE] that uses a static and offline scheduling strategy and can handle both preemptive and non-preemtive tasks.

2.2.3 OSEK and AUTOSAR

The most important standards of the last years for RTOSs in the automotive sector are OSEK/VDX [OSE, Hom09] and AUTOSAR [AUT11, AUT]. Parts of AUTOSAR are based on OSEK. We describe both standards shortly. OSEK, because it is used as specification for the RTOS and still used in AUTOSAR. AUTOSAR is important for the software specification in this thesis, since the models of Amalthea are based on the specification of AUTOSAR.

OSEK/VDX OSEK/VDX (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug) is a set of standards for RTOSs in the automotive industry. OSEK provides interfaces that allow abstracting the specification of the application layer from the operating system, e.g., by abstracting from the used hardware or network [OSE]. A further advantage is, that each RTOS that is compliant to the OSEK-OS standard can read and execute an application specified by this standard. In other words: An application written for an OSEK compliant RTOS can be adapted for all other OSEK compliant RTOSs.

Some parts of OSEK/VDX are standardized by ISO as ISO 17356 [ISO05a]. The most important part is OSEK-OS [OSE05, ISO05b], which defines the basic standard for an OSEK-conform RTOS. OSEK-OS defines a static, event-triggered RTOS for embedded systems. Static means, that all system information (like tasks, OS resources, timer, semaphores, etc.) are defined before compilation and cannot be changed during runtime. For the specification a specific language is used, the OSEK Implementation Language (OIL) [OSE04b, ISO05d]. Event-triggered means that all tasks are activated by specific events, e.g., hardware interrupts. Time-based (periodic) activation of tasks has to be realized by using clock alarms that can fire a task activation. Since this becomes complex and hard to maintain, the extension *OSEK/VDX time triggered operating system* [OSE01] was released that additionally allows time-triggered tasks. Furthermore OSEK-COM [OSE04a, ISO05c] was specified as a standard for inter-task communication.

OSEK/VDX specifies four conformance classes. Each conformance class specifies different combinations of functionalities that have to be implemented by the RTOS, e.g., different kinds of tasks or the amount of running tasks. OSEK/VDX allows to specify preemptive and non-preemptive schedules. Since each task defines if it can be preempted or not, mixed schedules are also possible. The schedulability analysis is applied before the system starts, i.e., OSEK/VDX uses an offline scheduler. For further information, we refer to the OSEK-OS [OSE05] specification.

AUTOSAR AUTOSAR [AUT] is a standard for software development in the automotive industry and has been developed “[...]to improve the design and integration of automotive SW components[...]” [WDNZ$^+$14]. It follows the MDSD paradigm and distinguishes between different layers for development.

Figure 2.5 shows the complete AUTOSAR architecture. In the following, we describe the AUTOSAR layers that are most important for this thesis.

The `AUTOSAR Software` layer contains software components that describe the application software. Software components cannot interact with each other directly, but have to use the `Runtime Environment` (RTE) [AUT14c]. Thus, components can be specified independently from each other and from the underlying hardware- and OS-layers, since the RTE is used as an adapter. Each component can use a specified `AUTOSAR Interface` for interacting with the RTE. Each component specifies its interfaces using different kinds of ports. The concrete behavior of a component is implemented in so-called *Runnables* [AUT14a]. From the view of the model, this implementation is a black-box and it is assumed to work correctly and use the specified interfaces only, the ports of the component respectively.

The RTE is used as “[...]communication center for inter- and intra-ECU information exchange[...]” [AUT11]. Thus, it routes the communication of the software components and can hide the distribution of the components from each other. Therefore, it abstracts from the used communication channels, e.g., CAN or Flexray. Since the RTE is application specific, it has to be tailored for each ECU. Some parts can be generated or derived automatically from configurations. The RTE can access the `Basic Software` and, therefore, provides a common interface for the components to this layer.

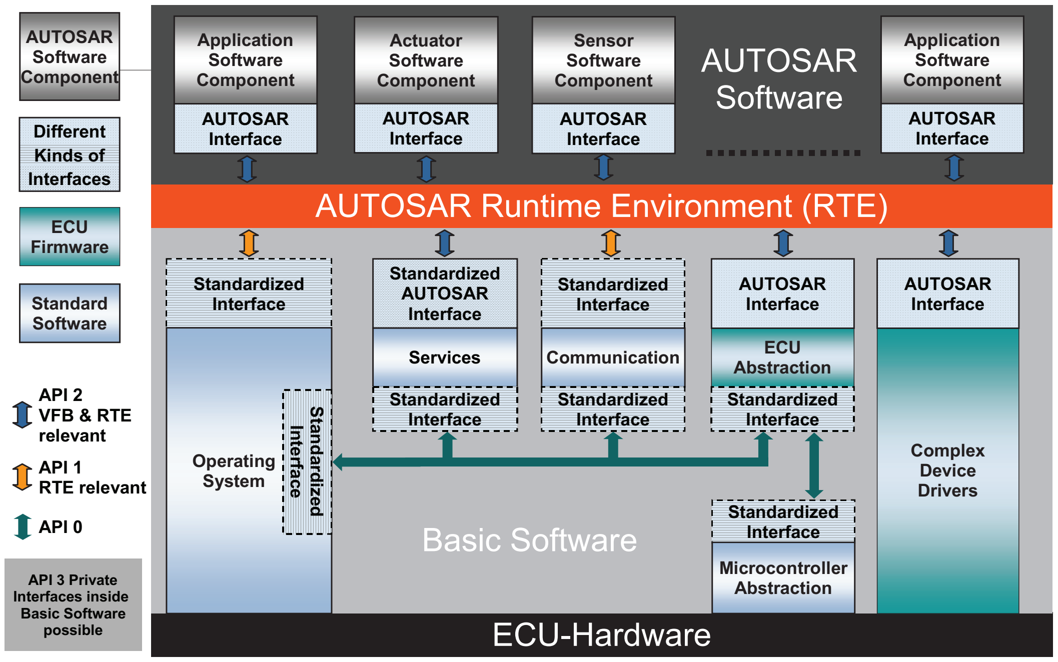


Figure 2.5: AUTOSAR Architecture [AUT11].

The `Basic Software` defines software parts that do not "[...]fulfill any functional job[...]" [AUT11], but provide services to the software components via the RTE [AUT11]. It contains several components, e.g., for `Services`, `Communication`, or the `Operating System`. It is important to state, that AUTOSAR uses OSEK-OS [OSE05] as basis for the operating system [AUT11]. Thus, each RTOS used in an AUTOSAR compliant system has to be OSEK compliant. The operating system executes defined tasks. Each task executes dedicated runnables of the software components. Hence, the correct execution of the runnables is handled by the scheduling of the operating system.

Furthermore, AUTOSAR defines an approach for the system development, the *AUTOSAR Methodology* [AUT14b]. It is not a complete process, but it only defines "dependencies of activities on work-products" [AUT11]. However, for this thesis, we focus on the software components. Thus, for further information regarding the methodology we refer the interested reader to the specification [AUT14b].

2.3 Parallel Execution of Software

Executing software parts in parallel, which means that several processors are used to solve a problem cooperatively [Fos95], is more challenging than executing them sequentially. In this section, we introduce the main concepts and challenges for parallel execution of software.

One important term for our thesis is *partitioning*, which is "[...] intended to expose opportunities for parallel execution" [Fos95] of the software, where a "[...] good partition divides into small pieces both the computation [...] and the data on which this computation operates" [Fos95]. Thus, in the partitioning, software is separated into tasks that can be executed in parallel. Additionally, Foster [Fos95] defines a *mapping*, which specifies which task is executed on which processor core. Usually, the main goal of a mapping is to minimize the execution time. For this, two strategies are proposed that are also applied in the mapping algorithm, which we will use in our thesis [Fos95]:

1. We place tasks that are able to execute concurrently on different processors, so as to enhance concurrency.

2. We place tasks that communicate frequently on the same processor, so as to increase locality.

Communicate means accessing the same data, e.g., global variables. If different tasks have access to the same data, this might result in conflicts since writing/reading the same variable at the same time can lead to data inconsistencies. Thus, mechanisms are needed to avoid such problems. The most known mechanism is using a semaphore [And99] to ensure mutually exclusive access to a variable.

If more than one semaphore is used, this can lead to a deadlock in the program. This might get more complex if an RTOS is used where tasks additionally get scheduled according to their priorities. However, this is well-investigated area and modern RTOS use specific protocols to avoid deadlocks even if several semaphores are used, e.g., the Priority Ceiling Protocol [But11].

2.4 Integer Linear Programming

In this thesis, we use Integer Linear Programming [Sch00] to solve problems of allocating software components to ECUs. In the following, we explain the basic concepts of this topic. In Integer Linear Programming [Sch00], constraint problems are encoded in Integer Linear Programs (ILPs). An ILP typically consists of a set of variables $x_1, ..., x_n \in \mathbb{R}$, a set of inequations $ie_1, ..., ie_m$, and an optimization function $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$. ILP-Solver can optimize values for the variables regarding a defined optimization goal, e.g., to maximize or minimize ϕ. The solution, provided by the Solver, describes values for all variables such that all inequations are fulfilled.

Additionally, ϕ is maximized (minimized respectively). For example, the following inequations specify an ILP [Wik]:

$$\begin{aligned} x_1 - x_2 &\leq 1 \\ 2x_1 + 3x_2 &\leq 12 \\ 3x_1 + 2x_2 &\leq 12 \\ x_1, x_2 &\geq 0 \\ x_1, x_2 &\in \mathbb{Z} \end{aligned}$$

With $\phi : max(x_1)$, one optimal result of the solved ILP is $\{x_1 = 1, x_2 = 2\}$. In general, solving an ILP is NP-hard. Nevertheless, it is used to determine optimal solutions for complex problems. In this thesis, we consider ILPs, since they are used in MechatronicUML (cf. Section 2.5.5) and Amalthea (cf. Section 2.6.2.2) to determine an allocation of software components to ECUs that has to fulfill several constraints.

2.5 MechatronicUML

MechatronicUML [BDG+14a] is a design method for CPS. It provides a domain-specific modeling language, a method to specify software for CPS model-driven, and a tooling platform. The modeling language is based on UML [Obj11b]. In this section, we introduce features that are important for this thesis. Figure 2.6 shows the development process of this method.

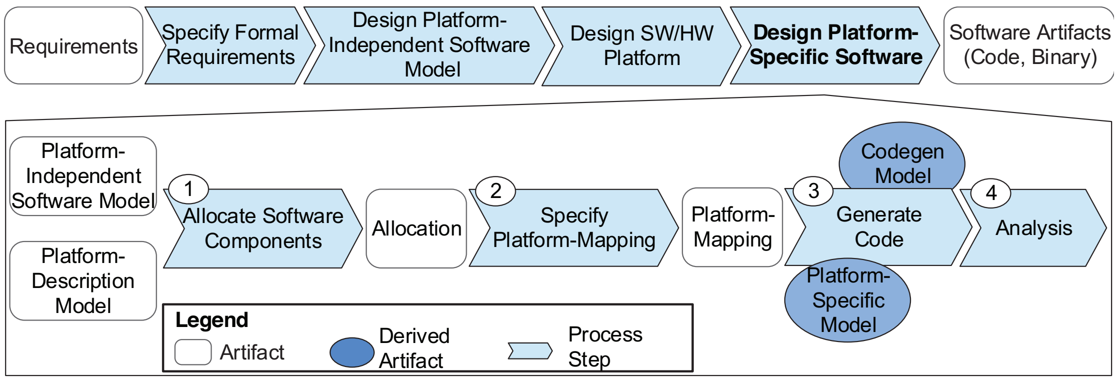

Requirements
Specify Formal Requirements
Design Platform-Independent Software Model
Design SW/HW Platform
Design Platform-Specific Software
Software Artifacts (Code, Binary)
Platform-Independent Software Model
Platform-Description Model
1
Allocate Software Components
Allocation
2
Specify Platform-Mapping
Platform-Mapping
Codegen Model
3
Generate Code
Platform-Specific Model
4
Analysis
Legend
Artifact
Derived Artifact
Process Step

Figure 2.6: MechatronicUML Development Process and Sub Process for the Step “Design Platform-Independent Software Model” (Source: [BDG+14a]).

In **Step 1**, requirement engineers specify the requirements for the software formally. After that, in **Step 2**, the PIM is specified by the software engineer, which can be verified regarding specified safety requirements using model checking techniques (cf. Section 2.5.4). Thus, possible modeling mistakes and risks can be detected early. Furthermore, export mechanisms to simulation environments can be used to analyze the behavior of the system, e.g., to MatLab [HRB$^+$14] or Modelica [PHMG14]. Then, in **Step 3** the target platform, described by the PDM, is modeled by the platform engineer. A platform describes an execution environment, i.e., the used hardware, operating system, and additional software that is needed for the execution of the software system. At last, in **Step 4** the allocation engineer designs the PSM by combining PIM and PDM. Furthermore, in this step the tooling provides an export mechanism that enables the developer to generate source code for different target platforms.

In the following, we describe all models, process steps, and concepts that are used in this thesis in more detail. First, we explain foundations for specifying the PIM in MechatronicUML. In Section 2.5.1, we describe components and component instances. After that, in Section 2.5.2, we introduce so-called RTCPs that are used to specify communication contracts between components. Then, in Section 2.5.3, we introduce RTSCs that are used in MechatronicUML to model the systems behavior. In Section 2.5.4, we give an overview of the verification approach that can be applied to the PIM. At last, in Section 2.5.5, we explain concepts for the platform-related modeling and the export to source code (Step 4 in the process).

2.5.1 Components and Component Instance Configurations

MechatronicUML follows the design approach of component-based development [SGM02]. A component is a software entity with high cohesion encapsulating its inner structure, i.e., it is not accessible or visible by other components. For communication with other components, components can have so called *ports*. Figure 2.7 shows a component diagram for the overtaker of the running example. The component `overtakerVehicle` has three discrete ports for external communication.

Furthermore, MechatronicUML provides hierarchical component models. For this, we distinguish between *structured components* and *atomic components*. Atomic components specify a behavior and do not contain other components, like `overtakerDriver` in Figure 2.7. Structured components contain so-called *component parts* that are typed over other components. `OvertakerVehicle` is a structured component with seven component parts. Each component part specifies a cardinality that describes how often this component will be instantiated at runtime. Each component part has a cardinality of [1..1], i.e., each part will be instantiated exactly once. The behavior of a structured component is defined as the composition of the behavior of its component parts.

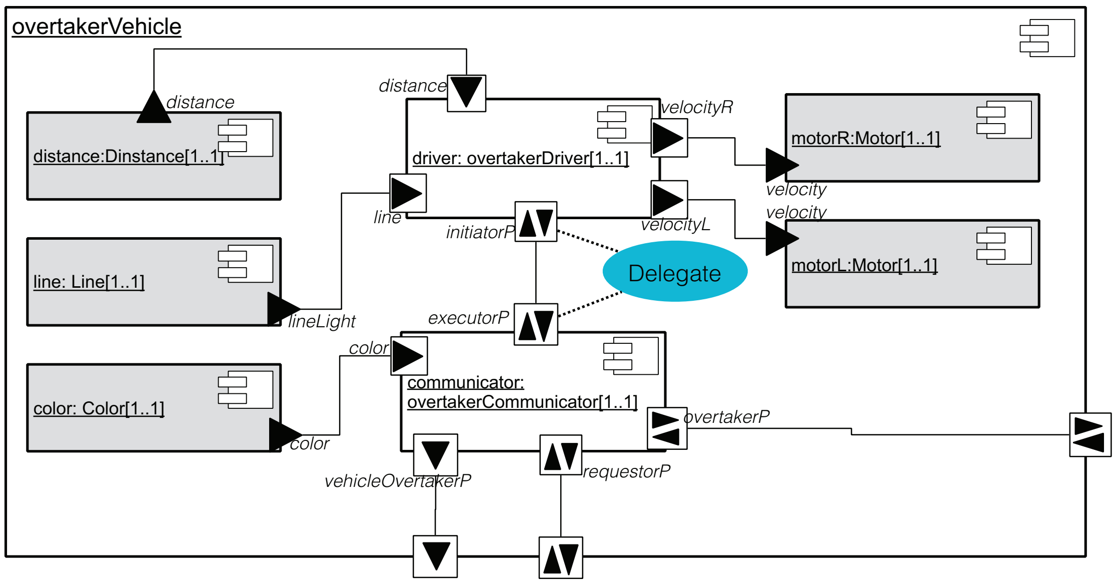


Figure 2.7: Structured Component “overtakerVehicle” and its Component Parts.

Additionally, MechatronicUML distinguishes between *discrete components*, *continuous components*, and *hybrid components*. A discrete component is a component with specified discrete behavior, e.g., `overtakerDriver`. Continuous components represent an abstract view of sensors and actuators of the system, which are accessed by the software system, like all gray-filled component parts in the figure. Structured components are called hybrid structured component, if they contain both discrete components and software components. In the example in Figure 2.7, `OvertakerVehicle` is a structured hybrid component.

Components can communicate via their defined ports. Similar to the components, MechatronicUML distinguishes between *discrete ports*, *continuous ports*, and *hybrid ports*. Discrete ports are used for communication between discrete components, e.g., `initiatorP`. A port has to define the messages, dedicated message buffers, and a behavior. This behavior is also specified by an RTSC. The communication between discrete components is message based, means asynchronous. Continuous ports are ports of continuous components, like `velocity` or `distance`, i.e., they provide or receive the value of the sensor (the actuator respectively). A continuous port can either be an incoming port or an outgoing port, but never both at the same time. Hybrid port are used in discrete components to access the value of a continuous component or its continuous port respectively, like `velocityR` on component `overtakerDriver`. It specifies no behavior, but the referenced value can be accessed by the behavior of the component and all discrete ports.

When all components are specified, the developer can configure an instance of the software system. Such a configuration is called Component Instance Configuration (CIC). It contains instances of components, their ports, and the connection of the ports. The CIC in Figure 1.3 on page 5, for example, is an instance of the component `overtakerVehicle` in Figure 2.7.

2.5.2 Real-Time Coordination Protocols

RTCPs define a contract between two communication partners, i.e., between connected ports. Figure 2.8 shows the RTCP `Delegation`, which is applied to the ports `initiatorP` and `executorP` in Figure 2.7.

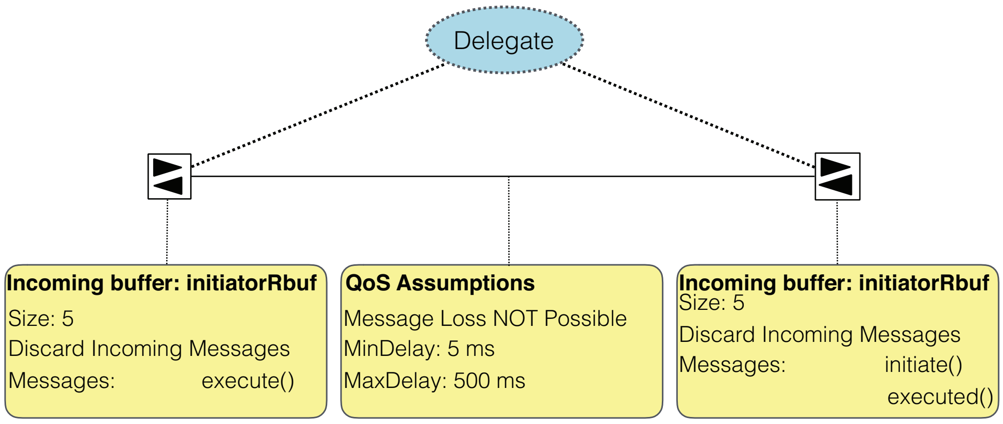

Delegate
Incoming buffer: initiatorRbuf
Size: 5
Discard Incoming Messages
Messages: execute()
QoS Assumptions
Message Loss NOT Possible
MinDelay: 5 ms
MaxDelay: 500 ms
Incoming buffer: initiatorRbuf
Size: 5
Discard Incoming Messages
Messages: initiate() executed()

Figure 2.8: Real-Time Coordination Protocol Delegation and its Defined Properties.

For each communication partner, a role is defined. Each role has a defined behavior, specified by an RTSC. If an RTCP is applied to ports of components, the behavior of the port has to refine the role behavior correctly. Additionally, each role defines incoming and outgoing message buffers and dedicated message buffers to store incoming messages. Furthermore, beside other QoS assumptions, the protocol defines a *min-delay* and a *max-delay*, which specify the time interval in which the message has to be transmitted.

2.5.3 Real-Time Statecharts

RTSCs are used to specify the behavioral models in MechatronicUML. They are based on Harel's statecharts [Har87] and UML state machines [Obj11b] and are enriched by timing features as used in timed automata [AD94], like clocks, clock constraints, and state invariants. States represent "certain situations within the system" [BDG+14a] and can be changed by firing a transition. In an RTSC there is exactly one active state at any time. Figure 2.9 shows an excerpt of the RTSC of the component `overtakerCommunicator`. For better readability, we show only two regions. Figure B.6 in Appendix B.1.3 shows the complete diagram. A state

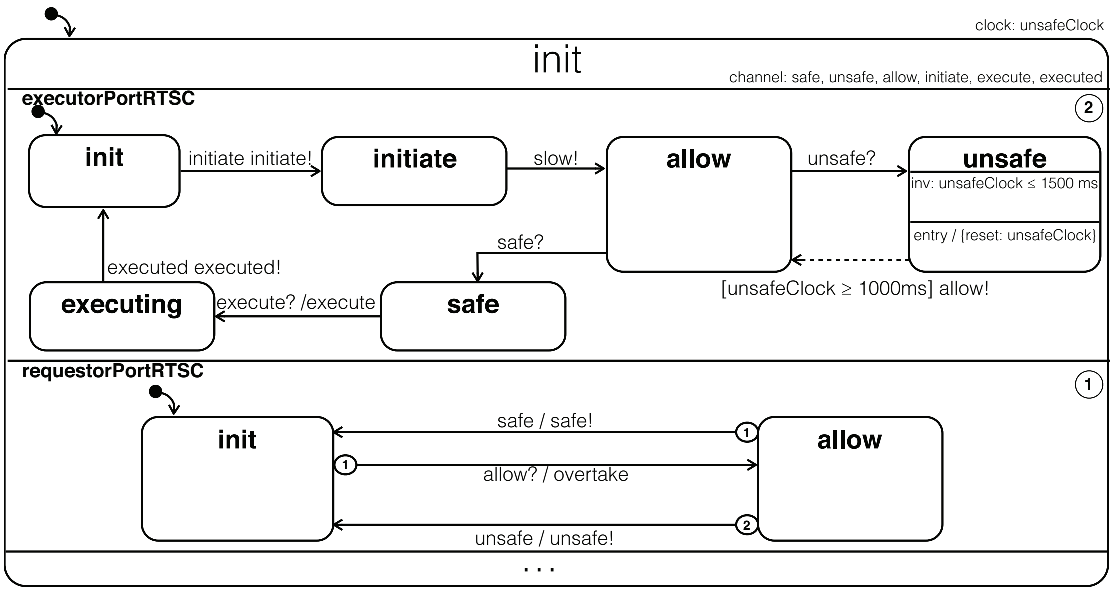


Figure 2.9: Excerpt of the RTSC of Component "overtakerDriver".

can have state events that are executed when the state is entered (entry event), not left at evaluation (do event), or left respectively (exit event). Each event can execute an action (cf. 2.5.3.4) and reset clocks (cf. 2.5.3.1). For example, in the shown RTSC, an entry event is defined for the state `unsafe` for the region `executorPortRTSC`. In this event, the clock `unsafeClock` is reset. The behavior of an action is implemented in the *Action Language* and can access variables and clocks of the RTSC. Additionally, a state can define a state invariant (cf. 2.5.3.3), which is used to enforce a state change until a defined point in time.

Furthermore, according to statecharts, RTSCs support hierarchical states. A hierarchical state embeds one or more RTSCs and is called *composite state*. A composite state has regions that contain RTSCs again. If a state embeds more than one RTSC, it is called *orthogonal*. Each region has a priority that is used to define a sequential execution order of the regions to ensure a deterministic execution. Both shown regions do not contain any composite state. The state `init`, which contains all regions is a composite state, or to be more precisely an orthogonal state.

The currently active state can be changed by a transition from source state to target state. A transition is enabled and may fire, if all conditions, like guard and clock constraints, are fulfilled. Of course, an implicit but essential condition is that the source state is the currently active state of the RTSC. Furthermore, a transition may declare additional conditions, like a *synchronization channel* (cf. Section 2.5.3.6) and a *trigger message event* (cf. Section 2.5.3.5). All outgoing transitions for a state define a unique priority. When evaluating the transitions of the currently active state of an RTSC, the transitions are evaluated in the order of their priorities. For example, if the RTSC of region `requestorPortRTSC` is in state `allow`, the transition with priority `2` is evaluated at first. Only if this transition cannot fire, the transition with priority `1` is evaluated and possibly fired. If a transition fires, several effects will be executed [BDG$^+$14a]:

State Deactivation: The currently active state of the RTSC is set to inactive. Thus, at this point in time, no state of the RTSC is active. Furthermore the effects of the exit events of the left state are executed.

Transition Effects: Effects of the transition are executed. If a transition defines a *trigger message* in the transition conditions, this message is consumed by the transition, i.e., it is taken from the message buffer. Additionally, the action of the transition is executed. This action may access the parameters of the consumed trigger message. Furthermore, all clock resets are executed. Executing effects are assumed to be not interrupted.

State Activation: Finally, the target state is set to be the active state of the RTSC. Furthermore, effects defined in the entry event are executed.

A transition can furthermore define a deadline. All of the three steps and their sub-steps have to be finished until this specific deadline. In this thesis, we assume that all three steps are executed atomically and, therefore, cannot be preempted.

2.5.3.1 Clocks

Clocks represent the time passing of the system. A clock has a time value and a fixed time unit. It is assumed that all clocks are in synchronization, i.e., the time passing is synchronously for all clocks. Every RTSC can specify clocks that can be accessed from events, actions, and clock constraints of all states and transitions of this RTSC. Additionally, it can be accessed by all child RTSCs, i.e., by states and transitions of the RTSCs of all regions of composite states. By *accessed* we mean the clock is read (for example in clock constraints) or written, i.e., the clock is reset to 0 – setting the clock to a specific time value is not possible. For example, the clock `unsafeClock` has the time unit `ms` and is defined in the RTSC `overtakerCommunicatorRTSC`. Note, that the clock is used in the RTSCs of the embedded regions.

2.5.3.2 Clock Constraints

A clock constraint defines a constraint over a clock of the RTSC. It refers to a clock, defines a comparison operator ($<, \leq, >, \geq, =, and \neq$), and an expression to that the clock is compared. This expression can be a simple number, but can also be defined in the action language and, therefore, can use other variables or even values of hybrid ports as well. A clock constraint can be used at a transition as part of the enabling conditions and as a condition for state invariants. Both situations are used in the example RTSC: In the state `unsafe` of region `executorPortRTSC` an invariant is defined and at the outgoing transition a clock constraint is specified, such that the transition can only be fired if the value of the clock `changeClock` is greater than `1000 ms`.

2.5.3.3 Invariants

An invariant contains a non-empty set of clock constraints (cf. 2.5.3.2) which all have to be fulfilled whenever the state is active [BDG$^+$14a]. Or, to put it another way, the state has to be left before the invariant gets invalid. In the example, for the state `unsafe` an invariant is defined, i.e., the state has to be left before the clock `unsafeClock` has reached `1500ms`. In the verification step (cf. 2.5.4), it is assumed that no invariant will be violated at runtime. This assumption is crucial, since there is no behavior defined for the case that an invariant is violated. Thus, the verification of all safety properties that are based on such an invariant would be invalid if an invariant would be violated a runtime.

2.5.3.4 Actions and Operations

In state events and when firing a transition, an action can be executed. The implementation is done in the *Action Language* provided by MechatronicUML. Using this language, values of hybrid ports and accessible variables can be read or written. The execution of an action can "neither be canceled nor interrupted" [BDG$^+$14a].

Furthermore, operations can be defined for each RTSC. Comparable to functions in programming languages, an operation may have parameters and a return type. The implementation can also be specified in the *Action Language*.

2.5.3.5 Messages

RTSCs that represent the behavior of a role or a port can send and receive corresponding messages. In particular, receiving a message is modeled at a transition and is part of the transition conditions. Thus, the transition *waits* until a specific message is received by the port and available in the message buffer. For example, in Figure 2.9, the transition from state `init` to state `initiate` in region executorPortRTSC is enabled only, if a message of the type initiate is available in the message buffer of the port. Analogous, when a transition fires, a trigger message can be defined, i.e., a message is sent. It is assumed that handling and routing a message is done by a middleware service and has not to be considered by the component.

2.5.3.6 Synchronization Channels

Synchronization channels are used for modeling transitions that fire simultaneously. A synchronization channel can be accessed by all RTSCs that are embedded in the composite state that defines the channel. In the example in Figure 2.9, several synchronization channels are used, e.g., initiate, allow, or safe. Each transition may declare one synchronization channel. Either as *sending* channel (indicated by *!*) or as *receiving* channel (indicated by *?*) We define a transition as a *partner-transition* of a transition, if the following condition holds: The transition defines a *sending* synchronization channel, the *partner-transition* defines a *receiving* synchronization of the same channel and vice versa, and both transitions are not contained in the same RTSC. For example, in Figure 2.9, the transition from state init to state allow in region requestorPortRTSC and the transition from state allow to state safe in region executorPortRTSC are partner transitions using the channel safe.

Two transitions fire simultaneously if both transitions are partner transitions and both transitions are enabled. Firing these transitions, will first execute the effects of the *sending* transition and then of the *receiving* transition. Note that deadlines of both transitions have to be respected. Furthermore, we assume that the execution of both transitions is atomic and cannot be preempted.

2.5.3.7 Urgency

We distinguish between *urgent* and *non-urgent transitions* [BDG^{+}14a]. An urgent transition fires immediately when it is enabled, i.e., there is no time passing between enabling and firing. Non-urgent transitions may postpone the firing. In Figure 2.9 on page 24, the transition from state unsafe to allow is a non-urgent transition (indicated as dashed arrow). Hence, this transition can fire when the clock unsafeClock is in the interval $[1000ms, 1500ms]$.

2.5.4 Verification Approach

An advantage of MDSD is that formal verification techniques can be applied in an early stage of development using model checking [CGP00]. MechatronicUML provides a verification approach on PIM level to ensure safety requirements [DGH15]. The domain-specific language MTCL is used for the formal specification of the properties. MCTL is based on TCTL (Timed Computational Tree Logic) [ACD93] and allows to refer to concrete model elements in MechatronicUML models [DGH15].

One problem in model checking is the *State Explosion Problem* [BK+08], which describes the fact that the state space that have to be explored gets large even for small models. Thus, applying model checking to a complete MechatronicUML model is not advisable. Hence, a compositional approach for model checking is applied in MechatronicUML [GTB+03, HBDS15]. For this, the RTSCs are translated into timed automata and the model checker Uppaal [BLL+96] is used for the verification. First, verification requirements are specified in MCTL for each RTCP, e.g., that no deadlock can occur. After that, a refinement check is used to verify that the behavior of the port refines the role behavior of the RTCP correctly. In a third step, verification on the level of the component is applied to ensure that no deadlock will occur [GTB+03].

2.5.5 Platform-Specific Modeling

MechatronicUML provides a process for creating a platform-specific model. It requires the PIM and the PDM as input. The PDM describes the target platform of the system by specifying hardware, like ECUs and communication channels and system properties like the operating system [Dan13]. Figure 2.10 shows the process in detail.

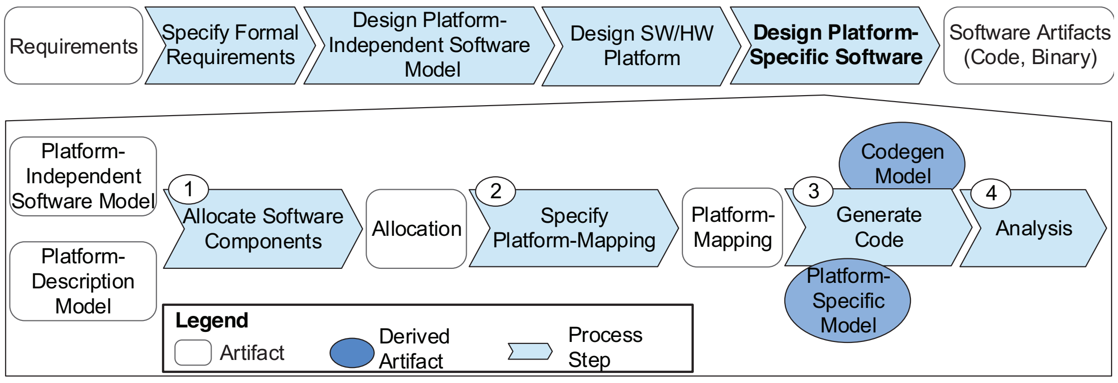

Requirements
Specify Formal Requirements
Design Platform-Independent Software Model
Design SW/HW Platform
Design Platform-Specific Software
Software Artifacts (Code, Binary)
Platform-Independent Software Model
Platform-Description Model
1
Allocate Software Components
Allocation
2
Specify Platform-Mapping
Platform-Mapping
Codegen Model
3
Generate Code
Platform-Specific Model
4
Analysis
Legend
Artifact
Derived Artifact
Process Step

Figure 2.10: MechatronicUML Process and Sub Process for the Step “Design Platform-Specific Software”. [BDG+14b]

First, in **Step 1**, the software components specified in the PIM are allocated to ECUs specified in the PDM. This allocation is done semi-automatically. On the one hand, an allocation has to fulfill constraints regarding the memory, e.g., that the memory consumption of all software components is not higher than the size of the ECU memory, which can be derived automatically. Furthermore, the allocation engineer may define so-called *allocation constraints* using a domain specific language, the *Allocation Specification Language (ASL)* [PH15] to specialize the allocation. For example, using such constraints, the allocation engineer may specify that two specific software components have to be allocated on the same ECU. For referencing software model and hardware model in the constraints, the ASL specifies a context model. This model is called *OCL-Context* and referes to the PDM (represented by a Hardware Platform Instance Configuration (HPIC) in MechatronicUML) and to the PIM (represented by a CIC)). Thus, in the ASL-constraints elements of the HPIC and the PIM can be referenced. The ASL provides four different kinds of allocation constraints [PH15]. In this thesis, we only use the constraint kind *Required Resource*. The actual implementation of the allocation constraints is done in OCL. After the specification of all constraints, a corresponding ILP is created, solved, and translated back to MechatronicUML. The result is an *Allocation* that describes the allocation of software component instances to ECUs that satisfies all defined constraints.

In **Step 2**, a *platform-mapping* is created. In this mapping, continuous components of the PIM are connected to sensors and actuators of the PDM. In particular, in this step the actual implementation of the sensor access has to be set. Using the PIM, the allocation, and the platform-mapping, a *Platform Specific Model* (PSM) is created. The PSM is used as input for the source code generation since it contains all information needed by a code generator. After that, in **Step 3** source code is generated. The current tooling of MechatronicUML provides generating C code. In the last step (**Step 4**), analysis of the executed software can be applied.

2.6 Amalthea

In this section, we describe parts of the Amalthea project [AMA] and tool platform that are important for this thesis since we reuse some parts in our approach. Amalthea is a research project founded by ITEA 2 providing a method and a tool platform for automotive CPS following the MDSD paradigm. It aims to be compatible with the AUTOSAR standard and to support the development for multi- and many-core environments. Hence, software and hardware meta models are similar to the AUTOSAR specifications [Ama13c]. The development process covers several steps starting with requirements engineering and ending with the deployment of the software. Figure 2.11 shows the development process steps defined by Amalthea.

In the first step (RE), requirements for the system are defined. In a next step, the *variability modeling* (VM) takes place. Here, possible variants of software and

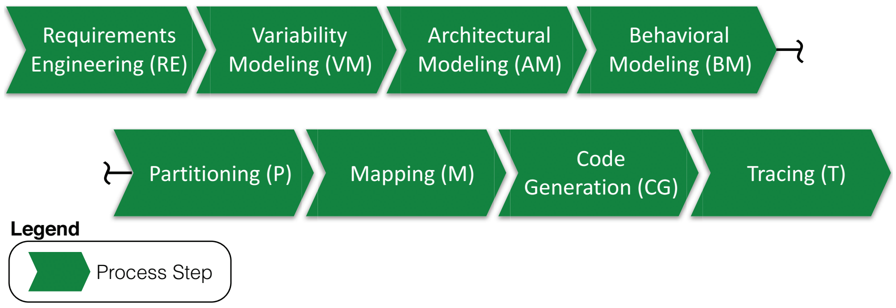


Figure 2.11: Development Process of Amalthea [Ama14]

hardware are designed by specifying feature diagrams. Next, in the *Architectural Modeling* (AM) the structure of the software system is designed, i.e., software components and their connections are modeled. After that, in the *Behavioral Modeling* (BM) the behavior models that describe the behavior of the software components are specified. Here, Amalthea does not stick to one specific modeling but enables the developers to use models of their choice, e.g., Matlab Simulink or even manually written C code. In parallel, a hardware description model is specified to describe the target platform [Ama14].

Then, the *Partitioning* (P) algorithm is applied to separate the software into system tasks. These tasks are mapped to ECU cores in the *Mapping* (M) step. In the end, source code for each ECU is generated (CG). However, the code generation is not mature and does not cover the generation of behavioral models by default. External commercial tools like a generator provided by Yakindu for state machines can be applied. Additionally, Amalthea provides trace formats that can be used to trace the execution of the software at runtime and use this information for refinement of the system specification and partitioning (T).

In the following sections, we describe the models and the parts of the process steps that are relevant for this thesis in more detail. In Section 2.6.1, we give an overview of the models provided by Amalthea that we use in this thesis. After that, in Section 2.6.2, we present the partitioning and mapping concepts and implementations. Finally, in Section 2.6.3, we shortly show the simulation and analyses concepts.

2.6.1 System Specification

Amalthea provides several models and views for describing hardware, software, variants, and analysis of the system [Ama13c]. However, in this thesis we don't use all kinds of models since some models are not relevant for the topic of this thesis, e.g., the variant modeling. In this section, we give an overview of the model (elements) that are relevant for this thesis.

2.6.1.1 Platform Modeling

Amalthea provides a detailed hardware description model to specify all hardware parts of the system [Ama13b]. The hardware is indeed involved in the execution and provides important parameters for the partitioning and mapping [KK13]. In this thesis, we do not focus on the modeling of hardware but of software. We assume that the hardware engineers provide the hardware model. For further information to the hardware description model, we refer to the specification in [Ama13b].

Beside the hardware, a platform provides an operating system. Amalthea defines a dedicated *OS-model* [Ama14] for the operating system. It mainly describes the scheduling environment for each ECU core that is defined in the hardware model. Each Scheduler defines an ECU core that contains this scheduler. Furthermore, the scheduler defines a *scheduling strategy*, which is used at runtime. Amalthea provides several possible scheduling strategies, e.g., Earliest Deadline First (EDF) [But11] or OSEK-compliant priority-based scheduling. All provided strategies can be found in [Ama13c]. Thus, it is possible that each core uses a different scheduling strategy. The current mapping algorithm uses priority-based scheduling for multi-core ECUs and EDF scheduling for single-core ECUs (cf. Section 2.6.2).

