Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Generating Code for Distributed Deployments of Cyber-Physical Systems Using the MechatronicUML

David Stürner

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Steffen Becker
Supervisor:	Prof. Dr. Steffen Becker
Commenced:	November 8, 2021
Completed:	May 8, 2022

Abstract

Models are applied in engineering disciplines to describe systems from a higher level of abstraction. In Model-Driven Software Engineering (MDSE), formal models are used to design and verify software systems and to infer platform-specific models and implementations. The MechatronicUML is an MDSE method specifically designed for distributed cyber-physical systems (CPS). This thesis explores how the MechatronicUML may be used for generating code. The exact state of previous code generation approaches is not precisely known. The objective of this thesis is to design and implement a MechatronicUML-based code generator for distributed deployments of CPS. Previous code generation approaches are analyzed for this purpose and one approach is selected and extended to support a particular type of robot car as a target platform. A taxonomy for model-based code generation is proposed to structure the analysis of the previous approaches. Based on the selected previous approach, a code generator is presented and implemented. Additionally, an automotive application scenario is used as a case study for evaluating the concept and the implementation of the presented code generator. This code generator supports modeling the distributed deployment of a CPS with the MechatronicUML and generates platform-specific source code which can be successfully compiled and deployed on the Arduino-based robot cars. Ultimately, the thesis presents a proof of concept to generate the code for a distributed CPS based on the MechatronicUML.

Kurzfassung

Ingenieurswissenschaften nutzen Modelle, um Systeme von einer höheren Abstraktionsebene zu beschreiben. In der Modellgetriebenen Softwareentwicklung (MDSE) werden formale Modelle dazu verwendet, um Softwaresysteme zu entwerfen, zu verifizieren und um plattformspezifische Modelle und Implementierungen daraus abzuleiten. Die MechatronicUML ist eine MDSE Methode die speziell auf verteilte cyber-physische Systeme (CPS) abzielt. Diese Arbeit untersucht, inwiefern die MechatronicUML zur Codegenerierung genutzt werden kann. Der genaue Zustand voriger Ansätze zur Codegenerierung ist nicht im Detail bekannt. Das Ziel dieser Arbeit ist es, einen Codegenerator für verteiltes CPS zu entwickeln. Der Codegenerator soll der auf der MechatronicUML basieren. Zu diesem Zweck werden die vorhergehenden Codegenerierungsansätze analysiert und einer der Ansätze wird zur Weiterentwicklung für eine spezielle Sorte Roboterautos als Zielplattform ausgewählt. Zum Zweck einer strukturierten Analyse der vorigen Ansätze wird außerdem eine Taxonomie für Modellgetriebene Codegeneratoren vorgestellt. Schließlich stellt die Arbeit das Konzept und die Implementierung eines Codegenerators vor, der auf dem ausgewählten vorhergehenden Ansatz basiert. Außerdem wird eine Fallstudie zur Evaluierung des vorgestellten Codegenerators durchgeführt. Dieser Codegenerator unterstützt die Modellierung eines CPS mit der MechatronicUML und generiert plattformspezifischen Quellcode, welcher erfolgreich kompiliert und auf die Arduino-basierten Roboterautos aufgespielt werden kann. Schlussendlich legt diese Arbeit die Machbarkeit der Codegenerierung für ein verteiltes CPS basierend auf der MechatronicUML dar.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface. 12

ApiMappingML API Mapping Modeling Language. 37

ApiML API Modeling Language. 37

BPMN Business Process Model and Notation. 16

CIC Component Instance Configuration. 20, 56

CPS cyber-physical system. 1

DDS Data Distribution Service. 38

DDS-XRCE DDS for eXtremely Resource Constrained Environments. 76

DSL Domain-Specific Language. 16

ECU Electric Control Unit. 6

EMF Eclipse Modeling Framework. 15

EPL Eclipse Public License. 62

HPIC Hardware Platform Instance Configuration. 25

I2C Inter-Integrated Circuit. 6

IDE Integrated Development Environment. 33

IOPT Input-Output Place-Transition. 43

IP Internet Protocol. 75

LHS lefthand side. 11

MDA Model Driven Architecture. 9

MDSE Model-Driven Software Engineering. 1

MOFM2T MOF Model to Text Transformation Language. 14

MQTT Message Queuing Telemetry Transport. 75

OCL Object Constraint Language. 14

OMG Object Management Group. 9

PIM Platform-Independent Model. 9

PLC Programmable Logic Controller. 26

PSM Platform-Specific Model. 9

PWM Pulse Width Modulation. 131

QoS Quality of Service. 21

QVTo QVT Operational. 12

RAM Random Access Memory. 26

RDBMS Relational Database Management System. 12

RHS righthand side. 11

ROM Read-only Memory. 26

RTCP Real-Time Coordination Protocol. 19, 56

RTSC Real-Time Statechart. 22, 56

SML Scenario Modeling Language. 43

TCP Transmission Control Protocol. 21

UML Unified Modeling Language. 12

WCET Worst Case Execution Time. 35

1 Introduction

A cyber-physical system (CPS) is a software system that combines real-world, physical aspects with computation and communication technologies [Zan17]. Embedded computers are used to process sensor signals and control physical actuators. Such systems can be found in aircrafts, automotive systems and traffic control, trains, manufacturing, water and energy infrastructure and many other applications [Lee15; Zan17]. Special challenges arise when these embedded computing systems do not only replace formerly mechanical controllers, but when they add additional functionality and make use of distribution and communication capabilities [Wol09]. Models are a key abstraction used in engineering disciplines to design complex systems. In Model-Driven Software Engineering (MDSE), models are used to design the system, allowing simulation and verification, and to derive artifacts and implementations [Obj14]. Based on formal models, such implementations may be created in a fully or partially automated way.

1.1 Problem Statement

The MechatronicUML is an MDSE method specifically designed for distributed CPS [DPP+16; Fra22b; HFK+16]. It provides a process for engineering CPS along with a set of modeling languages and tools. The proposed process comprises all stages of software engineering, starting with the modeling of requirements and concluding with the creation of software artifacts. There have been several endeavors for code generation from the MechatronicUML, but as opposed to the requirements engineering [HFK+16], the platform-independent modeling [DPP+16] or the hardware platform description modeling [DP], these endeavors are not documented in one unified technical report. Thus, the particularities of these code generation approaches are unknown, as well as their state and adaptability.

The goal of this thesis is to explore how the MechatronicUML can be used to generate source code for the distributed deployment of a CPS. Instead of manually implementing the source code and deploying it to the target environment, the software an be designed with the MechatronicUML and the code can be generated. This is depicted in Figure 1.1: The application scenario is modeled with the MechatronicUML and on this higher layer of abstraction, it is transformed into source code. The generated code is deployed to a concrete target platform. More specifically, the vision is to create a code generator for a laboratory environment of robot cars. With their driving and sensing capabilities, these robot cars are suitable to represent autonomous vehicles in a laboratory environment and serve as a target platform for the code generation. This envisioned code generator allows rapid prototyping of application scenarios by modeling them with the MechatronicUML and deploying the generated code on the robot cars for experiments and further analysis.

In particular, this thesis focuses is on code generation for distributed deployments: A code generator for distributed deployments must consider the particularities of a distributed target platform including communication.

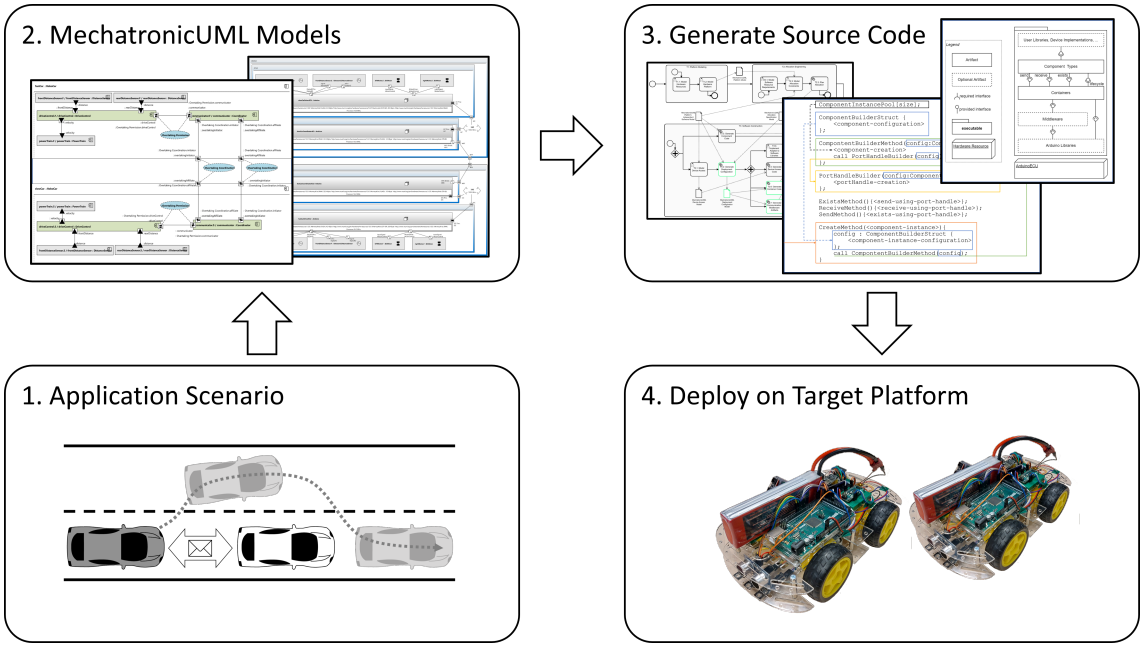


Figure 1.1: The goal is to generate code for a distributed CPS application modeled with the MechatronicUML.

1.2 Research Questions

In connection to the problems this thesis seeks to solve, the following research questions are formulated. They are answered in the course of this thesis. The answers to **RQ1.1** and **RQ1.2** refine the problem space while the remainder of the questions describes the solution space.

RQ1.1: What specific application scenario can be used to demonstrate the MechatronicUML modeling capabilities and the code generation?

RQ1.2: What type of robot car is used as the target platform?

RQ2.1: What criteria are applied to assess the previous MechatronicUML-based code generation approaches?

RQ2.2: What is the state of the previous MechatronicUML-based code generation approaches?

RQ3.1: Which (parts of) previous approaches are reused, and why or why not?

RQ3.2: What are the missing capabilities of a MechatronicUML-based code generator for the desired application scenario?

RQ3.3: How are these missing capabilities implemented?

RQ4.1: Can the application scenario be modeled with the MechatronicUML in a way that is suitable for code generation?

RQ4.2: Does the code generator produce valid source code for the modeled application scenario?

1.3 Solution Approach

As a basis for developing the envisioned code generator and to answer the aforementioned research questions, this thesis includes the analysis of previous code generation approaches with the MechatronicUML for distributed deployments. The goal of the analysis is to assess the state of these approaches' concepts and implementations in order to decide which previous endeavors could be reused or extended. Figure 1.2 sketches the solution approach followed in this thesis. Based on the analysis, a code generation concept is presented and a corresponding code generator is implemented. Finally, the code generator is used to implement an application scenario.

Furthermore, in order to structure the analysis of the previous MechatronicUML-based code generation approaches, a taxonomy for model-based code generation is proposed. It describes and categorizes important facets of model-based code generation approaches. The taxonomy is designed for a structured comparison of different model-based code generation approaches. It serves to define the properties of an ideal candidate approach, and to assess and compare the existing approaches.

The solution approach also involves a case study. As an exemplary use case, a specific application scenario from the domain of autonomous vehicles is designed. It is used as a running example throughout this thesis. The application scenario describes a cooperative overtaking maneuver between two autonomously operating cars. Arduino-based robot cars are the target environment for this application scenario, and thus have to be equipped with appropriate sensing and driving capabilities. The implementation of this application scenario serves as a case study to demonstrate and evaluate the code generator.

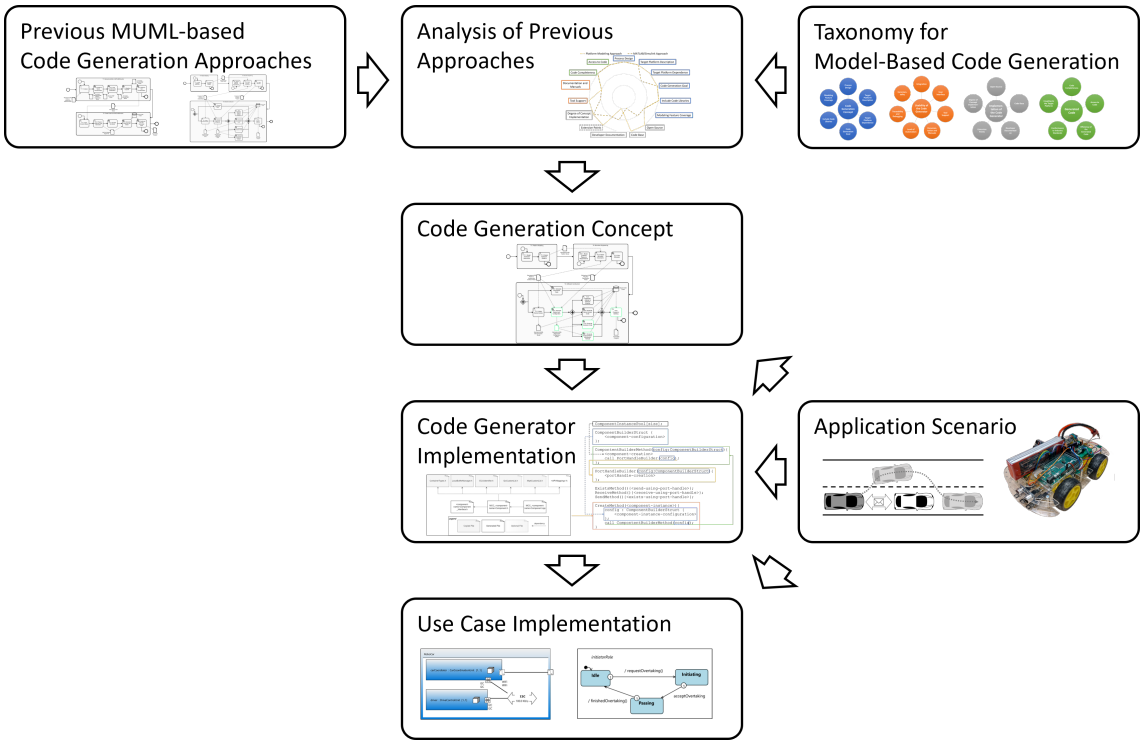


Figure 1.2: The solution approach which is followed in this thesis to develop a concept and implementation for the envisioned code generator.

In conclusion, the contributions of this thesis are (i) a taxonomy for model-based code generation, (ii) an analysis of previous code generation approaches using the MechatronicUML, and (iii) the concept and implementation of a code generator which is suitable to support the robot car laboratory environment. Additionally, as a paradigmatic use case, an application scenario for the robot cars is designed and implemented using the MechatronicUML modeling features and the newly created code generation capabilities.

Thesis Structure

This results in the following structure of the thesis. It is complemented by an appendix that contains supplementary material which may be especially relevant for readers who seek to reproduce, improve, extend or adapt the presented implementations.

Chapter 2 – Application Scenario: First, the application scenario is described as it is used as a running scenario in this thesis. It consists of an exemplary use case and the description of the robot car target environment.

Chapter 3 – Foundations: Afterwards, the theoretical foundations of this thesis are introduced, covering model transformations and a definition of *model-based code generation* as well as the MechatronicUML, its modeling languages and tools, and the MATLAB/Simulink tools.

Chapter 4 – Related Work: Thirdly, related work is described. This chapter incorporates related work in the area of taxonomies in MDSE, and related code generation approaches with a special focus on MechatronicUML-based approaches for distributed deployments.

Chapter 5 – A Taxonomy for Model-Based Code Generation: In this chapter, the first contribution is presented: A taxonomy for model-based code generation approaches with a total of 23 facets from four different perspectives.

Chapter 6 – Concept for the Code Generation: Using this taxonomy, the previous code generation approaches are analyzed to identify a suitable candidate for the application scenario. The chapter concludes by presenting a code generation concept tailored to the robot car platform.

Chapter 7 – Implementation of the Code Generator: Then, this concept is implemented: different extensions of metamodels, adaptions of model-to-model transformations as well as a new model-to-text transformation are presented.

Chapter 8 – Implementation of the Application Scenario: This newly implemented code generator is then used, along with the numerous MechatronicUML modeling features, to design and implement the application scenario.

Chapter 9 – Evaluation: The fitness of the presented concepts and implementations is evaluated by summarizing and discussing the results of the application scenario implementation.

Chapter 10 – Conclusion: Finally, the thesis is concluded by summarizing the most important achievements and limitations, and providing an outlook for future research.

2 Application Scenario

This section describes the application scenario which is used throughout this thesis. It serves as a use case in order to demonstrate the application of the MechatronicUML for modeling software, modeling hardware platforms and eventually generating source code for a distributed deployment. Therefore, Section 2.1 introduces the *cooperative overtaking* scenario as the running example for the application logic, while Section 2.2 describes an exemplary hardware platform to realize this running example with.

2.1 The Cooperative Overtaking Scenario

The MechatronicUML is a software engineering method for CPS with a special focus on coordinating autonomous systems at real-time (see Section 3.3). The *cooperative overtaking* has been used frequently in literature to demonstrate the features of the MechatronicUML's modeling languages and tools [BCD+14; DGB+14; Poh18]. There are several variations of the cooperative overtaking scenario. In this subsection, one specific instance of the cooperative overtaking is introduced. It is used as running example in this thesis.

The cooperative overtaking involves two autonomously operating vehicles. The situation can be described as follows: Both vehicles travel on the same road into the same direction, and the road has a fast lane (e.g., the road might be a freeway). The vehicle in front is driving on the right lane and will be called the *affiliate*. The vehicle in the back is also traveling on the right lane, and because it runs faster, it will approach the affiliate. The second vehicle will be called the *overtaker* as it attempts to pass the affiliate using the fast lane, with the goal to maintain its faster traveling speed.

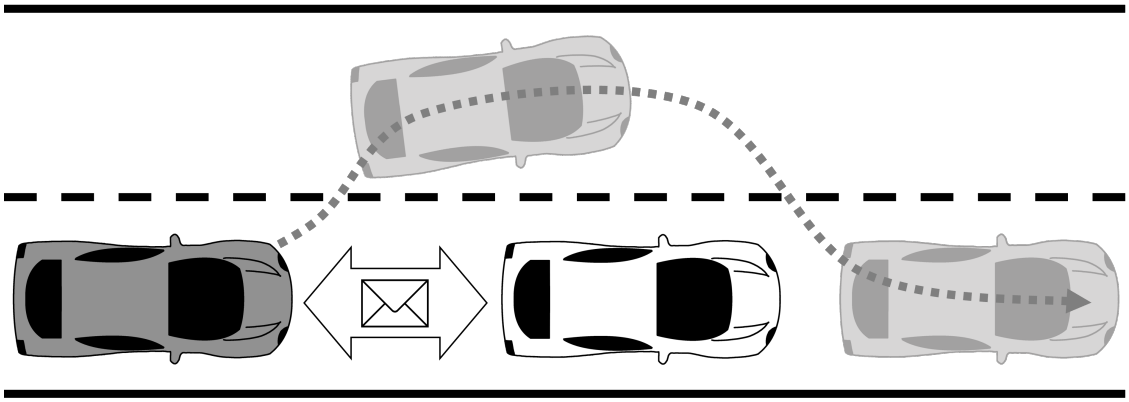


Figure 2.1: The cooperative overtaking of two autonomous vehicles

This scenario is also visualized in Figure 2.1 with the overtaker being represented by a grey car, and the affiliate as a white car. Passing a slower car is only possible if the overtaker keeps a speed difference over the other vehicle throughout the overtaking process. In the particular scenario presented here, the assumption is that both cars maintain their exact speed, i.e., the affiliate does not accelerate while being passed. Consequently, this requires a cooperation between the overtaker and the affiliate: Both vehicles exchange messages via wireless communication in order to coordinate on the overtaking process. The overtaker will only pass the affiliate once it agrees on being passed and hence indicates that it is aware of the overtaking process. Thus, the overtaker can be sure that the affiliate will not accelerate while being passed. Moreover, the scenario is regarded under the assumption that a started overtaking maneuver can and is always executed successfully, i.e., is never aborted.