2.6.1.2 Software Modeling

Amalthea is compatible to the AUTOSAR standard [Ama12]. Thus, their software models are related. In Amalthea, the software is described by *runnables* [Ama13c]. Corresponding to AUTOSAR, a runnable is an executable entity, which is executed by system tasks. A runnable contains the instructions that describe the behavior that have to be executed. At model level, the runnable is an abstract representation of this behavior. Similar to a task, each runnable has to provide a WCET and an activation kind, and may define a relative deadline. In Amalthea, the WCET is described by the amount of instructions of the runnable. Similar to tasks, the activation kind can be sporadic (event-triggered) or periodic (time-triggered). In this thesis, we only use periodically activated runnables. The relative deadline defines the point in time until executing the runnable has to be finished. Additionally, a runnable specifies all accesses to variables (called *labels* in Amalthea) and semaphores. For this, each runnable defines *label accesses* and *semaphore accesses*. In this thesis, we use only *label accesses* in the specification. For labels, we distinguish between *read label accesses* and *write label accesses*. Beside implicit dependencies defined by label accesses, constraints between runnables can be specified. These runnable constraints describe relation of two (or even two groups) of runnables. In this thesis, we only use the *runnable sequencing con-*

straint. This constraint specifies, that a runnable has to be finished before another runnable can start.

In our approach, we use label accesses and the runnable sequencing constraint to preserve the semantics of MechatronicUML when executed by runnables. In the following, we describe label accesses and runnable sequencing constraints in more detail.

Read/write accesses describe the kind of access of a runnable to a dedicated variable. These dependencies might be needed to avoid read/write conflicts for runnables that are executed in parallel. Figure 2.12 shows a read access of `R1` (red dashed arrow) and a write access of `R2` and `R3` (red solid arrow) to variable `X`. Thus, `R2` and `R3` have conflicting write accesses. This conflict can be solved for

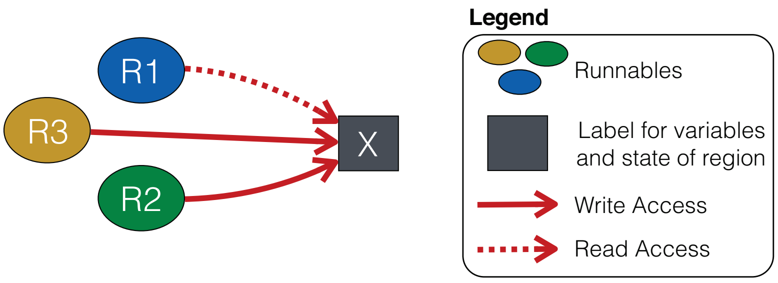
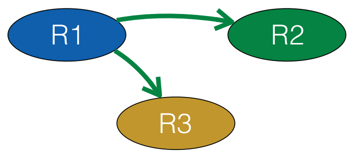


Figure 2.12: Read/Write Accesses to the Variable X.

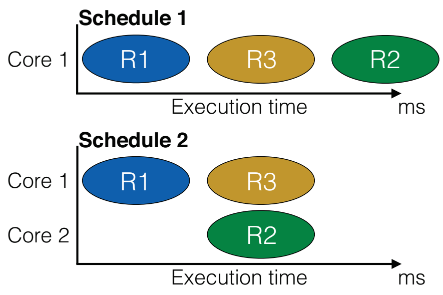
example by ensuring that `R2` and `R3` are not executed in parallel. In this thesis, we use this graphical syntax for read/write accesses and sequencing constraints.

Runnable sequencing constraints describe a dependency in the execution order of two (or more) runnables. Figure 2.13a shows a graphical representation of the *runnable sequencing constraint* for the runnables `R1` and `R2` and `R1` and `R3` respectively (green arrows). Figure 2.13b shows two feasible schedules for the runnables. `Schedule 1` shows an execution on a single-core ECU, `Schedule 2` on a multi-core ECU. Note that in `Schedule 2` two ECU cores are used but the execution of `R2` is delayed until `R1` is finished, although `Core 2` does not execute other code. Thus, such constraints can result in execution gaps and prohibit fully utilization of the processors.

Furthermore, tasks can be specified for the execution of the software. Each task executes a set of runnables in a specific order. Which runnables are executed by which task and in which order can be computed automatically in the *partitioning* (cf. Section 2.6.2). This computation is based on the runnable properties. Therefore the task properties (WCETs, periods, and deadlines) are derived from the runnable properties. At runtime, each task is executed by a scheduler. This is assignment is also computed automatically in the so-called *Mapping* (cf. Section 2.6.2).



(a) Runnable Dependency Graph.



(b) Resulting Feasible Schedules.

Figure 2.13: Runnable Sequencing Constraint and the Corresponding Feasible Execution of the Runnables.

2.6.2 Partitioning and Mapping

When software and platform are specified, the *partitioning* and *mapping* can be applied to compute an execution order for the runnables automatically that respects all dependencies of the runnables [Ama13a]. In the following, we describe both steps in more detail because we use them to determine a multi-core schedule. First, in Section 2.6.2.1, we describe the Partitioning, where runnables are grouped to tasks. After that, in Section 2.6.2.2, we describe the mapping, where these tasks are mapped to ECU cores (their schedulers respectively).

2.6.2.1 Partitioning

In the partitioning, a given set of runnables with specified runnable properties and runnable dependencies is separated into sub-groups of runnables that can be executed by one task. Using the runnable dependencies, a directed graph can be constructed that express these dependencies. AMALTHEA provides a partitioning approach that addresses "minimal sequential runtime" of the tasks (the executed runnables respectively) and "minimal communication" between the tasks [FHKK], where communication means the use of common labels. Several analyses are used to determine a feasible partitioning. These analyses and their execution order are depicted in Figure 2.14.

In the following, we describe each of the analyses shortly [Ama14]:

Activation Analysis: Runnables of a software system can have different activation times. Similar to tasks, a runnable can be activated sporadically or periodically. We focus on periodic activations only. In the Activation Analysis, runnables are grouped by their activation times. In Figure 2.14, this is indicated by different colors.

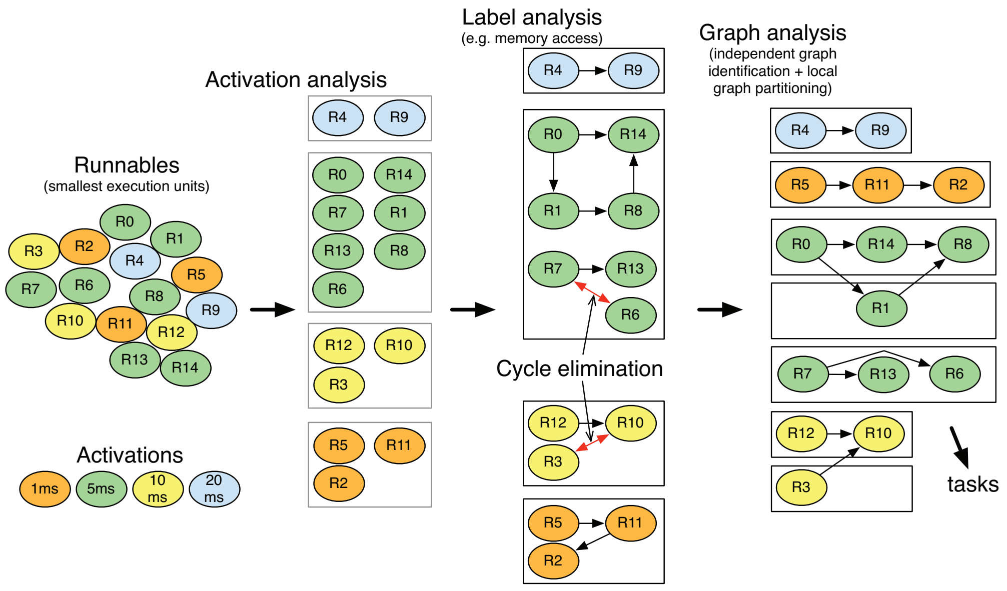


Figure 2.14: Analyses within the Partitioning of Amalthea (Source: [Ama14])

Label Analysis: In this step, read/write accesses to labels are analyzed. If two runnables have access to the same label, a *runnable sequencing constraint* is specified. In the example, black arrows from one runnable to another indicate these constraints. Such constraints describe, that both runnables cannot be executed in parallel.

Cycle elimination: If two runnables have write access to the same label, cyclic constraints can occur. In the figure, red arrows indicate such cycles. Since further computations for partitioning and mapping need *acyclic* graphs, such cyclic runnable sequencing constraints are resolved by converting an access precedence. In Amalthea, this step is processed automatically. Thus, after the cycle elimination, further graph analyses can be performed since the output is an acyclic graph.

Graph Analysis: At the end, an analysis on the graph is performed to find independent sub graphs. Such independent graphs can then be used to specify task prototypes that are transformed to tasks in a later step. Each task executed the runnables of the corresponding sub graph, such that all sequencing constraints are fulfilled.

2.6.2.2 Mapping

After the partitioning, the computed tasks have to be assigned to executing cores, such that all runnable dependencies and runnable properties are satisfied. This step is called *mapping* and known to be NP-hard [Fos95]. For single-core environments, there are several well-elaborated strategies to find a feasible scheduling, e.g., applying the EDF algorithm. For multi-core environments, this task gets more complex since all runnable dependencies have to be respected even if parallel execution of the runnables is possible. Furthermore, for large software systems a scheduling might have to fulfill some additional optimization goals, e.g., to minimize the energy consumption.

The mapping approach in Amalthea does currently support the optimization goal regarding energy consumption. The implementation is based on an approach by Zhang et al. [ZHC02]. For this, the mapping algorithm utilizes the runnable model that contains all runnables and corresponding constraints and the hardware model that describes the target ECU. Currently, the mapping approach does only consider one ECU. The support for ECU networks will be developed in the successor project Amalthea 4public.

If the ECU contains only one core, according to [ZHC02], the EDF scheduling strategy is applied. For ECUs with several cores, a priority-based strategy is used. Any mapping, found by this approach is a feasible scheduling and is schedulable by the target platform. This topic is currently under development and further optimization goals, like load balancing, will be usable in Amalthea 4public.

2.6.3 Simulation and Analysis

Amalthea provides additional features for analyzing of the software in simulation and during execution on hardware. It provides two trace formats: BTF (Best Trace Format) and HTF (Hardware Trace Format). These trace formats can be used to log and analyze the execution of the software, e.g., by using the commercial tool *Timing Architects Tool Suite* [Arc]. Additionally, Amalthea provides an export mechanism to this simulation tool. The BTF format is also readable by the non-commercial tool *Trace Compass* [Pol], which is available as Eclipse plugin.

Furthermore, Amalthea provides the possibility to define specific events and specify timing constraints for these events. For this, the existing domain-specific language Timing Augmented Description Language (TADL) [PFKH$^+$12] has been adapted. Using such timing constraints and the traced software, timing analyses can be applied to verify timing constraints during the execution. For further information, we refer to the specification of Amalthea [Ama13c].

3 Process and Contributions

In this thesis, we present an approach for executing software that is developed with the MechatronicUML method on multi-core platforms. To deploy the software specified in the PIM to a multi-core environment, we extend the sub process of the process step `Design Platform-Specific Software` of the MechatronicUML development process [BDG+14a] (cf. Figure 2.10). In the following, we call this process *PSM process*. Currently, in the PSM process, a task is created for each component instance of the PIM and allocated to an ECU. Timing properties are not considered. Hence, period, deadline, and WCET of the tasks are derived manually. However, since we are going to deploy the software to a multi-core platform in a semantic preserving way, additional steps are needed to enable parallel execution and to ensure timing constraints of the PIM. In this chapter, we give an overview of the complete process that is used to determine a multi-core scheduling for MechatronicUML. In particular, we discuss the integration of additional steps into the current PSM process and how existing steps have to be adapted. Figure 3.1 shows all steps of the new PSM process that are in the scope of this thesis.

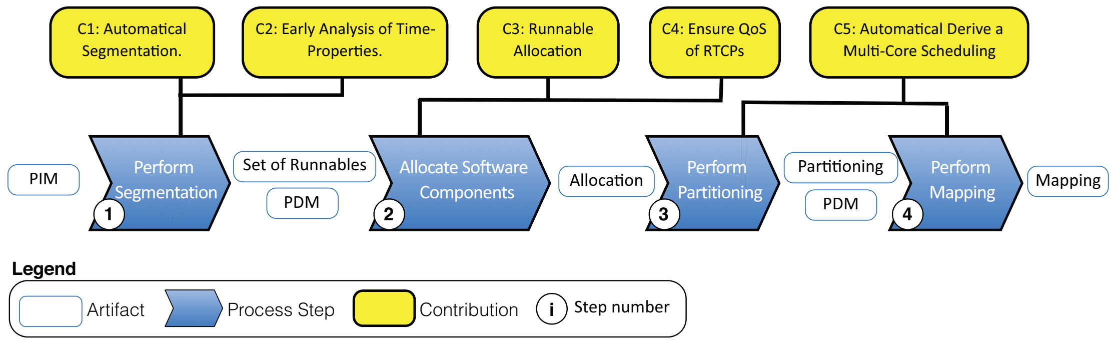


Figure 3.1: Development Process for Deriving a Multi-Core Scheduling.

For the new PSM process, we define new process steps (**1**, **3**, and **4**) and extend an existing step (**2**). We assume that PIM and PDM are already defined at this point of development and can be used as input artifacts for the process steps. The PIM defines a CIC and the PDM defines a platform specification. Since we are going to use methods of MechatronicUML and Amalthea where the PDM is used as input, we do not stick to one specific meta-model for the PDM, but assume that the PDM can be transformed from one to the other meta-model without losing information or changing the semantics, e.g., by a horizontal and exogenous model transformation (cf. Section 2.1.2). In the following, we describe each process step, the order of the steps in the process, and their input and output artifacts. After that, we state the contributions of our thesis for each process step.

3.1 Process Steps

At first, **in Step 1**, the software is separated into smaller pieces that can be executed independently. We call this step *Perform Segmentation.* The input for this step is the PIM of the software that is described by a CIC. The segmentation describes the separation of software into independently executable parts. According to the AUTOSAR standard, we call these parts *runnables.* For each runnable, runnable properties are defined (WCET, period, and deadline). Additionally, dependencies and constraints between the runnables are defined, e.g., accesses to shared variables. Furthermore, each runnable has a reference to its behavioral element in the PIM. This reference is not mandatory for the segmentation, but it is useful for later steps, e.g., allocation, code generation, or analyzing the execution. The output artifact of this step is a set of runnables with defined runnable properties.

In **Step 2** the software is allocated to ECUs, i.e., we define which runnables are executed on which ECU. This step is called *Allocate Software Components.* Hence, the input for this step is a set of runnables (produced by the segmentation) and the PDM. Since an allocation can affect the communication latencies, we ensure in this step that all QoS assumptions of the used RTCPs (cf. 2.5.2) are respected. The output artifact of this step is a *runnable allocation* that describes the allocation of runnables to ECUs.

In **Step 3**, we define for each of the ECUs which runnables can be executed by one task. According to Foster [Fos95], this step is called *Perform Partitioning* (cf. Section 2.3). Thus, the input for this step is a set of runnables per ECU. Then a partitioning is determined for each set of runnables. A partitioning is a set of tasks, which execute the runnables defined in the input. Thus, the output artifact of this step is a partitioning for each ECU, i.e., a set of tasks per ECU.

In **Step 4**, we define for each ECU which task is executed on which ECU core. This step is called *Perform Mapping* [Fos95] (cf. Section 2.3). The input for this step is the partitioning for each ECU. In the mapping, each task gets assigned to an ECU core that will execute this task at runtime. The mapping has to respect dependencies between the tasks and their runnables. The output of this step is a mapping of tasks to ECU cores.

Finally, the remaining steps of the MechatronicUML PSM process, which are not in the scope of this thesis, can be applied, i.e., steps for *Platform Mapping*, *Code Generation*, and *Analysis* (cf. Figure 2.10 on page 28). In the *platform mapping*, hardware abstractions in the PIM, like continuous components, get enriched by concrete platform dependent parts, e.g., concrete API calls for sensors and actuators. Afterwards, the newly created PSM, provided for each ECU, is transformed into source code for the target platform and compiled to an executable. Finally, the software can be executed and logged to analyze the execution on the target platform. For this, so-called traces can be used for further analyses of the software system, e.g., by using methods of Amalthea (cf. Section 2.6.3).

3.2 Contributions of this Thesis

The main goal of this thesis is to enable multi-core execution of software specified in MechatronicUML. Hence, defining the extended PSM process is the first contribution. In Chapter 4, we propose concepts for deriving a multi-core scheduling for one multi-core ECU. Hence, in this chapter Steps 1, 3 and 4 are concerned. After that, in Chapter 5 these concepts are used to extend the process for a system with several multi-core ECUs. Thus, in this chapter mainly Step 2 is concerned. In the following, we state our main contributions to each step.

The current PSM process does not provide a segmentation. Besides adding a segmentation step to the process, we state the following contributions:

C1 - Automatical Segmentation Approach: In this thesis, we discuss different strategies for an automatic segmentation for one multi-core ECU. Thus, the software, represented by a CIC, is analyzed for possible parallel execution automatically. Furthermore, we provide methods to derive runnable properties automatically, which ensure a semantic preserving execution of the software.

C2 - Early Analysis of Time Properties: We discuss how the segmentation can be used to find problems in the software on PIM level, e.g., by analyzing timing properties in comparison to the WCET and period of the runnables. Hence, we allow an early detection of problems regarding the execution.

In Step 2, currently only component instances are considered in the allocation. In our approach, we extend this step by enabling the allocation of runnables, or to refer in allocation constraints to runnables respectively. Furthermore, we provide a method to ensure the QoS of RTCPs during the allocation. Hence, we state the following contributions:

C3 - Runnable Allocation: We enable the use of runnables in the existing allocation step of the PSM process. In particular, we enrich the PIM that is referenced in the allocation by runnables automatically. These runnables can be used to formulate constraints regarding the allocation.

C4 - Ensure QoS of RTCPs: Using runnables in allocation constraints, we provide a technique to derive allocation constraints automatically that ensures the QoS assumptions of the RTCPs in the allocation step, i.e., we ensure that the max-delay of all RTCPs is respected in the allocation.

Having a set of runnables per ECU, Step 3 and 4 are used to determine a feasible set of tasks and a schedule for each ECU. We propose to apply existing algorithms for this purpose. In particular, we transform the MechatronicUML models into the meta-model of Amalthea [AMA] and use existing approaches for partitioning and mapping [Ama13a]. Hence, we state the following contribution:

C5 - Derive a Multi-Core Scheduling: We enable determining a partitioning and mapping for the set of runnables automatically by translating the runnables to the meta model of Amalthea and applying existing algorithms for partitioning and mapping for multi-core environments.

The PSM process provides three additional steps (platform mapping, code generation, and analysis) that are not in the scope of this thesis. However, these steps might also profit by the extensions done to the former steps. For example, Amalthea provides trace formats for tracing the execution of the software in real-time simulations or on hardware. Additionally, imports and exports for mature analysis- and optimization-tools are available and since we provide a transformation from MechatronicUML to Amalthea these tools can be used for analyzing the software. Both trace formats support the tracing of runnables and since we specify a reference from each runnable to its corresponding element in the PIM, this analysis can be used for improving the PIM in MechatronicUML. Thus, our approach can be used to improve the Analysis-Step of the PSM process.

4 Multi-Core Execution for MechatronicUML

In this chapter, we present a systematic method to derive a multi-core scheduling for MechatronicUML models. At this point of this thesis, we assume that the whole modeled software system is supposed to be executed on *one* multi-core ECU. Furthermore, we do not consider communication explicitly. Later, in Chapter 5 we present an approach how this method can be applied to systems that use ECU-networks and consider communication of component instances. Hence, we focus on the segmentation step (Step 1 in Figure 3.1 on page 37) in this chapter. Additionally, we discuss how partioning (Step 3) and mapping (Step 4) can be applied. Step 2 (Allocation) and its integration into the approach of this chapter is further discussed in Chapter 5.

Figure 4.1 shows the subset of the process presented in Chapter 3 with the relevant steps for this chapter, i.e., Step 1, 3, and 4. In the following, we explain each process step in more detail. We assume that the PIM as well as the PDM are defined and use both models as input for our process. Since we use MechatronicUML for defining the PIM in this thesis, we assume that the PIM is represented by a CIC. Since we derive runnables in Step 1 and a mapping in Step 4, the results of this process are first runnables with defined runnable properties and runnable dependencies for the given PIM, and second a mapping (cf. 2.6.2.2) of tasks to ECU cores that execute the behavior of the PIM, such that the semantics of MechatronicUML are preserved.

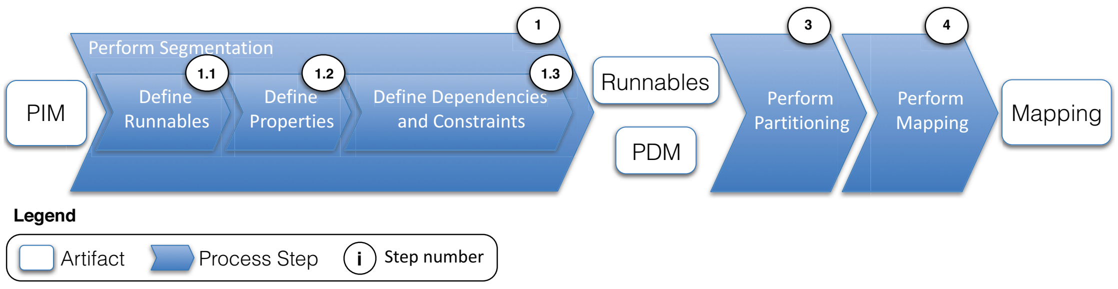


Figure 4.1: Detailed Development Process for Deriving a Multi-Core Scheduling for One Multi-Core ECU.

We first have to apply a segmentation, where we separate the software specified by the PIM in independently executable parts, so-called *runnables* **(Step 1.1)**. For each runnable, we have to define runnable properties. The runnable properties are WCET, period, and deadline **(Step 1.2)** (cf. Section 2.2.3). Since runnables are abstract descriptions of executable behavior and classified by their properties, these properties are essential for determining a feasible scheduling. Additionally, we have to specify possible dependencies of these runnables. Runnable dependencies are the use of shared variables and constraints regarding the execution order of runnables **(Step 1.3)**. Both might lead to a conflict if runnables are executed in parallel on different cores. Hence, runnable dependencies and runnable constraints are important information for determining a feasible scheduling.

After the segmentation, runnables have to be grouped and mapped to tasks without violating runnable constraints, runnable dependencies, or runnable properties. This is done in the *partitioning* **(Step 3)**. In a last step, we have to assign these tasks to cores and define a scheduling for each core of the ECU, such that all runnable dependencies are fulfilled at runtime. This is done in the *mapping* **(Step 4)**. In this thesis, we will utilize concepts, algorithms, and tooling of the Amalthea platform to compute a partitioning and a mapping [Ama13a].

Following these steps to derive a multi-core scheduling for MechatronicUML models, this chapter is structured as follows: In Section 4.1, we state requirements that have to be fulfilled by this approach. In Section 4.2, we describe different strategies for the segmentation of MechatronicUML models. Additionally, we state how runnable properties can be derived from the MechatronicUML models. In Section 4.3, we explain how the steps for partitioning and mapping can be performed using the algorithms of Amalthea and propose an approach for reducing the complexity of the input models for these steps. Finally, in Section 4.4, we discuss the different segmentation strategies and conclude this chapter.

4.1 Requirements

In this section, we state the requirements for the multi-core execution of software that is specified with the MechatronicUML method. In particular, we define four functional requirements. These requirements will be used as basis for the evaluation goals, which are defined and discussed in Chapter 8. We index the requirements as **R1** to **R4**, and refer in the ongoing thesis to these indices only. In the following, we describe each requirement shortly:

R1: Enable Parallel Execution: Multi-core environments can increase the performance of a system. However, software has to be separated into parts that are executable in parallel, if it shall be executed on a multi-core ECU. We defined runnables as smallest executable software units and therefore runnables are suitable for parallel execution. In our approach, we aim to provide parallel execution of the software to benefit from the multi-core architecture. Hence, the resulting mapping has to contain parallel execution of the derived runnables and tasks.

R2: Reduce amount of runnables and dependencies: Following **R1**, specifying as much runnables as possible increases the option for parallel execution of the software. Smaller runnables might have less read/write conflicts of variables with other runnables, but the number of runnable constraints rises to ensure the correct control-flow of the program. This increases the complexity for determining a feasible scheduling for the system: Since the computation of the partitioning mainly depends on the number of runnables, specified dependencies, and constraints, our approach has to generate as less runnables, dependencies, and constraints as possible without degrading **R1** significantly.

R3: Preserve Semantics: One important requirement is that all functional and non-functional requirements of the system are satisfied at runtime. On PIM level, model checking techniques are used to ensure the fulfillment these requirements. Executing the software on a platform adds additional parameters that have not been considered during the verification step on PIM level, e.g., the time for execution that is based on the frequency of the processor. Thus, a requirement for the resulting schedule is to ensure that the semantics of MechatronicUML are respected in the generated source code and during the execution on the target platform. In particular, the control-flow of the software has to be equivalent to the control flow specified by the MechatronicUML semantics. Furthermore, message exchange has to be respected regarding the QoS assumptions of the RTCPs. Additionally, timing requirements have to be considered. We state this in the separate requirement **R4**.

R4: Respect Timing Properties: Ensuring timing properties like invariants and clock constraints is important to ensure the semantics of MechatronicUML. Since we are going to provide a method to derive a multi-core scheduling for software with specified real-time behavior on an RTOS, we state this issue as a dedicated requirement. The provided method has to ensure, that all clock constraints, invariants, and deadlines are respected in the resulting scheduling.

4.2 Define Runnables

In this section, we present an appraoch for the segmenation of a CPS defined with MechatronicUML. Hence, starting point for the segmentation is a CIC. In general, we have to use the root-CIC of the system, because the root-CIC of a MechatronicUML model is the root for the complete software system. This CIC contains all component instances of the system, e.g., the complete *Overtaking With Approacher* system. The root-CIC of the running example contains component instances that will be allocated to different vehicles. Hence, it describes a distributed CPS with several ECUs. Thus, the root-CIC of the running example is not suitable for this section, because it is unreasonable to allocate components of the overtaker and components of the overtakee to the same ECU. In this chapter, we focus on the overtaker only and assume that it is deployed to one multi-core ECU. Hence, we use the CIC contained in component `overtakerVehicle` as root-CIC and assume that all contained component instances will be allocated to the same ECU. Figure 4.2 shows this CIC.

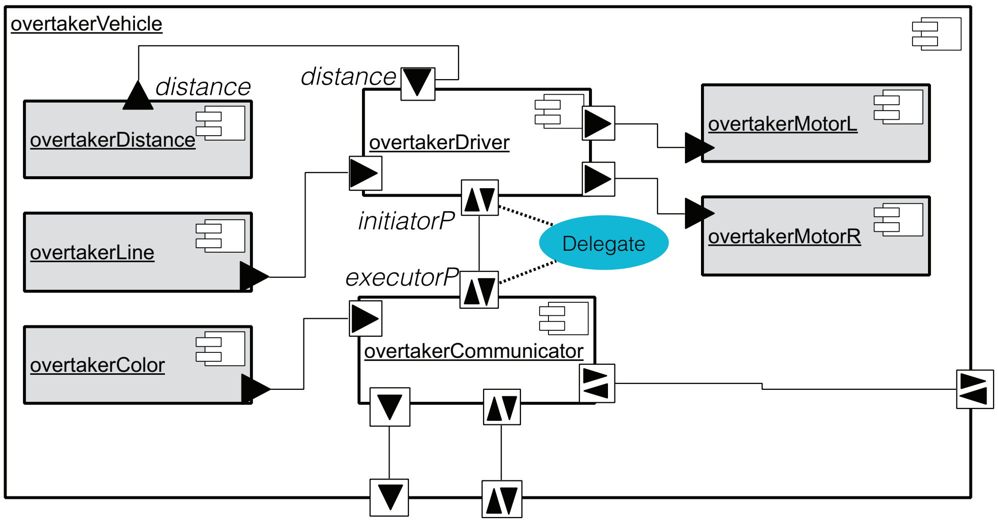


Figure 4.2: Structured Component Instance "overtakerVehicle" and its CIC.

For better readability, we ommit some identifieres of the elements and show only the name for each component instance and the names for relevant ports. Figure B.2 in Appendix B.1.1 shows the original diagram with all information. The CIC in Figure 4.2 contains two discrete component instances with specified behavior (white-filled) and five continuous components that represent sensor/actuator components (grey-filled). Each discrete component instance has hybrid ports to read sensor values and to write actuator values. Additionally, discrete ports are used for asynchronous communication, e.g., the port `initiatorP` of the discrete component instance `overtakerDriver`.

This section is structured as follows: Section 4.2.1 discusses segmentation approaches for discrete components like `overtakerCommunicator`. In Section 4.2.2 and 4.2.3, we present an approach to derive runnable properties for the runnables of discrete component instances. Since continuous components do not specify a behavior in form of RTSCs, but read and write values of sensors and actuators, we present an appropriate approach for the segmentation of continuous components in Section 4.2.4.

4.2.1 Segmentation for Discrete Component Instances

There are several possible abstraction levels to derive runnables for MechatronicUML models. In this section, we present three different approaches for the segmentation of MechatronicUML models that allow parallel execution of the PIM. These approaches are based on the requirements `R1` and `R2` as these two are mainly relevant for deriving runnables. On the one hand, we present an approach that is highly suitable for requirement `R2` since the number of generated runnables is low: *One Runnable per Component Instance*. On the other hand, we present an appraoch that is highly suitable for requirement `R1` since the possible parallel execution is high: *One Runnable per Transition*. Additionally, we present an approach that provides a good trade-off between the number of generated runnables and parallel execution: *One Runnable per Region*. For each approach, we derive resulting dependencies of the generated runnables and will discuss pros and cons related to the requirements stated in Section 4.1.

Each discrete component instance has to provide an RTSC that describes its behavior. Beside this RTSC there is no behavioral application code for the component instance itself. Therefore, we consider only the RTSC of the component instance in this section.

Each region of an RTSC has a priority to explicitly define an order for sequentially execution of all regions of the RTSC. This, however, prohibits parallel execution of the components behavior explicitly without having an advantage regarding the execution. In Appendix A, we discuss the fact that disregarding region priorities will not affect the execution order in a critical way. Hence, we propose to disregard region priorities and to change the semantics of MechatronicUML in the future.

In the following, we discuss three different approaches: In Section 4.2.1.1, we discuss the approach to generate one runnable per component instance. Next, we discuss the approach to generate one runnable for each transition in the RTSC in Section 4.2.1.2. Finally, in Section 4.2.1.3, we discuss the approach to generate one runnable per region of the RTSC.

4.2.1.1 One Runnable per Component Instance

Component instances are (corresponding to the definition of their component type) "[...] software entit[ies] with high cohesion" [BDG+14a]. Other components cannot access the behavior, but have to communicate via the defined ports of the component [Obj11b]. Therefore, a component instance can be seen as an independent part in the software system. The only dependencies between component instances result from connected ports. Communication via such ports can be implemented asynchronous. Thus, no dependencies are created for the communicating partners. Hence, component instances are suited for parallel execution and, therefore, we define one runnable per component instance. Each runnable executes the behavior of the corresponding component instance. This approach is similar to the currently used approach in MechatronicUML that was introduced in our former work [Gei13]. In this approach, each component instance is mapped to one task. In contrast to that, we now define one runnable per component instance, which allows a more sophisticated partitioning and mapping. Considering the example in Figure 4.2 on page 44, we get only two runnables: one for `overtakerDriver` and one for `overtakerCommunicator`.

Runnable Dependencies Since component instances have no access to the behavior of other component instances, they also cannot access variables or clocks of other RTSCs. The only possibility to communicate with other component instances is to communicate via ports. However, discrete ports represent asynchronous communication between components and therefore do not result in any direct dependencies between the communicating components. Hybrid ports are used to read or write specific sensor values and therefore result in a dependency between the discrete component instance and the connected continuous component. In this section, we focus on discrete component instances only. The dependency to continuous components is discussed in more detail in Section 4.2.4.

Discussion In the following, we discuss the segmentation approach with regard to the requirements presented in Section 4.1. Applying this approach to the discrete component instances of overtakerVehicle of the running example, we get two runnables in total. Additionally, we have to define five label accesses to read and write the connected continuous components. This approach is the most straightforward one, because there are no runnable dependencies that we have to consider in this segmentation for variables of the component behavior. It is highly suitable for requirement **R2**, since the amount of runnables is equal to the amount of discrete component instances. Requirement **R1** is only fulfilled partially by this approach, since it enables the parallel execution of component instances, but not the parallel execution of the component instance behavior. This might also have parts that are independent from each other and could be executed in parallel, which is not considered by this segmentation.

The execution of the behavior is not affected directly by this segmentation, since the complete RTSC is executed in one runnable. Hence, we argue that the semantics are preserved, if runnable properties are derived correctly and the generated source code does respect all MechatronicUML semantics. Thus, we state that **R3** is also fulfilled. Requirement **R4** depends on the concrete RTSCs of the component instance and might not be fulfilled in all cases: For example, if the WCET of the runnable is higher than a deadline of a transition in the RTSC, the deadline might be missed. Therefore, it might be useful to enable parallel execution of the internal component behavior itself.

In the following, we show that it can be an advantage to execute parts of the component instance in parallel. Figure 4.3 shows an excerpt of the component RTSC `overtakerDriver` . We show only two regions: `overtakerDrivingRTSC` and `initiatorPortRTSC`. The region `overtakerDrivingRTSC` specifies the driving

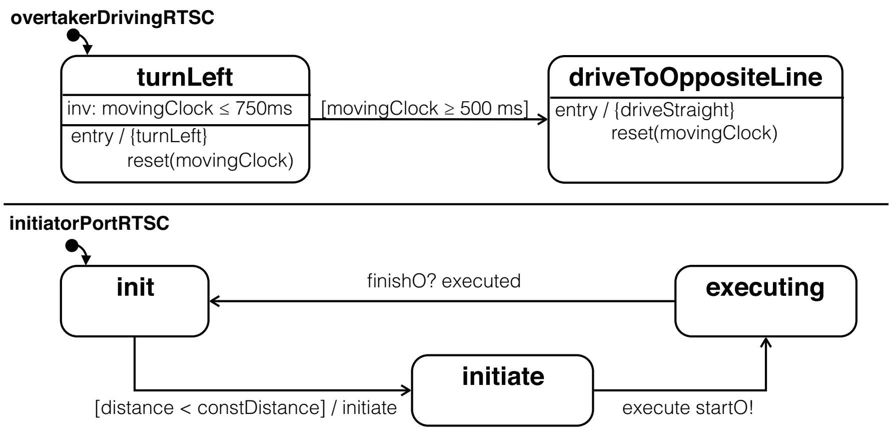


Figure 4.3: Different Periods are Possible if Regions Are Executed in Parallel.

behavior of the vehicle. When the vehicle starts to overtake another vehicle, it has to turn left for a specific time. This time is modeled by the clock constraint and the state invariant in the state `turnLeft`. We assume that this region has to be evaluated every `125 ms`. We show later on in Section 4.2.3.5, why this value is useful.

Additionally, we assume that in the state `executing`, an entry action is used that is very time-consuming. Furthermore, we assume that the determined period of the corresponding runnable is high, e.g., `1000 ms`. Therefore, it would be convenient to evaluate region `initiatorPortRTSC` every `1000 ms` and region `overtakerDrivingRTSC` every `125 ms`. It gets critical, if we assume that the WCET for executing the runnable for `initiatorPortRTSC` will take more time than the period of the runnable for region `overtakerDrivingRTSC`. If so, this segmentation is not able to provide a feasible scheduling since the runnable for the component instance would have to be reactivated before it has finished. Thus, in this case parallel execution with different periods is not only advantageous, but mandatory for a feasible scheduling.

Beside in situations, like in the example in Figure 4.3, parallel execution of the component behavior is needed in other approaches. For example, Schubert presents an approach in [Sch15] to find save reconfiguration states for the reconfiguration approach of MechatronicUML [HB13, Hei15]. In this approach, components define a reconfiguration behavior that is defined by an RTSC. In particular, additional to the component behavior for each atomic component it defines an RTSC *ReconfigurationManager* for planning and executing the reconfiguration and an RTSC *Monitoring* to monitor current values of the system. Schubert [Sch15] assumes explicitly for the approach, that these three RTSCs have to be executed in parallel to ensure correct and save reconfigurations. Thus, parallel execution of component code is needed in some cases.

4.2.1.2 One Runnable per Transition

In this section, we discuss a segmentation approach that aims to enable maximal parallel execution of an RTSC. We need to use the smallest possible software part as a runnable to maximize the potential parallel execution of the component instance. To reach this, we can use the smallest executable unit as a runnable. For this, we do not use the technical view as done in compiler optimization like variable assignments [Muc06], but we focus on the logical view, i.e., we focus on the semantics of MechatronicUML.

In each evaluation step, the transitions of the currently active state in all regions are evaluated in the order of their transition priorities if they are ready to fire or not. If there is one enabled transition, the transition effects are executed and the state is changed. After that, the evaluation step for this region is finished. Thus, there is at most one state change per region in one evaluation-step. The order of executed effects is fixed and parallelization is not applicable at this point. Consequently, we define a state change, a transition, as the smallest logical software part and therefore we will propose an approach for a segmentation with one runnable per transition in this section.

In this segmentation, each runnable is responsible for one transition and, therefore, it has to check if this transition is enabled. Hence, according to the definition for executing a transition (cf. Section 2.5.3), each runnable has to execute the effects of the exit event of its source state, transition effects, and effects of the entry event of the target state. Additionally, it has to set the new state to the currently active state of its region. The main idea of this approach is similar to an appraoch presented in [Tee12], where each state is mapped to one task. Similarities and differences to this approach are further discussed in Section 6.3.

Runnable Dependencies In contrast to the segmentation *One Runnable per Component Instance*, in this approach, we have to consider dependencies between the runnables, because they read and write common variables or have to be executed in a specific order. In the following, we describe all possible dependencies between these runnables that can be derived from the RTSC:

Variables Transitions may access variables in different ways: If the transition has a specified guard, variables of the containing RTSC might be read. Consequently, we have to specify a read access to each variable that is referenced in the guard. Additionally, a transition can execute an action. This action can read and write any variable of the RTSC. Thus, we have to define a corresponding dependency for each read or write access in this action.

Hybrid Ports Every region and therefore each transition has access to the values provided by the hybrid port instances of the component instance. Assuming, the value of the hybrid port instance is stored in a dedicated variable, read/write accesses similar to variables have to be defined.

Clocks Since clocks can be read in clock constraints and can be reset in state events, read accesses as well as write accesses to all clocks of the RTSC are possible. Hence, we define a read access for reading the clock and a write access for resetting the clock.

Transition Priorities Outgoing transitions of a state have a unique priority (cf. Section 2.5.3). These priorities define the order of evaluation. Thus, they are fundamental to make the execution of the RTSC deterministic. In terms of program-flow, this can be compared to nested if-statements. Consequently, it is mandatory to respect this defined execution order. Thus, we have to define *runnable sequencing constraints* for the runnables of all outgoing transitions for each state, that ensure the specified execution order. Figure 4.4 shows these constraints for one state with four outgoing transitions. Each transition is mapped to one runnable. For each transition a runnable sequencing constraint to the next transition in the execution order is specified. These runnable sequencing constraints therefore ensure the correct execution order of the transitions. The number of constraints for one RTSC can be estimated by $\mathcal{O}(N)$, where N is the number of all transitions (of all regions).

Regions A transition has no access to variables or clocks that are defined in other regions that are on the same or a lower hierarchy level (cf. Section 2.5.3).