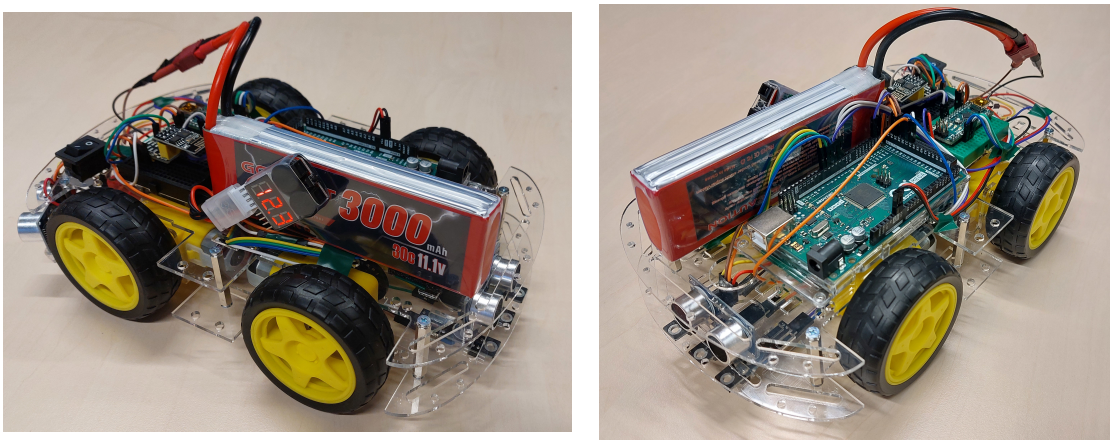
In a real-world scenario, other factors and decisions are also reasonable, such as the affiliate accelerating to at least the same speed as the overtaker and thus making the passing unnecessary. Furthermore, other vehicles could be included in the decision making process, or even a section controller which collects data for a specific section of the respective road [Poh18]. Similarly, maps and navigation systems could contribute relevant information on speed limits or number and length of fast lanes. And the scenario could also be ported to a different road situation where overtaking is only possible by temporarily using an oncoming lane (e.g., on a highway). These situations are not considered in the specific example that is introduced here.

The definition of the application scenario provides an answer to the first research question, **RQ1.1:** What specific application scenario can be used to demonstrate the MechatronicUML modeling capabilities and the code generation? The cooperative overtaking is defined and chosen inspired by previous research. Furthermore, it fits to the envisioned laboratory environment for robot cars to demonstrate automotive application scenarios, and it is realized using the available hardware which is described next.

2.2 The Robot Car Target Platform

The target platform described in this section is used to realize the running example presented above. While the cooperative overtaking scenario describes the application logic, the target platform describes the hardware and thus completes the application scenario for the MechatronicUML and the code generation. Thus, this section answers the research question, **RQ1.2:** What type of robot car is used as the target platform? For the purpose of this thesis, small robot cars are used to represent autonomous vehicles in a laboratory environment. The robots are composed of several Electric Control Units (ECUs) and equipped with driving, sensing and communicating capabilities. They are able to operate and drive autonomously and are thus suitable to represent vehicles as in the aforementioned running example.

Figure 2.2 shows the robot cars. They are based on two Arduino microcontrollers which can communicate with each other using Inter-Integrated Circuit (I2C). Furthermore, one Arduino board employs a WiFi module for wireless communication. The robot cars are equipped with infrared sensors to detect a line on the ground, with two ultrasonic sensors to measure the distance at the car's front and rear, and with four simple DC motors attached to the car's four wheels. Additionally, a set of libraries is implemented to realize the basic functionality of the robot cars. Arduino microcontrollers are programmed using a subset of C++ and C programming language features [Ard22; Söd22].



(a) Front-left view.

(b) Right-side view.

Figure 2.2: The robot cars with their chassis holding the microcontrollers, sensors and actuators.

The created libraries comprise the driving behavior including following a line on the ground which can effectively represent a lane of a road, and reacting to distance measurements by adapting the driving behavior, e.g., stopping before crashing into an obstacle or other vehicle. All details about the employed hardware as well as the implemented libraries is supplied in Appendix A. The implemented libraries for the robot cars are referenced as *robot car libraries* within this thesis.

3 Foundations

This chapter sums up the theoretical foundations for this thesis, beginning with a definition of the term *model-based code generation* in Section 3.1 as the object of interest of the presented work. Next, foundations of model transformations are covered in Section 3.2, before the MechatronicUML is introduced: The MechatronicUML including its tool suite are described in Section 3.3, the modeling language for platform-independent modeling in Section 3.4, and the modeling language for the hardware platform description modeling in Section 3.5. Finally, a short introduction to MATLAB/Simulink concludes the chapter.

3.1 Model-Based Code Generation

A central idea of MDSE is creating value from software models, e.g., by using them for simulation, verification or for code generation [Obj14]. Generating code transforms the software from its abstract model level to an executable state. In the course of this thesis, this is called *model-based code generation*.

In the domain of MDSE, there are many different model transformation languages originating from academic publications, open source tools or commercial tools [CH06; KC15]. Such model transformation languages and tools have the capability to effect the transformation from a model instance into a target representation as described above. This target representation may be another model (from the same or from a different modeling language) or a text file, which can also a be source code file that can be compiled into executable program code.

The Object Management Group (OMG) initiative Model Driven Architecture (MDA) is a well-known process in the domain of MDSE. As one of its core ideas, the MDA describes that a Platform-Specific Model (PSM) can be derived from a Platform-Independent Model (PIM) by the means of model transformation [Obj14]. The PSM is more tightly coupled to a technology, and therefore suited to infer an actual implementation, i.e., generating code. In practice, one single transformation step is often not sufficient for code generation from the PIM level. Instead, a chain of transformations is created, potentially using different transformation languages and tools for the single steps.

The term *model-based code generation* refers to the overall transformation from model instance to source. Model-based code generation includes the transformation process as well as its implementation using different transformation languages and tools [SCDP07]. Furthermore, model-based code generation also depends on the modeling language that the code generation starts with, the implementation of the modeling language's metamodel and its tooling, e.g., an editor and simulator.

The term *model-based code generation approach* is used when referring to a specific instance of model-based code generation. A model-based code generation approach is characterized by four essential parts that it is composed of. These parts are visualized in Figure 3.1 and are explained in the following:

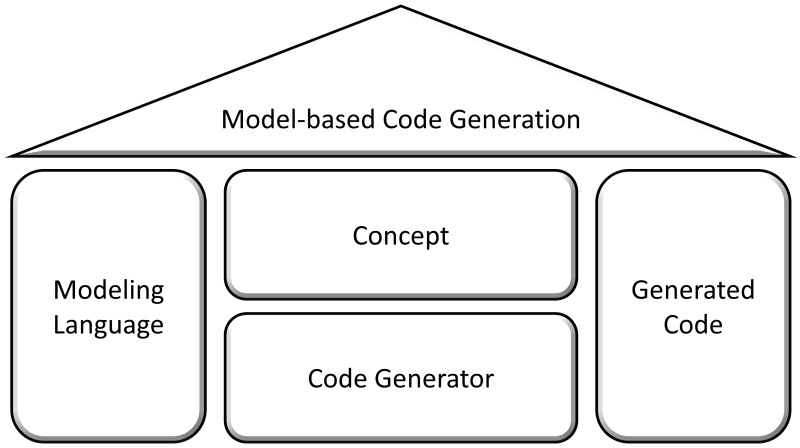


Figure 3.1: The characteristic elements of model-based code generation approaches

Modeling Language The starting point of model-based code generation is a model instance of a specific modeling language. Typically, a concrete model-based code generation approach is built for exactly one source modeling language (or modeling language family).

Concept For each model-based code generation approach, there is a theoretical concept defining the logical steps that are applied to transform a model to source code. The concept may also define how the target platform for the code generation is modeled, and which kinds of model transformation (see Section 3.2) are used to execute the transformation steps. The logical concept may not necessarily be explicitly described or documented, but it may be inferred from the implementation of the code generator.

Code Generator The code generator is the implementation of the code generation concept: An executable piece of software that receives a the representation of a model instance as input and produces source code. The code generator does not have to be fully automated: User interaction may be required in the course of code generation, e.g., to annotate the model or provide additional information for the code generation as part of the overall code generation process. Consequently, the code generator typically employs model transformation tools that are able to implement the intended kinds of model transformation.

Generated Code Finally, the generated code is also a characteristic asset of a model-based code generation approach as the generated code is the final output product. The term *generated code* in this context also includes configuration files, build or deployment artifacts, depending on the concrete code generation approach.

3.2 Model Transformations

As already stated in the previous Section 3.1, model transformations have the capability to create a new representation from a model instance. Such model transformations can for example be applied to generate textual source code from a model instance. Consequently, model transformations are an important foundation for model-based code generation as it typically employs model transformation tools to implement a transformation chain for code generation.

Model transformations are often used in a *forward engineering* way, crossing abstraction levels to get from a more abstract, technology independent viewpoint to a more concrete viewpoint [Obj14]. Nonetheless, model transformations can just as well translate between representations within the same level of abstraction, or to a higher level of abstraction. And they can also be used to combine several input models into one unified output format.

A key term in the context of model transformations are *transformation rules*. A transformation rule describes which elements of the source model or source representation are transformed into which elements of the target model or target representation [CH06]. Thus, they are the smallest unit of transformation and describe one granular transformation step; an entire model transformation is made up of multiple of such rules in combination. In this terminology, the term *rule* does not imply a declarative rule definition. An individual transformation step may also be defined as an imperative procedure or function. Therefore, the term *transformation rule* is used as an umbrella term for the general notion of a single transformation step.

As transformation rules may be applicable in multiple directions, instead of calling the concerned domains of a transformation the *source* and *target* domain, they may also be called the lefthand side (LHS) and righthand side (RHS) [CH06]. Depending on the exact way of rule definition and implementation of the model transformation tool, transformations may be possible both from the LHS and the RHS.

While model transformations can be distinguished by a detailed comparison of their features (cf. Section 4.1), at the top level there are two main categories: model-to-model and model-to-text transformations [CH06; KC15]. “The distinction between the two categories is that, while a model-to-model transformation creates its target as an instance of the target metamodel, the target of a model-to-text transformation is just strings.” [CH06] For the sake of completeness, Czarnecki and Helsen and Kahani and Cordy mention text-to-model transformations as a third category useful for reverse engineering, but do not examine them more closely in their work respectively [CH06; KC15]. As the context of this work is code generation which is forward engineering by its intention, this section is also limited to model-to-model transformations in Section 3.2.1 and model-to-text transformations in Section 3.2.2.

Occasionally, the term model-to-code transformation is used in literature [Sel03]. As compiler technology enables to transform textual source code to machine or byte code, model-to-text transformations can typically be applied to generate textual source code [CH06]. However, the source code of a programming language is not a meaningless sequence of characters but rather a representation of the programming language’s metamodel. That also fits the category of model-to-model transformation, so a clear distinction may sometimes not be possible. Therefore, a chain of model-to-model and model-to-text transformation steps appears useful, which corresponds to the presented definition of *model-based code generation* in Section 3.1.

3.2.1 Model-to-Model Transformations

The output of a model-to-model transformation is an instance of the target metamodel [CH06]. The target metamodel can be the same or a different one than the source metamodel. Czarnecki and Helsen define six kinds of model-to-model transformations.

Direct manipulation approaches typically offer some kind of Application Programming Interface (API) to directly alter a representation of the model [CH06]. These approaches are usually not complete environments for creating an entire model-to-model transformation, but can be used and enabled by programming languages such as Java. Thus, it might for instance be useful to first copy the model instance and then applying the manipulations to it, so the original source model would still be present after the transformation.

Structure-driven approaches follow such a copying strategy [CH06]. However, these approaches copy single model elements from LHS to RHS. This typically happens in two phases: first, the structure of the target model is created and then, the elements' attributes are set.

Template-based approaches also copy attribute values from the LHS to the RHS, but they do not first build the RHS model's structure [CH06]. Instead, the structure is already given by a template typically following the syntax of the RHS modeling language, which will then be filled with values from the LHS model.

Relational approaches describe the transformation rules based on mathematical relations between the LHS and RHS model [CH06]. Such relational descriptions are typically declarative, and build a set of constraints to solve in order to effect a correct transformation [KC15]. Such relational systems can often be used in both directions.

Similarly, *graph-transformation based approaches* are based on the mathematics of graphs [CH06]. Graphs are used as formal representations to describe which model elements remain unchanged, are added, deleted or altered in the transformation.

Lastly, there are also *operational approaches*. They are typically based on the metamodeling formalism that is used for the LHS model, and extend it with imperative programming constructs and model query facilities [CH06]. Thus, such approaches form an own programming language that allows to specify a transformation entirely using the transformation tool, and in a way that feels natural to imperative programmers. Therefore, these approaches are also described as imperative or constructive approaches [KC15]. The QVT Operational (QVTo) language is such a transformation language that uses an operational approach [CH06; KC15].

The QVTo language is specified by the OMG[Obj15], and "Ecplise QVTo is the only actively maintained QVTo implementation" [Wil22]. Each QVTo transformation is defined by a signature using the keyword `transformation`. The signature names the transformation and specifies the input and output entity with the respective metamodel reference. E.g., Listing 3.1 shows the transformation `uml2Rdbms` that is applicable to an entity of the `SimpleUML` metamodel as input and an entity of the `SimpleRDBMS` metamodel as output. These metamodels are just examples here, and they are used to demonstrate a simple transformation from the Unified Modeling Language (UML) to an Relational Database Management System (RDBMS) model. The entry point of each QVTo transformation is the operation with the name `main`.

Besides the main operation, the mapping operations are the core of a QVTo transformation [Obj15]. A mapping is defined using the keyword `mapping` and is the means to specify a transformation rule in QVTo. Each mapping is defined for a metaclass from the input metamodel. After two colons follows the name of the mapping and an optional parameter list in round brackets, and after another colon the return type is specified. The return type is a metaclass of the output metamodel. The signature of a mapping may be completed by `when` and `where` clauses for pre- and post-conditions [Obj15]. If not specified otherwise, the body of a mapping specifies the population: attributes of the return

Listing 3.1 A simple QVTo transformation based on [CH06].

```
1  transformation uml2Rdbms(in uml:SimpleUML, out rdbms:SimpleRDBMS);
2
3  main() {
4    uml.objectsOfType(Package)->map packageToSchema();
5    uml.objectsOfType(Attribute)->map attributeToColumn();
6  }
7
8  mapping Package::packageToSchema():Schema {
9    name := self.name;
10   tbls := self.elems->map classToTable();
11 }
12
13 mapping Class::classToTable():Table
14   when { self.isPersistent=true; } {
15   name := self.name;
16   key := object Column {
17       name := self.name + '_tid';
18       type := 'NUMBER';
19     };
20   cols := key;
21 }
22
23 mapping Attributes::attributeToColumn():Column }
24   ...
25 }
26 ...
```

type are assigned values using the given instance of the input type. The input instance is therefore referenced using the keyword `self`. Finally, mappings are called on instances of the respective type using the `map` keyword and the name of the respective mapping. Next to mapping operations, there are also *query* and *helper* operations. These are simple functions to encapsulate computations that provide a result value which is needed elsewhere, but do not implement transformation logic. As opposed to a query, a helper function may have side-effects on its parameters.

Listing 3.1 shows a sample QVTo transformation with a main operation that calls the mappings `packageToSchema` and `attributeToColumn` (lines 3-6). The latter mapping is not detailed further in the example. The `packageToSchema` mapping is defined for the metaclass `Package` of the `SimpleUML` metamodel and returns a `Schema` of the `SimpleRDBMS` metamodel (line 8). The mapping assigns values to the `Schema`'s attributes `name` and `tbls` using information from the input instance via the `self` keyword as well as by calling another mapping (line 9-10). The mapping `classToTable` defines a precondition as as `when` clause (line 14) and also creates a new object using the `object` keyword (lines 16-19). This example demonstrates some of the aforementioned QVTo language elements; for the full specification see [Obj15]. It has to be noted though that the Eclipse QVTo implementation does not adhere to all details of the specification regarding the concrete syntax [Sof20]. Therefore, consult the wiki page at [Wil22] for concrete implementation questions.

3.2.2 Model-to-Text Transformations

The output of a model-to-text transformation is a string of characters [CH06]. These strings may equally be documents, configuration files or source code. In literature, model-to-text transformations are categorized in two different kinds: Visitor-based and template-based approaches [CH03; CH06; KC15].

Visitor-based approaches traverse a representation of a model in order to generate text for each model element they "visit-on their trajectory [CH03]. Such a representation of a model is often tree-based to allow structured traversing [KC15]. Typically, the transformation rules specify which text or code is generated upon visiting a certain modeling element, often enriched with specific constraints or conditions. Visitor-based approaches are not used in this thesis, therefore see [KC15] for examples of visitor-based model transformation tools.

Template-based approaches generate the transformation output based on text templates [KC15]. A template defines the structure of the generated text and optionally the different files to be generated. As such, the template is very close to the generation target, i.e., the produced text or source code [CH06]. Typically, a template consists of template text that is written to the output exactly as it is in the template, as well as some kind of metacode that allows to access model elements of the source modeling language. This access to model elements may be used to produce the specific text required for the desired transformation.

Acceleo[1] is a model-to-text transformation tool employing a template based-approach [KC15]. It is advertised as a tool for creating code generators and implements the MOF Model to Text Transformation Language (MOFM2T) specification of the OMG [Mad18a]. The MOFM2T defines a language for model-to-text transformation based on *templates* and *queries* [Obj08]. Templates can be described as functions that do not have a return type, but write text to the transformation output. Queries on the other hand are functions that return a value of a certain type, which are used or evaluated in templates. Both these top-level language elements can further be structured into *modules* for large, complex transformations [Obj08]. Modules can import other modules in order to make use of their templates and queries.

Listing 3.2 shows a simple module called `main`, with a template called `main` and a query called `toJavaVisiblitiy`. The Module `main` is an Acceleo *main module*; not due to its name, but due to the `@main`-annotation in line 6 [Mad18b]. Main modules are the entry point into an Acceleo-transformation. To be precise, the specific template with the `@main`-annotation is the entry point, but the module inhabiting this template is still called main module. Acceleo uses the text-explicit mode of MOFM2T, delimiting the metacode by `[` and `]`. That means, all text outside of the delimiters are directed to the output without modifications, and the text within delimiters is the MOFM2T code.