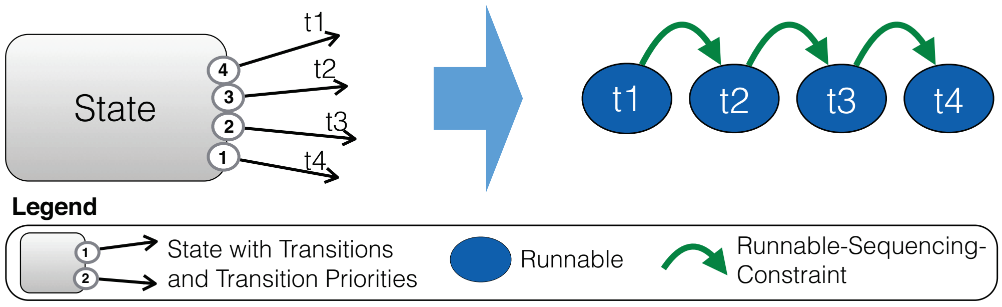
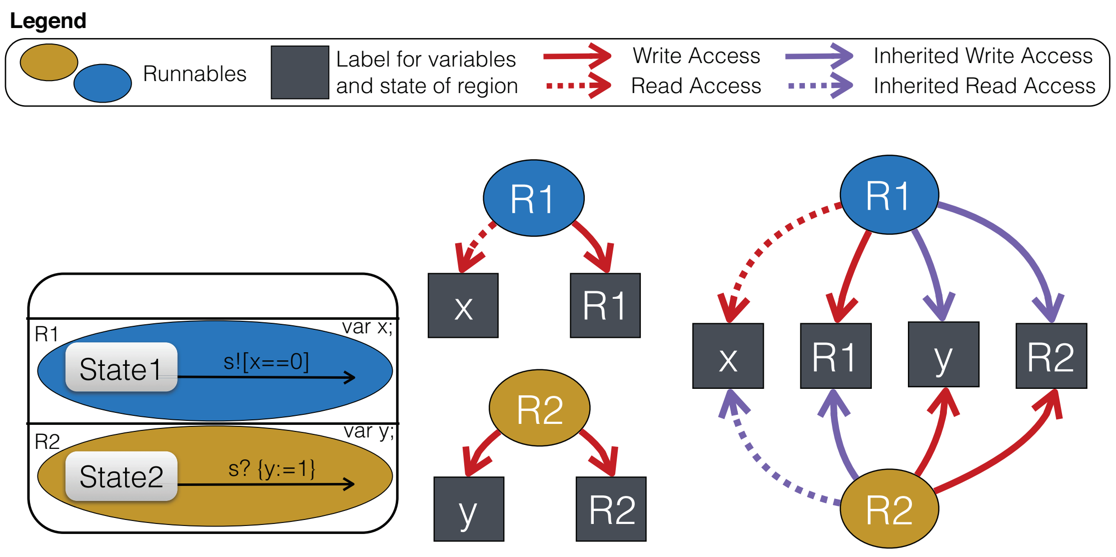

State
4
3
2
1
t1
t2
t3
t4
t1
t2
t3
t4
Legend
1
2
State with Transitions and Transition Priorities
Runnable
Runnable-Sequencing-Constraint

Figure 4.4: Runnable Sequencing Constraints for Transition Priorities.

However, variables that are declared in a shared higher hierarchy, e.g., the Root-RTSC, could be accessed by transitions of different regions. Consequently, there are dependencies to other regions (their containing RTSCs respectively). However, these dependencies are already specified by the dependencies created for the variable accesses. Transitions (without defined synchronization channel) are not able to change the current state of other regions. Therefore, no dependencies to the current-state-variable of other regions are needed.

Synchronization Channels Synchronization Channels are used to execute two transitions of different regions synchronized, i.e., both transitions fire at the same time (cf. Section 2.5.3.6). Thus, synchronization channels are explicitly defined dependencies between two regions in the RTSC. When two transitions are in synchronization and all clock constraints and guards of both transitions are enabled, both transitions will fire at once, i.e., first the effects of the sender transition are executed and second the effects of the receiver transition. No other behavioral code must be executed in between. Therefore, one solution is to combine the runnables of both transitions into one runnable. All dependencies and runnable sequencing constraints that are defined for both runnables will be applied to the new runnable to ensure the correct execution order. However, a transition with specified synchronization channel might have more than one possible partner transition. Thus, all combinations of possible partner transitions have to be considered. Therefore, instead of merging the runnables, we propose to extend the existing runnable for each transition with defined synchronization channel by the functionality to evaluate all possible partner transitions and to execute their effects. Consequently, this runnable will inherit all dependencies (to variables, clocks, and the current state of the containing region) and all runnable sequencing constraints of all partner transition runnables. Figure 4.5 shows an RTSC with two regions.



(a) Examplary RTSC with Synchronized Regions. (b) Label Accesses. (c) Label Accesses With Synchronization.

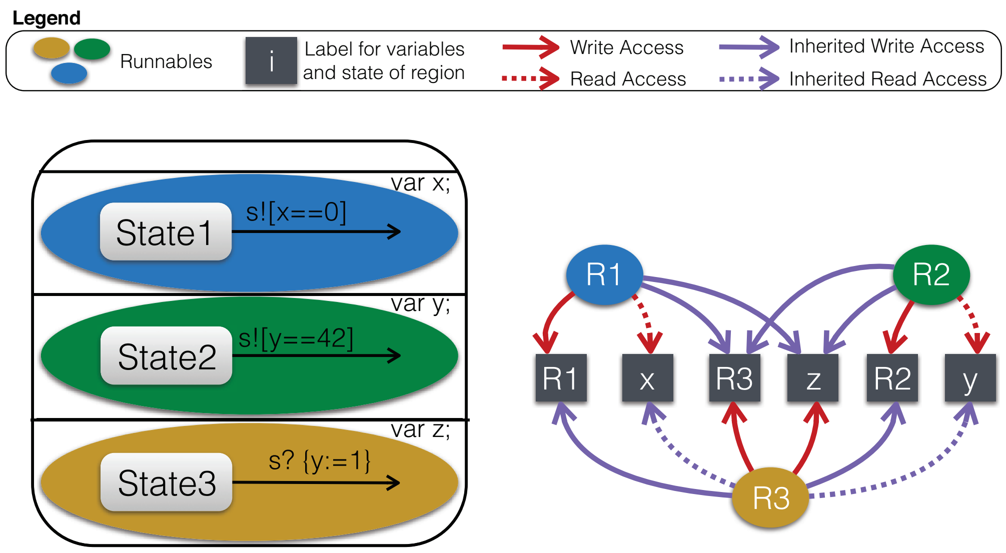
Figure 4.5: Label Accesses Without and With Synchronization of Transitions.

Additionally, the read/write accesses to the declared variables and to label for the currently active state of the region are shown. Figure 4.5b shows the accesses without using synchronization, Figure 4.5c with synchronization. The more possible partner transitions there are the more dependencies will occur and prohibit parallel execution of the corresponding runnables.

Naturally, runnables for the partner transitions will also inherit additional dependencies. However, these dependencies might be different for partner transitions, although both use the same synchronization channel. To illustrate this, Figure 4.6a show an RTSC with three regions and Figure 4.6b the resulting label accesses for every runnable.

In regions `R1` and `R2` transitions with *sending* synchronization channels are defined, in region `R3` there is a *receiving* synchronization channel. The transition `R3` does have other read/write accesses, since it has to handle two possible partner-transitions, whereas both the other transitions have only one possible partner transition. This effect will be intensified, if a synchronization channel is used for more than one transition in a region.

Do Events We assume that each runnable executes one transition. Firing a transition does also includes executing the exit event of the source state and the entry event of the target state. If a state defines a do event no transition will execute it. Thus, we have either to avoid the use of do events or to



(a) RTSC with Synchronized Regions. (b) Resulting Label Accesses.

Figure 4.6: Synchronization Partner Transitions Do Not Necessarily Have the Same Read/Write Accesses.

define an additional runnable that executes the do event. This runnable has to be executed after all transition runnables. Another approach is to merge all runnables for the outgoing transitions with the runnable for the do event. Thus, such a runnable will then execute the evaluation of each transition (according to their period) and the do event, if needed.

Discussion Using this segmentation, the amount of runnables that can be executed in parallel is high. Applying this approach to the discrete component instances of the overtaker of the running example, we get 34 runnables, 89 label accesses, and 8 sequencing constraints in total. Since we defined a transition as the smallest logical software unit and we define one runnable per transition, there is no other segmentation that enables more possible parallelism. Consequently, requirement **R1** is obviously fulfilled.

Typically, the number of transitions in an RTSC is high and the more runnables we generate the more dependencies and sequencing constraints occur and have to be specified and considered in the partitioning. Especially, if shared variables or synchronization channels are used, the amount of conflicting write accesses increases significantly. Therefore, **R2** is not sufficiently fulfilled.

The semantics of MechatronicUML are fulfilled with this segmentation approach, since the execution order is preserved by the defined dependencies and the corresponding sequencing constraints for transition priorities and synchronization channels. Since we argue to disregard the region priorities of the RTSC, the order of state changes can differ from the one defined in [BDG+14a]. It might happen that – in contrast to the first segmentation approach – one region fires several transitions while another region does not fire one transition at all. The resulting behavior will not differ significantly from the expected behavior, since regions can be executed independently between their synchronization points as further discussed in Appendix A. Thus, requirement **R3** is satisfied.

The fulfillment of **R4** depends on the runnable properties. This segmentation approach is highly suitable to fulfill **R4** by construction: **R4** is violated, if some timing properties are violated. This can happen, for example, if the WCET of the runnable is greater than its deadline. We define a runnable for the smallest possible software part. Therefore, there is no segmentation approach that can define runnables with shorter WCET. Consequently, if the WCET of a runnable in this segmentation is too high, no other segmentation can improve this. Thus, the developer has to adapt the models for this software part.

A negative aspect of this segmentation approach is that we execute more runnables periodically than needed: In each region, there is exactly one state active at the point of evaluation. Consequently, only the outgoing transitions of this state could be enabled in this evaluation step. Runnables that evaluate transitions of other states will always abort after checking the current state of the region. We call such runnables *dead runnables*. Since this has to be done for each transition, this might result in a lot of overhead. Let n be the number of transitions in a region. In the worst case, there are $n-1$ dead runnables that execute only the check for the current state and do not execute program behavior. Hence, the response time of the system will increase without any benefit. This might increase the complexity for finding a feasible partitioning and mapping. Therefore, it might be useful to use one runnable for all transitions of the region, or to be more precisely: one runnable per region.

4.2.1.3 One Runnable per Region

In the following, we present a segmentation approach that provides a good trade-off between the number of generated runnables and parallel execution of the RTSC. The main idea for this segmentation approach in this section is to generate one runnable per root-region of the component behavior RTSC. A root-region is a region whose parent is the top-level init-state of the RTSC.

Usually, the RTSC that describes the behavior of a component instance consists of several regions: one region for each communication partner of the component instance and possibly several additional regions for synchronization issues. Every port defines its own behavior and therefore is (apart from used synchronization channels) independent from the behavior of the other ports. Consequently, each region can be seen as one logical part of the component behavior. Thus, such a region is suitable for a runnable.

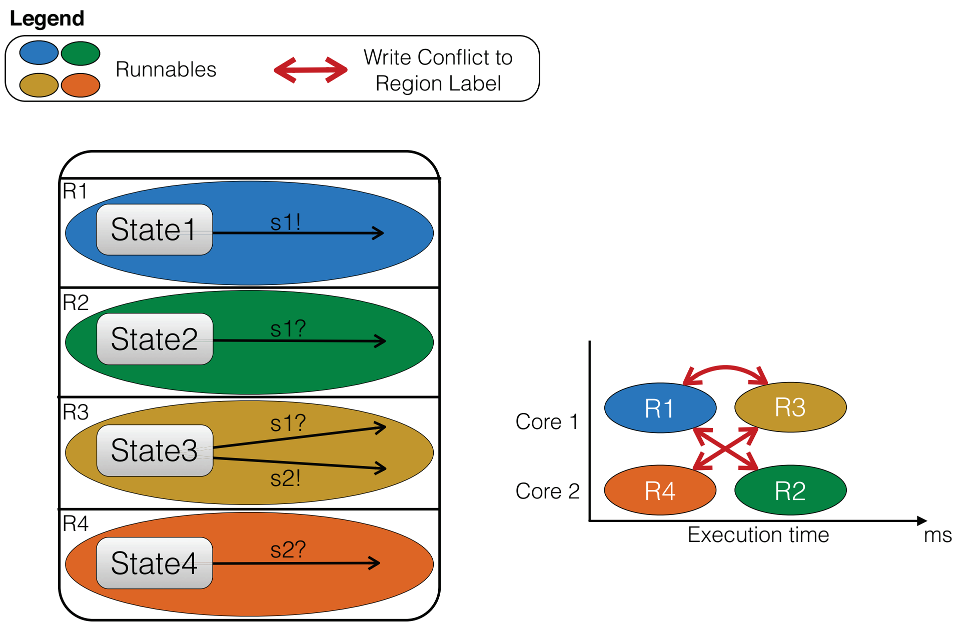
Beside the architectural reasons mentioned above, there are also technical reasons for this segmentation, since it avoids the overhead like in the *One Runnable per Transition*-segmentation, but still enables parallel execution of the RTSC. As described in Section 4.2.1.2, *One Runnable per Region* results in the same execution semantics as *One Runnable per Transition* but it creates no dead runnables. Furthermore, it does not decrease the possibility for parallelization, since all active states of the regions could still be executed in parallel, but it will decrease the number of runnables and the number of dependencies (there is no need to define runnable sequencing constraints for transition priorities). Additionally, the code-overhead can be reduced, since a *switch-case*-statement can be used to check the active state instead of calling an *if*-statement in each runnable.

Runnable Dependencies Regions of an RTSC are not completely independent, since they can use shared variables, clocks, or are even explicitly defined as dependent by using synchronization channels. In the following, we describe all possible dependencies between runnables that can be derived from the RTSC, which are quite similar to the dependencies in the previous chapter at some points.

Variables Similar to the segmentation in the section before (one runnable per transition), transition effects and state effects of different regions might access the same variables. Consequently, regions that access such variables depend on each other. States and transitions of a region can access variables that are defined in the contained RTSC and variables that are defined in any parent RTSC. As we define one runnable per root-region, only variables that are defined in the component instance RTSC can be accessed by different runnables. Consequently, only these variables have to be considered for deriving dependencies.

Clocks There are also clocks defined in an RTSC and visible to this RTSC and all its child RTSCs. Thus, similar to variables, we have to consider only clocks that are defined in the component instance RTSC. Other clocks cannot result in dependencies between runnables.

Synchronization Channels As described in Section 4.2.1.2, synchronization channels are explicitly defined dependencies between two regions. In contrast to a transition, in a region several synchronization channels can be used, since each transition of the region might use a different synchronization channel. Applying the concept of the section before and giving each region access to all synchronized regions is one solution. Following this concept, we would still have one runnable per region, but additional access dependencies. Thus, regions that have synchronizing transitions can still be executed autonomously, but must not be executed in parallel. Even if all regions do synchronize with other regions, parallel execution is still possible. Figure 4.7 shows an example for this.

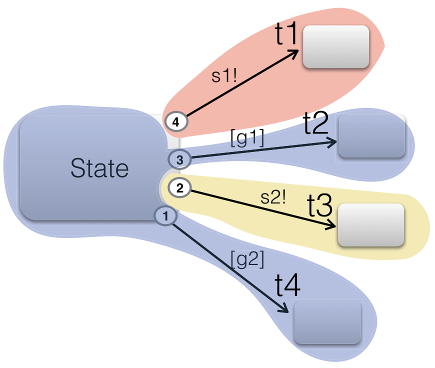


(a) Examplary RTSC with Synchronized Regions. (b) Resulting Parallel Runnable Execution.

Figure 4.7: Possible Execution Order for Regions with Synchronization Channels with Parallel Execution.

Region `R1` has partner transitions in region `R2` and `R3` defined by synchronization channel `s1`. Region `R3` additionally synchronizes with a transition in region `R4` using synchronization channel `s2`. Figure 4.7b shows a possible schedule for the four regions, that solves all write conflicts by specifying an execution order. Although all regions do synchronize, both cores do not have any gaps and the regions of the RTSC are executed in parallel. Thus, this concept does restrict parallel execution, but does not prohibit it completely.

However, if two regions with many transitions in total contain only one pair of transitions that are synchronized, these dependencies might be overdesigned and prevent more parallel execution than needed. Thus, another approach is to separate transitions with synchronization into dedicated runnables. Figure 4.8a shows a state of one region. The state has the four outgoing-transitions t1 to t4. t1 and t3 have defined synchronization channels to synchronize with transitions of other regions. For better readability, these regions are not shown in the Figure. We create one runnable for all transitions without synchronization. This runnable (the executed states and transitions respectively) is marked by blue background. Transitions with synchronization channels are executed by additional runnables (red for channel S1 and yellow for channel S2).



(a) Segmentation into Runnables

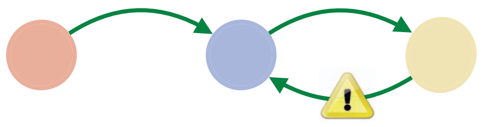
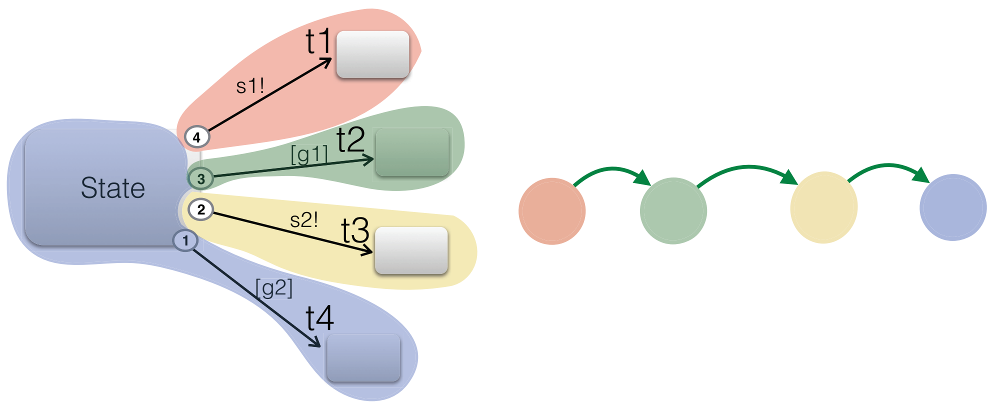

(b) Resulting runnable sequencing constraints contain a cycle.

Figure 4.8: Synchronized Transitions are Executed in Separate Runnables.

The advantage of this approach is that the dependencies resulting from synchronization channels are also moved to the new runnables. Thus, the region-runnables could still be executed in parallel. Moreover, the transition which is moved to a new runnable has a defined transition priority. Therefore, we have to define runnable sequencing constraints to ensure this execution order. Figure 4.8b shows the resulting constraints. The cyclic sequencing constraints cannot be satisfied, like the constraints between the blue and the yellow runnable. The blue runnable has to be finished before the yellow runnable can start, because transition t1 has a higher priority than transition t3. However, the yellow transition has a higher priority than transition t4. Thus, parts of the blue runnable have to be executed before the yellow runnable, others afterwards. This is not possible, because it is a contradiction by definition. We have two possibilities to solve this problem: disregarding transition priorities or resolving this cycle.

In the current semantics of MechatronicUML, in some cases the use of synchronization channels overrules the defined transition priorities [BDG+14a]. Hence, we could also disregard transition priorities, if synchronization channels are used and therefore one possibility is to define that all synchronized transitions are executed before the rest of the transitions. If we decide to respect the transition priorities, we have to resolve the cyclic constraints. For this, we have two options: On the one hand, we could merge all runnables that result in a cycle. In Figure 4.8, we would get two runnables: one for transition `t1` and one for transitions `t2`, `t3`, and `t4`. This can result in the our first solution for handling synchronization channel (keeping one runnable per region and to inherit the access-constraints). On the other hand, we could define additional runnables for all transitions that have a higher priority as a transition with synchronization channel and define dedicated runnable sequencing constraints. Figure 4.9a show this extended version for the example.



(a) Using An Additional Runnable to Respect Transition Priorities (b) Resulting runnable sequencing constraints have No Cycle.

Figure 4.9: Synchronized Transitions and Conflicting Tranisitions are Executed in Separate Runnables.

The runnable sequencing constraints do not contain a cycle anymore as shown in Figure 4.9b. Thus, this version is a feasible solution for the problem. However, using this approach, several additional runnables and dedicated constraints have to be specified, which increases the complexity a lot. Hence, we propose to choose the first solution (extend the runnables by the functionality to execute the firing of partner transitions).

Discussion Applying this approach to the discrete component instances of the overtaker of the running example, we get in total 6 runnables, 34 label accesses, and no sequencing constraints, if we extend the region-runnables by adding code for evaluation and execution of possible partner-transitions to handle synchronization channels. This approach enables parallel execution of regions of an RTSC.

However, synchronization channels might restrict the parallel execution of all regions, but we show that nevertheless there can be regions that are still executable in parallel. Therefore **R1** is sufficiently fulfilled. The complexity in terms of number of generated runnables, dependencies, and constraints is in general not high. Basically, we have one runnable per region, which is independent from the other regions. Using shared variables and synchronization channels results in dependencies between the region-runnables. If synchronization channels are frequently used, the amount of additional runnables and constraints will rise. We proposed two different approaches to handle synchronization channels. In the first approach, we extended the region-runnables by adding code for evaluation and execution of possible partner-transitions. In the second approach, we transferred synchronized transitions into separate runnables. This can lead to further runnables and sequencing constraints, which are needed to respect transition priorities. Thus, we expect the first approach to be less complex and therefore choose this approach in the ongoing thesis. Typically, only few synchronized transitions are used and therefore we argue that this segmentation approach fulfills **R2** sufficiently for the most models.

Similar to the *One Runnable per Transition*-approach, **R3** is only partially fulfilled, since we disregard region priorities. However, beside this, the semantics are preserved, even if several synchronization channels are used. Since the timing behavior depends on the period of the runnables, the fulfillment of **R4** does also depend on the period of the runnable, which is discussed later on in Section 4.2.3.7.

4.2.1.4 Intermediate Conclusion

In the sections before, we discuss three different segmentation approaches: *One Runnable per Component Instance* (cf. Section 4.2.1.1), and *One Runnable per Transition* (cf. Section 4.2.1.2), and *One Runnable per Region* (cf. Section 4.2.1.3). To allow parallel execution of the component behavior, we propose to disregard region priorities, since they prohibit parallel execution of regions (and their containing transitions). Disregarding region priorities may change the order of fired transitions, but only non-critically, because no safety-critical state-combinations are possible when synchronization channels are used. This aspect is further discussed in Appendix A.

For each segmentation approach, we define a systematic method to derive runnables, dependencies (read/write accesses), and sequencing constraints. We observe that the possible parallel execution does not only depend eminently on the number of generated runnables but does also depend on the number of read/write accesses and runnable sequencing constraints. Table 4.1 shows the amount of runnables, label accesses, and sequencing constraints for all segmentation approaches when applying them to the discrete component instances of the overtakerVehicle.

Requirement	**Segmentation Strategy**		
	Per Component	*Per Region*	*Per Transition*
Runnables	2	6	34
Label Accesses	5	34	89
Sequencing Constraints	0	0	8

Table 4.1: Number of Runnables, Label Accesses, and Sequencing Constraints for the Discrete Components of the Overtaker.

For the segmentation approaches *One Runnable per Region* and *One Runnable per Transition*, the use of synchronization channels increased the complexity significantly, since we have to define additional runnables and dependencies to preserve the MechatronicUML semantics: Each runnable needs an additional write access to the region variable of all partner transitions. Since this has to be done for both of the synchronously firing transitions, these write accesses are resolved by runnable sequencing constraints during the partitioning algorithm. Hence, using synchronization channels prohibits parallel execution of the runnables.

At this point of the thesis, we cannot discuss **R4**, because the fulfillment of the timing properties depends on the period of the runnables, which is discussed later on in Section 4.2.3. Hence, the fulfillment of **R4** is discussed in the conclusion of the whole chapter in Section 4.4. Table 4.2 shows an overview of the requirements **R1-R3** for each segmentation strategy. All three segmentation approaches fulfill the defined requirements, but only the strategy *One Runnable per Region* fulfills requirements **R1** - **R3** sufficiently.

Requirement	**Segmentation Strategy**		
	Per Component	*Per Region*	*Per Transition*
R1: Parallel Execution	(✓)	✓	✓
R2: # Runnables	✓	✓	(✓)
R3: Preserve Semantics	✓	✓	✓
R4: Time Properties	?	?	?

Table 4.2: Fulfillment of Requirements R1 - R3 for all Segmentation Strategies.

All segmentation strategies allow parallel execution of the software. The *One Runnable Per Component*-strategy, does not allow the parallel execution of an RTSC. Hence, we argue that **R1** is less fulfilled than by the other strategies that allow parallel execution of RTSCs. The strategies *One Runnable Per Component* and *One Runnable Per Region* provide a good relation between the number of runnables plus the number of defined dependencies to the number of component instances. For the *One Runnable Per Transition*-strategy, many so-called dead-runnables are generated that have to be scheduled but do not execute behavior code. Additionally, many runnable sequencing constraints have to be defined to respect the semantics of MechatronicUML, even if region priorities are disregarded. Hence, we argue that **R2** is fulfilled, but not sufficiently for the most cases and is not suitable for most real systems. Due to the construction of the runnables and the derived dependencies, all three strategies preserve the semantics of MechatronicUML. Thus, we argue that **R3** is fulfilled sufficiently by all strategies.

Each strategy might be suitable for different models since the runnables of the approaches have differences in the WCET: Let us again consider the excerpt of the RTSC of the component instance `OvertakerDriver` shown in Figure 4.3 on page 47. Per definition, the WCET of transition runnables will be shorter or equal than the WCET of region-runnables, which again will be shorter than the WCET of component-runnables. For example, the WCET to execute both regions in one runnable is the sum of the WCETs of both regions. Consequently, for software systems with component instances that have a low WCET and the processor utilization is low, the first segmentation approach is suitable. The advantage of using a multi-core ECU is not very high in this case, since the processors are hardly utilized. For software systems, where most transitions have high WCETs, *One Runnable per Transition* might be better. Especially, if the WCET of one transition is significantly higher than the deadline of another transition. The *One Runnable per Region* segmentation seems to be a good trade-off between the number of runnables and possible parallel execution, as long as synchronization channels are used cautiously. Hence, we suggest using synchronization channels as little as possible, since it increases the complexity for parallel execution on multi-core platforms significantly.

In our approach, we take operations and actions into account, when determining the WCET for a runnable. However, some implementations might be time-consuming and can increase the WCET of the runnable unnecessarily. To avoid this, the execution of operations and actions can be moved to additional runnables or tasks, like done for actions in [BGS05, Bur06] to reduce WCET of periodic runnables (cf. Section 6.3). This changes the execution order of the transition effects. Hence, we propose to integrate the execution of actions and operations into the periodic runnables.

Furthermore, at this point in development, the runnables can be analyzed for inconsistencies, conflicts, or possible improvements. For example, it can be detected at this point, if the WCET of a runnable is higher than its period. This is a problem because in this case a runnable would be activated again before it has finished its execution. A possible solution is to adjust the segmentation by applying a more fine-grained approach, e.g., *One Runnable per Region* instead of *One Runnable per Component Instance*. Another option is to adjust timing properties of the PIM to relax the conditions for the period, e.g., by changing clock constraints or invariants. Hence, analyzes and improvements regarding the scheduling can already be applied in an early stage by analyzing the runnables and utilizing this information in former process steps.

In summary, we state that three segmenation strategies fulfill all requirements (at least partially) as far as we can analyze them at this point. However, only the strategy *One Runnable per Region* fulfills the requirements sufficiently. We argue that *One Runnable per Region* is the strategy, which is applicable most usefully to CICs in general. *One Runnable per Component Instance* and *One Runnable per Transition* can be used in special cases, if needed.

4.2.2 Define Runnable WCET and Deadline

In this thesis, we define three properties for each runnable that are mandatory: worst case execution time (WCET), period, and deadline (cf. Sections 2.6.1.2 and 3.1). These runnable properties describe an abstract view of the resulting source code and provide needed information for the latter process steps *partitioning* and *mapping*. All three values are strongly related to the executed behavior of the component instance. Consequently, we will derive them from the behavior-RTSC of the component instance. In the following, we discuss how to derive WCET and deadline for runnables. Determining the period is a more complex task and is discussed separately in Section 4.2.3.

4.2.2.1 Runnable WCET

The WCET of a runnable describes the longest possible time the runnable needs to be executed on a specific platform (cf. computation time C_i of tasks in Section 2.2.1). Thus, the WCET of a runnable can differ if it is executed on different platforms. The time that is needed to execute a program (part) on a CPU depends on the frequency and the instruction set, caches, and pipelines of the CPU [Wil09]. Additional factors, like processor family (version), the programming language, and the used compiler (with or without optimization functionality) affect the actual WCET.

During the specification of the runnable, the target platform is not specified yet. Thus, one option is to choose the number of processor instructions as representation of the WCET as done in AMALTHEA (cf. 2.6.1). Then, for each processor, a concrete WCET can be determined based on the instructions. Another option is to measure the WCET for a possible target platform, e.g., by measuring it during execution or simulation as done for MechatronicUML. This has to be done obviously for each possible target platform (each ECU) that is used in the target system. The measured time has then to be annotated in the PIM for each runnable and each ECU combination.

Determining WCETs for specific target platforms is quite complex. Tools as *Simple Scalar* [ALE02, Sca] or *aiT* [FH04] provide the possibility to determine WCETs for specific architectures. Thus, in this thesis, we assume that the WCET for each runnable is determined by an appropriate method or tool and provided as an annotation for each runnable.

4.2.2.2 Runnable Deadline

Every runnable has a defined relative deadline (cf. Section 2.2.3). It is sufficient to provide a relative deadline since the absolute deadline can be computed automatically by adding the the relative deadline to the release time (cf. Section 2.2.1). Similar to the period of a runnable, the deadline depends on the execution of each transition of an RTSC. Every transition can define an own deadline, which is the time span until all transition effects have to be executed (in relation to the time, when the firing of the transition starts). When executed, the runnable evaluates all outgoing transitions of the currently active state. If there is an enabled transition, this transition fires and the transition effects are executed. Thus, once the transition effects are started, this execution cannot be preempted by other code of this runnable. Consequently, the runnable has to be finished until the deadline for the firing transition. Since the executed region can have several transitions with deadlines, the deadline of a runnable is defined as the minimum deadline-value of all transitions that are evaluated by this runnable. If no transition of the region has a defined deadline, no specific deadline for the runnable is set. Then, the deadline is set automatically to the period of the runnable, because it has to be finished before being called again.

4.2.3 Define Runnable Period

The execution of an RTSC has to evaluate all transitions and to execute the firing of a transition if an enabled transition is found. We propose to execute the RTSC stepwise, where each step checks the transitions conditions and fires at most one enabled transition. Since we focus on time-triggered execution we have to define a period for the execution for such steps of each runnable.

In this section, we discuss how the period for a runnable can be determined. Determining the period for executing a runnable has to respect the semantics of MechatronicUML, particularly deadlines, clock constraints, and invariants of the executed RTSC. Since we have to guarantee that every clock constraint and every invariant is respected at runtime, the period of a runnable depends on the *shortest* period-value of its transitions. Thus, we have to derive a period-value PV_t for each transition that will be executed by the runnable and set the period π of the runnable to the minimum period-value of all these transitions. Hence, we define

$$\pi_{runnable} = min(PV_t), \forall t \in runnable$$

Finding PV_t for a transition is not trivial, since it has to consider the semantically correct execution of the RTSC. Hence, we discuss the transition properties that affect the period-value. First, we discuss how PV_t can be derived for a transition in general (Section 4.2.3.1). After that, we discuss transition conditions influencing if and at which point in time the transition is enabled and, therefore, the PV_t: *guards* (Section 4.2.3.2), *clock constraints* (Section 4.2.3.3), *invariants* (Section 4.2.3.4), the *combination of clock constraints and invariants* (Section 4.2.3.5), and the *combination of guards and invariants* (Section 4.2.3.6).

4.2.3.1 Determine Period-Values

The period value PV_t of a transition t has to be short enough to guarantee that each enabled transition is detected. An enabled transition might become disabled again before firing, if the enabling conditions change to false, e.g., if a clock constraint is not valid anymore after a specific time or a guard changes to false again due to changing environment conditions. We assume that for each transition exists at least one enabling interval I_E in which this transition is enabled. Transitions that can never be enabled can be found using model checking in former development steps. Figure 4.10 shows an excerpt of the region `overtakerDrivingRTSC`, which is part of the RTSC of the component instance `overtakerDriver`. This RTSC defines the logic for adapting the speed if an ongoing vehicle is detected. It shows

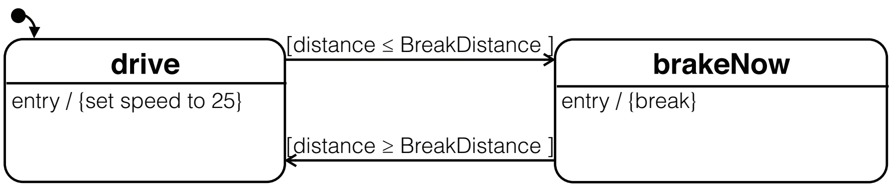
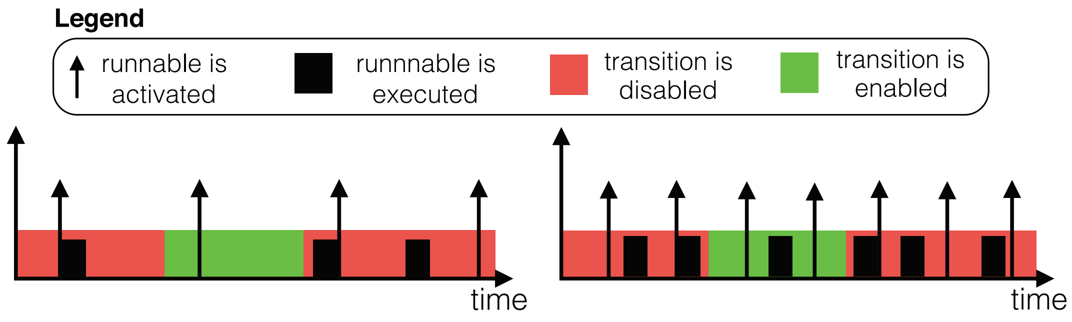

drive
entry / {set speed to 25}
[distance ≤ BreakDistance]
[distance ≥ BreakDistance]
brakeNow
entry / {break}

Figure 4.10: Urgent Transition in the RTSC of Component “overtakerDriver”.

the two states `drive` and `breakNow`. While the vehicle is driving, the state `drive` is active. If the distance to an ongoing vehicle gets lower than the constant value `BreakDistance` (let us assume 50 m), the current state of the RTSC will change to state `breakNow` where the speed is adapted.

Let us assume that the distance to an ongoing vehicle becomes 40 m. Hence, the transition to `breakNow` is enabled and should be fired. As the transition might not be fired immediately it is possible that the distance becomes 55 m again, before the transition got evaluated. This can happen, for example, if the change of the distance occurs before the executing runnable evaluates its transitions again. Figure 4.11 shows the interval when the transition is enabled (green) and the points in time when it gets evaluated by the execution of the runnable (black boxes). In Figure 4.11a, the enabling interval of the transition is missed, because



(a) A Transition is Missed. (b) A Transition is Fired.

Figure 4.11: Firing Transitions Depends on the Period of the Runnable.

the transition is evaluated too late, for the reason that the corresponding runnable is executed when the transition is already disabled again. In Figure 4.11b, the transition gets evaluated more frequently and the enabling of the transition is recognized and the transition gets fired. Thus, the period of a runnable has to be short enough to ensure that no enabled transition is missed. Consequently, the period-value of the transition has to be set to the half of the length of the shortest enabling interval I_E. Hence, we define

$$PV_t = \left\lceil \frac{min(I_E)}{2} \right\rceil$$

Urgent transitions (cf. Section 2.5.3.7) are more restrictive concerning their execution and must be considered seperately. Due to the semantics of MechatronicUML an urgent transition has to be executed immediately if it gets enabled. Immediately means without time-passing [FMMR12]. Gamati [Gam10] states that urgency can be implemented, if *immediately* can be relaxed to a specific time span. In general, such a value should be very small in comparison to the remaining executed code. We would have to set PV_t to this small value. This would lead to very short periods for all runnables that use urgent transitions and would not be practical. Without such an assumption, urgency cannot be implemented correctly, because executing code for a statechart will take time and thus the zero-time assumption cannot be fulfilled [FMMR12]. Consequently, a limitation of our approach is that the semantics of urgency cannot be guaranteed. Due to our definition of PV_t, the urgent transition might not be fired immediately after enabling, but it gets fired every time when it becomes enabled with a maximum delay of the period of the runnable.

Deriving I_E for a transition is not trivial in general. A possibility is to use a reachability analysis [Hei15] that can be applied for RTSCs as done in [Sch15]. In this analysis, a complete state-space for the system is built and can be analyzed for such intervals. However, in this analysis only the PIM is considered, but not the properties of the platform, e.g., the period of the runnables. Thus, the intervals, computed by this analysis, might not be exactly the same during the execution. Hence, this solution might also miss the execution of an enabled transition. For many combinations of enabling conditions, the enabling interval I_E can be determined without such an analysis. In the following sections, we present strategies to find this interval for different combinations of transition conditions.

4.2.3.2 Guards

Guards are part of the enabling conditions of a transition. A guard is specified by a logical expression and can refer to all variables and hybrid ports of the RTSC. Thus, a guard can change its state, if a referenced variable or hybrid port value changes. Consequently, the minimum of I_E of a guard results from changing the values for these variables and hybrid ports. The value for a hybrid port changes within a defined sampling interval. Hence, we can use the value of the sampling interval as the enabling interval. Variables that are used in the guard are also updated within a specific interval: Each variable is written by specific runnables and each of these runnables has a period, which defines the interval for updating the variable. Hence, we set the enabling interval I_E of the transition to the minimum of the periods of all runnables writing a variable or the hybrid port value of the guard. Hence, we ensure that the transition gets evaluated while enabled.

4.2.3.3 Clock Constraints

Clock constraints are used as enabling conditions for transitions (cf. 2.5.3.2). Missing a clock constraint at runtime does not violate any assumptions for the verification, if they are used as transition-guards only. If there are only greater- and greater-or-equal-constraints it is unproblematic as long as clocks cannot be reset by other runnables. Once the clock constraint changed to true, it will not be invalid until the clock is reset. To detect the time when the transition gets enabled as soon as possible, we set I_E to the clock constraint's value.

If it is possible, that clocks get reset by parallel executed runnables, the point in time of the reset depends on the scheduling. Therefore, we cannot find this point in time without runtime information and we cannot determine a useful period at this point in development, which ensures that the enabled transition is recognized. Thus, we limit our approach to clocks that are not allowed to be reset by other runnables. Additionally, we allow only static values for the clock constraints, i.e., natural numbers, because we cannot calculate the value for an expression at this point.

If there are both greater-or-equal-constraints and less-or-equal-constraints, we have to find the timeframe, when all constraints are fulfilled. For example, in Figure 4.12 four clock constraints are defined. It shows two greater-or-equal-

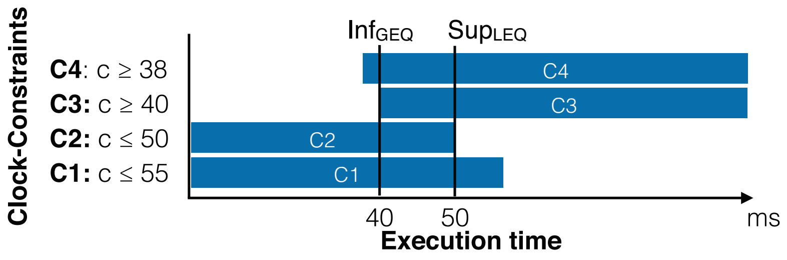


Figure 4.12: Determining the Interval where All Clock Constraints are Fulfilled.

constraints (`C3` and `C4`) and two less-or-equal-constraints (`C1` and `C2`). The diagram shows the interval (blue bar) where the clock constraint is fulfilled for each clock constraint. The corresponding transition is enabled between 40 and 50 milliseconds. Thus, it is useful to evaluate the transitions within this timeframe.

Let Sup_{LEQ} be the supremum of the less-or-equal-constraints and Inf_{GEQ} the infimum of all greater-or-equal-constraints. We can observe, that the interval where all clock constraints are fulfilled is bounded by these two values. We state that such an interval can be determined for every transition with clock constraints. A transition will never be enabled if such an interval cannot be found. This would be a modeling mistake and should be detected during verification of the PIM and, therefore, cannot not occur anymore at this point in the development process. The worst-case point in time to evaluate the transition is immediately before reaching Inf_{GEQ}. In the example, this is before 40 ms. We have to evaluate the transition again before the clock value is greater than Sup_{LEQ}, which is at 50 ms in the example. Consequently, we set the enabling interval I_E to the difference of Inf_{GEQ} and Sup_{LEQ}.

4.2.3.4 Invariants

An invariant defines that the state (which defines this invariant) has to be left before the invariant expires by using clock constraints. If there is an invariant and no transition with enabling conditions, it is sufficient to evaluate the transitions of the state once before the invariant is violated. Hence, for this case, we set the enabling interval I_E to the upper bound of the clock constraint of the invariant. This solution does only work if the clock (referenced by the invariant) is reset when the state is entered. Experience show that this is often done, but cannot be assumed to be done in general. In the worst case, the state is entered immediately before the invariant expires. If the clock is not reset, it is possible that the invariant gets violated. Therefore, similar to the use of clock constraints, we limit our approach to clocks that are reset when the state is entered.

If any transition defines additional enabling conditions, we have to consider these cases separately. Thus, we discuss the combination of an invariant and clock constraints in Section 4.2.3.5 and the combination with guards in Section 4.2.3.6.

4.2.3.5 Clock Constraints and Invariants

If a clock constraint at an outgoing transition and an invariant in the source state are used in combination, this has to be considered separately. Figure 4.13 shows an excerpt of the region `driveOvertakingRTSC` that we already used as an example in Section 4.2.1.1 on page 4.3. We consider only the region `overtakerDrivingRTSC`

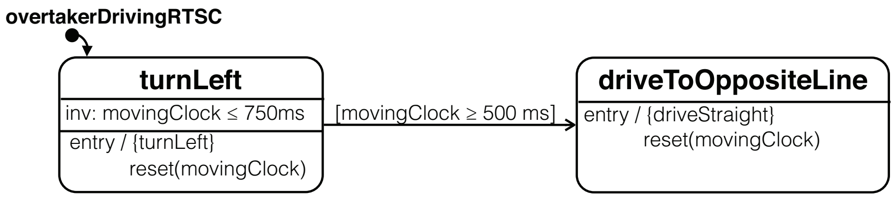
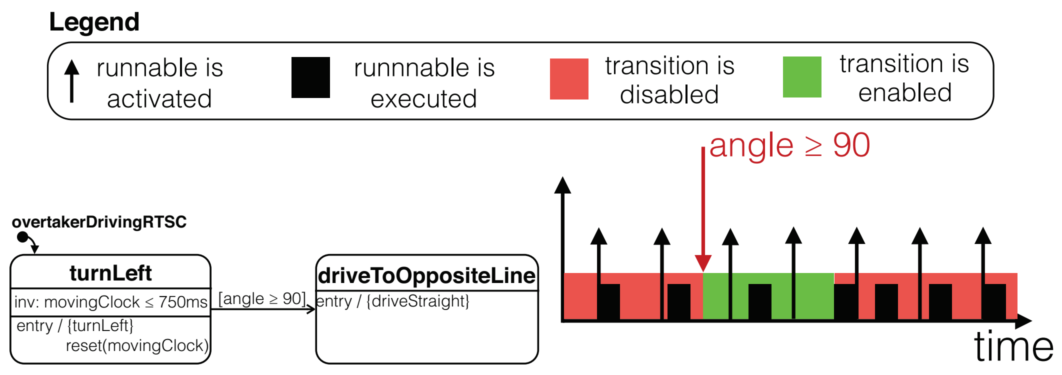


Figure 4.13: Example for Using an Invariant and a Clock Constraint in Combination.

that has two states `turnLeft` and `driveToOppositeLine`. After entering the state `turnLeft`, it can be left again after `500 ms`. This is specified by a clock constraint referencing the clock `movingClock`. Due to the defined invariant, the state has to be left before the `movingClock` reaches `750 ms`. To find the enabling interval I_E, we can apply the approach for clock constraints, since an invariant defines clock constraints that have to be fulfilled.

4.2.3.6 Guards and Invariants

If the currently active state defines an invariant and an outgoing transition specifies a guard that gets enabled before the invariant expires, we have to ensure that this transition gets evaluated and fired before the invariant expires. Figure 4.14a shows the same excerpt of the RTSC as in Figure 4.13, but we assume that the state is not changed after a defined time (defined by a clock constraint) but when a specific angle is reached. Thus, we replace the clock constraint by a guard.



(a) Guard and Invariant Used in Combination.
(b) Transition is Enabled in a Dedicated Interval.

Figure 4.14: Guard and Invariant in Combination Affect the Period-Value of a Transition.

Additionally, we assume that the guard becomes true before the invariant expires, since this can be verified by model checking in former development steps. Thus, the transition has to be evaluated by the runnable within the interval between the point in time when the guard becomes true and the point in time when the invariant expires. Figure 4.14b shows the interval and the corresponding activation times of the runnable with a corresponding period-value. The enabling interval is affected by the change of used variables and hybrid port values. Since we know that there is an enabling interval before the invariant expires, it is sufficient to set I_E to this interval. Consequently, we can apply the technique that is already applied when only guard are used. Hence, we ensure that the transition gets evaluated while enabled before the invariant expires.

4.2.3.7 Discussion and Limitations

In Section 4.2.3, we discussed how the period for runnables can be derived from RTSCs, such that the RTSCs are executed in a semantic preserving way. Using our approach, the period depends on the enabling-conditions of the transitions that are evaluated by the runnable. Since the derived period value PV_t for each transition is an upper bound, we choose the minimum of all determined period values (of all transitions) as the period of the runnable, i.e., we define

$$\pi_{runnable} = min(PV_t), \forall t \in runnable$$

Additionally, an upper bound π_{max} should be used for all periods of all runnables. If a determined period is higher than π_{max}, the period of the runnable is set to π_{max}. Using this value, it is ensured that each runnable is executed, even if no transition uses enabling conditions that allow to determine a period automatically.

We identified different combinations of transition conditions and state invariants to find the enabling interval I_E, which is used to determine the period value PV_t of a transition.We figured out that we cannot ensure all semantics in all cases when deriving the period statically. Thus, we state limitations for deriving the period. First, we have to assume that all referenced clocks are reset when the state is entered and cannot be reset by other runnables. This is indeed a hard limitation, because clocks can be reset by other state events according to the semantics of MechatronicUML. Furthermore, we allow only static time values as value in clock constraints. In MechatronicUML, expressions can be used as time value. This leads to the fact that the period cannot be determined statically (without runtime information). Second, in our approach urgent transitions are not fired immediately but might be fired with a delay of the length of the period of the executing runnable. However, using our approach, we can guarantee that each enabled transition is detected and fired. In summary, it can be stated that our approach is limited to clocks that cannot be reset by other runnables, are reset when entering the state, and use fixed time values. Hence, we propose to improve the use of clocks in the period determination in future work. Additionally, urgent transitions might be fired with a delay. Thus, we argue to consider this delay in the model checking approach in future work.

A further problem can occur, if variables are used in the guard of a transition. If a guard uses variables and hybrid ports, we state to set the enabling interval I_E to the period of the runnable that writes the values. For hybrid ports, we can use the sampling interval of the hybrid port. For runnables, this approach might become a problem, if there are two runnables and each is writing a variable which is part of a guard of the other runnable. This leads into a cylic calculation of the period and the periods of both runnables would be halved iteratively. Hence, in this case, we propose to use the same period for both runnables. This solution cannot guarantee that each enabling interval is detected and, therefore, an urgent transition might be missed or an invariant gets violated.

Our approach does cover only a subset of transition conditions of MechatronicUML in a semantic preserving way. We propose to improve determining the period in future work, e.g., by applying a reachability analysis or model checking that consideres PIM and PSM, e.g., the scheduler or periods of other runnables. Using our approach, we have to limit the use of clocks and shared variables. Otherwise, we propose to apply additional analyses to ensure that all invariants are met and no urgent transition is missed e.g., using the simulation and tracing of

Amalthea (cf. Section 2.6.3) and to adapt the period of runnables if any failures are found.

4.2.4 Segmentation for Continuous Component Instances

Continuous Component Instances are used to provide access to sensors and actuators. In the CIC of the running example (cf. Figure 4.2 on page 44), there are five continuous components (gray background). The values are read/written by connected hybrid software components. For example, the component instance `overtakerDriver` reads the value `distance` of the continuous component instance `overtakerDistance` using the hybrid port `distance`. This port defines the sampling interval, which denotes how often the value has to be updated. Thus, not only the continuous components but also the connected hybrid ports have to be considered when deriving runnables for continuous components.

One idea is to use the model normalization presented in [Ros14, BDG$^+$14b] before applying a segmentation. Using this normalization, each continuous component is mapped to a discrete compenent instance and a new RTSC is created. This RTSC has one region for reading the value from the sensor and one region per continuous port. Thus, for a simple continuous component with one outgoing port, we get two new regions. Since the generated regions have always the same structure, we can compute the number of resulting runnables for all three segmentation approaches presented for discrete component instances. After this normalization, we can apply the segmentation approaches of the sections before, because the new CIC does only contain discrete software components. An exemplary RTSC for the continuous component `overtakerDistance` is shown in Appendix B.2. Table 4.3 shows the resulting values for each segmentation approach.

Table 4.3: Number of Runnables for Continuous Component Instances per Segmentation Strategy.

Segmentation-Approach	# Runnables
One Runnable Per Component	1
One Runnable Per Region	$\#continuousports + 1$
One Runnable Per Transition	$\#continuousports + 2$

To keep the number of runnables low and to reduce computational overhead, we present an approach without reusing the approach for discrete component instances in the following. This approach generates one runnable per continuous component instance and is, therefore, better than *One Runnable per Transition* and *One*

Runnable per Region, and as good as *One Runnable per Component Instance* after applying the model-normalization presented in [BDG+14b]. Better means in this case less generated runnables and dependencies in total (cf. Requirement **R2**). A disadvantage is, that each continuous component instance has to be allocated to the same ECU as the discrete component instance that accesses the corresponding hybrid port.

In the following, we describe the approach for a continuous component instance representing a sensor. This approach can also be applied to actuators by switching read and write accesses. We create a new runnable for each continuous component instance that is used to read the value of the corresponding sensor. Additionally, a new variable has to be defined in which the recently measured value can be stored. For example, we create a new runnable for the continuous component instance `overtakerDistance` of the running example. This runnable calls an API-method to receive a new distance-value and stores it into a corresponding variable.

Runnable Dependencies Every continuous component provides a value for a hybrid component. We assume that this value is stored in one variable (label) that is accessed by both the continuous component and the hybrid component. Thus for each continuous component instance, there is exactly one variable, which is written by the runnable of the continuous component instance and read by the runnables of all connected software components. Thus, we have to consider a dependency between these runnables. Hence, we have to define a write access to the variable for each runnable of a continuous component instance with an outgoing port and a read access for each runnable of the corresponding discrete component instance that accesses this value. For incoming ports vice versa. Since reading the sensor-value does not depend on other runnables, there are no further runnable dependencies that we have to consider. Noteworthy is, that a write conflict is not possible for sensor values since only the runnable for the continuous component instance can write this value. RTSC runnables can only read this value. This does not work that way for actuator values. Only the continuous component runnable reads the value, but it might be written by several runnables since all actions (of states or transitions) can write this value and, therefore, each runnable of the RTSC might have a write access dependency to the value variable. Thus, it might be a good modeling convention to use values of hybrid ports in one region only, or at least in as few regions as possible.

Runnable Properties For each newly created runnable, we have to define WCET, deadline, and period. The WCET has to be measured, similar to the WCET of the runnables of discrete components. This measurement might be more complex than for runnables of discrete components since it does not only depend on software instructions: Reading a sensor by calling an API-function might take additional time, which is not computable based on the software only. For example, if the hardware of the sensor needs a fixed time to response when getting triggered, this has to be considered in the WCET as well. The hardware engineers must provide this time. Therefore, the WCET is the sum of software instructions of the runnable and the time for executing the corresponding API-method successfully.

The relative deadline for such a runnable is equal to its period since there are no special timing restrictions. The period of the runnable can be determined automatically: Each hybrid port instance that is connected to the continuous component instance has a sampling interval defining how often the sensor value needs to be updated. Consequently, we define the period of the runnable as the half of the minimum of all sampling intervals of all connected hybrid port instances.

Discussion In this section, we present an approach to map continuous component instances of a given CIC to runnables. This approach satisfies all of the defined requirements, if relevant. Continuous component instances have to handle the access to a sensor/actor, which is usually done by direct hardware accesses. Thus, parallel execution of such software parts is not advisable. Hence, requirement **R1** is not relevant for this case.

R2 is fulfilled since the approach generates the lowest possible number of runnables. The period of the runnable is less or equal to the sampling interval of each connected hybrid port. Thus, the value that is used in the corresponding RTSCs of the discrete components is updated in time regarding the sampling interval. Consequently, **R3** is fulfilled. Since a continuous component instance does not specify any real-time behavior, resulting runnables don't have to fulfill any time properties. Thus, **R4** is fulfilled by default.

We decide not to use the model-normalization [BDG$^+$14b] for the advantage of having less runnables. This decision might have a drawback. For our approach, we assume that the continuous component instance and all of the connected discrete component instances are allocated to the same ECU. Hence, our approach is limited to this restriction.

To profit from the advantages of both approaches in future, we propose to consider both options when generating runnables: If all discrete component instances that access the continuous component instance and the continuous component instance itself are allocated to the same ECU, the approach presented in Section 4.2.4 without applying a model normalization is used. If – for any reason – the continuous component instances are allocated to different ECUs, the normalization in [BDG$^+$14b] is applied for this continuous component instance. Thus, distribution of component instances to different ECUs is still possible, but the overhead of runnables is prevented.

4.3 Partitioning and Mapping

Following the process to derive the multi-core scheduling, we have to do the partitioning and the mapping after the segmentation. In this thesis, we are using the partitioning and mapping algorithms (and tooling) of Amalthea [Ama13a, Ama14] for these steps. We discuss how these algorithms can be used for our purposes in Section 4.3.1. After that, in Section 4.3.2 we propose a technique, how the complexity, i.e., the number of runnable dependencies, of the input-models for the partitioning and mapping can be reduced.

4.3.1 Applying Partitioning and Mapping

After the segmentation, the runnables have to be mapped to system-tasks for the target system. This step is called partitioning. Since the runnables have to be called periodically, periodic tasks are used to execute the runnables. Similar to runnables, tasks have a period, a deadline, a WCET and may have a priority (if priority scheduling is applied) (cf. Section 2.2.1). Period, deadline and WCET can be derived from the runnable properties of the executed runnables, e.g., by using the maximum of WCETs and the minimum of periods and deadlines.

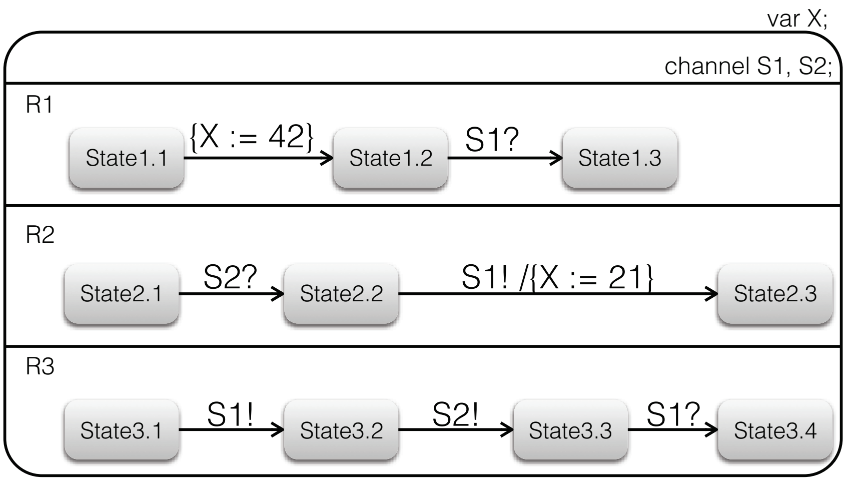
In the partitioning, the runnables are grouped by similarities, like period or commonly used variables. Amalthea provides a partitioning algorithm (cf. Section 2.6.2.1), which analyzes the set of runnables and constraints and groups them to possible tasks. This algorithm considers the period and label accesses of the runnables and specifies additional constraints to resolve cyclic label accesses. The partitioning algorithm aims to find a minimal number of tasks. However, for complex systems with many runnables and tight dependencies, this step can result in a lot of tasks, e.g., when all runnables have different periods. Then, each runnable would be mapped to one task. One possibility to avoid this is to adjust the periods of the runnables. This can be achieved by changing clock constraints in the PIM. It can also be done by setting the period of a set of runnables to the minimum of all values. Using the minimum, the safety requirements are still fulfilled, because we defined that the period for a runnable has to be *less or equal* than the determined value (cf. Section 4.2.3) to satisfy all safety assumptions. Other optimizations are possible, e.g., by reducing the amount of dependencies between the runnables. An approach for this is presented in the following section.

After the partitioning, the newly created tasks can be mapped to the cores of the ECU. Amalthea provides an algorithm for this, which allocates a set of tasks to the cores of one multi-core ECU regarding specific optimization criteria, like load balancing (cf. Section 2.6.2.2). However, if no scheduling can be found, the developer needs to adjust the models in former steps of the development, e.g., in the PIM. For example, the developer may adjust clock constraints, invariants, variable-accesses, or the use of synchronization channels to relax dependencies between runnables and therefore relax the constraints that have to be respected in the scheduling.

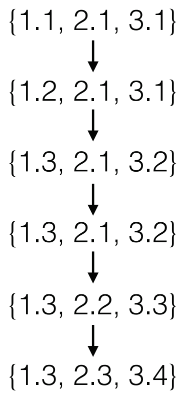
After applying partitioning and mapping of Amalthea, the resulting models can be utilized to analyze the system before its deployment and to improve the models in former development steps.

4.3.2 Reducing the Complexity of Partitioning

From the view of partitioning and mapping, a runnable is a black-box, which executes a piece of source code. The partitioning algorithm knows only meta-information like WCET, period, and possible side-effects like variable-accesses, but does not know anything about the internal control-flow. Therefore, it is possible that label accesses do not really conflict at runtime allthough the runnables are executed in parallel. Due to the semantics of MechatronicUML, the actual write commands can simply not be executed at the same time. For example, Figure 4.15a shows an RTSC with three regions.. Applying the *One Runnable Per Region* segmentation, we get three corresponding runnables `R1`, `R2`, and `R3`.



(a) RTSC with Three Regions, Whose Runnables have Write Conflicts to Shared Labels.



(b) Possible State-Combinations.

Figure 4.15: Regions with Shared Variables and Synchronization Channels Can Result in Superfluous Label Accesses.

`R1` and `R2` both write the variable `X`. Additionally, synchronization channels are used in all regions. Each region has a possible partner transition in each of the other regions. Consequently, the corresponding runnable has a write access to the label for the currently active state of each region with a partner transition.

The partitioning algorithm solves the cyclic dependency of the write access to the label of variable X by creating a runnable sequencing constraint, such that R2 is always executed after R1. Consequently, there is no conflict anymore, but that also prohibits the parallel execution of R1 and R2.

However, since we know the inner structure of the runnables, specified as RTSC, we can do further analysis for the label accesses and solve the conflict without prohibiting parallel execution. Figure 4.15b shows all possible state combinations that can occur according to the used synchronization channels. It is not possible that region R1 is in State1.1 and region R2 is in state State2.2 at the same time. Consequently, even if R1 and R2 are executed in parallel, they cannot execute the write command for X at the same time. Thus, in this case, we can disregard these write accesses.

The same can be done for the use of synchronization channels. If a synchronization channel is used at many positions and regions, it might happen that two specific regions will never use this channel together. For example, the transitions State1.2-->State1.3 and State2.2-->State2.3 are partner transitions. However, State1.2 and State2.2 can never be activated at the same time. Consequently, these transitions can never fire synchronously. Hence, we can disregard the use of the synchronization channel and the resulting write accesses to the current-state variables of the regions.

To automate the necessary analyses, we propose to use model checking for spotting if two states can be active at the same time. MechatronicUML does already use model checking via the model checker Uppaal [BLL+96] for the verification of safety requirements in RTCPs. Therefore, we reuse the domain specific temporal logic language MTCTL [DGH15] for MechatronicUML models (cf. Section 2.5.4). Using MTCTL, we can specify the corresponding queries for the model checker by referencing the states directly. We have to check each state that has an outgoing transition writing a specific label or using a synchronization channel if there is any other state of this kind in other regions. If not, we can disregard the corresponding label accesses. Listing 4.1 shows the MTCTL-queries for the access to label X (line 1) and for the use of synchronization channel S1 (line 2-4).

Listing 4.1: MTCL-Query for Synchronization Channel S and Access to Label X of Region R1 and R2.

```
AG not(R1.isInState(State1.1) and R2.isInState(State2.2));
AG not(R1.isInState(State1.2) and R2.isInState(State2.2));
AG not(R1.isInState(State1.2) and R3.isInState(State3.1));
AG not(R2.isInState(State2.2) and R3.isInState(State3.3));
```

It means, that for all possible program paths it holds at any time that no of the dedicated states are active simultaneously. Such queries have to be generated pairwise for each shared variable for all regions and for all possible partner transitions of used synchronization channels. Consequently, we can reduce the number of label access dependencies if there are such state combinations.

In Listing 4.1, the queries in lines 1 and 2 are fulfilled. Thus, the dependency resulting by the write access to `X` and the dependency resulting from the use of the synchronization channel `S1` are both superfluous. Hence, the runnables `R1` and `R2` can be executed in parallel.

4.4 Summary and Discussion

In this chapter, we present an approach to define a multi-core scheduling for software that is specified with MechatronicUML. For this, in Section 4.1, we first define four main-requirements that have to be fulfilled by the resulting scheduling.

After that, in Section 4.2.1, we discuss three different approaches for the *segmentation* for discrete component instances. A segmentation is a method to derive a set of runnables, runnable dependencies, and runnable sequencing constraints. We analyze three different strategies for a segmentation: a segmentation where one runnable is created for *every component instance*, for *every region* of the behavior-RTSC of all component instances, and for *every transition* of all RTSCs. Additionally, we derive runnable dependencies and runnable sequencing constraints, which ensure that the execution order of the runnables is correct regarding the semantics of MechatronicUML. All segmentation strategies fulfill our requirements at least partially as described in Section 4.2.1.4 for **R1** - **R3** and in Section 4.2.3.7 for **R4**. Table 4.4 summarizes the fulfillment of all requirements.

Requirement	**Segmentation Strategy**		
	Per Component	*Per Region*	*Per Transition*
R1: Parallel Execution	(✓)	✓	✓
R2: # Runnables	✓	✓	(✓)
R3: Preserve Semantics	✓	✓	✓
R4: Time-Properties	(✓)	(✓)	(✓)

Table 4.4: Fulfillment of Requirements R1 - R4 for all Segmentation Strategies.

The semantic preservation of time properties is only fulfilled, if limitations regarding clocks are considered (cf. Section 4.2.3.7). We argue that the *One Runnable per Region*-segmentation is a good trade-off between possible parallel execution and complexity in the sense of the number of runnables and runnable dependencies. Furthermore, we propose to disregard the region priorities defined by MechatronicUML. The reason is, that the only purpose of these priorities is to define a deterministic execution of an RTSC. This prohibits parallel execution explicitly. Disregarding these priorities does violate the current semantics of MechatronicUML but does not change the behavior of the RTSC critically (cf. Appendix A). Hence, we propose to alter the semantics definition of MechatronicUML regarding region priorities in the future.

Moreover, in Section 4.2.4 we present a segmentation approach for continuous parts of the defined software system. For this, we define one runnable per continuous component that reads/writes the corresponding sensor value. The value is stored in a specific value variable. Each runnable of an RTSC using this sensor value defines a corresponding read/write access to this value variable. To avoid conflicting write accesses to an actuator value, we propose to use hybrid port values in one region only.

After the segmentation, the runnables are grouped to tasks that execute the runnables. This step is called *Partitioning*. We present an approach how the complexity of the input model for the partitioning algorithm can be reduced. This is possible by analyzing the behavior of a runnable (specified by an RTSC) for superfluous write accesses to labels. For this, we present a technique to reduce the number of runnable dependencies by using MechatronicUML's Model Checking Approach. In Section 4.3.1, we discuss how the partitioning and mapping algorithms can be used for our purposes.

5 Scheduling for ECU Networks

Software component instances do not run in isolation but are connected to other component instances for communication. Since we consider software with hard real-time requirements, this communication has to fulfill hard real-time requirements as well. In MECHATRONICUML, RTCPs are used on PIM level to define Quality of Service (QoS) assumptions for communication between interacting components. These QoS assumptions are used to verify the fulfillment of these hard real-time requirements using model checking. Hence, the mapping has to ensure these assumptions, especially if more than one ECU is available in the system.

Typically, in CPSs more than one ECU is used. For example, modern cars use more than 70 ECUs to execute software for several software systems [Bro06]. Consequently, the software components of a system have to communicate across a network of ECUs, which might be connected via several bus systems [Bro06]. Thus, the mapping of software components to ECU cores gets more complex. First, there are more possible target ECU cores. Second, communication latency between two interacting components might increase significantly if these components are mapped to different ECUs. Third, runnable properties have to be respected in the mapping as discussed in Chapter 4.

In this chapter, we present an approach that enables the developer to specify a multi-core scheduling for an ECU network that explicitly takes communication requirements in hard real-time systems into account. Figure 5.1 shows the process for this approach.

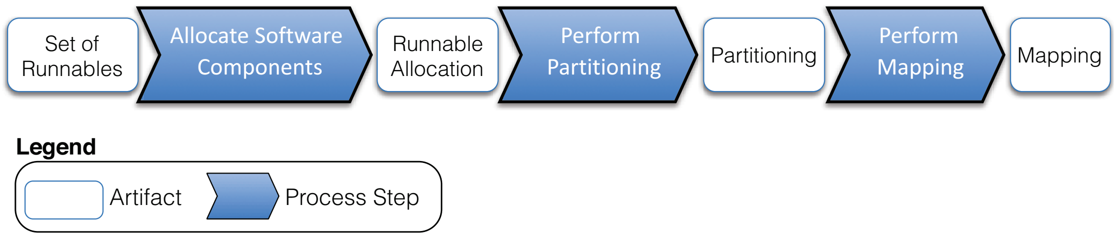


Figure 5.1: Development Process for ECU Networks.

The input for the process is a set of runnables with defined runnable properties produced by the segmentation as presented in Section 4.2. First, in the `Allocation`, we allocate runnables to ECUs. Since all runnables that belong to one component instances are highly dependend, we propose to allocate all runnables of a component instance to the same ECU. Hence, in this step, we allocate component instances to ECUs with respect to the runnable properties and to the communication requirements. At this point of development, we cannot apply a full schedulability analysis. Nevertheless, it is important that schedulability is considered in the allocation step. In our approach, we ensure that the processing capacity of the ECU is not exceeded by the the processing requirements for the allocated runnables, i.e., we discard allocations that are not schedulable already in this step. Additionally, we ensure that the real-time requirements for the communication are respected in the allocation. In particular, we extend the allocation step of the current MechatronicUML PSM process (cf. Section 2.5.5) by enabling to specify allocation constraints with regard to the runnables. The result is a set of runnables for each ECU. Second, we apply the steps `Partitioning` and `Mapping` to each ECU as described in Section 4.3.

The result of this approach is a valid allocation of component instances to ECUs that respects hard real-time requirements for the communcation in distributed systems and the runnable properties for all runnables. Additionally, a partitioning and mapping and, therefore, a scheduling is determined for each ECU.

This chapter is structured as follows: In Section 5.1, we discuss the requirements for this approach. Afterwards, we discuss how the schedulability requirement and the QoS assumptions of the RTCPs can be ensured during the allocation in Section 5.2. Finally, in Section 5.3, we apply the approach to the running example and discuss the results.

5.1 Requirements

In this section, we state the requirements for our approach by extending the requirements of Section 4.1. Hence, this approach has also to fulfill requirements **R1** to **R4**.

R5: Ensure Understandability: Finding a feasible allocation is not trivial. Thus, it is important that the specification of the allocation is still understandable and usable by the developer.

R6: Ensure schedulability: Partitioning and mapping is applied for each allocation. Thus, the schedulability of the allocation, resulting by our appraoch, should be considered.

R7: Preserve real-time communication requirements: The semantics of all RTCPs have to be respected in the generated source code and during the execution on the target platform. In particular, the defined timing behavior (specified by the `max-delay` in RTCPs) has to be ensured.

Additionally, we make some assumptions that simplify the explanation of the approach. In the following, we state and explain these assumptions.

A1: Runnable Properties: All properties of all runnables that were generated for a component instance are known, i.e., we can assume that values for the WCET, period and deadlines are given. This implies, that the segmentation of the model is already finished.

A2: Message loss: In any system, the communication channel can have errors or fail completely. Consequently, the sent message will not arrive in time or will never even reach the receiver. This can, in general, be detected on a lower layer, e.g., the data link layer [Zim80]. In case, such system failures occur, the system has to provide a fail-safe behavior. For our approach, we abstract from such failures and assume that each communication channel works without errors.

5.2 Defining Allocation Constraints

In this section, we present a method how the approach presented in Chapter 4 can be extended for multiple ECUs. As mentioned before, we propose to use a stepwise approach. The current allocation step of MechatronicUML allocates component instances to one ECUs. After that, a task is created for each component instance, which corresponds to the steps partitioning and mapping. We need to extend the concepts to reuse this step for the allocation of runnables that will be grouped to tasks in a later step. We state two important conditions regarding the runnables and their execution on multi-core ECUs as necessary conditions for a valid allocation and scheduling.

1. **Schedulability:** All runnables that are allocated to one ECU have to fulfill a necessary condition for schedulability: the total processor utilization of all allocated runnables has to be lower than 100%, i.e., the processing capacity of the ECU is not exceeded (cf. **R6**).
2. **Communication Latency:** All properties of all RTCPs of the system have to be fulfilled by the allocation and still hold at runtime (cf. **R7**).

Additionally, we present how these conditions can be expressed formally, which is needed for reusing the allocation approach of MechatronicUML by using the ASL.

In Section 5.2.1, we discuss how *Schedulability*-condition can be ensured. After that, in Section 5.2.2, we show how the *Communication Latency*-condition can be ensured, i.e., we analyze how the properties of the RTCPs can be preserved in general at runtime. Finally, in Section 5.2.3, we present how the results can be integrated into the existing allocation step of MechatronicUML.

5.2.1 Condition for Schedulability

It is important to consider the schedulability of the software already during the allocation. In this section, we present a method to ensure a necessary *Schedulability*-condition for runnables. At his point in time, we cannot apply a full schedulability analysis for the software, because we only have specified runnables, but do not know anything about possible tasks. Tasks will be determined in the partitioning algorithm in a later process step. Nevertheless, similar to Jatzkowski et al. [JKR15] , we can 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. Consequently, the processing capacity of each ECU C_{ECU} is defined as

$$C_{ECU} = |ECUCores| \tag{5.1}$$

Each runnable has a defined Period π and a WCET. Note that the WCET does depend on the executing ECU and may vary if the runnable is allocated to different ECUs. Let $WCET_{Runnable,ECU}$ be the WCET for a runnable and a specific ECU. We define the utilization factor of a runnable $U_{Runnable}$ for a specific ECU as

$$U_{Runnable} = \frac{WCET_{Runnable,ECU}}{\pi_{Runnable}} \tag{5.2}$$

If the sum of the utilization factors of all runnables exceeds the processing capacity of the ECU, it is impossible to find a scheduling for a given set of runnables (independent from the resulting tasks). If this sum equals the processing capacity of the ECU, the resulting scheduling would have to execute all runnables without any idling time. Since this is not realistic for complex systems, we define as a necessary condition that this sum has to be *less* than the processing capacity of the ECU.

$$\sum\nolimits_{r \in Runnables(ECU)} U_r < C_{ECU} \tag{5.3}$$

5.2.2 Condition for Communication Latency

An RTCP defines a contract between communication partners. Each communication partner has to refine the behavior of a role, defined by the RTCP (cf. Section 2.5.2). In this section, we present an approach how the max-delay defined in RTCPs can be ensured during the allocation.

An RTCP provides QoS assumptions, like the min/max delay for the communication. Furthermore, it refers to RTSCs that define the behavior of the roles and are refined for the RTSCs of each port that implements the specific role. We have to consider each instance of RTCPs in the component instance configuration separately. Figure 5.2 shows the RTCP that is applied to component `overtakerDriver` and `overtakerCommunicator` in the running example.

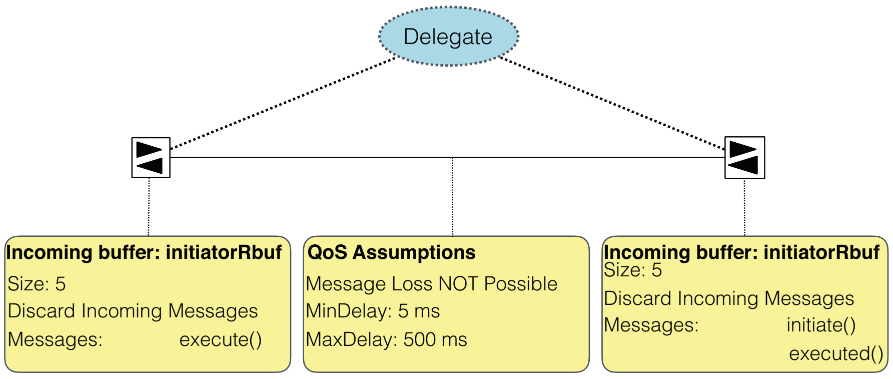

Delegate
Incoming buffer: initiatorRbuf
Size: 5
Discard Incoming Messages
Messages: execute()
QoS Assumptions
Message Loss NOT Possible
MinDelay: 5 ms
MaxDelay: 500 ms
Incoming buffer: initiatorRbuf
Size: 5
Discard Incoming Messages
Messages: initiate() executed()

Figure 5.2: RTCP Delegate with QoS Assumptions.

The RTCP defines a `min delay` of `5 ms` and a `max delay` of `500 ms`. Thus, the message has to be transmitted within this time interval. Otherwise, the message will be dropped, e.g., by the middleware of the system. For our approach, we focus on the max delay. If the message arrives before `min delay`, we assume that the middleware will retain the message until it can be delivered. Delivering a message relies basically on time for generating and sending the message, transmitting it from sender to receiver, and queuing it until the receiving process recognizes the message [TBW95]. The max-delay defines the maximum time span we have for sending, transmitting, and receiving the message. Thus, we have to consider these time frames for deriving the condition. Figure 5.3 shows these time frames in an exemplary schedule.

At r_{send} the message is sent by the sender runnable. At MW_r the middleware recognizes that a message has been sent. We call the interval between these points as *Time for Sending* and denote it by t_s. After that, the message is transmitted to the target and is put into the corresponding message buffer of the component instance at MW_d. We call the interval between MW_r and MW_d *Time for Trans-*

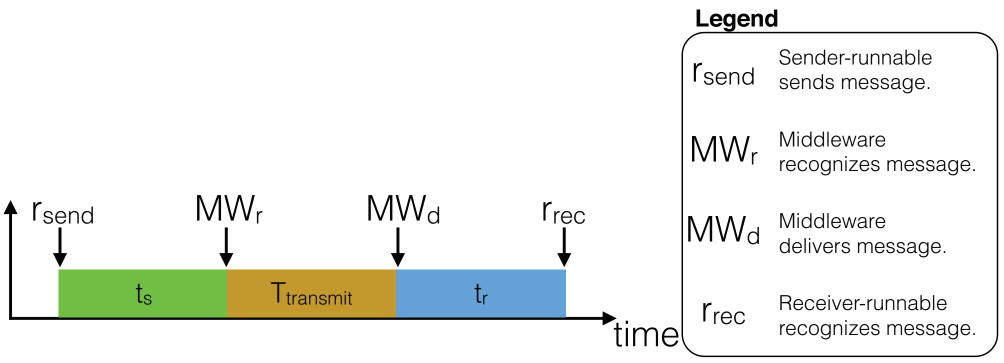

Legend
r_{send} Sender-runnable sends message.
MW_r Middleware recognizes message.
MW_d Middleware delivers message.
r_{rec} Receiver-runnable recognizes message.
r_{send}
MW_r
MW_d
r_{rec}
t_s
$T_{transmit}$
t_r
time

Figure 5.3: Transmitting a Message can be Separated into Three Time Slots.

mitting and denote it by $t_{transmit}$. At r_{rec}, the receiver runnable evaluates its transitions and recognizes the new message. We call the interval between MW_d and r_{rec} *Time for Receiving* and denote it by t_r.

In the following, we discuss the worst case estimation for each of these values. For this, we consider the communication between the ports `initiatorP` and `executorP` in the running example, which refine the roles of the RTCP `Delegate` shown in Figure 5.2. We assume that there is exactly one runnable for the sender (*sender runnable*) and one runnable for the receiver (*receiver runnable*). In the following, we consider the case, that both components and, therefore, both runnables are allocated to different ECUs. Figure 5.4 shows an exemplary schedule of the runnables for both ECUs.

For better readability, we abstract from tasks, but only show the execution of runnables. The *sender runnable* is released at its arrival time a_s(i.e., the execution is triggered by its task) and terminates before its absolute deadline d_s. Thus, the message is sent between these two points in time. The runnable for receiving the message is released at its arrival time a_r and terminates before its absolute deadline d_r. Thus, the message is received between these two points in time, if it is available in the message buffer of the component. Note, that the concrete point in time for sending and receiving cannot be determined. The execution of a runnable r (blue bar in the figure) depends on the starting time s_r and finishing time f_r, which are not necessarily the same points in time like arrival time a_r and relative deadline d_r (cf. Section 2.2.1).

Time for Sending Sending a message means to specify the message parameters and to delegate it to a middleware. The time for specifying the message and its parameters is negligibly short. We assume this time as a point in time and denote it as *message triggering time*. As stated before, we assume that sending and delivering the message is done by a middleware [BDG+14a]. This middleware will also be

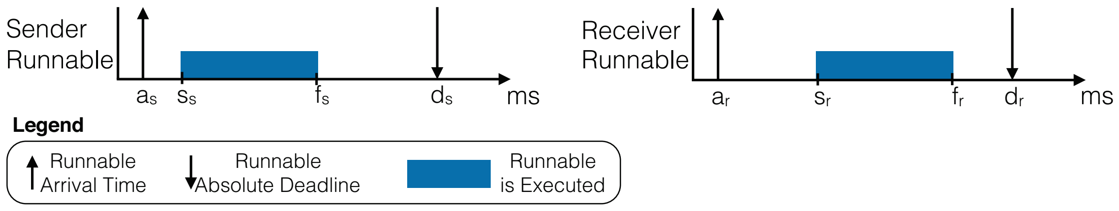


Figure 5.4: Exemplary Schedule for Sender Runnable and Receiver Runnable.

scheduled by the system. Thus, the time passed from *message-triggering-time* until the message is recognized by the middleware, does depend on the period/schedule of the middleware. We call this time *Time for Sending* and denote it by t_s (cf. Figure 5.3).