The example in Listing 3.2 demonstrates the specification of an output file (line 7). This file is created as part of the Template `main`, which is applicable to an entity of type `Class` of the UML metamodel. Within this template, different attributes of the type `Class` are used to create the source code of a basic Java class and its member variables. Line 12 calls the Query `toJavaVisibility` which is defined in lines 19-29. This query returns a `String` and its body is implemented, as the MOFM2T specifies, using the Object Constraint Language (OCL) [Obj08]. The example shows a very simple transformation generating a java class with member variables. When generating the output, the

[1] https://www.eclipse.org/acceleo/

Listing 3.2 Simple Java class generation with Acceleo.

```
1  [comment encoding = UTF-8 /]
2  [module main('http://www.eclipse.org/uml2/2.0.0/UML')/]
3
4  [template public main(class : Class)]
5
6  [comment @main /]
7  [file (class.name.concat('.java'), false, 'UTF-8')]
8  package [class._package.URI /];
9
10 public class [class.name.toUpperFirst() /] {
11   [for (attribute : Property | class.attribute)]
12   [toJavaVisibility(attribute.visibility) /] [attribute.type.name /] [attribute.name /];
13   [/for]
14 }
15 [/file]
16
17 [/template]
18
19 [query private toJavaVisibility(visibility : VisibilityKind) : String =
20 if (visibility=VisibilityKind::public) then
21   'public'
22 else if (visibility=VisibilityKind::private) then
23   'private'
24 else if (visibility=VisibilityKind::protected) then
25   'protected'
26 else
27   ''
28 endif endif endif
29 /]
```

template follows the 'what-you-see-is-what-you-get'-principle, which also includes the whitespace in the template [Obj08]. Thus, in order to adhere to the usual Java formatting, the query call in line 12 is not indented inside the *for block*, and the the attribute access to the model elements follows right after. This is only an implementation design choice. Besides the for block in lines 11-13, the MOFM2T also supports *let blocks* to define variables and *if blocks* for simple branching [Obj08]. Furthermore, the import of other modules is not demonstrated in this simple example. Line 2 only shows the declaration of the module including the metamodels it references, in this case the UML 2.0 Eclipse Modeling Framework (EMF) metamodel. An *import block* would follow right after the module declaration. For the full language specification and how Acceleo implements it, see [Obj08] and [Mad18b].

3.3 The MechatronicUML and its Tool Suite

The MechatronicUML is a model-driven software engineering method, specifically created for the design of mechatronic systems or CPS [DPP+16]. It "particularly focuses on the software for the real-time coordination of mechatronic systems. The coordination is achieved by exchanging

messages between systems which amounts to complex discrete state-based behavior in each of the systems." [DPP+16] The MechatronicUML provides several methods and a set of Domain Specific Languages (DSLs) in order to support the entire software development cycle, starting from the requirements up to the finished, executable software [DPP+16; HFK+16]. This endeavor resulted in the MechatronicUML software engineering process. This process is visualized, using the Business Process Model and Notation (BPMN), in Figure 3.2. It describes the overall vision for the MechatronicUML, and puts the individual work that contributes methods and languages into context. The primary goal of this thesis is to use the MechatronicUML to generate code for the application scenario described in Chapter 2.

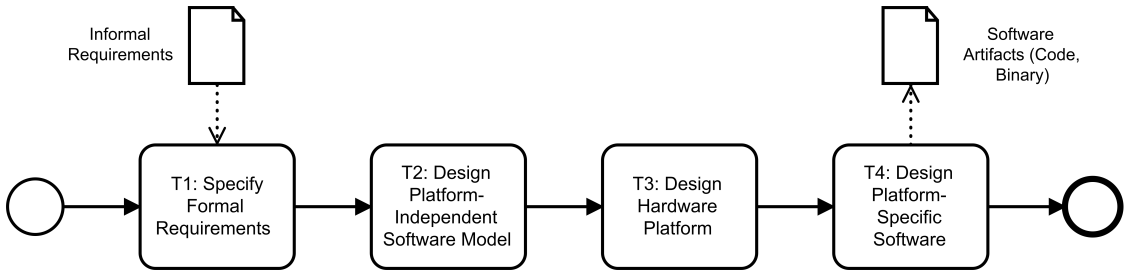


Figure 3.2: The MechatronicUML software engineering process [BCD+14; DP; DPP+16; HFK+16]

First of all, the MechatronicUML Requirements Engineering Method comes along with a requirements modeling language that allows to model the system requirements formally as scenarios [HFK+16]. It implements the first task of the MechatronicUML process: *T1: Specify Formal Requirements*. The MechatronicUML Requirements Engineering Method is not detailed further in this thesis.

Secondly, the MechatronicUML Design Method provides a process and a language for platform-independent modeling of CPS [DPP+16]. It implements the second task of the MechatronicUML process, *T2: Design Platform-Independent Software Model*. The platform-independent modeling language is the core of the MechatronicUML and is referred to as MechatronicUML PIM for the remainder of this thesis. It provides several modeling perspectives to model both the structural as well as the behavioral elements of CPS, focusing on the coordination of several autonomous subsystems at real-time via message-based information exchange. Section 3.4 explains the most important concepts and modeling features of the MechatronicUML PIM.

Thirdly, the MechatronicUML Hardware Platform Description Method implements the third task of the MechatronicUML process, *T3: Design Hardware Platform*, by proving a process and a modeling language tailored to hardware resources and platforms [DP]. The MechatronicUML Hardware Platform Description Modeling language is referred to as MechatronicUML HPDM in this thesis. Its modeling features are explained in Section 3.5. Originally, the the third process step was intended to define the hardware platform as well as foundational software assets such as operating system APIs [DP; DPP+16]. The latter is not implemented by the MechatronicUML HPDM, and was moved to the fourth step of the MechatronicUML process.

For the fourth step of the MechatronicUML process, *T4: Design Platform-Specific Software*, there is no technical report available equivalent to the MechatronicUML Requirements Engineering Method [HFK+16], the MechatronicUML Design Method [DPP+16] or the Mechatronic-

UML Hardware Platform Description Method [DP]. Some research has been done attempting to fill this gap, however, the exact state and maturity of the concepts and implementations are unknown. Exploring this previous work is a contribution of this thesis and the previous code generation endeavors are presented in Section 4.2.

Furthermore, there is the MechatronicUML Tool Suite [DGB+14] is available[2] for download via the MechatronicUML project website[3]. The MechatronicUML Tool Suite is an Eclipse-based tool that unites several plugins which implement the aforementioned features of the MechatronicUML [DGB+14]. It was developed between 2011 and 2016, and the latest release was version 1.0 [Fra22a; Hei16] which can also be seen in Table 3.1.

Version	Release Date	Eclipse Version
0.1	Nov 2011	Helios (3.6)
0.2	Mar 2012	Helios (3.6)
0.3	Aug 2012	Indigo (3.7)
0.4	Mar 2014	Kepler (4.3)
0.5	unknown	Luna (4.4)
1.0	Aug 2016	Neon (4.6)

Table 3.1: An overview about releases of the MechatronicUML Tool Suite based on [Fra22a; Hei14d; Hei17a].

The features of the MechatronicUML Tool Suite do not cover the entire MechatronicUML process. Specifically, the MechatronicUML Requirements Engineering is not implemented by the MechatronicUML Tool Suite [Hei14b; Hei14c; Hei16]. The tool development started by implementing the MechatronicUML PIM and adding different features for verification and portability to other development tools [Hei14d]. Later on, the MechatronicUML HPDM was added as well as a C code generator [Hei14b; Hei14d; Hei16]. The final version 1.0 contains the following features: the MechatronicUML PIM, the MechatronicUML Verification (Uppaal Adapter), the MechatronicUML HPDM, reconfiguration, platform-specific modeling, several C code generators and a Modelica Adapter [Hei16]. Up until at least version 0.4, there was also a MATLAB/Simulink adapter, which was discontinued later (see Section 4.2.2 and Section 6.3) [Hei14c; Hei17b]. This feature set demonstrates that the MechatronicUML Tool Suite has been the target environment for implementing the majority of methods and languages that have been contributed to the MechatronicUML process. Overall, the plugins for the MechatronicUML Tool Suite cover the MechatronicUML development process starting at task T2. Therefore, the MechatronicUML Tool Suite is an object of research in this thesis, especially for analyzing the state, maturity and capabilities of the previous code generation approaches (see Section 4.2, Section 6.2 and Section 6.3).

[2] The download was not working at the time of writing this thesis.

[3] http://www.mechatronicuml.org/de/download.html

3.4 MechatronicUML Platform-Independent Modeling

The MechatronicUML PIM language allows the platform-independent modeling of the software for a CPS [DPP+16]. This comprises modeling the structure of a distributed CPS and the behavior of each single component as well as the coordination behavior of those independent components (see Section 3.4.2). The application scenario that is introduced in Section 2.1 is used as a running example in this section to demonstrate the most important modeling language elements of the MechatronicUML PIM. This section focuses on the modeling features themselves; an explanation of the modeled software as a whole can be found later in Chapter 8. Also, the MechatronicUML PIM allows sophisticated modeling of the software structure and behavior with many variations. Only a subset of these variations is explained in this section, focusing on the modeling elements that are needed for this work. Specifics of the action language as well as the entire set of modeling features for reconfigurable components are not discussed in this work. For all details about the MechatronicUML PIM see[DPP+16].

The MechatronicUML PIM is widely supported by its implementation in the MechatronicUML Tool Suite (see Section 3.3). There are different diagram editors implementing the concrete syntax of the modeling language and those editors were used to create the models shown in this section. The section starts off by introducing the MechatronicUML component model in Section 3.4.1, followed by an introduction to the message-based communication modeling in Section 3.4.2 and concludes with an overview about the behavior modeling in Section 3.4.3.

3.4.1 Component Modeling

The MechatronicUML PIM employs a component model to specify the structure of the system [DPP+16]. The component model explicitly distinguishes between *component types* and *component instances*. Whenever the term *component* is used, it refers to the *component type*, whereas a *component instance* is explicitly called as such.