There are two different possibilities for the point in time when the middleware receives the message for further processing: Inside of the sending runnable (by calling a middleware function) or by a dedicated middleware task (cf. [RTI]). We argue for distinguishing two different cases for the former one: 1. Calling the method during the transition effects, and 2. Calling the method at the end of the runnable (or trigger the middleware-task by an event). For the former case, due to its definition, t_s will be zero (or at least negligibly short) if the middleware receives the message immediately at the *message triggering time*. This case would increase the WCET of the sender runnable by the time that is needed for executing the middleware method. For the latter one, the execution of the runnable is not affected. Hence, we argue to use this strategy and assume that the middleware receives and processes the message immediately after the runnable terminated. In general, this time can be estimated by the period of the sender runnable. In general, the upper bound for the deadline can be estimated by the period π of the runnable. Hence, we set $t_s = \pi_{senderrunnable}$. Note, that this value does not change, if the middleware is not executed after the runnable, but after the executing task. The reason for this is, that all runnables within one task and, therefore, the task itself do have the same period.

Time for Transmitting Transmitting a message from one to another component takes time, especially if sender runnable and receiver runnable are executed on different ECUs, which is our assumption in the following. We assume, that the time to transmit a message via a specific communication channel between two ECUs can be estimated by a worst-case time. We denote this time by $t_{transmit}$ (cf. Figure 5.3). This time depends on the type of the communication channel, e.g., CAN or FlexRay. Furthermore, it may depend on the utilization of the channel. However, we assume that this time is determined by the hardware-engineer and assume that it is defined in the model. Hence, each communication channel in the PDM has to provide a value for $t_{transmit}$. Additionally, the communication layers of the system, e.g., a middleware, take additional time, which also has to be considered for this time value. Thus, in this thesis we assume the sum of these time values to be fixed for each ECU-pair (per communication channel in the PDM) and described by $t_{transmit}$. For two specific ECUs e_1 and e_2, we denote $t_{transmit}$ as $t_{transmit(e_1,e_2)}$. Messages have to be routed via several ECUs if the communicating ECUs are not connected directly. In this case, $t_{transmit}$ depends on the communication time of each involved pair of ECUs. In this thesis, we do not consider this case and assume that all ECUs are directly connected, e.g., via the same bus.

Time for Receiving *Time for Receiving* describes the time it takes from delivering the message (MW_d in Figure 5.4) to the point in time when the message is received by the RTSC (r_{rec} in Figure 5.4). We denote this time by t_r (cf. Figure 5.3). According to the semantics of MechatronicUML [BDG+14a], the max delay is fulfilled when the message is put into the buffer of the receiver. This is atMW_d, which is defined by the point in time when the middleware delivers the message to the message buffer of the receiving port. In this case, t_r would be zero. However, since we have to ensure, that the message in the buffer is recognized in time by the receiving runnable, we also take the time into account it takes for recognizing the message in the buffer.

Since we want to find a worst-case estimation for t_r, we have to consider the execution of the receiver runnable. Figure 5.5 shows two execution cycles of the receiver runnable.

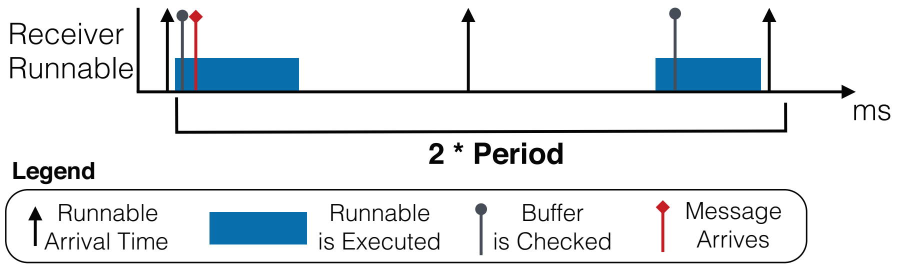


Figure 5.5: Periodic Execution of Receiver Runnable and Corresponding Receiving Time.

We assume that the message is put into the message buffer immediately after the receiver runnable checked the buffer. Hence, in this execution the message is not received by the runnable (the corresponding transition respectively). Since the receiver runnable is activated periodically, it has to be completely finished 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 upper bound for t_r.

Defining the Condition Using the upper bound values for t_s, $t_{transmit}$, and t_r, we can express a constraint implied by the QoS-assumptions of the RTCP. Let T_{RTCP} be the max-delay value of the RTCP. Since T_{RTCP} specifies the upper bound for sending, transmitting, and receiving the message, the sum of these three values has to be smaller than the max-delay. Consequently, for each RTCP that is used in the system, the following inequation has to hold:

$$t_s + t_{transmit} + t_r \leq T_{RTCP} \tag{5.4}$$

As $t_{transmit}$ depends on source and target ECU, it depends on the allocation of component instances to ECUs, if inequation 5.4 can be satisfied or not for each RTCP. Thus, they have to be respected in the allocation step in the development process.

5.2.3 Specification with the Allocation Specification Language

In this section, we show how the inequations 5.3 and 5.4 can be used in the current allocation step of MechatronicUML. Since there are many possibilities to allocate the runnables to ECUs in an ECU network, finding a feasible allocation is not a trivial task. Such a problem can be solved by translating it into an ILP and solving it by an existing approved ILP-Solver (cf. Section 2.4).

Currently, in the allocation step of MechatronicUML, an ILP is already used to find a feasible allocation of component instances regarding allocation constraints (cf. Section 2.5.5). Thus, we propose to express the inequation for each RTCP as an equivalent ILP-constraints and add them to the existing ILP for the allocation step. The resulting allocation will then respect both, the constraints defined in the allocation step and the constraints for the RTCPs. However, since the concept of runnables is currently not considered in the allocation step of MechatronicUML, we need to extend this step.

There are two possibilities to extend the ILP by constraints for the new conditions: Formulate the constraints using the ASL or generate ILP-constraints directly and add them to the ILP before the solver is called. The latter would be a static solution because it is fixed to one specific ILP-description, which might restrict the choice of an ILP solver. An additional disadvantage is, that maintaining the generated constraints is difficult, especially for developers without knowledge of ILP specifications.

In MechatronicUML, the ASL is used to specify custom constraints for the ILP. The advantage of using the ASL is, that it is easier to understand than large ILPs, especially for a developer without knowledge of ILP. Additionally, it is independent of the latter solving, e.g., the ILP-solver or even using other optimization techniques, because it is described by an ILP meta model, which is independent from the latter ILP syntax, e.g., for different solvers. Thus, we propose to specify the constraints resulting from the RTCPs in the ASL and adding it to the ASL-specification. Hence, all constraints for the ILP are defined in one common solver-independent language and can be adjusted by the developer before executing the ILP-solver. Thus, formulating the ILP-constraints for the RTCPs in this language and adding them to the specification file will increase the usability and maintainability for the developer.

Two steps are needed to extend the allocation step of MechatronicUML to express the inequations 5.3 and 5.4 in the ASL: First, we have to *extend the OCL-Context* of the ASL (cf. Section 2.5.5) to enable the use of runnables properties in the ASL. Second, we have to *specify the ASL-constraints*. After that, the ASL can be used to generate all corresponding ILP-constraints. Both steps are described in the following.

5.2.3.1 Defining the Constraint Context

The ASL uses the OCL-Context model that references the PDM (which specifies all available ECUs) and a CIC (which contains all component instances of the system) (cf. Section 2.5.5). Currently, the ASL supports referring to component instances in its constraints. However, for our approach, we also need to reference to runnables, since we are interested in the properties of the sender runnable and receiver runnable for the communication-constraint and for the utilization factor of each runnable. Thus, we have to extend the OCL-Context model of the ASL, such that the runnables are accessible in the ASL-constraints. An idea to solve this is to change the context of the ASL, such that runnables instead of component instances are allocated to ECUs. This might result in allocations where runnables of the same component instance are allocated to different ECUs, e.g., parallel regions of an RTSC. Since such regions are quite related and use shared variables and synchronization channels, it might not be useful to allocate them to different ECUs. Consequently, we propose to keep using component instances in the allocation but to reference all dedicated runnables for each component instance additionally. The advantage is, that we still allocate component instances to ECUs, but we can use the properties of the runnable within the ASL-constraints.

Therefore, we propose to extend the PIM of MechatronicUML by runnables. Since all runnable properties except the WCET are platform-independent, it does not mix the concepts of PIM and PSM. The WCET can be annotated for each ECU by a WCET-Extension that is already defined in the MechatronicUML Meta-Model for component instances and can be reused for our purposes. Figure 5.6 shows the OCL-Context model of the ASL including the referred extended PIM.

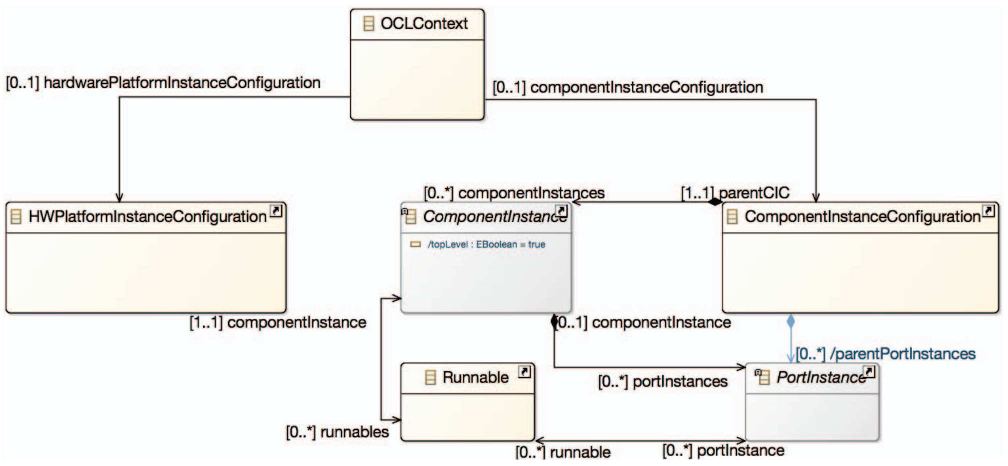

OCLContext
[0..1] hardwarePlatformInstanceConfiguration
[0..1] componentInstanceConfiguration
HWPlatformInstanceConfiguration
[0..*] componentInstances
[1..1] parentCIC
ComponentInstance
/topLevel : EBoolean = true
ComponentInstanceConfiguration
[1..1] componentInstance
[0..1] componentInstance
[0..*] /parentPortInstances
Runnable
[0..*] portInstances
PortInstance
[0..*] runnables
[0..*] runnable
[0..*] portInstance

Figure 5.6: ASL-Context-Model with Runnables.

The `OCLContext` is the root, which can be seen as the entry-point for each ASL-constraint. It refers to the HPIC that describes the hardware of the system and the CIC that describes the software of the system, e.g., the PIM. Note that the new class `Runnable` is not added in the Context-model, but in the PIM meta-model of MechatronicUML and, therefore, accessible by the context-model of the ASL. The advantage is, that the ASL is still usable for the current concepts, but is additionally able to refer to runnables as needed for our concepts. Each component instance refers to a set of runnables that are used to execute the behavior of the component. Additionally, we need a reference from the port instance (that refines the role of the RTCP) to the corresponding runnable to express the ASL-constraint.

5.2.3.2 Specifying Allocation Constraints

We have to find an allocation, for which the conditions 5.3 and 5.4 hold. Thus, we have to specify constraints for each possible combination of ECUs and runnables. An advantage of using the ASL is, that many of the resulting constraints for these conditions can be expressed by one ASL-constraint [PH15]. The ASL defines four different kinds of constraints. We reuse the constraint kind *RequiredResource* for the new constraints since it provides the needed functionality to generate the ILP-constraints. Using the extended context-model for the ASL, each of both inequations can be specified in one ASL constraint that hides the concrete implementation from the developer. Listing 5.1 shows the concrete call of the ASL-constraints.

Listing 5.1: ASL constraint specification for the new constraints.

```
--ensure RTCP max delays in allocation
        constraint requiredResource maxDelay4RTCPs {
                ...
                ocl self.RTCPMaxDelayConstraints();
        }

--ensure necessary condition for scheduling
        constraint requiredResource validUtilization{
                ...
                ocl self.validUtilizationFactors();
        }
```

The developer has to specify a requiredResource constraint for each of both conditions. We ommit some ASL specific parts of the constraints, indicated by three dots. The concrete OCL code to determine all constraints is called by one OCL-Operation for each condition. Hence, the developer does not need any domain-knowledge to use our approach in the allocation. These ASL-constraints are translated to corresponding ILP-constraints during the allocation automatically. In the next section, we show the expected number of ILP-constraints shortly and how they can be derived for both conditions.

5.2.3.3 Derive ILP-constraints

In this section, we show the resulting ILP-constraints for our conditions that we derive from our allocation specification automatically. In general, inequation 5.4 can directly be translated into an ILP-constraint. For this, we have to add an ILP-decision-variable (cf. Section 2.4) to the constraint. Let x_{r_s,e_x,r_r,e_y} be the decision-variable that is 1, if runnable r_s is allocated to ECU e_x and runnable r_r is allocated to ECU e_y. In all other cases it is 0. Then, we can define the following ILP-constraint:

$$(t_s + t_{transmit(e_x,e_y)} + t_r) * x_{r_s,e_x,r_r,e_y} \leq T_{RTCP} \tag{5.5}$$

where t_s is the *Time for Sending*, t_r is the *Time for Receiving*, and $t_{transmit(e_x,e_y)}$ the time for sending a message from ECU e_x to ECU e_y.

Obviously, this inequation has to hold for every combination of the RTCPs with every possible pair of ECUs. Thus, we have to specify $|ECU|^2 * |RTCP|$ many constraints, because we have to consider all possible pairs of ECUs for each RTCP.

Furthermore, inequation 5.3 can also be translated directly into an ILP-constraint by adding a decision-variable to the inequation. Let $x_{r,e}$ be this decision-variable that is 1, if runnable r is allocated to ECU e. In all other cases, it is 0. Let C_e be the processing capacity of ECU e. Then, we define e the following constraint for each ECU.

$$\sum\nolimits_{r \in Runnables} (U_r * x_{r,e}) < C_e \tag{5.6}$$

Thus, we have to define $|ECU|$ many ILP-constraints. Note, that U_r is not a static value for each runnable but does depend on the executing ECU (cf. equation 5.2 on page 82).

For both new inequations, we have to define $|ECU|^2 * |RTCP| + |ECU|$ many ILP-constraints in total that can be expressed by only two ASL-constraints as shown in Listing 5.1.

5.3 Example and Discussion

In this Section, we discuss our approach with regard to the requirements stated in Section 5.1 by applying the constraint for ensuring the max-delay of RTCPs to the running example. For this, we assume that the segmentation of the CIC is already finished and therefore, we have a defined CIC with references to dedicated runnables. We assume that the segmenation *One Runnable Per Region* is applied for discrete component instances. Figure 5.7 shows a sketch of the CIC for component overtakerVehicle and the resulting runnables.

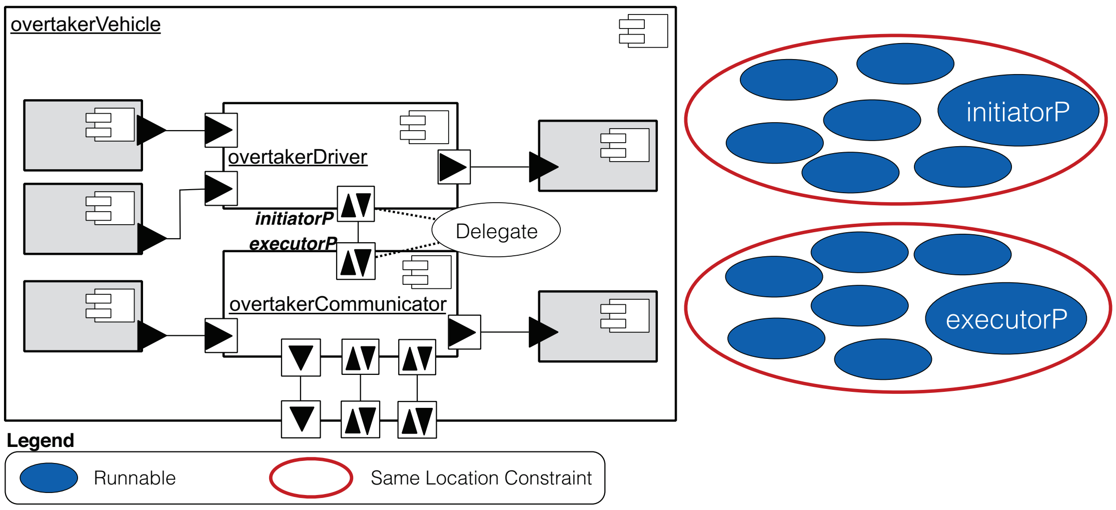


Figure 5.7: CIC for OvertakerVehicle and the Resulting Runnables.

As example, we apply the communication latency condition (cf. inequation 5.4) in the allocation. Hence, we focus on the runnables that communicate and, therefore, contain port behavior, i.e., runnables `initiatorP` and `executorP`.

Additionally, we assume that the PDM is defined and accessible by the ASL. The developer may also define ASL-constraints to ensure important allocation requirements, e.g., that all component instances of the overtaker are allocated to the ECUs of one vehicle using the *sameLocation* constraint of the ASL. Additionally, we assume that all continuous component instances and the connected discrete component instances are allocated to the same ECU. This allows us to use one runnable per continuous component instance as presented in Section 4.2.4. In Figure 5.7, this is indicated by the red lines. Additionally, the RTCP `Delegate` is used between the ports `initiatorP` and `executorP`, which defines a max delay of $500ms$ (cf. Figure 5.2 on page 83). Let us assume that the developer added the ASL constraint for communication latency condition (cf. inequation 5.4) to the allocation specification. Table 5.1 shows exemplary values for the period of sender runnable and receiver runnable that are used in the constraints.

Table 5.1: Exemplary Runnable Properties for the Example-CIC.

Runnable	Period π	t_s	t_r
initiatorP-Runnable	180 ms	180 ms	360 ms
executorP-Runnable	100 ms	100 ms	200 ms

Thus, corresponding to Section 5.2.2, we have $t_s = \pi_{initiatorP} = 180ms$ and $t_r = 2 * \pi_{executorP} = 200ms$. Furthermore, we assume that $T_{transmit}$ between ECU e_1 and ECU $e2$ has the constant value of $150ms$. $T_{transmit}$ for the communication on the same ECU has the constant value of $0ms$, e.g., when global variables are used for implementation. However, if more ECUs are involved and, therefore, $T_{transmit}$ may vary between different ECU-pairs, $T_{transmit}$ can be annotated to the used communication channel.

In the following, we show the four ILP-Constraints resulting from the ASL-constraint for the RTCP `Delegate`. For better readability, we focus on one direction of the communication, i.e., sending a message from `initiatorP` to `executorP`. Additionally, the result for each inequation is shown using the values from Table 5.1.

$$\begin{aligned}(t_{s,iniP}+T_{transmit(e1,e1)} + t_{r,exeP}) \quad & *x_{CI(iniatorP),e1,CI(executorP),e1} \leq Delegate_{maxDelay}\\ (180+0+200) \quad & *x_{CI(iniatorP),e1,CI(executorP),e1} \leq 500 \qquad (5.7)\end{aligned}$$

$$\begin{aligned}(t_{s,iniP}+T_{transmit(e1,e2)}+t_{r,exeP}) \quad &*x_{CI(iniatorP),e1,CI(executorP),e2} \leq Delegate_{maxDelay}\\ (180+150+200) \quad &*x_{CI(iniatorP),e1,CI(executorP),e2} \leq 500\end{aligned} \tag{5.8}$$

$$\begin{aligned}(t_{s,iniP}+T_{transmit(e2,e1)}+t_{r,exeP}) \quad &*x_{CI(iniatorP),e2,CI(executorP),e1} \leq Delegate_{maxDelay}\\ (180+150+200) \quad &*x_{CI(iniatorP),e2,CI(executorP),e1} \leq 500\end{aligned} \tag{5.9}$$

$$\begin{aligned}(t_{s,iniP}+T_{transmit(e2,e2)}+t_{r,exeP}) \quad &*x_{CI(iniatorP),e2,CI(executorP),e2} \leq Delegate_{maxDelay}\\ (180+0+200) \quad &*x_{CI(iniatorP),e2,CI(executorP),e2} \leq 500\end{aligned} \tag{5.10}$$

The results show, that the inequations 5.7 - 5.10 can only be fulfilled, if both runnables are executed on the same ECU. Consequently, the QoS assumption of the RTCP can only be satisfied, if both component instances are allocated to the same ECU. Since we specified that all continuous component instances have to be allocated to the same ECU as the connected component instance, the resulting allocation is, that all runnables have to be allocated to the same ECU. Thus, the second ECU is not used. This might be a problem, if additional ASL-constraints are used that restrict the allocation of all component instances to the same ECU, e.g., when the constraint for the utilization factor cannot be fulfilled by this allocation. Hence, the developer should check all runnable properties and the RTCP for possible changes to relax the conditions for the ILP.

For each ECU, the resulting runnable sets can then be processed further by the Partitioning- and Mapping-Step. At the end, a mapping that contains tasks is provided to execute the software on each ECU.

We argue that all requirements (**R1** - **R7**) are sufficiently fulfilled by our approach. Since we do not change any property of the runnables, **R1** to **R4** are fulfilled, if a feasible scheduling can be found for the resulting allocation. Otherwise, the developer may refine the models and check the allocation constraints. Using our stepwise approach, the complexity of the complete problem is solved step by step and, therefore, we state that it is maintainable. Additionally, we integrated our concepts into the current allocation step of MechatronicUML and extended the ASL by enabling the possibility to reference runnables in the constraints. Thus, the complexity of the ILP as well as the understandability for the developer is appropriate. Hence, we argue that **R5** is fulfilled. By considering the processor utilization, we consider the schedulability during the allocation step. Since we only check a necessary condition for schedulability, schedulability cannot be ensured in all cases. However, it supports the developer by finding a feasible and schedulable allocation by discarding allocation that would not be schedulable. Thus, we argue that **R6** is only partially but sufficiently fulfilled. The QoS-assumptions of all RTCPs are fulfilled, if a feasible allocation can be found by the ILP-solver, since we add corresponding constraints for each RTCP to the ILP. Thus, **R7** is also fulfilled.

6 Related Work

In this section, we discuss publications and approaches related to our thesis. We identified three main categories for structuring this chapter. In Section 6.1, we discuss other component-based approaches for designing CPS. Section 6.2 presents approaches that focus on the deployment of CPS, i.e., approaches for allocating software to the platform and for generating source code. Finally, we discuss existing approaches for deployment in the context of MechatronicUML in Section 6.3.

6.1 Component-Based MDSD for CPS

There are already several component-based approaches for developing software systems. For a component-based approach, an appropriate underlying component model is needed. Surveys by Lau and Wang [LW07] and by Crnković et al. [CSVC11] classify component models in general. However, for our thesis only component models for developing CPSs are relevant. Hŏsek et al. [HPB+10] present a survey for component-based approaches for this domain. Furthermore, this survey focuses on approaches that also provide a method and tooling. Additionally, in [BDG+14a] several component models are presented that are related to MechatronicUML. In the following, we state the most related approaches and state similarities and differences to our approach. We focus on approaches that consider at least partially concepts for partitioning, mapping, or deployment.

ProCom ProCom [VSC+09, BCC+08] provides a component model for embedded systems. The main focus of ProCom is the development of real-time systems in the automotive and telecommunication domains. ProCom contains two layers: *ProSys* and *ProSave*. ProSys is the top layer and describes concurrent running subsystems that can communicate asynchronously. ProSave is used to describe each of these systems and consists of passive components that define ports for data-exchange and trigger-ports. Passive components are not executed periodically but have to be triggered by an event that is received via a trigger-port. Data ports are used to exchange data between components. Similar to the RTSCs of MechatronicUML, ProCom provides a modeling language that is based on Final State Machines enriched by features of Timed Automata. Based on that, ProCom provides functionality to compute (real-time related) dependencies of the model that can affect the scheduling. Nevertheless, ProCom does not provide a view to model the behavior of the ProSave components. It assumes that the behavior will be provided by a C function. Additionally, ProCom provides a code synthesis [BC11] that aims to generate code that preserves the semantics of ProCom at runtime. In particular, the code synthesis aims for a correct order of executing data accesses and provides for this purpose a verification approach based on timed automata. Similar to the partitioning in Amalthea, ProCom provides an approach in [BC11] to group data ports that access the same data to determine groups of ports that can be executed concurrently. However, in contrast to our approach, the resulting system is mainly event-triggered, which does not allow a statical timing analysis like in our approach. Since the behavior of the components is provided by a C function, model checking and a model-driven segmentation like in our approach is not possible. Nevertheless, in [BC11] a formalization of the generated code is provided, which might be interesting for future work.

SOFA HI SOFA HI [PWT+08, PKH+11] is based on the SOFA 2.0 [HPB+05] and defines a component model for embedded systems with "emphasis on timing and memory constraints" [PWT+08]. It provides a hierarchical component model with clearly defined interfaces. Thus, it distinguishes primitive components that correspond to a piece of source code and composite components that encapsulate other components. SOFA HI also provides a code generation that aims to fulfill timing requirements [PKH+11]. In particular, it is used for generating code skeletons for all primitive components. Furthermore, in the deployment glue-code is generated that ensures the correct communication between the components and the System API. The generated code is organized in so-called containers that can be deployed to the platform. In contrast to our approach, the behavior of the components is not specified model-driven and, therefore, a model-driven segmentation like in our approach is not applicable. Furthermore, the component behavior is not considered during the code-generation or for the deployment. Nevertheless, in SOFI HI analyses on the deployed system, e.g., WCET analysis is suggested, which should be considered for our approach as well in future work.

PECOS The PECOS approach [NAD+02, GCW+02] aims to enable component-based software development for embedded systems. In PECOS, three different component kinds are used: Active Components, Passive Components, and Event Components. Active Components are executed in an own thread. Passive Components are not executed in an own thread, but can be called synchronous by Active Components. Event Components are triggered by special system events, e.g., by hardware interrupts. In PECOS, each component is implemented manually and has to define a fix cycle time. This cycle time is comparable to the period in our approach. Thus, in contrast to our approach, PECOS does not allow an automatically segmentation, since all properties (including the implementation of the behavior) has to be specified by the developer manually.

MEMCONS MEMCONS (Model-based EMbedded CONtrol Systems) [MSAK15] aims to provide a model-driven framework for embedded systems to support a seamless development from requirements to deployment. MEMCONS supports the exchange formats for AUTOSAR and OSEK models (AUTOSAR XML and OIL). It also supports the development for multi-core environments. Since it follows the AUTOSAR methodology, it provides component-based development of the system and platform independent specification of the application components. It also provides an automatic mapping approach for mapping tasks to multi-core ECUs. Furthermore, an analysis of timing constraints can be applied for the deployed system. However, in contrast to our approach, MEMCONS does not focus on early verification of the PIM. Furthermore, the behavior of the software components is not specified model-driven and cannot be used for segmentation as done in our approach.

CHESS/CONCERTO The CONCERTO project [prob, pro14] (follow-up of the CHESS project [proa]) aims to be a seamless development process for CPS. It adapts the UML-Profile MARTE [SG13] for modeling the system. Similar to MechatronicUML, it provides views to define the PIM. In particular, it is possible to model the system's structure with components in the *component view* and specify the behavior using activity diagrams and state machines in the *functional view*. Since modeling time constraints is not considered, the used state machines are less expressive than the RTSCs in MechatronicUML. In contrast to our approach, this approach doesn't focus on the specification and verification of time constraints but provides analyses for real-time schedulability. Furthermore, it provides models to describe the hardware and the PSM. Support for multi-core ECUs is not in the focus of CHESS, but is developed in the upcoming CONCERTO project.

6.2 Deployment of Safety-Critical Software

In the deployment of safety-critical software, several aspects have to be considered. In particular, we focus on the safe execution of software. Safe means, that all safety requirements, e.g., timing constraints, are fulfilled at runtime. In this section, we present related work for two important aspects: In Section 6.2.1, we present approaches that deal with the source code generation for models that explicitly define timing behavior, i.e., we focus on the execution of timed automata (TAs) [AD94]. In Section 6.2.2, we present approaches regarding scheduling of safety-critical software on RTOS.

6.2.1 Source Code Generation for Time-Critical Systems

In this section, we state approaches for generating code for time-critical systems. Since RTSCs use features of timed automata and Uppaal [LPY97] is used for verification, we present approaches that generate source code from TAs automatically.

TIMES Tool TIMES [AFM+04] is a tool for schedulability analysis based on TAs. It allows for schedulability analysis for preemptive and non-preemptive strategies for mixed tasks sets (tasks with timing, precedence, and resource constraints (cf. Section 2.2.1)). Tasks can have arbitrary arrival times (periodic and sporadic), which are modeled by so-called TATs. A TAT is a TA, where the locations are enriched by real-time tasks [FPY02]. Each task is described by "worst execution time and deadline, and possibly other parameters such as priorities etc. for scheduling" [AFM+02]. Furthermore, they assume that the task behavior is provided in source code. Additional concepts allow to specify an expression that modifies global/shared variables for communication between the tasks [AFM+02].

Using these input information, TIMES can then provide a schedulability analysis of the given tasks. Using the model checker Uppaal [LPY97], it can be proven, if a set of tasks is not schedulable for all cases. Additionally, TIMES uses an approach to automatically generate C-Code for the execution of the specified system as further described in the next paragraph.

In contrast to our approach, the focus of TIMES is schedulability and not to find properties of the tasks. However, TIMES allows quite mature schedulability analysis, if mixed task sets are used. Thus, this can be helpful, to find schedulings for MechatronicUML if mixed task sets or event-triggered task sets are used.

Code generation for timed automata The concepts used in TIMES for generating code from timed automata are presented in [Amn03]. Since the concepts in general are also applicable for generating code without the purpose of schedulability analysis, we present these concepts in this paragraph in more detail. In [Amn03], Amnell state that two types of non-determinism can occur in timed automata that have to be considered in the code generation: *external non-determinism* and *time non-determinism*. External non-determinism means "that several actions may be simultaneously present from the environment" [AFM+04], i.e., several transitions are enabled. Similar to MechatronicUML, they introduce priorities for transitions that are used to order the transition execution to resolve this non-determinism. Time non-determinism means "[...] that enabled transition[s] can be taken at any time point [...]" and is solved by "adopting the so-called maximal-progress assumption [Wan91]" [AFM$^+$04]. This means, that the controlling task fires all enabled transitions until no transition is active. They state, that the code generation is platform independent since it also generates a runtime-system that handle task activation and system events. However, 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. Nevertheless, in future work it might be interesting to evaluate executing a MechatronicUML system using this approach.

In [OCH] and [PKM], further approaches are presented to generate executable code for timed automata. In [OCH], Opp et al. focus on memory- and speed-efficient execution of the generated code. For this, they realize the execution by generating functions for each location of the automaton. Additionally, Opp et al. restrict the TA for the behavior specification to deterministic features. Hence, invariants and urgent transitions are not supported in this approach. Furthermore, they allow modeling an additional TA that describe the environment of the system. This TA is not restricted to deterministic features and can use therefore urgency and invariants. Environment does not mean the executing platform, but an environment that provide input data to test the specified behavior. However, this automata is not considered in the code generation for the system behavior.

In [PKM] on the other hand, Pedersen et al. present an approach, where invariants are allowed in the specification. However, they do not analyze if it all invariants can be guaranteed at runtime. To support the developer by finding such violations at runtime, a warning is generated, if an invariant gets violated.

In contrast to our approach, in both approaches properties of the target platform are not considered, e.g., the number of ECU cores or scheduling strategies. Furthermore, both approaches do not consider distributed systems. However, using timed automata for testing the environment might be interesting to test the behavior of RTSCs. This concept might also be useful for determining intervals of enabled transitions as needed in the concept in Section 4.2.3.5 and 4.2.3.6.

6.2.2 Scheduling of Safety-Critical Software

There are several approaches for scheduling software in general. Simple strategies like Rate Monotonic [But11], advanced strategies like Priority Server [But11], and strategies like the Priority Ceiling Protocol to prevent deadlocks in priority-based systems. In [DB11], Davis and Burns present in a survey the most important scheduling-approaches for hard real-time systems. However, in this section we do not focus on scheduling in general, but on approaches that focus on considering schedulability during the model-driven specification of the software. For this, we discuss two approaches, presented in [LBD$^+$12] and [MAAK15], that focus on modeling parts of the RTOS or the scheduler. Furthermore, we show an approach for determining task priorities presented in [LSC13], which seems to be interesting for future work. Moreover, we show an approach by Lukasiewycz et al. [CM05] for scheduling the execution of hardware controllers.

Modeling the RTOS In [LBD$^+$12], Lelionnais et al. present an approach where the target RTOS is explicitly modeled in the PDM. For specifying the RTOS, the domain-specific modeling language RTEPML (Real-time Embedded Platform Modeling Language) [BD10] is used. In particular, RTEPML is extended by elements to describe the behavior of the RTOS, i.e., tasks and semaphores. Following the MDA approach, the PIM that describes the application behavior and the PDM that specifies the target RTOS are transformed into a PSM. Lelionnais et al. [LBD$^+$12] state that the advantage is, that verification techniques like model checking can then be applied to the PSM, which then consider both the application behavior and the behavior of the underlying system resources. In contrast to our approach, where we use meta-models of MechatronicUML and Amalthea for PDM, in [LBD$^+$12] concrete hardware properties are not considered. Furthermore, distributed systems and multi-core environments are not taken into account. However, extending this approach for resource management on multi-core environments might be useful to improve the allocation regarding resource management in MechatronicUML. This should be compared to the resource management applied by the partitioning and mapping of Amalthea.

Generating OIL In [MAK14], Macher et al. present an approach for enhancing a model-driven framework for system engineering and safety engineering. The main focus is to ensure the seamless fulfillment of safety requirements in the whole development process of the system. The approach allows to specify AUTOSAR compliant software systems and to generate the systems structure, i.e., component structures with interfaces, basic software structure, and information for safety artifacts. One focus of the approach is gathering tracing information for the system. For this, they use a model-driven description of the target RTOS [MAAK15]. Since the target is an AUTOSAR system, they provide a model-driven editor to create OIL-specifications (cf. Section 2.2.3) for the target system. In contrast to our approach, they do not focus on the behavior specification, but on the systems structure. However, currently MechatronicUML does not provide a mature PDM for describing RTOS artifacts. Amalthea, on the other hand, provides a PDM that is high related to the AUTOSAR specifications. Concepts for tracing between PIM and the target RTOS might be reusable, especially the functionality of importing and exporting OIL files into model-based descriptions.

Determining Task Priorities In [LSC13], Lukasiewycz et al. present an approach to derive task priorities in event-triggered systems. The input for the algorithm is a task graph and a mapping. A task graph describes all tasks of the system and their message-communication. The mapping describes the assignment of tasks and messages to resources, e.g., ECUs or busses. They assume that each task has a defined period, deadline and WCET. Lukasiewycz et al. [LSC13] present an algorithm to determine optimal priorities for these tasks to ensure a feasible scheduling with low response-times in event-triggered systems. In contrast to that, we focus on time-triggered systems only and use algorithm of Amalthea to find a feasible scheduling. However, we do not consider priorities of tasks in our approach explicitly. Hence, this approach might be interesting to improve the task priorities for the mapping in our approach.

Scheduling for Controller In [CM05], Caspi and Maler discuss possibilities how the execution of hardware controllers can be scheduled deterministically and, therefore, can be integrated into software system. They discuss the problem that controller often use *timeless* descriptions (using mathematical functions), whereas in RTOS the execution of the controller program is triggered “every T time units”. Besides the discussion how to realize the implementation of the controller, Caspi and Maler [CM05] propose techniques how to integrate such program snippets into a system running on an RTOS. Furthermore, they discuss first ideas for solving the problem of controllers in distributed systems. However, in this approach only hardware controller are considered explicitly. Nevertheless, Caspi and Maler [CM05] propose a time-triggered execution of the controller code, similar to our approach of specifying one runnable per continuous components (that are basically an abstract description of hardware controllers). Currently, the actual implementation of continuous component can be realized by using tools like Matlab/SimuLink. Hence, the approach in [CM05] might be an interesting extension for this topic in future work.

6.3 Deployment of MechatronicUML

There are already approaches how software specified by MechatronicUML can be deployed to a platform. In particular, approaches exist that investigate how source code can be generated and how it has to be scheduled. The most related are approaches by Burmester et al. [Bur06, BGS05], and Teetz [Tee12]. In the following, we describe each of both approaches and their relation to our approach.

Burmester et al. In [Bur06], Burmester presents an approach for *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. In this work, RTSCs are used to describe the discrete behavior of software components of the system. Burmester derives a partitioning and scheduling strategy for the correct execution of RTSCs on single-core environments [Bur02, BGS03]. This thread is called the *main thread*. In particular, the approach was implemented for an (object-oriented) Real-time Java environment. Each RTSC is executed by one periodic thread (which is equivalent to a task in the terms of RTOS). Hence, it is comparable to the *One Runnable Per Component* segmentation and, therefore, prohibits parallel execution of the RTSC. Until the end of its period, this task "checks for all transitions which can be triggered from the beginning of its last period until and during the duration of the current period" [BGS03]. If an enabled transition is found, transition effects and actions are executed and the new state is set. In [BGS03], Burmester et al. propose to execute actions with WCETs in separate threads that are too high to be executed within the period of the main thread. In our approach, we suggested adapting this concept, to decrease the WCET of runnables. In contrast to our approach, this approach does not consider multi-core platforms.

Additionally, Burmester et al. propose a technique to determine periods for the main-thread. For this, two equations are proposed that determine a minimal and a maximal value for this period. Then a period is chosen, such that "at least one equation [holds] for every transition" [BGS03]. However, they focus on the execution of transitions, especially on executing actions without violating the deadline. In contrast to our approach, the use of invariants and clock constraints is not considered in these equations.

Simulation of OSEK/VDX tasks In [Tee12], Teetz presents an approach to map the PIM of MechatronicUML to RTOS tasks. In particular, the target platform is a proprietary simulation environment by INCHRON [INC]. Similar to our approach, the target platform is conform to the OSEK/VDX standard. Teetz maps in the approach each state of all RTSCs to one system task. A system event is triggered, when an outgoing transition of the task gets fired. Then, all transition effects are executed and the currently active task terminates. The triggered event activates the task that is responsible for the target state of the transition. Thus, in contrast to our approach this approach is event-triggered, although only events are used that were set by the system. Grouping similar tasks, like runnables in our approach, is not possible due to the definition of tasks in OSEK/VDX. Furthermore, parallel execution in multi-core environments is not considered. Teetz uses fixed values for the WCET of tasks and states that each task continuously executes the evaluation of transitions. Tasks release the processor for $5ms$ if no enabled transition is found to allow other tasks to be executed. Since this approach is event-triggered, no technique for periods is proposed. Also invariants, clock constraints, and deadlines are not explicitly considered and have to be tested in the simulation environment. EventChains are used to specify timing constraints that can be automatically verified in the simulation environment and is related to the timing constraint concept in Amalthea. This concept should be considered in the tracing of the software.

7 Implementation

For the purpose of evaluation, we implemented the concepts introduced in Chapter 4 and Chapter 5 prototypically. This chapter addresses the description of this concrete implementation and the integration into the current tooling support of MechatronicUML and Amalthea. This chapter is structured as follows. In Section 7.1, we introduce technologies that were used for the implementation. In Section 7.2, we give an overview of the implemented concepts in relation to the process presented in Chapter 3. In Section 7.3, we describe extensions of the meta-model we created and explain how we implemented the segmentation presented in Chapter 4. In Section 7.4 we explain the implementation for the concepts presented Chapter 5. In Section 7.5, we explain the implementation for transforming the models to Amalthea and how to apply the partitioning and mapping algorithms. Finally, in Section 7.6, we state known limitations to the current implementation.

7.1 Used Technologies

In this section, we introduce the technologies we used for implementing our concepts. Since the tooling of MechatronicUML and Amalthea are implemented as Eclipse-projects, all of our implementations are provided as Eclipse-plugin also. For the used technologies, we follow accepted standards as the MDA [Obj03] approach that provides several standards for specific tasks in MDSD approaches. All of the used technologies are part or implementations of these standards. First, we explain main concepts of *Ecore* and the *Eclipse Modeling Framework* (EMF) [Ecla], since we use these technologies for implementing our meta-models. After that, we introduce languages that were used for implementation: *Object Constraint Language* (OCL) and *Query/View/Transformation operational* (QVTo). Both languages are already used in the tooling of MechatronicUML and is a mature tooling for our purposes.

Ecore and EMF The meta-models for MechatronicUML and Amalthea are defined using the Ecore meta-meta-model provided by the Eclipse Modeling Framework (EMF) [Ecla]. Ecore is an implementation of the MOF subset EMOF (Essential MOF) [Obj11a] defined in the MDA standard. It can be used to describe meta-models by classes and their associations, i.e., as a class diagram. Since we extended the meta-model of MechatronicUML for the concepts of Chapter 5, we integrated our changes into the current Ecore file of the MechatronicUML meta-model.

OCL The Object Constraint Language [Obj10] (OCL) is a declarative programming language to specify constraints and invariants for meta-models defined with MOF. It is part of the MDA standard and part of the UML. In general, it is used to describe constraints and invariants in models that are part of the static semantics. In the ASL, OCL is used to query Tuples of model elements that describe ILP constraints. Thus, we use OCL mainly for this purpose when extending the ASL. Additionally, OCL is part of all model transformation standards of the OMG, e.g., QVTo, which is explained in the next paragraph.

QVTo QVTo stands for *Query/View/Transformation operational* [Obj15] and is also part of the MDA standard. QVTo is an imperative model transformation language. An implementation of QVTo is integrated [Eclb] in the Eclipse Modeling Tools. Thus, it allows for writing transformations for models of meta-models defined by EMF (Ecore files). Since both the meta-model of MechatronicUML and the meta-model of Amalthea are implemented by EMF, we use the Eclipse QVTo implementaion for the implementation of the model-to-model transformations.

7.2 Overview

In this section, we give an overview of the implemented concepts. We extended the meta-model of MechatronicUML and implemented the concepts of Chapter 4 and 5 using QVTo and OCL. In the following, we give an overview to show which parts are currently implemented, and in which order they have to be applied to the models. Figure 7.1 shows transformation steps that are considered in the current implementation and the resulting model artifacts.

The input model is a CIC of the complete system, e.g., the whole *Overtaking With Approacher* scenario. For the segmentation, we extend the meta-model of MechatronicUML and implement an endogenous, vertical (cf. Section 2.1.2) QVTo transformation **(1)**, which is provided as an Eclipse plugin and is further described in Section 7.3. The reduction of runnable constraints (cf. Section 4.3.2) is not implemented yet, but can be applied by the developer using the current model checking approach for RTCPs **(2)** manually. The result of this step is a CIC enriched by runnables, labels, and label accesses.

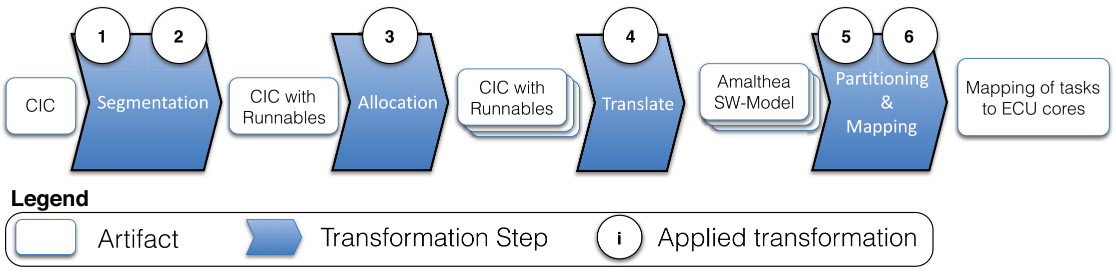


Figure 7.1: Overview of Implemented Concepts.

Additionally, we implemented the extension of the ASL as OCL operations **(3)**, such that they can be used during the allocation. Note, that the OCL-Context of the ASL is extended by the meta-model extension and does not need to be extended separately. The result of the allocation step is a set of CICs. In particular, there is one CIC defined for every ECU, e.g., a CIC for each of the three vehicles of the running example.

Furthermore, we provide an exogenous, horizontal QVTo transformation to AMALTHEA **(4)**. We create one AMALTHEA model file for each ECU (its CIC) that specifies the runnables, and their dependencies and properties, e.g., we create one AMALTHEA model for each vehicle. Hence, at the end, partitioning and mapping can be applied by the tooling of AMALTHEA (**(5)** and **(6)**). The result of this tool-chain is a mapping of tasks to ECU cores for each AMALTHEA model file and can be used as input for the code generator.

Figure 7.2 shows the created and used Eclipse-plugins for the implementation. The actual implementation of the corresponding concepts is described in the following sections.

For the implementation, we extend the existing meta-model of MECHATRONICUML (cf. Section 7.3). This meta-model is provided in the Eclipse plugin *de.uni_paderborn.fujaba.muml*. Additionally, we provide the Eclipse plugin *de.fraunhofer.ipt.muml.MUML2Runnable* that contains the two model transformations for the segmentation (cf. Section 7.3). We provide the extension to the ASL (cf. Section 7.4) in the Eclipse plugin *de.fraunhofer.ipt.muml.AllocationRepository*. Furthermore, we provide a model transformation from the extended MECHATRONICUML meta-model to the meta-model of AMALTHEA in the plugin *de.fraunhofer.ipt.muml.MUML2Amalthea* (cf. Section 7.5). The used meta-models of AMALTHEA are represented as one plugin in the figure. Actually, for each of the models *central*, *common*, *sw*, and *constraint* a dedicated plugin exists. Additionally, we show one plugin that represents the plugins for partitioning and mapping of AMALTHEA.

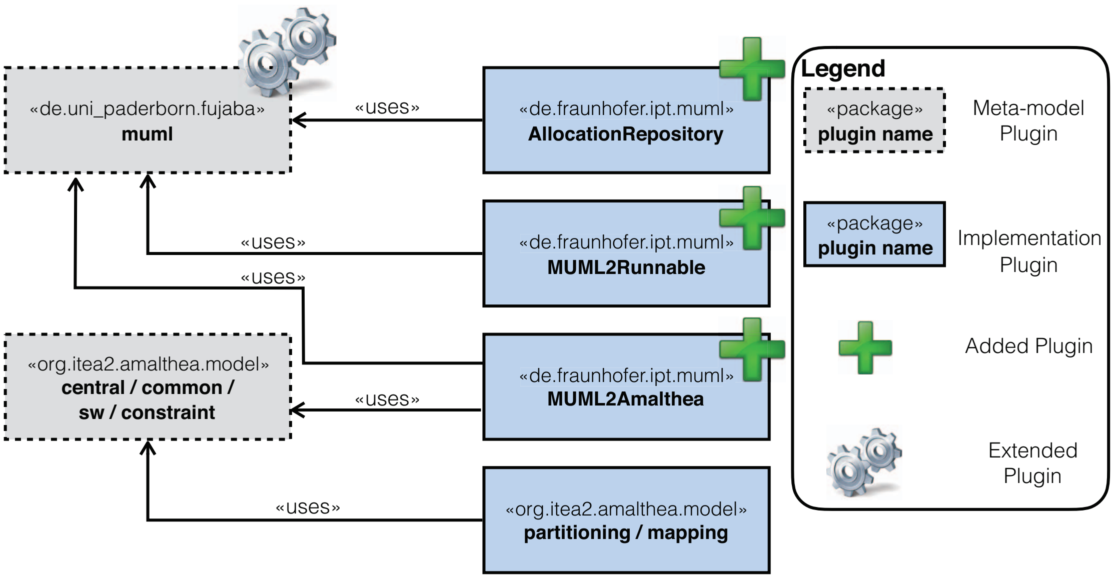


Figure 7.2: Plugins Used for Implementing our Concepts.

7.3 Segmentation

In this section, we explain the implementation for the segmentation of MechatronicUML models. First, we present the meta-model for the segmentation. Second, we describe the model transformation for the segmentation.

Meta-Model According to our concepts, we extend the meta-model of MechatronicUML to allow a segmentation. Figure 7.3 shows the concrete class diagram of the extended meta-model. For better readability, we show only affected excerpts of the model.

All added elements are yellow-filled. Existing elements that are referenced are grey-filled. The added enumeration-type is green-filled. We add the new class `Runnable` to the meta-model. It provides attributes for a name (since it inherits from `NamedElement`) and a period. We specify the period and deadline by a `TimeValue` that specifies a value and a time unit. The WCET can be added by specifying a WCET extension, which is already provided by MechatronicUML. Since *NamedElement* inherits from *ExtendableElement*, `Runnable` can be extended by WCET extensions. Furthermore, each runnable refers to the component instance that contains the corresponding RTSC. If the corresponding region of the runnable does implement the behavior of a discrete port instance, the runnable additionally refers to this discrete port instance.

Furthermore, we add a class `Label` and `LabelAccess` to specify read and write accesses to variables in the system. Every `LabelAccess` specifies an `AccessKind`,

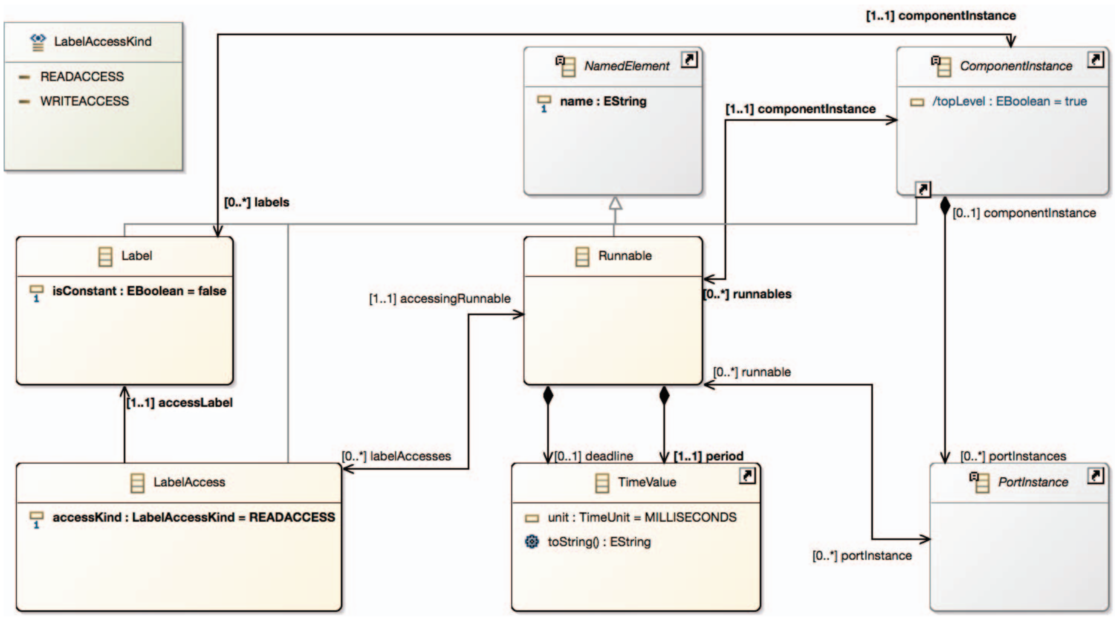


Figure 7.3: MechatronicUML Meta-Model Extended by Runnables (similar to [Ama13c]).

i.e., it is a read or write access to a label. Additionally, each LabelAccess belongs to one specific runnable.

Model Transformations We provide a model transformation to create a segmentation for a given MechatronicUML model automatically. In particular, we implemented the segmentation strategies *One Runnable Per Component* and *One Runnable Per Region*. For the latter one, we had to choose a strategy for handling synchronization channels. We decide to extend the region-runnables by the dependencies for all synchronization channels (cf. Section 4.2.1.3). Both transformations derive runnables and values for the period of each runnable based on the concepts of Section 4.2.3. Additionally, all labels and label accesses for every runnable are determined. Both transformations are implemented as a QVTo transformation in the Eclipse plugin `de.fraunhofer.ipt.muml.MUML2Runnable` on the attached DVD. After executing the segmentation transformation, the developer has to create WCET extensions for every runnable. Thus, at the end the MechatronicUML model is extended by runnables with defined runnable properties, labels and label accesses.

7.4 Allocation

For the allocation concepts, we extend the concepts for the allocation step of MechatronicUML. We extend the ASL by providing two new ASL constraints that can be used by the developer and correspond to the concepts proposed in Section 5.2.3: `maxDelay4RTCPs` and `validUtilization`. For both constraints, we implement corresponding OCL operations (RTCPMaxDelayConstraints() and validUtilizationFactors ()). These operations have to be executed on the OCL-Context of the ASL and return a set of tuples that correspond to the needed ILP constraints. Both constraints are of the ASL constraint type `requiredResource` (cf. Section 5.2.3.2). These operations can then be used in the ASL specification file to add them to the ILP. Listing 7.1 shows the specification of both constraints in the ASL file as already shown in Section 5.2.3.2.

Listing 7.1: ASL Constraint Specification for Conditions for Schedulability and to Ensure Real-Time Requirements of the Communication.

```
--ensure RTCP max delays in allocation
        constraint requiredResource maxDelay4RTCPs {
                lhs communicationTime;
                rhs MaxDelay;
                descriptors (c1, e1), (c2, e2);
                ocl self.RTCPMaxDelayConstraints();
        }

--ensure necessary condition for scheduling
        constraint requiredResource validUtilization{
                lhs utilizationTime;
                rhs ecuCapacity;
                descriptors (ci, ecu);
                ocl self.validUtilizationFactors();
        }
```

We do not implement the OCL operations for the constraints in a single operation, but stepwise. Hence, it is also possible to add constraints only for specific RTCPs by replacing `ocl self.RTCPMaxDelayConstraints();` with the OCL-Operation for only one specific RTCP.

According to the process, the result of this step is an allocation of component instances to ECUs. Every component instance refers to its dedicated runnables.

7.5 Partitioning and Mapping

We use the algorithms for partitioning and mapping of the Amalthea toolchain. Thus, we have to translate the MechatronicUML model into an Amalthea model. Since both meta-models are implemented with Ecore, we also apply a QVTo transformation. The model extension to the MechatronicUML meta-model (cf. Section 7.3) is inspired by the meta-model of Amalthea. In this transformation, we translate only runnables, their labels, dependencies, and properties. The component instances and their behavior are not translated. We expect that the source code generation for the runnables in later steps will use the MechatronicUML specification. The transformation is implemented in the Eclipse plugin `de.fraunhofer.ipt.muml.MUML2Amalthea` on the attached DVD.

Once the model is translated to Amalthea, existing algorithms for partitioning and mapping can be executed. This is mainly done in three steps that have to be executed.

1. **Partitioning:** The runnables are grouped to so-called *Task Prototypes*.
2. **Generate Tasks:** The Task Prototypes are translated into tasks.
3. **Perform Mapping:** The tasks are mapped to a hardware model, i.e., to ECU cores. Additionally, a scheduling is determined.

For a more detailed description of Amalthea, we refer to its documentation [Ama14].

7.6 Limitations

The implementation of the proposed concepts is not yet complete. In this section, we state known limitations of the current implementation. Furthermore, there are additional steps that should be applied to provide a seamless tooling for the complete process. In the following, we describe these missing implementation parts.

Hardware transformation The PDM of the system is currently not considered. MechatronicUML and Amalthea both provide a meta-model for the PDM, but the focus of the meta-models is different. For example, Amalthea does consider scheduler explicitly as part of the PDM. Nevertheless, we expect that the hardware model can be transformed from MechatronicUML to Amalthea. This step is needed, since we need the hardware description in MechatronicUML for the allocation step and in Amalthea for the mapping step. Currently, the PDM has to be modeled in both meta-models. Thus, we propose to provide a model transformation to map the PDM from MechatronicUML to Amalthea to enable a seamless tooling. Another idea is to use the PDM specification of Amalthea in the allocation step of MechatronicUML by adapting the OCL-Context of the ASL.

Reducing Label Accesses Currently, the approach for reducing the amount of dependencies (cf. Section 4.3.2) is not implemented. However, since model checking can be applied, the current implementation for the RTCP model checking can be used to apply this step manually. For this, the developer needs to find all accesses to labels and synchronization channels manually and has to define MTCL queries for all combinations. This results in many queries and therefore doing this step manually is very error-prone. Thus, we propose to automate this step and to reduce superfluous label accesses in the CIC (with runnables) automatically.

Source Code Generation Currently, we do not generate code for the generated mapping. This step can also be done automatically. Amalthea provides an export mechanism to create OIL files for the mapping. MechatronicUML provides a C code generator for CICs. Hence, the RTOS code can be generated using the Amalthea models, the behavioral code can be generated from the corresponding MechatronicUML models. Thus, we propose to automize the code generation using the information of both meta-models.

Since MechatronicUML also provides a generation of a middleware to handle the communication of distributed systems, it should be investigated if a retransformation of the mapping into MechatronicUML would be useful. After that, the current code generation can be applied. However, this might need further meta-model extensions in MechatronicUML.

Tooling Issues When applying the constraint for the schedulability condition for runnables in the allocation specification, an exception occurs ("no matching content for any()"). Until now, we have not been able to fix this problem. However, the ASL provides an evaluation view for OCL Operations. In this view (pure OCL evaluation), no problems occur. Thus, we were able to evaluate the correctnes of our implementation of the corresponding OCL Operation. Nevertheless, the use of this constraint in an allocation specification is currently not possible. The constraint for ensureing the RTCP QoS is not affected by this bug.

8 Case Study

In this chapter, we describe the evaluation of this thesis. We use a case study for evaluation and focus on the correctness of the transformations and as a proof of concepts. We assume a transformation to be correct if all relevant elements are considered in the transformation and all computed values are correct. To increase the validity of our case study, we follow the guidelines by Kitchenham et al. [KPP95].

Since a structured method for deriving metrics is needed, we apply the Goal-Question-Metric (GQM) method [Bas92, VSB99] for defining evaluation metrics. Hence, we use a template provided by the GQM method for defining goals. As evaluation example, we use the *Overtaking with Approacher* scenario that is also used as running example in this thesis.

This chapter is structured as follows: In Section 8.1, we present our evaluation hypotheses and corresponding goals, questions, and metrics. After that, in Section 8.2 we present the planning of the evaluation. In Section 8.3, we present the measurement of the data and state problems that occurred. At last, in Section 8.4 we analyze the measured data and give an interpretation of the results.

8.1 Define Hypotheses and Goals

In this section, we state our hypotheses for the case study in Section 8.1.1 Afterwards, we summarize the used goals, questions, and metrics in Section 8.1.2. The original GQM definitions for these goals, questions, and metrics are shown in Appendix C.1.

8.1.1 Hypotheses

We state three hypotheses. H1 is concerned with the correctness of the implemented transformations, which is important for the validity of the results. H2 is concerned with the segmentation and the resulting mapping and scheduling as presented in Chapter 4. H3 is concerned with the ASL extension, presented in Chapter 5.

H1: Transformations are correct. We expect, that the transformations for segmentation and to the meta-model of Amalthea are correct and correspond with the proposed concepts.

H2: Segmentation results in a feasible scheduling. We expect, that the segmentation approaches result in a feasible partitioning and mapping. Hence, we expect that the segmentation of a CIC is schedulable.

H3: Using new ASL constraints results in a feasible allocation. We expect that applying the new ASL constraints result in a correct allocation that respects both stated constraints. It is detected if no feasible allocation can be found.

8.1.2 Goals, Questions, and Metrics

In the following, we summarize the used goals, questions, and metrics, that help us validating the hypotheses. Since we follow the GQM method, we use a template provided by the GQM method for describing the goals. As suggested by the GQM method, we started the definition of goals in the beginning of this thesis and updated them during the development. For each goal, we define corresponding questions and for each question, we define metrics that are used to answer the question Additionally, we define a hypothesis for the measured value of each metric. All goals, corresponding questions, and metrics are shown in Appendix C.1.

For our case study, we focus on the evaluation of the correctness of the used transformations. Additionally, we apply partitioning and mapping algorithms and analyze the resulting task set. Hence, we separate the goals into two parts. In Section 8.1.2.1, we present the goals with regard to the correctness of the transformations and to the allocation constraints. In Section 8.1.2.2, we present the goals with regard to schedule and execution relevant information.

For our case study, we focus on the evaluation of the correctness of the used transformations. Additionally, we apply partitioning and mapping algorithms and analyze the resulting task set. Hence, we separate the goals into two parts. In Section 8.1.2.1, we present the goals with regard to the correctness of the transformations and to the allocation constraints. In Section 8.1.2.2, we present the goals with regard to schedule and execution relevant information.

8.1.2.1 Correctness

One key aspect in the evaluation is the correctness of our transformations. Thus, we define two goals for correctness: We define *Goal 1* with two dedicated questions that concern the correctness of the transformations for the segmentation. The original GQM-template for the goal, the questions and the metrics is shown in Appendix C.1.1 on page C.1.1.

Goal 1: Ensure correctness of the segmentation transformations.

> **Question 1.1:** Is the transformation complete, i.e., are all MechatronicUML elements considered?
>
> **Question 1.2:** Are all elements translated correctly?

The metrics for Question 1.1 focus on the completeness of the transformation. We check, if all relevant time properties of the RTSCs are considered, i.e., clock constraints, invariants, sampling-intervals, and the max-delay of RTCPs. Metrics for Question 1.2 are used to check if all relevant language elements are considered in the transformations and if the computed values are correct, i.e., the number of runnables, their properties, and dependencies. The questions and metrics are not specific for one segmentation strategy, but can be used for both evaluated segmentation strategies.

We define *Goal 2* with three dedicated questions that concern the correct creation of ILP constraints using the new ASL Constraints. The original GQM-template for the goal, the questions, and the metrics is shown in Appendix C.1.1 on page 140.

Goal 2: Ensure correctness of the extended allocation approach.

> **Question 2.1:** Is the transformation complete?
>
> **Question 2.2:** Are the timing information used in the maxDealy- constraints correct?
>
> **Question 2.3:** Are the timing information used in the utilizationFactor-constraints correct?

The metrics for Question 2.1 concerns the number of considered RTCPs and the number of resulting ILP-constraints. The metrics for Question 2.2 focus on the correct computation of the time values for the constraints that ensure the max-delay and metrics for Question 2.3 focus on the correct computation of the utilization factors for the schedulability-constraint.

8.1.2.2 Execution

Furthermore, we evaluate a goal regarding the execution of the produced mapping and scheduling. We define *Goal 3* that focuses on schedulability (including parallel execution) of the produced mapping. The original GQM-template for the goal, the questions, and the metrics is shown in Appendix C.1.1 on page 139.

Goal 3: Improve deriving a feasible scheduling.

Question 3.1: Are all RTCPs respected in the resulting allocation?

Question 3.2: Is the resulting task set schedulable?

Question 3.3: Is parallel execution possible?

The metric for Question 3.1 is used to check if all max-delays of all RTCPs are respected in the resulting allocation. The metric for Question 3.2 is used to check if a feasible scheduling is found for the resulting task set. Metrics for Question 3.3 check, if parallel execution of the software is possible.

8.2 Preparing Case Study

We use the *Overtaking With Approacher* scenario for our case study. In particular, we use the CIC of the *Overtaker*, because it uses all relevant model elements. The CIC consists of continuous components, discrete components, and an RTCP. The RTSCs of the discrete components have several regions, use clock constraints, state invariants, and urgent transitions and read and write the values of hybrid ports. As platform model, we reuse an existing model of a multi-core ECU that is a model of a "Freescale MPC5668G based evaluation board" [Ama14] and is already used in an evaluation example for Amalthea [Ama14]. This ECU provides two ECU cores. For evaluating the mapping for a single-core ECU we use the same model but removed one ECU core from the model.

For comparing the results with our expectations, we manually create reference models. In particular, we create a reference model for each implemented segmentation approach. Additionally, we derive all expected ILP constraints manually. We use these reference models as expected results and compared the results of the transformation with them.

We define an evaluation plan to have a structured execution of the evaluation. In the following, we state the main steps of this plan.

Evaluate the Transformations: We evaluate the correctness of the implemented transformations. For this, we execute the segmentation transformations and compare the results with the reference models. In particular, we execute the transformations for the segmentation strategies *One Runnable Per Component*, which does not allow parallel execution of the components behavior, and *One Runnable Per Region*, which allows parallel execution of the component behavior. Additionally, we execute the transformation from MechatronicUML meta-model to the Amalthea meta-model. For comparing the results with the reference models, questions and metrics of Goal 1 are used.

Evaluate the Segmentations: We evaluate the resulting scheduling for the segmentation strategies *One Runnable Per Component* and *One Runnable Per Region*. We use the results of the segmentation transformations of the first evaluation step for this. For each segmentation strategy, we apply the partitioning and mapping algorithm of AMALTHEA. We apply the algorithms once for an ECU with two cores and once for an ECU with one core. After that, we analyze the resulting mapping. In particular, we check if parallel execution is possible. For this, questions Q3.2 and Q3.3 are used.

Evaluate the ASL Extensions: We evaluate the extended allocation step. In particular, we compare the generated ILP constraints with the expected constraints. Furthermore, we use different time values for the period of the sender runnable and receiver runnable and for the transmit time. Table C.1 shows these time values. These times are used to check that the resulting allocation will be valid, (1) if the communication partners are allocated to different ECUs, (2) if the communication partners are allocated to the same ECU, and (3) no valid allocation can be found. Furthermore, we use constant values for the $T_{Transmit}$: $100ms$ for inter-ECU communication and $15ms$ for intra-ECU communication. For comparing the results, questions and metrics of Goal 2 and question Q3.1 are used.

For the schedulability-constraint, we have to specify the WCET for each runnable. Since the WCETs for our example are too short, we scaled the values, such that an evaluation of the correctness is more expressive. Table C.2 shows the WCETs in instructions (used in the mapping step) and the scaled values in ms (used for the ASL-evaluation) for the *One Runnable Per Component* segmenation strategy. Table C.3 shows the values for the *One Runnable Per Region* segmenation strategy.

8.3 Measuring Evaluation Data

Corresponding to the defined execution plan, we gather the values for all metrics. Table C.7 in Appendix C.3 on page 151 summarizes the expected values and the actual measured values for each metric and for both segmentation strategies. Additionally, the last column of the table shows, if the hypothesis for the values is fulfilled by the measured data. We use these data as a basis for the measurement and the interpretation. In the following, we summarize the results of the evaluation.

Evaluate the Transformations: We evaluate the QVTo transformation regarding the segmentation approaches. First, we execute the transformation for the *One Runnable Per Component* segmentation. The result is 7 runnables, 5 labels, and 10 label accesses. These results do match our expectations. The periods and deadlines are also as expected. Table C.4 in Appendix C.3.1 shows the values for period and deadline for each runnable.

Second, we execute the transformation for the *One Runnable Per Region* segmentation. The result is 11 runnables, 37 labels, and 39 label accesses and these results differ from our expectations. We apply further analyzes of the measured values to find the reasons for the differences, which are discussed in Section 8.4. The periods and deadlines of all runnables match our expectations. Table C.5 in Appendix C.3.1 shows the values for each runnable.

Finally, we apply the transformation from the MechatronicUML model to the Amalthea model for both approaches. All runnables, runnable dependencies, and runnable properties that are specified in MechatronicUML are translated correctly to corresponding elements in Amalthea.

Evaluate the Segmentations: First, we apply the partitioning and mapping algorithms to the set of runnables produced by the *One Runnable Per Component* segmentation strategy. The result is a feasible scheduling with 3 tasks. 2 are mapped to one core and 1 to the other core. Appendix C.3.3.2 shows all tasks, their mapping to ECU cores, and the corresponding runnables per task.

Second, we apply the partitioning and mapping algorithms to the set of runnables produced by the *One Runnable Per Region* segmentation strategy using the same hardware platform. The result is a feasible scheduling with 7 tasks. 5 are mapped to one core and 2 to the other core. Appendix C.3.3.1 shows all tasks, their mapping to ECU cores, and the corresponding runnables per task.

Finally, we apply partitioning and mapping again using a single-core ECU as the hardware platform. The algorithms of Amalthea find a feasible scheduling for both segmentation approaches.

Evaluate the ASL Extensions: We evaluate the new created ASL-constraints. First, we evaluate the constraint for ensuring the max-delay of RTCPs (Inequation 5.3). The result is a set of 4 corresponding ILP constraints. The values of the constraint match our expectations and correspond to the values proposed in Section 2.5.2. Additionally, we use different values for the periods of the sender runnable and receiver runnable to test the cases that (A) an valid allocation with two ECUs is found, (B) a valid allocation with only one ECU is found, and (C) no valid allocation is found. Table C.1 in

Appendix C.2.1 shows these values. For each value combination, the results are as expected (cf. Table C.6 in Appendix C.3.2).

Second, we evaluate the constraint regarding the necessary condition for schedulability (Inequation 5.3). Unfortunately, due to an exception during executing the allocation procedure, we are not able to evaluate this ASL-constraint in the allocation step. However, since the ASL-constraint is implemented in OCL, we are able to evaluate the OCL expression using the *EvaluationView* of the ASL. The output is not a set of ILP-constraints, but of the corresponding OCL-Tuples that represent these ILP-constraints. All resulting tuples (and the values for the utilization factors) match our expectations. Thus, we argue that the implementation of the ASL-constraint is correct, but is for unknown reasons currently not compatible with the ASL interpreter.

8.4 Analyzing the Results and Conclusion

The measured evaluation data show, that the segmentation transformations work as expected. For the *One Runnable Per Component* strategy, all values are as expected. We get one runnable per component and one label per hybrid port. Each value of the hybrid port is written by the corresponding continuous component and read by the hybrid component.

In the *One Runnable Per Region* strategy, we recognized some differences to the reference models. 1. more labels are generated as expected and 2. the amount of runnable accesses (37) to the amount of labels (34) is unusual. Hence, we analyzed the model and found reasons that explain the measured values. First, we discovered, some variables and hybrid port values that are specified for an RTSC are never accessed by a state event or transition effects. Second, in the evaluation example, many constant values are used. Such values are initialized before the RTSCs or the corresponding runnables are executed, e.g., when the system starts. Hence, such labels are only read but have no specified write access. Altogether, we argue that this segmentation transformation is correct also. Nevertheless, the initializing of constants and variables has to be considered explicitly in the code generation, such that all variables are initialized before the runnables start. All things considered, we state that hypothesis **H1** is fulfilled.

Partitioning and mapping result in a feasible scheduling for both segmentation strategies. For the *One Runnable Per Component*-strategy, both discrete component instances are allocated to different cores and are therefore executed in parallel. The connected continuous components are allocated to the same core. This mapping is useful since the component instances are executed in parallel and all dependent runnables are mapped to the same core. Hence, no conflicts can occur at runtime.

For the *One Runnable Per Region*-strategy, 7 tasks are created in the partitioning. 5 tasks execute only one runnable each. The resulting mapping shows that two runnables (for two regions of the overtakerCommunicator) are mapped to one core and the rest of the runnables to the other core (including the other regions of overtakingCommunicator). Thus, the regions of the overtakerCommunicator are executed in parallel. Another result of the evaluation is, that it mainly depends on the defined WCETs, which runnables are executed in parallel. Since we use a rough estimation for the WCETs, other mappings are possible if more sophisticated WCETs are used.

In summary, we find a feasible single- and multi-core scheduling for all of our test cases and for both segmentation approaches. Using the *One Runnable Per Region* strategy allows to execute the component behavior in parallel for our evaluation example. Hence, we argue that hypothesis **H2** is fulfilled.

The generated ILP-constraints that represent the constraint for ensuring the max-delay of the RTCP are as expected. For all used values for periods, the correct values were determined (cf. Table C.1 and Table C.6). Additionally, the resulting allocation is as expected. The evaluation for the constraint for the necessary condition for schedulability is more complicated since we had to evaluate the OCL-tuples instead of ILP-constraints. Additionally, we cannot test the resulting allocation, because of the runtime exception during the execution outlined above. However, we translate the resulting OCL-tuple into ILP-constraints manually. All values are as expected and, therefore, we argue that the constraints work. In summary, we show that using our extension and the proposed constraints help to find an allocation that respects the semantics of MechatronicUML. If no feasible allocation can be found, it is also detected during the allocation. Altogether, we state that hypothesis **H3** is fulfilled.

The case-study shows that our concepts and the implementation work as expected and all three hypotheses are fulfilled. The segmentation approaches *One Runnable Per Component* and *One Runnable Per Region* result in a feasible multi-core scheduling for our evaluation model. Additionally, we show that the extended allocation approach enables specifying allocation constraints with regard to runnables and to ensure the max-delay of RTCPs. Nevertheless, we propose to do a more expressive evaluation of the concepts to evaluate the concepts with more sophisticated models to evaluate the approach with more runnables and more varying WCETs. Furthermore, we propose to evaluate the advantage of using a multi-core scheduling in comparison to a single-core scheduling. For this, models of real-world systems with more runnables might be needed.

The most important threats to validity are as follows: 1. Currently, we do not support all possible modeling constructs of MechatronicUML, e.g., when deriving the period we restrict the use of clocks to local clocks. Additionally, we do not support expressions as values for clocks but only static time values. 2. We evaluated our concepts and implementation based on only one MechatronicUML example model. For more expressive results, larger and real-world examples are needed. 3. We did not execute the generated scheduling on a real multi-core ECU and assume that the generated source code is correct with regard to the semantics of MechatronicUML. Assuming that, we argue that the behavior is executed correctly, but could not show this during our evaluation explicitly. 4. We did not a full evaluation. Hence, there are several additional features that should be evaluated in future. As suggested by the GQM method, we defined additional evaluation goals in the beginning of writing this thesis. These goals aim to *Correctness* of the transformations, *Efficiency* of the new process, and *Preserving Semantics* during execution. Unfortunately, in the end, we are not able to evaluate all defined goals. Not evaluated goals and questions can be found in Appendix C.4 and might be used for further evaluation in future work.

9 Conclusion and Future Work

In this chapter, we summarize this thesis and give initial ideas for future work. In Section 9.1, we conclude the concepts developed within this thesis. After that, we state initial ideas for future work in Section 9.2.

9.1 Conclusion

Finding a real-time scheduling for software in safety-critical, distributed CPSs is a complex task, especially if multi-core ECUs are used as target platform. Finding a feasible schedule gets even more complex if the executed software has to fulfill hard real-time requirements. The main contribution of our approach is a semi-automatic method to derive a multi-core scheduling for software with real-time requirements in the context of CPSs. We use the MechatronicUML method to specify the software since MechatronicUML allows to specify real-time constraints in the software behavior. Our approach enables the developer to automatically derive a multi-core scheduling that respects the semantically correct execution of the specified real-time behavior in distributed CPSs.

We provide a method for an automatic segmentation for CICs to separate the component-RTSCs into runnables and to derive their dependencies automatically. The segmentation contains several steps: Separating the PIM into runnables, determining constraints between these runnables like the access to shared variables and determining runnable properties. Runnable properties are period (since we focus on time-triggered activation), WCET, and a deadline for every runnable. This segmentation allows for an early analysis of timing behavior, e.g., that period and WCET do not result in a conflict. We discuss three different approaches for the segmentation of discrete component instances (cf. Section 4.2.1). In particular, we apply the segmentation to the behavior specification of the component instance, i.e., an RTSC. We conclude that defining one runnable per region provides a good trade-off between parallel execution and the number of runnables and their dependencies. Using this segmentation strategy, we enable parallel execution of the component's behavior, which can be useful if regions differ in their WCET and period. We figure out, that the use of synchronization channels results in strong dependencies between runnables and should, therefore, be used cautiously.

Furthermore, we propose a segmentation strategy for continuous components (cf. Section 4.2.4), where we define one runnable per continuous component. Additionally, we present an approach to determine a period for the runnables automatically. The period of a runnable is important since it affects the semantically correct execution of the software. MechatronicUML uses RTSCs that allow specifying real-time features as in timed automata, like clock constraints or invariants. These real-time features are considered in the model checking of the PIM. We show that a badly chosen period could prohibit the semantically correct execution of these features and would, therefore, neglect parts of the model checking. The period of a runnable depends on the enabling conditions of all transitions that have to be executed by the runnable. Thus, for determining the period, we discuss combinations of possible transition-conditions, i.e., guards, clock constraints, state invariants, and urgent transitions. We figrued out that the semantics of two language features cannot be ensured during runtime in all cases since we have to limit our approach to RTSCs where clocks cannot be reset by other runnables: urgent transitions and invariants. Nevertheless, we show that both features can be executed correctly if a specific delay of firing a transition is allowed.

Furthermore, we provide an approach to find a multi-core scheduling for distributed systems, i.e., in systems where several ECUs are used. This step is a complex task since it has to ensure hard real-time requirements regarding the communication of software components and has to fulfill several additional allocation constraints. We enable the developer to find a feasible allocation for a distributed system that respects all hard real-time requirements of the communication by extending the concepts for the current allocation step of MechatronicUML. In this step, the domain-specific constraint language ASL is used to specify allocation constraints. Currently, the ASL does only consider properties of component instances. We extend the ASL by the possibility to refer to runnables and their properties. We enrich the ASL by two constraints that refer to the properties of the runnables: 1. A constraint to ensure that the allocation of runnables does not exceed the processing capacity of the target ECUs. 2. A constraint that ensures the max-delay property of the QoS assumptions of the RTCPs in the system. Hence, this approach allows to compute a feasible allocation automatically that respects allocation constraints and communication requirements of the system.

There are already mature approaches that allow to compute a real-time scheduling for a set of runnables with defined runnable properties and runnable dependencies, i.e., for the partitioning and mapping of the runnables. We argue to use existing algorithms of the Amalthea toolchain for partitioning and mapping. Hence, we provide an automatic transformation of the segmented PIM from MechatronicUML to Amalthea and apply algorithms for partitioning and mapping. Since the runtime of the algorithms is mainly affected by the number of runnables and their dependencies, we provide an approach to reduce the number of dependencies by using model checking techniques automatically before translating the model to AMALTHEA. The central idea of this approach is to analyze the RTSCs of the runnables to find specified write accesses to labels of runnables that are superfluous.