There are two kinds of components: *atomic components* and *structured components* [DPP+16]. Atomic components are the bottom layer of the component model: they have an individual behavior. This behavior specifies the component's internals, and some of this behavior is exposed to other component via ports. Depending on the kind of behavior a component contains, it may be a *discrete component* or a *continuous component*.

Discrete components contain state-based behavior and represent the application logic of the software [DPP+16]. Discrete components interact with other discrete components via the exchange of messages via *discrete ports* (see Section 3.4.2). Figure 3.3 visualizes the concrete syntax of the discrete atomic component `Coordinator` which has three different discrete ports. In general, the MechatronicUML PIM also supports different cardinalities of ports, making them so-called *multi-ports*. However, multi-ports are not be used in this work and are therefore not explained here.

Figure 3.3 also shows two continuous atomic components: The `DistanceSensor` and `PowerTrain` components. Their behavior is not state-based, but instead, they represent a continuous controller of a CPS. These controllers can also be described as *sensors* and *actuators*. Sensors are components that collect data from the real world, whereas actuators perform physical actions in the real world. Continuous components have *continuous ports* to expose their functionality. The `DistanceSensor`

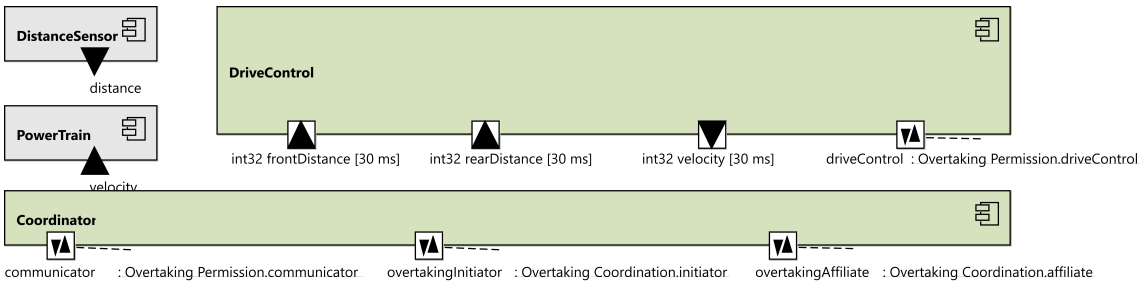


Figure 3.3: The atomic components of the overtaking scenario modeled with the MechatronicUML.

component has a *continuous out-port* called `distance` delivering the distance value which is measured by the sensor. Thus, it models the software component for measuring the distance to another car which is one functional aspect required for the cooperative overtaking. The `PowerTrain` component has a *continuous in-port* allowing it to receive a `velocity` value to affect its behavior, hence, encapsulating the car's driving capabilities as another functional part of the cooperative overtaking scenario.

The behavior of discrete components is formally specified using the MechatronicUML PIM (see Section 3.4.3). The behavior of continuous components however is not specified using the MechatronicUML, but they still expose behavior via continuous ports. These ports specify a specific data type[4] and a name so that discrete components can interact with continuous components to fulfill their behavior. For such an interaction, a discrete component needs a *hybrid port*. Figure 3.3 shows the `DriveControl` component. This component encapsulates the behavior for the autonomous driving of the car. As such, it has a discrete port and also two hybrid in-ports as well as one hybrid out-port. These hybrid ports may be used for interaction with the `DistanceSensor` and `PowerTrain`. The discrete port `driveControl` is used for communication with the `Coordinator` component. The `Coordinator` component is designed to encapsulate the functionality of coordinating an overtaking process with another vehicle and to instruct the `DriveControl` component accordingly.

In addition to atomic components, there are structured components [DPP+16]. Structured components do not contain a behavior directly, but get their behavior by the components they embed. Structured components may embed atomic components as well as structured components. If a structured component only embeds discrete components, then it is called *discrete structured component*. If not, i.e. it embeds discrete and continuous parts, is is called *hybrid structured component*. Figure 3.4 displays the hybrid structured component `RoboCar` which represents the software for the cooperative overtaking scenario. Structured components instantiate the components they embed as *component parts*.

The ports of interacting component instances are connected with each other using assembly connectors, or they are connected to delegation ports that expose specific behavior of the overall structured component using delegation connectors. The `RoboCar` structured component depicted in Figure 3.4 consists of one instance of the `Coordinator` component and one instance of the `DriveControl` component. Their port `driveControl` and `communicator` are connected by an assembly connector and annotated with the Real-Time Coordination Protocol (RTCP) `Overtaking Permission`, which is explained with more detail in Section 3.4.2. The `driveControl` component instance communicates with

[4]For a list of data types see [DPP+16].

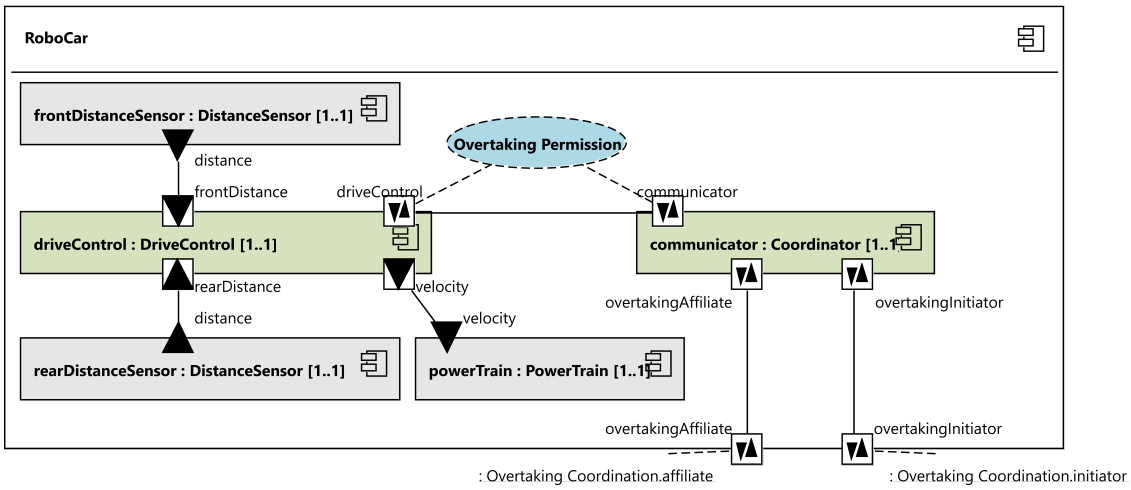


Figure 3.4: The structured component of the overtaking scenario modeled with the MechatronicUML.

the `communicator` instance whether it is allowed to start the overtaking process. The `communicator`'s other ports are connected to delegation ports, exposing the functionality for initiating a cooperative overtaking endeavor (`overtakingInitiator`) or for joining it (`overtakingAffiliate`) to the outside. Furthermore, the `RoboCar` component contains two instances of the `DistanceSensor`, one for the front and the rear of the car each, and another instance of type `PowerTrain`. All these continuous component instances are connected to the `driveControl` instance so that it can fulfill its intended behavior of controlling the car autonomously.

The MechatronicUML PIM component model distinguishes between components and component instances. Structured components embed component instances as *component parts*. A *Component Instance Configuration (CIC)* contains component and connector instances [DPP+16]. A CIC may either be used to express the internals of a structured component as just described. Then it is called *embedded component instance configuration*. Or, a CIC may be used to express a concrete instance of the modeled software. Then it is called a *root component instance configuration*. Figure C.1 in Appendix C.1.1 shows the root CIC of the cooperative overtaking example with the two `RoboCar` instances `fastCar` and `slowCar` to model the overtaker and the affiliate. Similar to the UML, the MechatronicUML PIM's concrete syntax identifies instances with a name separated from the instance's component type by a colon, e.g., `fastCar : RoboCar` [DPP+16]. An embedded component instance is typed over the component part in addition to the component type when the parent component is instantiated, e.g., `driveControl.F / driveControl : DriveControl`. Furthermore, the RTCPs are also instantiated in a CIC diagram, however, they do not have an instance name.

3.4.2 Communication Modeling

The *Real-Time Coordination Protocol (RTCP)* is the MechatronicUML PIM language element to describe the message based interaction between discrete components [DPP+16]. At the core, each RTCP consists of two *roles* and one *role connector*. The roles are the communication participants, and for each role, an RTCP defines the sent and received messages called *sender message types* and

receiver message types as well as a *message buffer*. For the role connector, an RTCP defines Quality of Service (QoS) assumptions. Thus, the RTCP defines the communication contract between these two roles.

Figure 3.5 shows the RTCP `Overtaking Permission` between the roles `driveControl` and the `communicator`. The `driveControl` role has one incoming message buffer of size 5 that discards new messages in case the buffer is full, and receives messages of type `grantPermission` and `denyPermission`. Similarly, the `communicator` role has a buffer for the message types `requestPermission` and `executedOvertaking`. Thus, the sender message types of the `driveControl` role are `requestPermission` to tell the `communicator` that a car in front was detected that could be passed, and `executedOvertaking` to tell the communicator that the overtaking was executed. Conversely, the sender message types of the `communicator` role are `grantPermission` to tell the `driveControl` that it is allowed to start overtaking and `denyPermission` to prohibit an overtaking maneuver. The role connector is assumed to be reliable, i.e., no message loss occurs, preserve the message order and to have a maximum transmission delay of 500ms.