In conclusion, our approach enables an automatic segmentation to runnables, finding an allocation with respect to allocation constraints and hard real-time constraints for the communication, and to find a scheduling for multi-core platforms automatically. It contributes by an automatical analysis of component-based software descriptions with real-time behavior for possible parallel execution. Additionally, it assists the developer to tackle complex tasks that are error-prone when done manually like finding a scheduling that respects both hard real-time requirements and runnable dependencies. Hence, our approach helps to safe costs and time as well as increase the efficiency when deriving a feasible scheduling for safety-critical software with hard real-time requirements. We based our approach on MECHATRONICUML, but it is also applicable to other model-driven approaches that allow specification of real-time behavior for distributed CPSs.

9.2 Future Work

In this section, we give initial ideas for future work in the context of our concepts.

Semantic Preservation of Urgent Transitions In this thesis, we focus on time-triggered activation of runnables and tasks. We identify, that a semantic preserving execution (with regard to the current model checking approach) of urgent transitions cannot be guaranteed in all cases by using this approach. In event-triggered systems, runnables and tasks are activated by system events that can occur at arbitrary points in time. System events can be triggered, when a sensor value or variable that affect transition conditions of urgent transitions changes. Thus, triggering a runnable that contains an urgent transition by such an event might be more suitable than the time-triggered approach. There are also approaches that allow both time-triggered and event-triggered tasks as in [VDHBL$^+$12]. Such approaches should be evaluated in future for the implementation of urgent transitions. However, even in a time-triggered system, we can simulate an event-triggered execution. This can be done by changing the implementation for the RTSC. Whenever a variable is set to a new value, all urgent transitions that use this value are evaluated and possibly fired. Hence, an urgent transition is fired immediately. However, this might add additional dependencies between runnables and increase the WCET of the runnable that sets the new value, because the time for firing such related transitions has to be added. Thus, we propose to evaluate an event-triggered or mixed (event-triggered and time-triggered) execution of MechatronicUML models.

Determining Runnable Period In our presented approach, we restrict some features of MechatronicUML for the determination of the period for a runnable. For example, we assume that clocks that are used for clock constraints and state invariants get reset when the state is entered. In MechatronicUML, such clock resets are not mandatory and a clock can have another value than 0 when the state is entered. Thus, the time until an invariant expires or the fulfillment of a clock constraint changes is shorter. Hence, it would be useful to analyze the worst-case clock value that is possible when the state is entered as mentioned in Section 4.2.3.7. An idea for finding such values is to apply a reachability analysis for the runtime model as done in [Hei15]. However, since in this analysis only the PIM is considered, an extended reachability analysis is needed, which considers properties of the PDM additionally, e.g., the scheduling explicitly.

Determining Tight WCETs In this thesis, we do not focus on WCETs and assume that time values for WCETs of runnables are provided by engineers, e.g., by measuring them via appropriate tools. We argue to base the WCET on the number of instructions that are generated for a runnable. Another option we use is to measure the WCET for each target ECU or its processing cores respectively and annotate them to the runnables. These methods might result in unprecise values for the WCETs, since we derived them manually and only used rough estimations. WCETs can be influenced by several additional factors, e.g., caches, co-processors, or processor pipelining [Wil09]. The precision of the WCET affects the scheduling directly. Furthermore, the utilization factor of the runnable will also be affected and, therefore, can result in another allocation. Hence, we propose to improve the determination of WCETs to improve allocation and mapping. A further idea is to automize the step, e.g., by analyzing RTSCs to estimate the WCETs automatically based on features used in the RTSC.

Static WCET for Communication Channels We assume that the WCET of sending a message from one ECU to another is a constant value. This assumption might not hold for real systems because ECUs might use different kinds of communication channels that have different bandwidths. For some communication channels, the WCET might change with the channel's utilization, like CAN-busses. For this case, a schedulability analysis can be performed to predict the utilization of the channel before runtime. Hence, the approach for extending the ASL should be improved by more precise WCETs for the used communication channels. It is not sufficient to simply measure the bandwidth of each channel because the utilization of the channel is affected by the current allocation. Thus, the determination of the communication WCET has to be applied during computing the allocation, e.g., by considering it during solving the ILP for the allocation problem.

Component Instances with Different Complexity In this thesis, we discuss three different segmentation strategies. We state, that all of them can be used for parallel execution of software specified with MechatronicUML. In the evaluation, we compare two of the strategies for the same input model. However, for large models with many component instances it might be useful to use different strategies for different component instances as stated in Section 4.2.1.4. For example, component instances that have a low WCET in total and many dependencies, the strategy *One Runnable Per Component* is advisable. For component instances where the WCET of the regions differ widely and only a few dependencies are defined, the strategy *One Runnable Per Region* is more suitable. It is possible, that both components are part of the same CIC, i.e., the CIC is heterogenous regarding the runnable properties of its component instances. Therefore, using different segmentation strategies seems promising. Hence, we propose to allow choosing the segmentation approach more flexible, e.g., by choosing it for each component instance separately. This can be done automatically by analyzing WCET and period of all regions.

Automatic Code Generation In this thesis, we do not focus on the generation of source code. We assume that the source code can be generated straightforward from the given models. In particular, we assume to generate source code for the behavior of the runnables from MechatronicUML and source code for the RTOS (OIL files and glue code) from Amalthea. The current source code generation of MechatronicUML provides a generator for C code but targets a single-core ECU. We state that the execution of all transition effects has to be done automically, i.e., during the execution of transition effects, the runnable (or the corresponding task) cannot be preempted. This has to be ensured in the generated source code. Hence, we propose to enrich the current source code generation for MechatronicUML by multi-core support.

Additionally, code for a middleware is generated in the source code generation of MechatronicUML. This middleware is used for ensuring correct communication of components if the components are allocated to different ECUs. Amalthea provides a source code generator for system specific information, e.g., OIL files. Hence, we propose to develop a source code generator that automatically generates source code for the target platform using the Amalthea model for the system information (task, ECU cores, scheduling) and the MechatronicUML model to generate the source code for the runnables.

Verifying Real-Time Requirements In Section 3, we state that our approach can be used to improve the analysis of the software at runtime. Since this is not in the focus of this thesis, we state it as future work and give initial ideas for this task in the following. Amalthea provides methods for generating trace-formats in the executed software. Execution means either to simulate the tasks in dedicated simulation environments or to execute the software on hardware. Currently, the trace formats consider events of tasks, events of runnables, and read and write accesses of labels [Ama13b]. Furthermore, event chains [Ama13a] can be defined that can be verified on the measured trace of the software. Thus, we propose to evaluate real-time requirements using event chains on runnable level. Since we store the currently active state of each region in a label, state changes of all RTSCs can be traced by observing these labels when using the tracing formats of Amalthea. In [Tee12], a similar approach for a simulation environment for OSEK tasks in the context of MechatronicUML is applied (cf. Section 6.3) and evaluated using event chains.

Appendix A

Disregarding Region Priorities

In this thesis, we propose to disregard region priorities. Region priorities are used to explicitly define an order for sequentially execution of all regions of an RTSC. The only purpose of region priorities is to define an execution order of all regions, if the RTSC is executed on a single-core ECU [BDG^{+}14a]. They are used to define a deterministic execution of an RTSC. On a multi-core ECU, this is a hard limitation, since it prohibits parallel execution explicitly. However, we argue that there is no reason to execute all regions of an RTSC in a specific order or with the same period.

Synchronization channels, on the other hand, are used to specify points in the regions that have to be executed in synchronization. Thus, these synchronization points define explicit points in the execution, which ensure that no infeasible state configuration is possible. By a state configuration we denote a set of states of parallel regions that are active at the same time. Figure A.1 shows to regions of the RTSC of the component instance `overtakerCommunicator`.

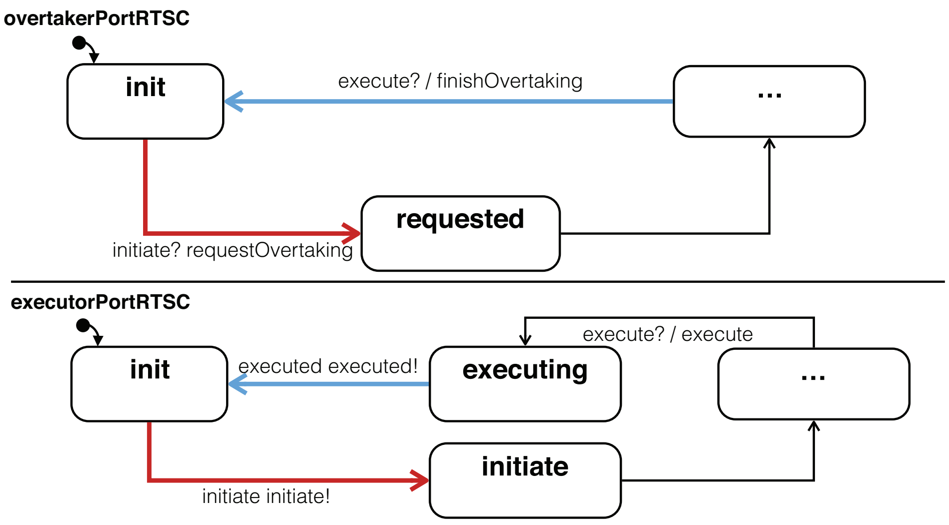


Figure A.1: Region Priorities Can be Disregarded.

For better readability, we hide some states denoted by the state with three dots. The complete RTSC with both regions can be found in Appendix B.6. Additionally, we colored transitions that are in synchronization and, therefore, are the synchronization points where both regions have to be in sync. The initial state configuration of the regions is {`init`, `init`}. After firing the red transitions, each state configuration until the next sychronized transition is not critical (with regard to the specified synchronization channels), because the order of actions, events and state changes that are processed between such synchronization points should not lead to infeasible state configurations. An infeasible state configuration would be {`init`, `executing`}. This, however is prohibited by the usage of synchronization channel `executed` (blue).

Thus, the order of execution of the regions is not mandatory, because at least at the synchronization points regions have to wait for other regions to synchronize. Hence, in this thesis we propose to disregard region priorities, since it will not affect safety requirements of the system that are ensured by the usage of synchronization channels.

Appendix B

Model Diagrams

B.1 Model Diagrams for the Running Example

In this section, we show model diagrams for the running example *Overtaking With Approacher*. In Section B.1.1, we show diagrams for components and component instances used within in this thesis. In Section B.1.2, we show the diagrams for the used RTCP. In Section B.1.3, we show the RTSCs for the discrete compenents that are used in the thesis. For the remaining diagrams, we refer to the model specification on the DVD.

B.1.1 Components and Component Instances

In this section, we show the component diagram and compenent instance diagrams that are used in this thesis. Figure B.1 shows the component instance for the whole example scenario *Overtaker With Approacher*. For better readability, we ommit the atomic component instances of each vehicle. Figure B.2 shows the component diagram of the overtakerVehicle with all compenent parts. Figure B.3 shows an instance of the overtakerVehicle and the containing CIC.

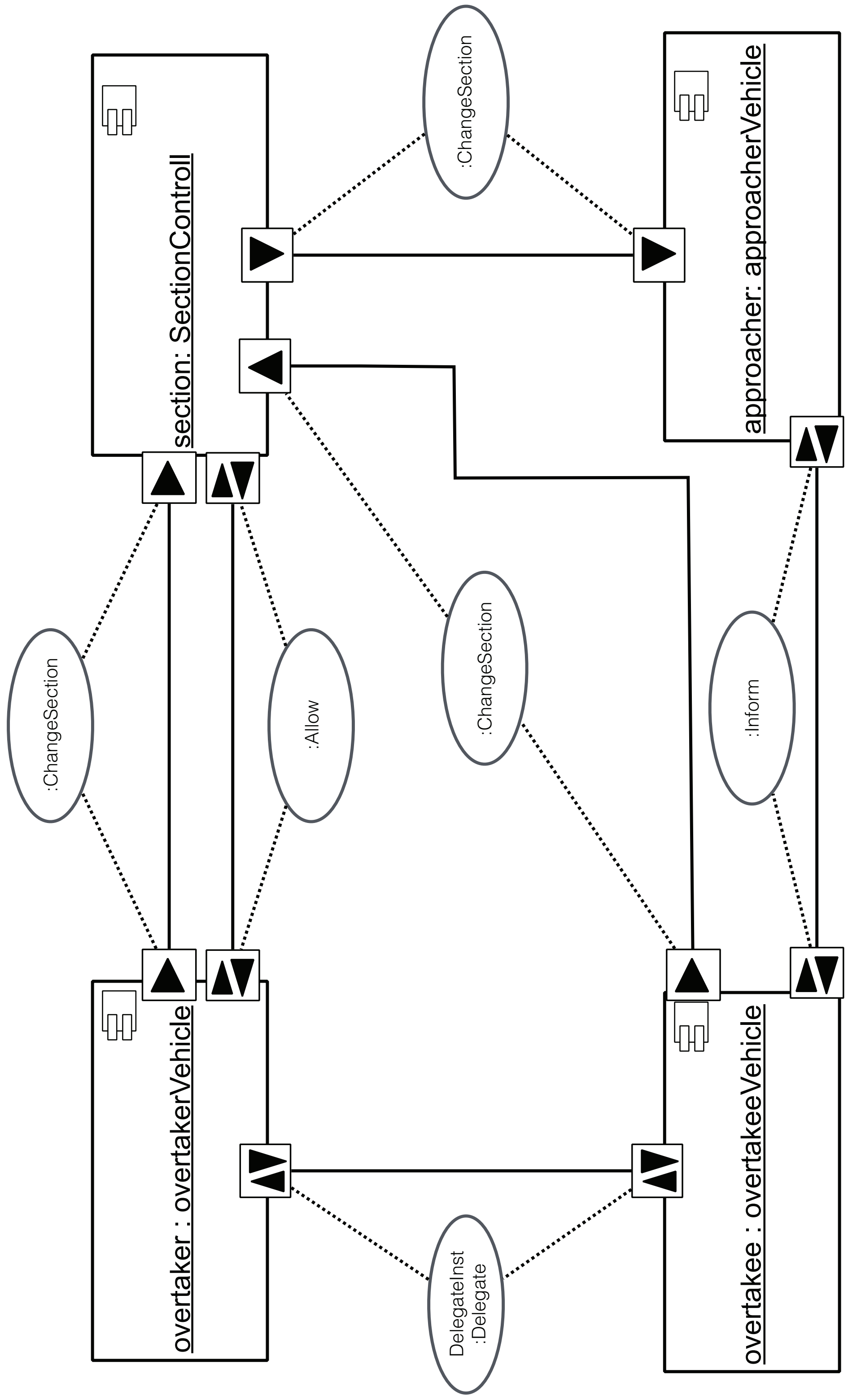


Figure B.1: Component Instance Configuration Diagram for "Overtaking With Approacher".

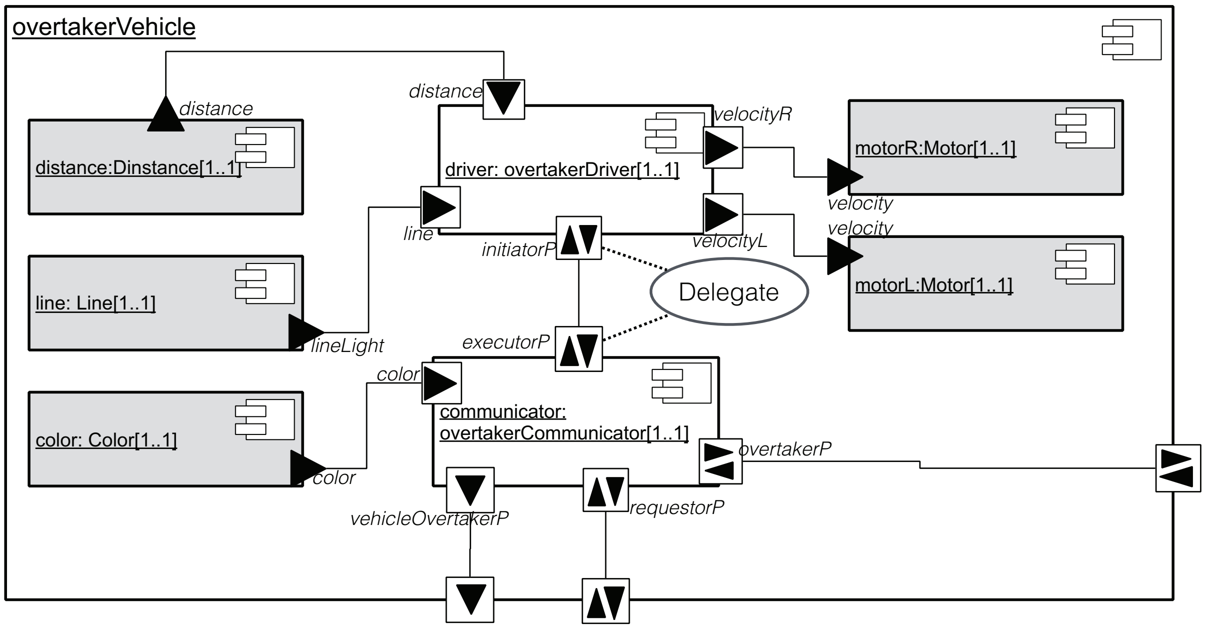
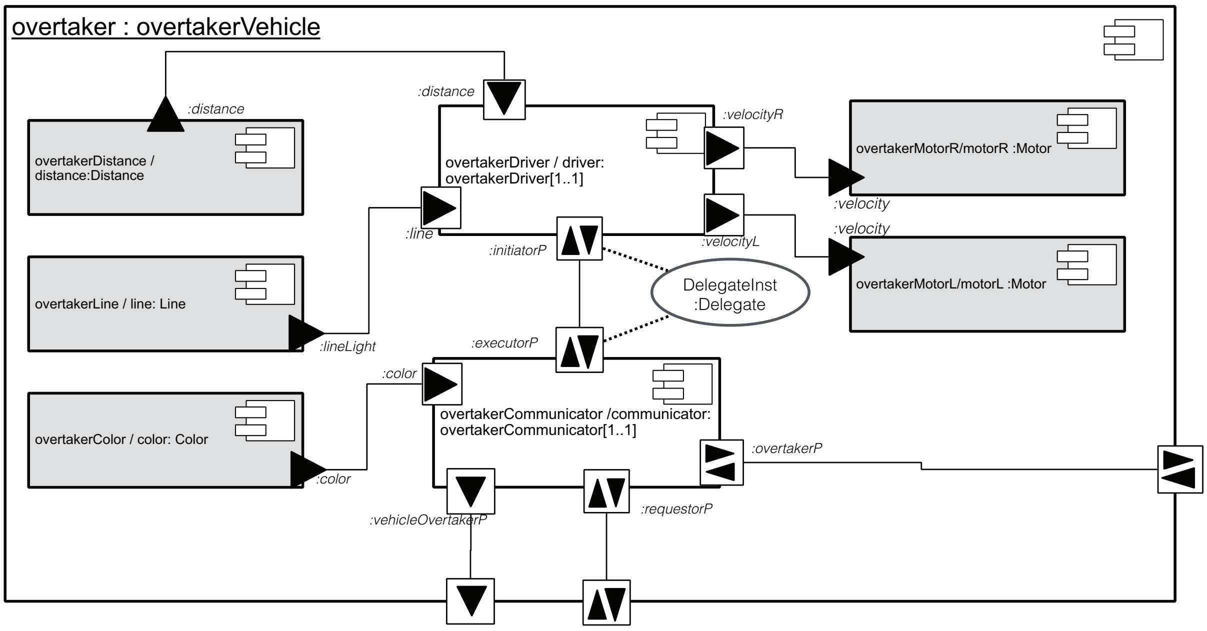


Figure B.2: Component Diagram for component “overtakerVehicle”.



overtaker : overtakerVehicle
:distance
overtakerDistance / distance:Distance
:distance
overtakerDriver / driver: overtakerDriver[1..1]
:velocityR
overtakerMotorR/motorR :Motor
:velocity
:velocity
:line
:initiatorP
:velocityL
DelegateInst :Delegate
overtakerMotorL/motorL :Motor
overtakerLine / line: Line
:lineLight
:executorP
:color
overtakerCommunicator /communicator: overtakerCommunicator[1..1]
overtakerColor / color: Color
:overtakerP
:color
:vehicleOvertakerP
:requestorP

Figure B.3: Component Instance Diagram for component “overtakerVehicle”.

B.1.2 Real-Time Coordination Protocols

Figure B.4 show the RTCP `Delegate` that were already shown in the concept chapters of the thesis. For the diagrams of the role RTSCs, we refer to the original model specification on the attached DVD. The component RTSCs in Figure B.5 and Figure B.6 contain one region each that refines the role behavior of this RTCP.

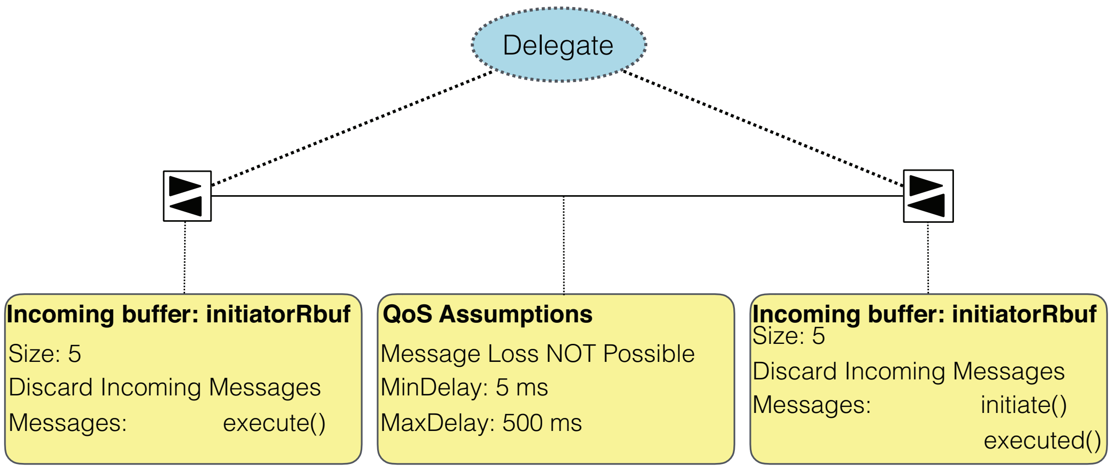

Delegate
Incoming buffer: initiatorRbuf
Size: 5
Discard Incoming Messages
Messages: execute()
QoS Assumptions
Message Loss NOT Possible
MinDelay: 5 ms
MaxDelay: 500 ms
Incoming buffer: initiatorRbuf
Size: 5
Discard Incoming Messages
Messages: initiate() executed()

Figure B.4: RTCP “Delegate” of component “overtakerDriver”.

B.1.3 Real-Time Statecharts

In this section, we show the component RTSC of the component parts of the overtakerVehicle. Figure B.5 shows the component RTSC of the component part `overtakerDriver`. Figure B.6 shows the component RTSC of the component part `overtakerCommunicator`.

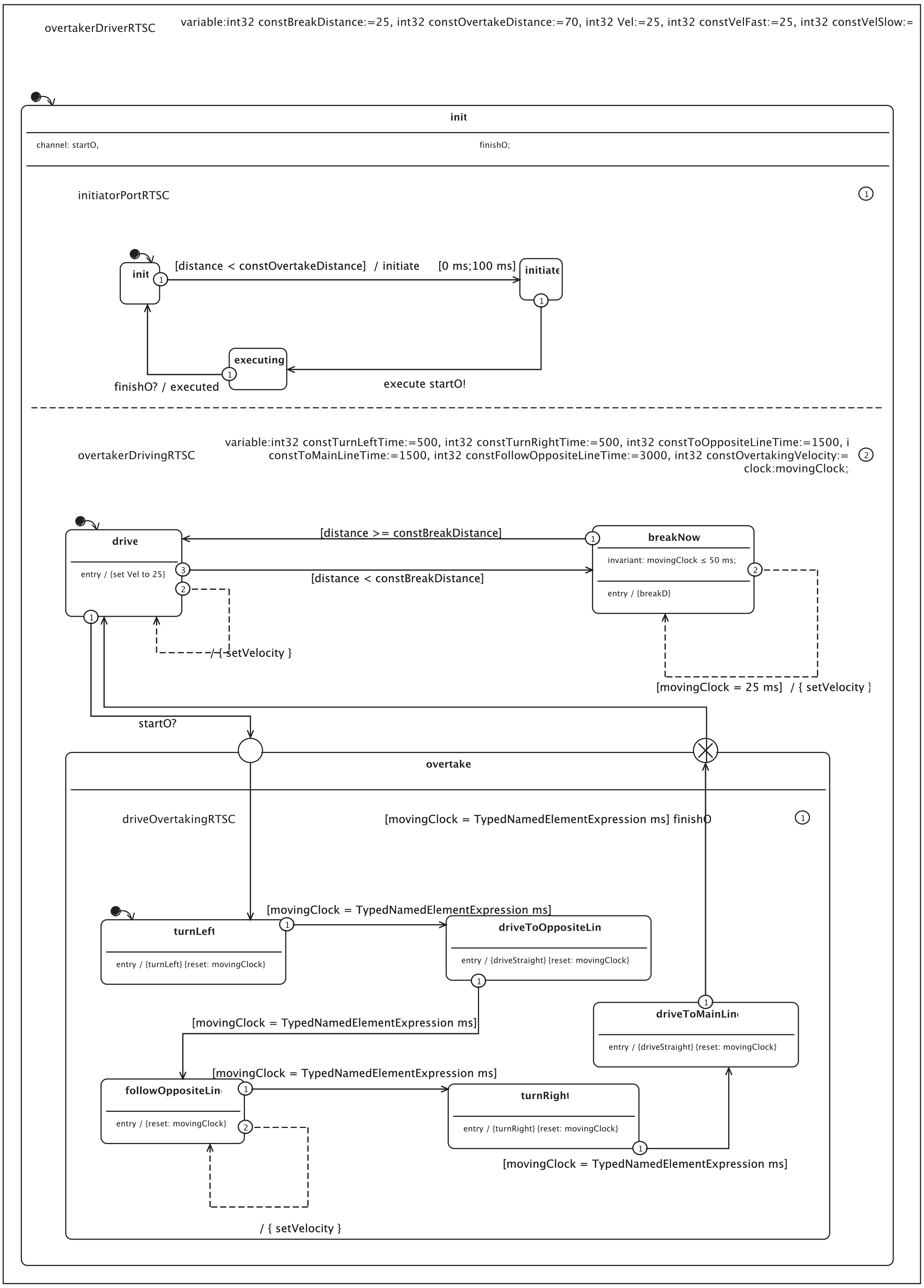

overtakerDriverRTSC
variable:int32 constBreakDistance:=25, int32 constOvertakeDistance:=70, int32 Vel:=25, int32 constVelFast:=25, int32 constVelSlow:=
init
channel: startO,
finishO;
initiatorPortRTSC
init
[distance < constOvertakeDistance] / initiate
[0 ms;100 ms]
initiate
executing
execute startO!
finishO? / executed
overtakerDrivingRTSC
variable:int32 constTurnLeftTime:=500, int32 constTurnRightTime:=500, int32 constToOppositeLineTime:=1500, i
constToMainLineTime:=1500, int32 constFollowOppositeLineTime:=3000, int32 constOvertakingVelocity:=
clock:movingClock;
drive
entry / {set Vel to 25}
[distance >= constBreakDistance]
[distance < constBreakDistance]
breakNow
invariant: movingClock ≤ 50 ms;
entry / {breakD}
/ { setVelocity }
[movingClock = 25 ms] / { setVelocity }
startO?
overtake
driveOvertakingRTSC
[movingClock = TypedNamedElementExpression ms] finishO
turnLeft
entry / {turnLeft} {reset: movingClock}
[movingClock = TypedNamedElementExpression ms]
driveToOppositeLin
entry / {driveStraight} {reset: movingClock}
[movingClock = TypedNamedElementExpression ms]
driveToMainLin
entry / {driveStraight} {reset: movingClock}
[movingClock = TypedNamedElementExpression ms]
followOppositeLin
entry / {reset: movingClock}
turnRight
entry / {turnRight} {reset: movingClock}
[movingClock = TypedNamedElementExpression ms]
/ { setVelocity }

Figure B.5: Component RTSC of component “overtakerDriver”.

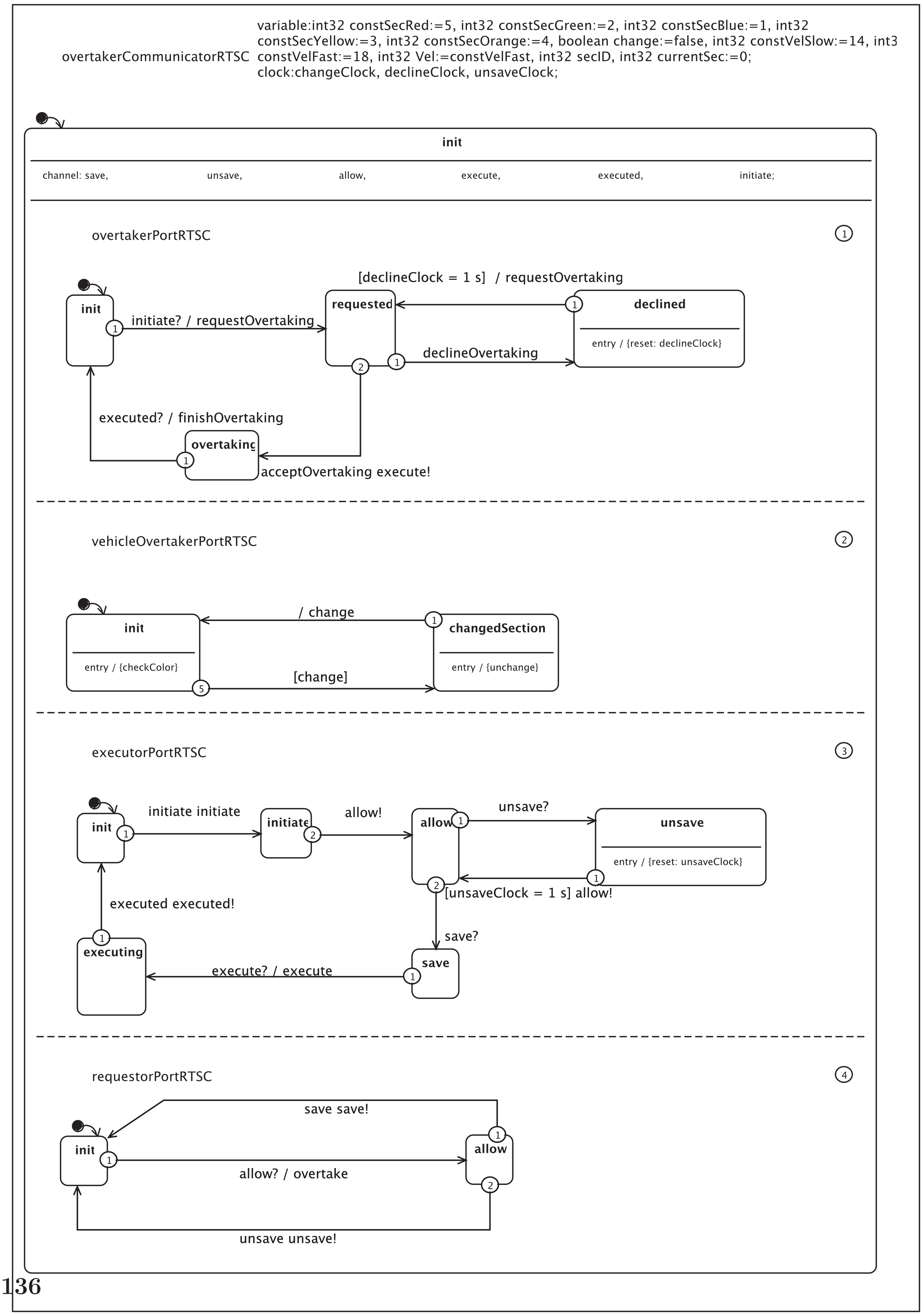

overtakerCommunicatorRTSC
variable:int32 constSecRed:=5, int32 constSecGreen:=2, int32 constSecBlue:=1, int32 constSecYellow:=3, int32 constSecOrange:=4, boolean change:=false, int32 constVelSlow:=14, int3 constVelFast:=18, int32 Vel:=constVelFast, int32 secID, int32 currentSec:=0;
clock:changeClock, declineClock, unsaveClock;
init
channel: save, unsave, allow, execute, executed, initiate;
overtakerPortRTSC
init
initiate? / requestOvertaking
requested
[declineClock = 1 s] / requestOvertaking
declined
entry / {reset: declineClock}
declineOvertaking
executed? / finishOvertaking
overtaking
acceptOvertaking execute!
vehicleOvertakerPortRTSC
init
entry / {checkColor}
/ change
changedSection
entry / {unchange}
[change]
executorPortRTSC
init
initiate initiate
initiate
allow!
allow
unsave?
unsave
entry / {reset: unsaveClock}
[unsaveClock = 1 s] allow!
executed executed!
executing
save?
save
execute? / execute
requestorPortRTSC
init
save save!
allow
allow? / overtake
unsave unsave!

Figure B.6: Component RTSC of component “overtakerCommunicator”.

B.2 Model Diagrams for Normalized Continuous Components

In the following, we show RTSCs that are created in the normalization for continuous components [Ros14, BDG+14b]. Figure B.7 shows the RTSC that is created for the normalized continuous compenent "light". Its calls an API function and sends the corresponding value to the connected discrete component instance periodically. Figure B.8 shows the adapted component RTSC of the discrete component instance "overtakerDriver" that contains the connected hybrid port. A new region is created for the hybrid port. In this region, the value (sent by the normalized continous component) is received and written into a corresponding RTSC variable.

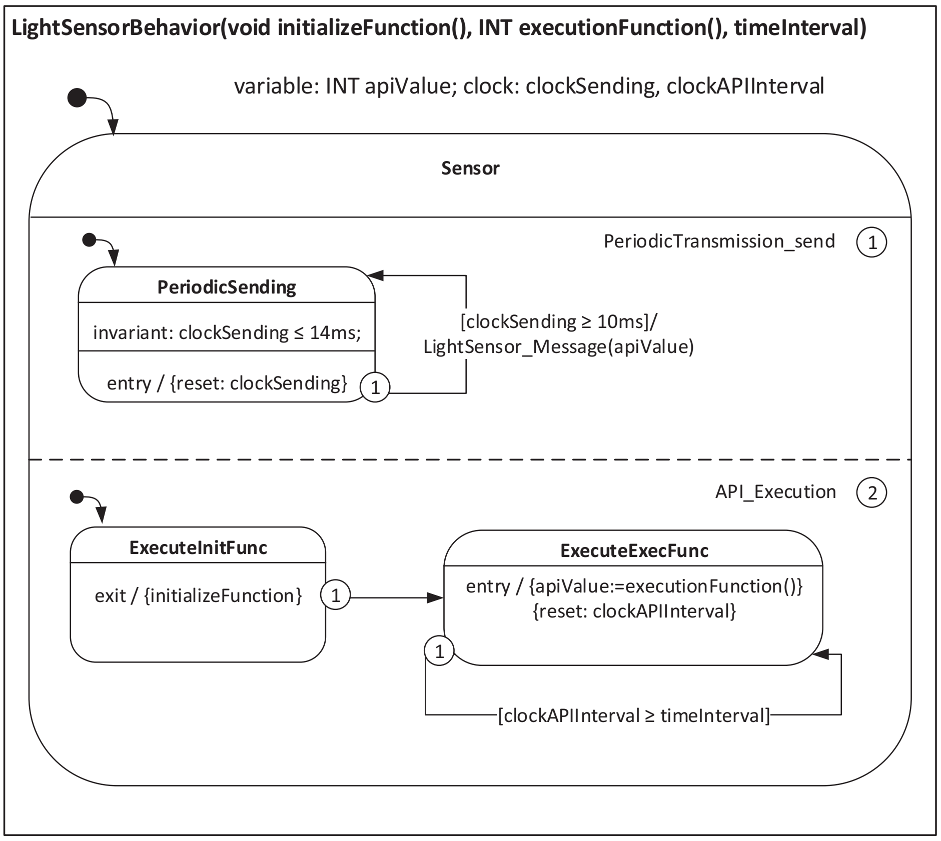

LightSensorBehavior(void initializeFunction(), INT executionFunction(), timeInterval)
variable: INT apiValue; clock: clockSending, clockAPIInterval
Sensor
PeriodicTransmission_send 1
PeriodicSending
invariant: clockSending ≤ 14ms;
entry / {reset: clockSending} 1
[clockSending ≥ 10ms]/ LightSensor_Message(apiValue)
API_Execution 2
ExecuteInitFunc
exit / {initializeFunction} 1
ExecuteExecFunc
entry / {apiValue:=executionFunction()} {reset: clockAPIInterval}
1
[clockAPIInterval ≥ timeInterval]

Figure B.7: Component RTSC for the Transoformed Continuous Component Instance (Source [BDG+14b]).

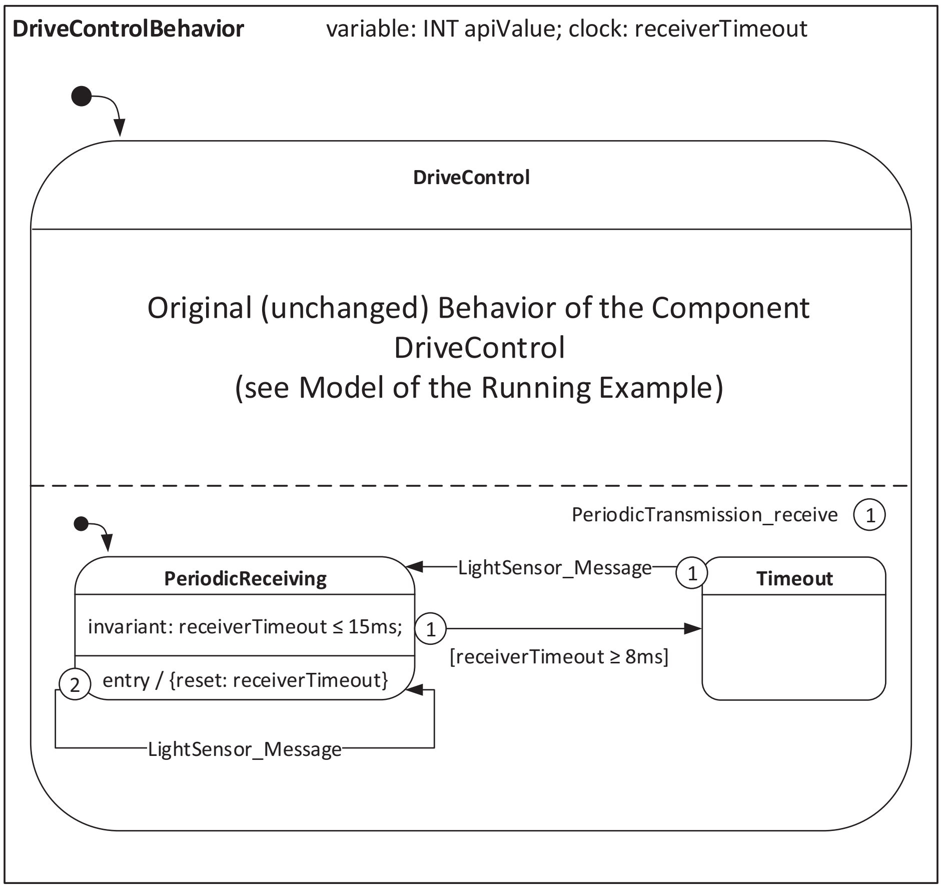

DriveControlBehavior
variable: INT apiValue; clock: receiverTimeout
DriveControl
Original (unchanged) Behavior of the Component DriveControl (see Model of the Running Example)
PeriodicTransmission_receive
1
PeriodicReceiving
invariant: receiverTimeout ≤ 15ms;
entry / {reset: receiverTimeout}
2
LightSensor_Message
1
[receiverTimeout ≥ 8ms]
LightSensor_Message
1
Timeout

Figure B.8: Component RTSC for the Transoformed Discrete Component Instance (Source [BDG+14b]).

Appendix C

Evaluation

C.1 Evaluation Goals

In this section, we show the original GQM templates for the evaluation goals that we used for our case study. Additionally, we show all dedicated questions and metrics with corresponding hypotheses. In Section C.1.1, we present the goals regarding the correctness of the transformations and the ASL-constraints. In Section C.1.2, we present the goals focusing on the execution of the software, i.e., mapping and scheduling.

C.1.1 Correctness

In this section, we present two goals that focus on the correctness of our implementation. Goal 1 is used to evaluate the model to model transformations for the implemented segmentation approaches. Goals 2 is used to evaluate the OCL operations for the ASL-constraints.

Goal 1	
Analyze	the MechatronicUML segmentation transformation
For the purpose of	ensuring
With respect to	correctness
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML to multi-core deployment process.

Q1.1: Is the transformation complete, i.e., are all MechatronicUML elements considered?	
M1.1.1	Are clock constraints considered?
H1.1.1	yes
M1.1.2	Are invariants considered?
H1.1.2	yes
M1.1.3	Are max-delays of RTCPs considered?
H1.1.3	yes
M1.1.4	Are sampling intervals considered?
H1.1.4	yes

Q1.2: Are all elements translated correctly?	
M1.2.1	Number of runnables.
H1.2.1	equal to number of regions in MechatronicUML model.
M1.2.2	Number of runnable dependencies.
H1.2.2	equal to manually determined dependencies and label accesses.
M1.2.3	Number of wrong properties, i.e., runnable period, -WCET, -deadline, and runnable dependencies.
H1.2.3	0
M1.2.4	Number of missing properties, i.e., runnable period, -WCET, -deadline, and runnable dependencies.
H1.2.4	0

Goal 2	
Analyze	the extended allocation approach
For the purpose of	ensuring
With respect to	correctness
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML to multi-core deployment process.

Q2.1: Is the transformation complete?	
M2.1.1	Number of considered RTCPs.
H2.1.1	Equal to the number of RTCPs in the MechatronicUML model.
M2.1.2	Number of ILP Constraints for RTCPs.
H2.1.2	$\#\text{RTCP}s * \lvert ECU \rvert^2$.
M2.1.3	Number of ILP Constraints for UtilizationFactors.
H2.1.3	$\lvert ECU \rvert$.

Q2.2: Are the timing information used in the maxDealy-constraints correct?	
M2.2.1	Time for Sending, Transmitting, and Receiving Messages of a RTCP.
H2.2.1	Corresponding to inequation 5.4.
M2.2.2	Right-hand-side of ILP constraint.
H2.2.2	Corresponding to MaxDelay of RTCP.

Q2.3: Are the timing information used in the utilizationFactor-constraints correct?	
M2.3.1	Utilization Factor for each ECU.
H2.3.1	Corresponding to inequation 5.3.
M2.3.2	Processing Capacity of each ECU.
H2.3.2	Corresponding to number of processor cores.

C.1.2 Execution

Furthermore, we evaluate goals that are concerned with the execution of the produced mapping and scheduling. We define *Goal 3* that is concerned with the schedulability (including parallel execution) of the produced mapping. Additionally, we define *Goal 4* which concerns basic evaluation of the executed software. In the following, we present both goals and their dedicated questions with metrics.

Goal 3	
Analyze	the MechatronicUML to Multi-Core Platform transformation
For the purpose of	improving
With respect to	finding a feasible schedule for the software
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML development process

Q3.1: Are all RTCPs respected in the resulting allocation?	
M3.1.1	Number of violated RTCPs in the resulting allocation.
H3.1.1	0

Q3.2: Is the resulting task set schedulable?	
M3.2.1	Schedulability of the resulting task set.
H3.2.1	schedulable

Q3.3: Is parallel execution possible?	
M3.3.1	Is parallel execution possible?.
H3.3.1	yes
M3.3.2	Is parallel execution of RTSCs possible?.
H3.3.2	yes
M3.3.3	Number of tasks per ECU core.
H3.3.3	$\frac{\#tasksintotal}{\#ECUcores}$

Goal 4	
Analyze	the source code generation for multi-core environments
For the purpose of	ensuring
With respect to	semantic preservation
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML to multi-core deployment process.

Q4.1: Are all timing constraints fulfilled?	
M4.1.1	Number of missed deadlines
H4.1.1	0
M4.1.2	Number of violated invariants
H4.1.2	0
M4.1.3	Number of vialoated RTCP max delays.
H4.1.3	0

C.2 Measured Input Parameters

In this section, we present input data for the evaluation. In particular, we show the input values for periods and WCETs used in the evaluation of the ASL-constraints. In Section C.2.1, we show the values for the periods to evaluate the constraint for ensuring the max delay of RTCPs. In Section C.2.2, we show the WCETs for all runnables that were used for the utilization factor and in the mapping step of Amalthea.

C.2.1 Adapted Period Values for Testing

In this section, we show in Table C.1 values that are used for evaluating the constraint regarding the max-delay of RTCPs. We used values for three different cases: A valid allocation is found for two ECUs **(A)**, a valid allocation is found for one ECU **(B)**, and no valid allocation can be found **(C)**. Note, that we assume that $t_{transmit}$ is $150ms$ if the runnables are allocated to different ECUs and $15ms$ if the runnables are allocated to the same ECU.

Table C.1: Runnable Period Values for Evaluation of Communication Time Constraint.

	Runnable	Period π	Time for Sending t_s	Time for Receiving t_r
A	initiatorP-Runnable	180 ms	180 ms	360 ms
	executorP-Runnable	100 ms	100 ms	200 ms
B	initiatorP-Runnable	50 ms	50 ms	100 ms
	executorP-Runnable	100 ms	100 ms	200 ms
C	initiatorP-Runnable	200 ms	200 ms	400 ms
	executorP-Runnable	200 ms	200 ms	400 ms

C.2.2 Worst-Case Execution Times

We measure WCETs for all generated runnables. For this, we measure the generated instruction (on a x86 machine with GCC) and corresponding WCETs in milliseconds. Since the measured time values are too small (all are smaller than 1ms), we scaled them up to get more expressive computation results. Nevertheless, the relations to the WCETs in instructions is kept.

Table C.2: WCETs for "One Runnable Per Component" Segmentation Strategy.

Runnable Name	Instructions	ms
H_overtakerDriver_overtakerDriver_distance	100	1 ms
H_overtakerDriver_overtakerDriver_line	100	1 ms
H_overtakerDriver_overtakerDriver_velocityR	100	1 ms
H_overtakerDriver_overtakerDriver_velocityL	100	1 ms
H_overtakerCommunicator_overtaker-Communicator_color	100	1 ms
overtakerDriverdriverovertakerDriver_Runnable	700	4 ms
overtakerCommunicatorcommunicator-overtakerCommunicator_Runnable	1010	9 ms

Table C.3: WCETs for "One Runnable Per Region" Segmentation Strategy.

Runnable Name	Instructions	ms
H_overtakerDriver_overtakerDriver_distance	100	1 ms
H_overtakerDriver_overtakerDriver_line	100	1 ms
H_overtakerDriver_overtakerDriver_velocityR	100	1 ms
H_overtakerDriver_overtakerDriver_velocityL	100	1 ms
H_overtakerCommunicator_overtaker-Communicator_color	100	1 ms
overtakerDriverdriverovertakerDriverinitiator-PortRTSC_Runnable	300	2 ms
overtakerDriverdriverovertakerDriverovertaker-DrivingRTSC_Runnable	400	2 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorovertakerPortRTSC_Runnable	250	2 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorvehicleOvertakerPortRTSC_Runnable	100	1 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorexecutorPortRTSC_Runnable	460	4 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorrequestorPortRTSC_Runnable	200	2 ms

C.3 Evaluation Results

In this section, we show the results measured during the evalution. In Section C.3.1, we show the determined values for the periods of the runnables for each segmentation approach. After that, in Section C.3.2, we show the computed values for the communication time in the ASL constraint. In Section C.3.3, we show the results of partitioning and mapping for each segmentation approach. Finally, in Section C.3.4, we present a table that summarizes all measured metrics that are used to answer the evaluation goals and questions.

C.3.1 Runnable Properties

In this section, we show the measured values for periods and deadlines for the running examples. Since all periods are smaller than a measured deadline, the deadline ist set to the value of the period in all cases. Table C.4 shows the values for the *One Runnable Per Component* segmentation strategy. Table C.5 shows the values for the *One Runnable Per Region* segmentation strategy.

Table C.4: Runnable Properties for "One Runnable Per Component" Segmentation Strategy.

Runnable Name	Period π	Deadline D_i
H_overtakerDriver_overtakerDriver_distance	25 ms	25 ms
H_overtakerDriver_overtakerDriver_line	25 ms	25 ms
H_overtakerDriver_overtakerDriver_velocityR	25 ms	25 ms
H_overtakerDriver_overtakerDriver_velocityL	25 ms	25 ms
H_overtakerCommunicator_overtaker-Communicator_color	25 ms	25 ms
overtakerDriverdriverovertakerDriver_Runnable	12 ms	12 ms
overtakerCommunicatorcommunicator-overtakerCommunicator_Runnable	25 ms	25 ms

Table C.5: Runnable Properties for "One Runnable Per Region" Segmentation Strategy.

Runnable Name	Period π	Deadline D_i
H_overtakerDriver_overtakerDriver_distance	25 ms	25 ms
H_overtakerDriver_overtakerDriver_line	25 ms	25 ms
H_overtakerDriver_overtakerDriver_velocityR	25 ms	25 ms
H_overtakerDriver_overtakerDriver_velocityL	25 ms	25 ms
H_overtakerCommunicator_overtaker-Communicator_color	25 ms	25 ms
overtakerDriverdriverovertakerDriverinitiator-PortRTSC_Runnable	25 ms	25 ms
overtakerDriverdriverovertakerDriverovertaker-DrivingRTSC_Runnable	12 ms	12 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorovertakerPortRTSC_Runnable	500 ms	500 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorvehicleOvertakerPortRTSC_Runnable	25 ms	25 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorexecutorPortRTSC_Runnable	500 ms	500 ms
overtakerCommunicatorcommunicatorovertaker-CommunicatorrequestorPortRTSC_Runnable	S (500 ms)	S (500 ms)

C.3.2 Values for Communication Time

In this section, we show the values that were measured during the evaluation of the max-delay constraint of the ASL. Table C.6 show the computed communication time for all three evaluated cases (cf. Table C.1).

Table C.6: Communication Times based on Adpated Periods.

	Sender Runnable	Receiver Runnable	Communication Time
A	initiatorP-Runnable	executorP-Runnable	380 ms
	executorP-Runnable	initiatorP-Runnable	460 ms
B	initiatorP-Runnable	executorP-Runnable	250 ms
	executorP-Runnable	initiatorP-Runnable	300 ms
C	initiatorP-Runnable	executorP-Runnable	600 ms
	executorP-Runnable	initiatorP-Runnable	600 ms

C.3.3 Partitioning and Mapping

In this section, we show the partitioning and mapping determined by the algorithms of AMALTHEA. Section C.3.3.1 show the results for the *One Runnable Per Component* segmentation strategy. Section C.3.3.2 show the results for the *One Runnable Per Region* segmentation strategy.

C.3.3.1 One Runnable Per Component

This section show the partitioning and mapping for the *One Runnable Per Component* segmentation strategy.

Partitioning In the following we show the partitioning computed by the partitioning algorithm of AMALTHEA. We show for each generated task the period and the executed runnables.

Task_PP1 (25 ms)
H_overtakerCommunicator_overtakerCommunicator_color

Task_PP0 (25 ms)
H_overtakerDriver_overtakerDriver_line
H_overtakerDriver_overtakerDriver_distance
overtakerDriverdriverovertakerDriver_Runnable
H_overtakerDriver_overtakerDriver_velocityL
H_overtakerDriver_overtakerDriver_velocityR

Task_PP2 (12 ms)
overtakerCommunicatorcommunicatorovertakerCommunicator_Runnable

Mapping In the following, we show the mapping computed by mapping algorithm of AMALTHEA. It shows the executed task for each core of the ECU.

Core 1 (e200z0)
Task_PP1
Task_PP2

Core 2 (e200z6)
Task_PP0

C.3.3.2 One Runnable Per Region

This section show the partitioning and mapping for the *One Runnable Per Region* segmentation strategy.

Partitioning In the following we show the partitioning computed by the partitioning algorithm of AMALTHEA. We show for each generated task the period and the executed runnables.

Task_PP3 (500 ms)
overtakerCommunicatorovertakerCommunicatorovertakerPortRTSC

Task_PP4 (500 ms)
overtakerCommunicatorovertakerCommunicatorexecutorPortRTSC

Task_PP1 (25 ms)
H_overtakerCommunicator_overtakerCommunicator_color overtakerCommunicatorovertakerCommunicatorvehicleOvertakerPortRTSC

Task_PP2 (12 ms)
overtakerDriverdriverovertakerDriverovertakerDrivingRTSC

Task_PP0 (25 ms)
H_overtakerDriver_overtakerDriver_distance H_overtakerDriver_overtakerDriver_line H_overtakerDriver_overtakerDriver_velocityR

Task_PP5 (500 ms)
overtakerCommunicatorovertakerCommunicatorrequestorPortRTSC

textbfTask_PP6 (500 ms)
H_overtakerDriver_overtakerDriver_velocityL

Mapping In the following, we show the mapping computed by mapping algorithm of Amalthea. It shows the executed task for each core of the ECU.

Core 1 (e200z0)
Task_PP3 Task_PP6

Core 2 (e200z6)
Task_PP0 Task_PP1 Task_PP2 Task_PP5 Task_PP4

C.3.4 Values for Evaluation Metrics

In Table C.7, we present the measured values for all metrics that are used to answer the evaluation goals and questions. We show the metric identifier, the stated hypothesis (expected results), and the results for both applied segmentation approaches. Additionally, we show by a tick, if the metric is correct with regard to the stated hypothesis.

Metric	**Per Component**			**Per Region**		
	Hypothesis	**Result**		**Hypothesis**	**Result**	
M1.1.1	yes	yes	✓	yes	yes	✓
M1.1.2	yes	yes	✓	yes	yes	✓
M1.1.3	yes	yes	✓	yes	yes	✓
M1.1.4	yes	yes	✓	yes	yes	✓
M1.2.1	7	7	✓	11	11	✓
M1.2.2	10	10	✓	34	37	✓
M1.2.3	0	0	✓	0	0	✓
M1.2.4	0	0	✓	0	0	✓
M2.1.1	1	1	✓	1	1	✓
M2.1.2	4	4	✓	4	4	✓
M2.1.3	2	2	✓	2	2	✓
M2.2.1	corresponding to inequation 5.4	cf. Table C.6	✓	corresponding to inequation 5.4	cf. Table C.6	✓
M2.2.2	500	500	✓	500	500	✓
M2.3.1	Corresponding to inequation 5.3	correct	✓	Corresponding to inequation 5.3	correct	✓
M2.3.2	2	2	✓	2	2	✓
M3.1.1	0	0	✓	0	0	✓
M3.2.1	schedulable	schedulable	✓	schedulable	schedulable	✓
M3.3.1	yes	yes	✓	yes	yes	✓
M3.3.2	yes	yes	✓	yes	yes	✓
M3.3.3	yes	yes	✓	yes	yes	✓

Table C.7: Evaluation Results for all Metrics.

C.4 Not Evaluated Evaluation Goals

In the beginning of writing this thesis, we started to derive goals that have to be fulfilled by a multi-core scheduling for MechatronicUML. Unfortunately, we were not able to evaluated all of these goals. Since these goals might be useful for future work, we present these goals in the following. In Section C.4.1, we show goals regarding measuring semantics preservation at runtime. In Section C.4.2, we show goals regarding efficiency of the development process.

C.4.1 Semantic Preservation

In this section, we present goals regarding the semantic preservation of MechatronicUML during the execution on a multi-core platform.

Goal 1	
Analyze	the MechatronicUML to Amalthea transformation
For the purpose of	ensuring
With respect to	semantic preservation
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML to multi-core deployment process.

Q1: Are the semantics of MechatronicUML preseverd in the executed source code?	
M1.1.1	Order of state changes in the whole system
H1.1.1	Corresponding to the defined MechatronicUML semantics
M1.1.2	Order of state changes and effects for synchronized transitions
H1.1.2	Corresponding to the defined MechatronicUML semantics
M1.1.1	Execution order of regions of a RTSC
H1.1.1	**Not** Corresponding to the defined MechatronicUML semantics

Q2: Are all timing constraints fulfilled?	
M1.2.1	Number of missed deadlines
H1.2.1	0
M1.2.2	Number of vialoated invariants
H1.2.2	0
M1.2.3	Number of not fulfilled time properties of RTCPs
H1.2.3	0

Goal 2	
Analyze	the extended allocation approach
For the purpose of	ensuring
With respect to	semantic preservation
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML to multi-core deployment process.

Q1: Are the semantics of RTCPs fulfilled at runtime?	
M2.1.1	Time for sending until receiving a message.
H2.1.1	less than the max-delay in the corresponding RTCP.

Goal 3	
Analyze	the source code generation for multi-core environments
For the purpose of	ensuring
With respect to	semantic preservation
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML to multi-core deployment process.

Q1: Are the semantics of MechatronicUML preseverd in the executed source code?	
M3.1.1	Order of state changes in the whole system
H3.1.1	Corresponding to the defined MechatronicUML semantics
M3.1.2	Order of state changes and effects for synchronized transitions
H3.1.2	Corresponding to the defined MechatronicUML semantics
M3.1.1	Execution order of regions of a RTSC
H3.1.1	**Not** Corresponding to the defined MechatronicUML semantics

Q2: Are all timing constraints fulfilled?	
M3.2.1	Number of missed deadlines
H3.2.1	0
M3.2.2	Number of vialoated invariants
H3.2.2	0
M3.2.3	Number of not fulfilled time properties of RTCPs
H3.2.3	0

C.4.2 Efficiency

In this section, we present goals, questions, and metrics regarding the efficiency of the executed software and of our development process.

Goal 4	
Analyze	the MechatronicUML to Multi-Core Platform transformation
For the purpose of	ensuring
With respect to	ensuring schedulability of the software
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML development process

Q1: Did the performance of the system improve?	
M4.1.2	Schedulability of the resulting task set.
H4.1.2	schedulable
M4.1.1	Response-Time per Region.
H4.1.1	decreased

Goal 5	
Analyze	the MechatronicUML to Multi-Core Platform transformation
For the purpose of	reducing
With respect to	effort during development
From the viewpoint of	the MechatronicUML engineer
In the context of	the MechatronicUML development process

Q1: Did the performance of the system improve?	
M5.1.1	Response-Time per Region.
H5.1.1	decreased
M5.1.2	Processor utilization.
H5.1.2	decreased

Q2: Did the number of manual steps for the developer decrease?	
M5.2.1	Difference of manual steps with and without our approach.
H5.2.1	less than 0
M5.2.2	Difference of number of manual defined properties with and without our approach.
H5.2.2	less than 0

Q3: Does our approach provides informationt o improve the current MechatronicUML process?	
M5.3.1	number of runtime-information for elements in the MechatronicUML PIM.
H5.3.1	less than 0

Bibliography

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994. ISSN 0304-3975. URL `http://www.sciencedirect.com/science/article/pii/0304397594900108`.

[AFM$^+$02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times b— a tool for modelling and implementation of embedded systems. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 460–464. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43419-1. URL `http://dx.doi.org/10.1007/3-540-46002-0_32`.

[AFM$^+$04] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In KimGuldstrand Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21671-1. URL `http://dx.doi.org/10.1007/978-3-540-40903-8_6`.

[ALE02] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb 2002. ISSN 0018-9162.

[AMA] AMALTHEA. Amalthea project page. `http://amalthea-project.org`. Last checked: 10.12.2015.

[Ama12] Amalthea. Deliverable: D4.1 specification of architecture. Technical Report 4.1, Amalthea, July 2012.

[Ama13a] Amalthea. Deliverable: D3.1 concept for a partitioning/ mapping/ scheduling/ timing-analysis tool. Technical Report 3.1, Amalthea, January 2013.

[Ama13b] Amalthea. Deliverable d3.2: Hardware model specification and examples of hardware models. Technical Report 3.2, Amalthea, January 2013.

[Ama13c] Amalthea. Deliverable: D4.2 detailed design of the amalthea data models. Technical Report 4.2, Amalthea, January 2013.

[Ama14] Amalthea. Deliverable: D3.4 prototypical implementation of selected concepts. Technical Report 3.4, Amalthea, April 2014.

[Amn03] Tobias Amnell. Code synthesis for timed automata. 2003.

[And99] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201357526.

[Arc] Timing Architects. Ta tool suite. `http://www.timing-architects.com`. Last checked: 10.12.2015.

[AUT] AUTOSAR. Autosar project page. `http://www.autosar.org/`.

[AUT11] AUTOSAR. *AUTOSAR 3.2 - Technical Overview*, April 2011. URL `http://www.autosar.org/fileadmin/files/releases/3-2/main/auxiliary/AUTOSAR_TechnicalOverview.pdf`. Document Identification No. 067, Version 2.2.2.

[AUT14a] AUTOSAR. Autosar 4.2.2 - glossary. Technical report, AUTOSAR, March 2014. Document Identification No. 055.

[AUT14b] AUTOSAR. *AUTOSAR 4.2.2 - Methodology*, March 2014. Document Identification No. 068.

[AUT14c] AUTOSAR, autosar. *AUTOSAR 4.2.2 - Specification of RTE*, March 2014. Document Identification No. 084.

[Bas92] Victor R. Basili. Software modeling and measurement: The goal/question/metric paradigm, September 1992.

[BBG05] Sami Beydeda, Matthias Book, and Volker Gruhn. *Model-Driven Software Development*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-28554-0.

[BC11] Etienne Borde and Jan Carlson. 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, CBSE '11, pages 129–138. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0723-9. URL `http://doi.acm.org/10.1145/2000229.2000248`.

[BCC+08] Tomáš Bureš, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis. Procom–the progress component model reference manual. *Mälardalen University, Västerås, Sweden*, 2008.

[BD10] Matthias Brun and Jérôme Delatour. *Contribution to the software execution platform integration during an application deployment process*. PhD thesis, Ph. D. dissertation, École Centrale de Nantes, Nantes, France, 2010.

[BDG+14a] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Sebastian Thiele, Wilhelm Schäfer, Matthias Meyer, Uwe Pohlmann, Claudia Priesterjahn, and Matthias Tichy. The mechatronicuml design method - process and language for platform-independent modeling. Technical Report tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn, March 2014. Version 0.4.

[BDG+14b] Jan Bobolz, Andreas Dann, Johannes Geismann, Marcus Hüwe, Arthur Krieger, Goran Piskachev, David Schubert, and Rebekka Wohlrab. Project group cybertron 2013/2014 final document, October 2014.

[BEP+07] Jacek Blazewicz, Klaus H. Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz. *Handbook on Scheduling : From Theory to Applications*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-28046-0.

[BFH+10] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526–545, April 2010. ISSN 0018-9219.

[BGS03] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Code generation for hard real-time systems from real-time statecharts. Technical Report tr-ri-03-244, University of Paderborn, Paderborn, Germany, October 2003. 1-15 pp.

[BGS05] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In Alan Hartman and David Kreische, editors, *Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA '05)*, volume 3748 of *Lecture Notes in Computer Science*, pages 25–40. Springer, Berlin/Heidelberg, November 2005. ISBN 978-3-540-30026-7.

[BK$^+$08] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008. ISBN 978-0-262-02649-9.

[BLL$^+$96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. *UPPAAL—a tool suite for automatic verification of real-time systems*. Springer, 1996. ISBN 978-3-540-61155-4.

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42. ACM, New York, NY, USA, 2006. ISBN 1-59593-375-1. URL `http://doi.acm.org/10.1145/1134285.1134292`.

[Bur02] Sven Burmester. Generierung von java real-time code für zeitbehaftete uml modelle. Diploma thesis, University of Paderborn, September 2002.

[Bur06] Sven Burmester. *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. PhD thesis, University of Paderborn, August 2006.

[But11] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. ISBN 978-1-461-40675-4.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000. ISBN 978-0-262-03270-4.

[CM05] Paul Caspi and Oded Maler. From control loops to real-time programs. In *Handbook of networked and embedded control systems*, pages 395–418. Springer, 2005.

[CSVC11] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel RV Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, 2011.

[Dan13] Andreas Dann. Ein plattform-beschreibungsmodell für die softwareverteilungsplanung mechatronischer systeme. Bachelors thesis, University of Paderborn, April 2013.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011. ISSN 0360-0300. URL `http://doi.acm.org/10.1145/1978802.1978814`.

[DGH15] Stefan Dziwok, Christopher Gerking, and Christian Heinzemann. Domain-specific model checking of mechatronicuml models using uppaal. Technical Report tr-ri-15-346, University of Paderborn, July 2015.

[Ecla] Eclipse Foundation. Eclipse modeling framework (emf). `http://www.eclipse.org/modeling/emf/`. Last checked: 10.12.2015.

[Eclb] Eclipse Foundation. QVT Operational. `http://projects.eclipse.org/projects/modeling.mmt.qvt-oml`. Last checked: 10.12.2015.

[FH04] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 377–383. Springer US, 2004. ISBN 978-1-4020-8156-9. URL `http://dx.doi.org/10.1007/978-1-4020-8157-6_29`.

[FHKK] Daniel Fruhner, Robert Höttger, Sebastian Köpfer, and Lukas Krawczyk. Partitioning and mapping for embedded mutlicore system utilization in context of the model based open source development environment platform amalthea.

[FMMR12] Carlo A Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling time in computing*. Springer Science & Business Media, 2012. ISBN 978-3-642-32331-7.

[Fos95] Ian Foster. *Designing and building parallel programs*. Addison Wesley Publishing Company, 1995. ISBN 0-201-57594-9.

[FPY02] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume

2280 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43419-1. URL `http://dx.doi.org/10.1007/3-540-46002-0_6`.

[Gam10] A. Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language.* Springer New York, 2010. ISBN 9781441909428.

[GCW$^+$02] Thomas Genßler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software: the pecos approach. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26. ACM, 2002.

[Gei13] Johannes Geismann. Codegenerierung für lego mindstorms - roboter. Bachelor's thesis, Heinz Nixdorf Institute, Software Engineering Group, University of Paderborn, Zukunftsmeile 1, 33102 Paderborn Germany, January 2013.

[GTB$^+$03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time uml designs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE'03, pages 38–47. ACM, New York, NY, USA, September 2003. ISBN 1-58113-743-5.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987. ISSN 0167-6423. URL `http://www.sciencedirect.com/science/article/pii/0167642387900359`.

[HB13] Christian Heinzemann and Steffen Becker. Executing reconfigurations in hierarchical component architectures. In *Proceedings of the 16th international ACM Sigsoft symposium on Component based software engineering*, CBSE '13, pages 3–12. ACM, New York, NY, USA, June 2013. ISBN 978-1-4503-2122-8.

[HBDS15] Christian Heinzemann, Christian Brenner, Stefan Dziwok, and Wilhelm Schäfer. Automata-based refinement checking for real-time systems. *Computer Science - Research and Development*, 30(3-4): 255–283, 2015. ISSN 1865-2034. Published Online June 2014.

[Hei15] Christian Heinzemann. *Verification and Simulation of Self-Adaptive Mechatronic Systems*. PhD thesis, University of Paderborn, September 2015.

[Hom09] Matthias Homann. *OSEK – Betriebssystem-Standard für Automotive und Embedded Systems. 3. Auflage.* mitp, 2009. ISBN 3-826-61552-2.

[HPB$^+$05] Petr Hnětynka, František Plášil, Tomáš Bureš, Vladimír Mencl, and Lucia Kapová. Sofa 2.0 metamodel. Technical report, Citeseer, 2005.

[HPB$^+$10] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. Comparison of component frameworks for real-time embedded systems. In *Component-Based Software Engineering*, pages 21–36. Springer, 2010.

[HRB$^+$14] Christian Heinzemann, Jan Rieke, Jana Bröggelwirth, Andrey Pines, and Andreas Volk. Translating mechatronicuml models to matlab/simulink and stateflow. Technical Report tr-ri-14-338, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2014. Version 0.4.

[INC] INCHRON. Inchron tool suite. `http://www.inchron.com/tool-suite/tool-suite.html`. Last checked: 10.12.2015.

[ISO03] ISO/IEC Standard. Software engineering – product quality – part 1: Quality model. ISO Standard 9126-1, ISO/IEC, 2003.

[ISO05a] ISO/IEC Standard. Road vehicles – open interface for embedded automotive applications – part 1: General structure and terms, definitions and abbreviated terms. ISO Standard 17356-1, ISO, 2005.

[ISO05b] ISO/IEC Standard. Road vehicles – open interface for embedded automotive applications – part 3: Osek/vdx operating system (os). ISO Standard 17356-3, ISO, 2005.

[ISO05c] ISO/IEC Standard. Road vehicles – open interface for embedded automotive applications – part 4: Osek/vdx communication (com). ISO Standard 17356-4, ISO, 2005.

[ISO05d] ISO/IEC Standard. Road vehicles – open interface for embedded automotive applications – part 6: Osek/vdx implementation language (oil). ISO Standard 17356-6, ISO, 2005.

[JKR15] Jan Jatzkowski, Marcio Kreutz, and Achim Rettberg. Hierarchical multicore-scheduling for virtualization of dependent real-time systems. unpublished, 2015.

[KK13] L. Krawczyk and E. Kamsties. Hardware models for automated partitioning and mapping in multi-core systems. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2013 IEEE 7th International Conference on*, volume 02, pages 721–725, Sept 2013.

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011. ISBN 978-1-4419-8236-0.

[Koz08] Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. Phd's thesis, Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften, Carl von Ossietzky Universität Oldenburg, Ammerländer Heerstr. 114-118, 26129 Oldenburg, 2008. URL `http://oops.uni-oldenburg.de/id/eprint/742`.

[KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4): 52–62, July 1995. ISSN 0740-7459.

[LBD$^+$12] Cédrick Lelionnais, Matthias Brun, Jérôme Delatour, Olivier Henri Roux, and Charlotte Seidner. Formal Behavioral Modeling of Real-Time Operating Systems. In *The 14th International Conference on Enterprise Information Systems (ICEIS (2) 2012)*. Wroclaw, Poland, June 2012. URL `https://hal.archives-ouvertes.fr/hal-01093794`.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1 (1-2):134–152, October 1997.

[LSC13] Martin Lukasiewycz, Sebastian Steinhorst, and Samarjit Chakraborty. 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. EDA Consortium, San Jose, CA, USA, 2013. ISBN 978-1-4503-2153-2. URL `http://dl.acm.org/citation.cfm?id=2485288.2485524`.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.

[MAAK15] Georg Macher, Muesluem Atas, Eric Armengaud, and Christian Kreiner. Automotive real-time operating systems: A model-based configuration approach. *SIGBED Rev.*, 11(4):67–72, January 2015. ISSN 1551-3688. URL `http://doi.acm.org/10.1145/2724942.2724953`.

[MAK14] Georg Macher, Eric Armengaud, and Christian Kreiner. Automated generation of autosar description file for safety-critical software architectures. *Lecture notes in informatics*, 2014.

[MJL06] Jonathan Musset, Etienne Juliot, and Stéphane Lacrampe. Acceleo référence. Technical report, Technical report, Obeo et Acceleo, 2006.

[MSAK15] Georg Macher, Michael Stolz, Eric Armengaud, and Christian Kreiner. Filling the gap between automotive systems, safety, and software engineering. *e & i Elektrotechnik und Informationstechnik*, pages 1–7, 2015. ISSN 0932-383X. URL `http://dx.doi.org/10.1007/s00502-015-0301-x`.

[Muc06] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 2006. ISBN 1-55860-320-4.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152: 125–142, 2006.

[NAD$^+$02] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P Black, Peter O Müller, Christian Zeidler, Thomas Genssler, and Reinier Van Den Born. A component model for field devices. In *Component Deployment*, pages 200–209. Springer, 2002.

[Obj] Object Management Group (OMG). Webpage. `http://www.omg.org`. Last checked: 10.12.2015.

[Obj03] Object Management Group (OMG). MDA Guide Version 1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`, June 2003. URL `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`.

[Obj10] Object Management Group (OMG). *Object Constraint Language, v2.2.* Object Management Group (OMG), , February 2010. URL http://www.omg.org/spec/OCL/2.2/. OMG Document Number: formal/2015-02-01.

[Obj11a] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification*. Object Management Group (OMG), , June 2011. URL http://www.omg.org/spec/MOF/2.4.1/. Document formal/2013-06-01.

[Obj11b] Object Management Group (OMG). Uml 2.4.1 superstructure specification, 2011.

[Obj15] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.2*. Object Management Group (OMG), , February 2015. URL http://www.omg.org/spec/QVT/1.2/. Document formal/2015-02-01.

[OCH] Daniel Opp, Mirko Caspar, and Wolfram Hardt. Code generation for timed automata system specifications consideringtarget platform resource-restrictions.

[OSE] OSEK. Osek/vdx goals and motivation. http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=4. Last checked: 10.12.2015.

[OSE01] OSEK/VDX. Osek - time-triggered operationg system. Technical report, OSEK/VDX, July 2001. URL http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf.

[OSE04a] OSEK. Osek communication version 3.0.3. Technical report, July 2004. URL http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf.

[OSE04b] OSEK VDX. Oil: Osek implementation language. Technical Report 2.5, OSEK/VDX, July 2004.

[OSE05] OSEK VDX. Osek/vdx operating systen 2.2.3. Technical report, OSEK/VDX, February 2005.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 55–71. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2829-5. URL http://dx.doi.org/10.1109/FOSE.2007.22.

[PFKH$^+$12] Marie-Agnès Peraldi-Frati, Daniel Karlsson, Arne Hamann, Stefan Kuntz, and Johan Nordlander. The timmo-2-use project: Time modeling and analysis to use. In *ERTS2012 International Congres on Embedded Real Time Software and Systems*, 2012.

[PH15] Uwe Pohlmann and Marcus Hüwe. Model-driven allocation engineering. In *Proceedings of the 30th ACM/IEEE International Conference on Automated Software Engineering*, ASE '15, pages 374–384. IEEE, 2015.

[PHMG14] Uwe Pohlmann, Jörg Holtmann, Matthias Meyer, and Christopher Gerking. Generating Modelica models from software specifications for the simulation of cyber-physical systems. In *Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '14, pages 191–198. IEEE Computer Society, August 2014.

[PKH$^+$11] Tomáš Pop, Jaroslav Keznikl, Petr Hošek, Michal Malohlava, Tomáš Bures, and Petr Hnětynka. Introducing support for embedded and real-time devices into existing hierarchical component system: Lessons learned. In *Software Engineering Research, Management and Applications (SERA), 2011 9th International Conference on*, pages 3–11. IEEE, 2011.

[PKM] Søren Pedersen, Jesper Kristensen, and Arne Mejholm. Automatic translation from uppaal to c.

[Pol] PolarSys. Trace compass. `https://projects.eclipse.org/projects/tools.tracecompass`. Last checked: 10.12.2015.

[proa] CHESS project. Chess: Composition with guarantees for high-integrity embedded software components assembly. `http://www.chess-project.org`. Last checked: 10.12.2015.

[prob] CONCERTO project. Concerto: Guaranteed component assembly with round trip analysis for energy efficient high-integrity multi-core systems. `http://www.concerto-project.org`. Last checked: 10.12.2015.

[pro14] CONCERTO project. D2.1 – multiview-based design of industrial multicore systems – initial version. Technical Report 1.1, CONCERT project, May 2014.

[PWT$^+$08] Marek Prochazka, Roger Ward, Petr Tuma, Petr Hnetynka, and Jiri Adamek. A component-oriented framework for spacecraft on-board software. In *Proceedings of DASIA*. Citeseer, 2008.

[Ros14] Mario Rose. Modellgetriebene generierung von api-konnektoren für eingebettete sensoren und aktuatoren. Bachelor's thesis, Software Engineering, University of Paderborn, Zukunftsmeile 1, 33102 Paderborn, April 2014.

[RTI] RTI. Rti connext dds cheatsheet. `https://community.rti.com/static/documentation/connext-dds/5.2.0/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_QoS_Reference_Guide.pdf`. URL `https://community.rti.com/static/documentation/connext-dds/5.2.0/doc/manuals/connext_dds/RTI_ConnextDDS_CoreLibraries_QoS_Reference_Guide.pdf`.

[Sca] Simple Scalar. Simplescalar webpage. `http://www.simplescalar.com`. Last checked: 10.12.2015.

[Sch00] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 2000. ISBN 0-471-98232-6.

[Sch15] David Schubert. Identification of safe states for reconfiguration in mechatronicuml. Master's thesis, University of Paderborn, July 2015.

[SG13] Bran Selic and Sébastien Gérard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Elsevier, 2013. ISBN 978-0-124-16656-1.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software : beyond object oriented programming*. ACM Press, New York, 2. ed. edition, 2002. ISBN 0-201-74572-0. Teilw. als Printed-On-Demand Ausg. mit späterem Erscheinungsjahr.

[SR90] JohnA. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *Real-Time Systems*, 2(4):247–254, 1990. ISSN 0922-6443. URL `http://dx.doi.org/10.1007/BF01995673`.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, December 1973. ISBN 978-3211811061.

[SVB$^+$06] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006. ISBN 978-0-470-02570-3. I-XVI, 1-428 pp.

[SW07] Wilhelm Schäfer and Heike Wehrheim. The challenges of building advanced mechatronic systems. In *Future of Software Engineering*, FOSE '07, pages 72–84. IEEE Computer Society, May 2007. ISBN 0-7695-2829-5.

[TBW95] K. Tindell, A. Burns, and A.J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, 1995. ISSN 0922-6443. URL `http://dx.doi.org/10.1007/BF01088855`.

[Tee12] Alexander Teetz. Werkzeuggestützter übergang von der plattformunabhängigen zur plattformabhängigen modellebene für eingebettete systeme im automobilbereich. Master's thesis, University of Paderborn, September 2012.

[VDHBL$^+$12] Martijn MHP Van Den Heuvel, Reinder J Bril, Johan J Lukkien, Damir Isovic, and Gowri Ramachandran. Rtos support for mixed time-triggered and event-triggered task sets. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 578–585. IEEE, 2012.

[VSB99] Rini Van Solingen and Egon Berghout. *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill, 1999. ISBN 978-0-077-09553-6.

[VSC$^+$09] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the procom real-time component model. In *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, pages 478–485. IEEE, 2009.

[Wan91] Yi Wang. *A calculus of real time systems*. Chalmers University of Technology, 1991. ISBN 91-7032-589-8.

[WDNZ$^+$14] Ernest Wozniak, Marco Di Natale, Haibo Zeng, Chokri Mraidha, Sara Tucci-Piergiovanni, and Sebastien Gerard. Assigning time budgets to component functions in the design of time-critical automotive systems. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 235–246. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-3013-8. URL `http://doi.acm.org/10.1145/2642937.2643015`.

[Wik] Wikipedia. Integer linear programming. `https://en.wikipedia.org/wiki/Integer_programming`. Last checked: 10.12.2015.

[Wil09] Reinhard Wilhelm. Determining bounds on execution times. In *Embedded Systems Handbook*, pages 14–23. CRC Press, 2 edition, 2009. ISBN 9781439807613.

[ZHC02] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z Chen. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference*, pages 183–188. ACM, 2002.

[Zim80] H. Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, Apr 1980. ISSN 0090-6778.