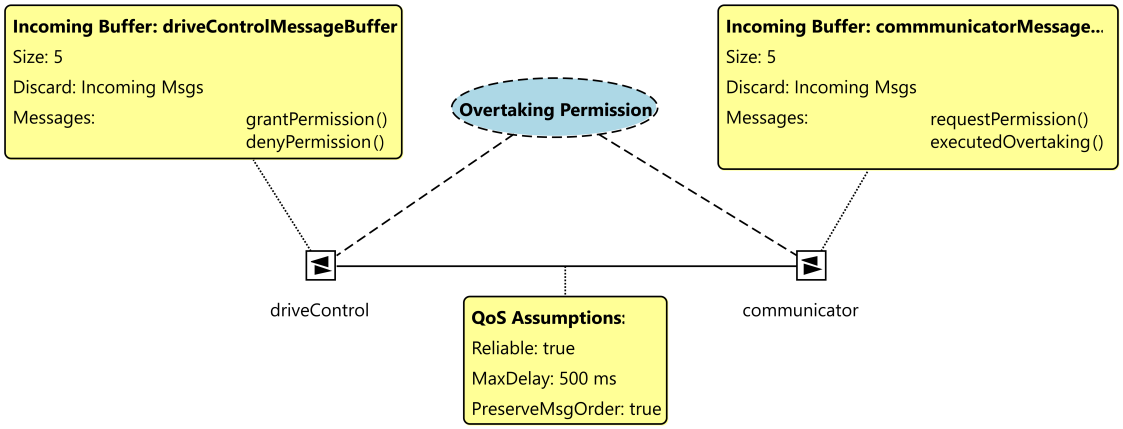


Figure 3.5: The Real-Time Coordination Protocol `Overtaking Permission`.

In the example in Figure 3.5, the roles are both *single in/out-roles*. That means, each role can be instantiated once inside of an RTCP instance, and each role sends and receives messages. There may also be roles that only receive (*in-role*) or send (*out-role*) messages, and that have a cardinality larger than one making them a *multi-role* [DPP+16]. The concrete syntax of these roles is different from the single in/out-roles shown in the example. In this thesis, only single roles are used and hence, the details of multi-roles are not explained here.

If a role receives messages, then it must define one or more message buffers assigning each incoming message type to exactly one message buffer [DPP+16]. These buffers allow the delayed consumption of messages by the respective role instance. It is important to note, that message buffers and messages are completely platform-independent, just like all other MechatronicUML PIM language elements. That means, that the messages and buffers model the communication on application level, but not, e.g., on network level where protocols like Transmission Control Protocol (TCP) would send messages in both directions to set-up the communication, and may use buffers to assemble and disassemble packets of higher layers. Therefore, a role connector models QoS assumptions in order to represent relevant properties of an assumed communication channel in a platform-independent way. So a MechatronicUML PIM message buffer is a logical buffer on application level that has a

fixed size and follows the FIFO (First In, First Out) principle. Whenever a message is available in the buffer, it may be consumed by the respective role instance. Thus, RTCPs enable asynchronous communication.

As the communication is asynchronous and messages are buffered, each role may operate independently of its communication partners [DPP+16]. To specify a reliable protocol for message-based real-time coordination, the behavior of each role must be formally specified. This behavior specification describes when messages are consumed or produced. If two components communicate using an RTCP, the respective component's ports must adhere to the behavior specified by the RTCP's roles. This behavior modeling is explained in the next section.

3.4.3 Behavior Modeling

Next to modeling the structure of a software system and its real-time coordination, the MechatronicUML PIM also supports modeling the behavior of the a CPS. More specifically, it allows to model the behavior of (i) roles of RTCPs, (ii) discrete ports and (iii) discrete atomic components [DPP+16]. The behavior of atomic components is not modeled with the MechatronicUML PIM. For behavior modeling of the above three cases, the MechatronicUML PIM introduces *Real-Time Statecharts (RTSCs)*.

At the core, RTSCs consist of *states* and *transitions* [DPP+16]. States represent a situation within the modeled system, and only one state can be active at a time. Transitions specify the possible state changes, and if no state is active, then a transition is active and will soon activate another state. Figure 3.6 shows a simple RTSC with three states and four transitions. The state `AutonomousDriving` is the *initial state*. Each RTSC must have exactly one initial state, and may optionally have one or several *final states* (not shown or used here). The RTSC in Figure 3.6 has two *variables* `distanceLimit` and `distance`. Variables are storage locations that can be accessed anywhere within the RTSC that defines them. For instance, they can be accessed in the *guard* of a transition as shown in Figure 3.6. Guards are one option to define a *condition* for a transition, i.e., restricting when a transition may be enabled. Conditions for transitions can also be *clock constraints*, *synchronization* (both explained later) or a *trigger message*. The latter is visible e.g., for the transition from `WaitForPermission` to `AutonomousDriving` being activated when a `denyPermission` message is available. If more than one transition can enabled at the same time, the transition *priorities* decide which transition is enabled: the priorities are fixed numbers, higher numbers represent higher priority, and the concrete syntax shows the priority value in a circle at the root point of the transition.

Besides conditions that restrict the activation of a transition, a transition may also have *effects* [DPP+16]. There are three kinds of effects: (i) *action*, (ii) *raise message* and (iii) *clock reset*. Actions are specified using the MechatronicUML PIM action language[5] and their concrete syntax either shows the action's name or its definition using the action language, both in curly brackets. Secondly, a transition may produce a message when it gets activated: a raise message effect. Figure 3.6 shows two raise message effects: `requestPermission()` and `executedOvertaking()`. The example demonstrates the concrete syntax of a raise message effect: the message type is always followed by an (optionally empty) parameter list in round brackets. Thus, the raise message effect can be distinguished from the previously mentioned trigger message guard. Thirdly, the effect of a

[5] The action language is not described here at all. Consult [DPP+16] for details.

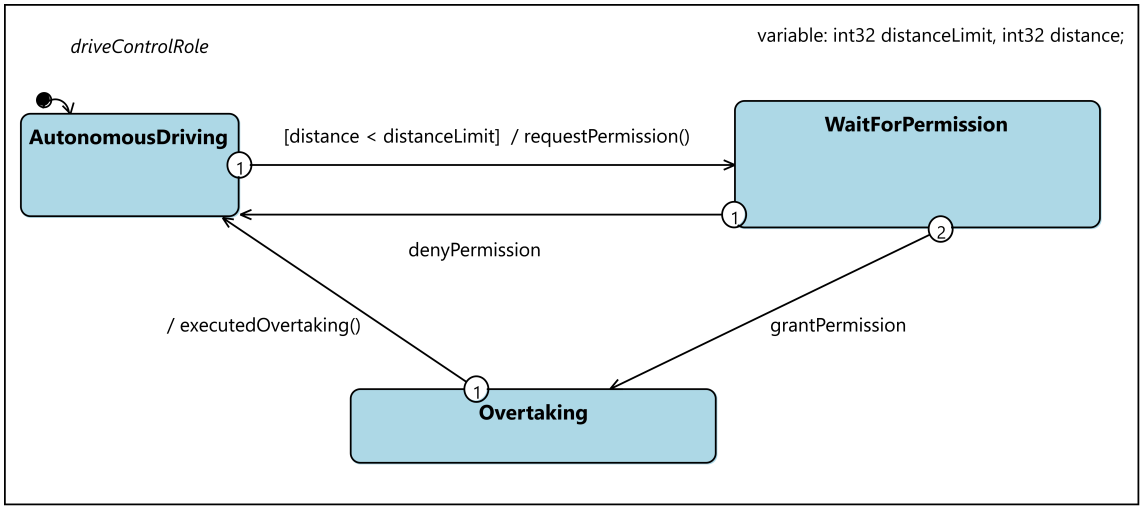


Figure 3.6: The Real-Time State Chart `driveControlRole`.

transition can be a clock reset. An RTSC has a finite number of *clocks* that model the time that is passing during the system execution [DPP+16]. All clocks of an RTSC start with the value zero, and their value is only ever incremented unless they are explicitly reset (e.g., as part of a transition effect or state effect). The clocks are used to model *clock constraints* that can be employed in states and transitions. States may have *invariants* for modeling real-time constraints: Using a set of clock constraints, the time a state may be active is constrained.

The RTSC in Figure 3.6 models the behavior of the `driveControl` role of the `OvertakingPermission` RTCP (see Figure 3.5): The role starts in the state `AutonomousDriving`, and when the `distance` value is smaller than the `distanceLimit`, a transition fires and releases a `requestePermission` message. Then, the `driveControl` role waits for a permission until it either receives a `denyPermission` message and transitions back to the initial state, or it receives a `grantPermission` message transitioning to the `Overtaking` state. From there, the `driveControl` can transition back to the initial state by sending an `extecutedOvertaking` message. This example demonstrates, how RTSCs are used to model the behavior or roles of an RTCP.

Similar to transitions, states may also have effects [DPP+16]. However, within states, effects may only appear for a *state event*. There are (i) *entry*, (ii) *do* and (iii) *exit* events of states. Entry events are triggered when a state is activated, and exits events are triggered on state deactivation respectively. Both event type's effects may employ actions and clock resets. Do events are triggered periodically with a given time interval as long as a state is active, and their effects may only be actions. Figure 3.7 shows the RTSC `DriveControlComponent`. In the `driving` region of the `DriveControl_main` state, the states each have a do event with a time interval of 1ms, each specifying an action effect.

The RTSC in Figure 3.7 uses some more advanced modeling features that have not been explained yet. Most importantly, that is the concept of *composite states* [DPP+16]. The state `DriveControl_main` is such a composite state. A composite state consists of several *regions*, that each contain exactly one RTSC. Those RTSCs are called *embedded RTSC*, as opposed to *root RTSCs*. The name of an embedded RTSC is the name of its region. These regions enable hierarchical RTSCs and also concurrent behavior within one RTSC. In the specific example in Figure 3.7, whenever the state `DriveControl_main` is active, the embedded RTSCs are active as well and may change their state.

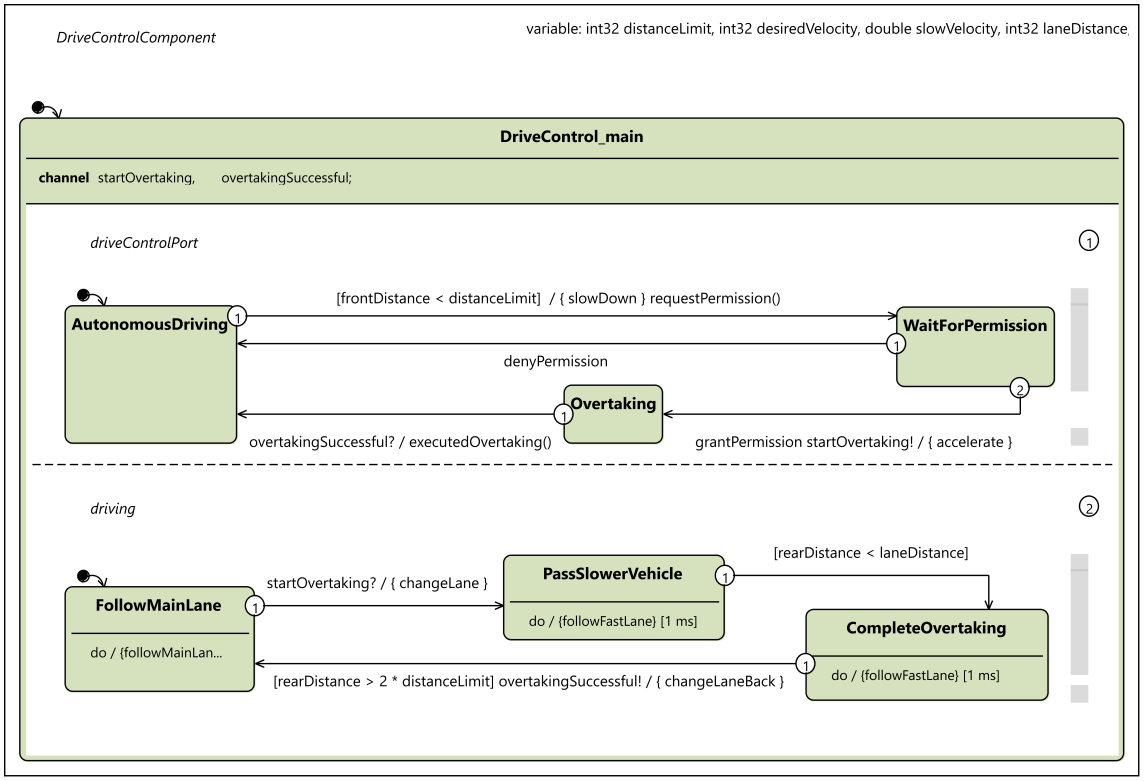


Figure 3.7: The Real-Time State Chart `driveControlComponent`.

The communication of discrete ports is specified using RTCPs, and the roles of an RTCP are defined via an RTSC each. Thus, if a role of an RTSC is assigned to a discrete port, the port must correctly refine the specified role behavior [DPP+16]. In practice, the role's RTSC is often copied to the RTSC defining the port behavior, and then refined with additional constraints or effects. This is also visible for the embedded RTSC `driveControlPort` in Figure 3.7: It specifies the behavior of the discrete port `driveControl` of the discrete atomic component `DriveControl` in Figure 3.3, and refines the `driveControlRole` RTSC given in Figure 3.6. It uses the same three states and four transitions, but adds action effects for slowing down and accelerating the vehicle, as well as adding synchronization with its orthogonal RTSC in the `driving` region.

As different embedded RTSCs are executed independently and concurrently, a means of *synchronization* is introduced by the MechatronicUML PIM to connect the different region's behavior [DPP+16]. The example in Figure 3.7 declares two *synchronization channels* for the `DriveControl_main` state: `startOvertaking` and `overtakingSuccessful`. A synchronization channel can be used to make two transitions of different orthogonal regions occur together in an atomic way. There is always one transition that issues the synchronization, i.e., when it fires it tells another transition to fire as well. This transition is called the sender transition, and its synchronization is expressed with the channel name and an exclamation mark, e.g. `startOvertaking!`. The receiver transition is the synchronization partner, which waits for the synchronization signal on the respective channel, declares the synchronization using the channel name and a question mark, e.g. `startOvertaking?`. A transition can not specify more than one synchronization. Additionally, synchronization channels may specify a data type (e.g., `channel successful[boolean];`) and then the sender and receiver may synchronize only on specific values received via this synchronization

channel (e.g., `successful[false]!` on the sender side, and for the receiver `successful[false]?`). This is called *synchronization with selector* and is also limited to synchronize exactly two transitions, i.e., one transition sending a specific value and one transition waiting to receive it.

Overall, the behavior of a discrete atomic component is also specified using an RTSC which unites the behavior of all the component's discrete ports that it embeds in one region each, and may add another region for its own component behavior [DPP+16]. Figure 3.7 shows exactly such an RTSC that contains the `driveControlPort` behavior as a region next to its own `driving` behavior. The `DriveControlComponent` RTSC has a set of variables, and can also access the variables of its hybrid ports (i.e., `frontDistance`, `rearDistance` and `velocity`). It's behavior can be described as follows: While being in the `AutonomousDriving` state, the vehicle follows the main lane of the road (`FollowMainLane` with a corresponding do event). Once it detects a car in front (`frontDistance < distanceLimit`), it slows down and requests the permission to overtake. Once the `grantPermission` message is consumed by the `driveControlPort` region, the car accelerates again and signals the `driving` region via synchronization that the overtaking may be started. The `driving` RTSC specifies the overtaking behavior, such that the vehicle would change the lane, pass the slower vehicle and once it has passed the affiliate (`rearDistance > 2*distanceLimit`), it would change back to the main lane and upon transitioning back to the initial state, it would indicate to the `driveControlPort` region via synchronization the the overtaking was completed successfully (`overtakingSuccessful!`). Then, the `driveControlPort` RTSC transitions back to the initial state by sending an `executedOvertaking` message indicating completion of the overtaking maneuver.

Thus, the behavior of discrete components, discrete ports and RTCP roles can be modeled using RTSCs. Importantly, the modeling of action effects is not visible in RTSC diagrams but still contains relevant parts of the implementation of the application scenario. These missing parts are explained when the application scenario is implemented as a whole in Chapter 8.

3.5 MechatronicUML Hardware Platform Description Modeling

The MechatronicUML HPDM modeling language supports modeling the hardware platform for CPS [DP]. This comprises the modeling of structured hardware platforms consisting of hardware resource instances. The exemplary hardware platform that is introduced in Section 2.2 is used as running example to demonstrate the modeling language elements of the MechatronicUML HPDMin this section. The description of the modeling language features focuses on the feature required to model the application scenario. For a complete description of all features, consult [DP].

Overall, the MechatronicUML HPDM distinguishes between *resource types*, *resource instances*, *platform types* and *platform instances* [DP]. The MechatronicUML HPDM defines a perspective for each of these modeling language features. Moreover, each of these language features builds upon its predecessor: First, *hardware resource types* model the hardware resources and variation points. Secondly, these resource types are instantiated and their variation points are fixed. Next, a *hardware platform type* is described with resource instances and also recursively with other hardware platform types that it may be composed of. Lastly, the platform type is instantiated to model a run-time view of the hardware system, including a fixed number of resource instances. The platform instance is specified in a *Hardware Platform Instance Configuration (HPIC)*.

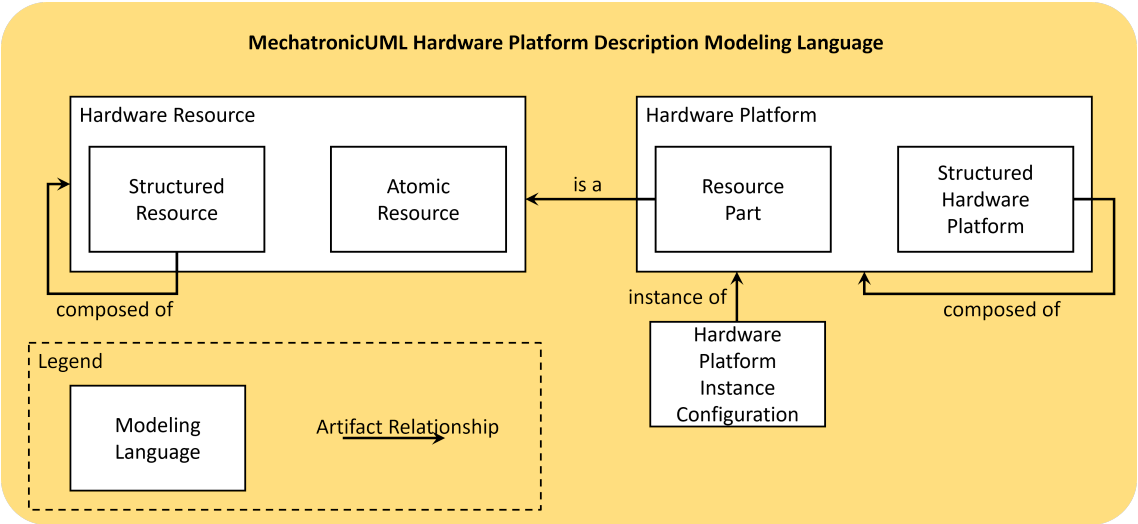


Figure 3.8: The MechatronicUML HPDM language parts and their relationships based on [DP].

Figure 3.8 shows the parts of the MechatronicUML HPDM language and their relationships. The modeling of hardware resources consists of atomic and structured resources which are described in Section 3.5.1. The hardware platform modeling uses hardware resources as resource parts and models structured hardware platforms which are also instantiated. This is explained in Section 3.5.2.

3.5.1 Hardware Resource Modeling

"The starting point for modeling the hardware platform is typically the description of the hardware platform and the hardware resources, e.g., ECUs, in form of product data sheets." [DP] The description of the exemplary hardware platform can be found in Section 2.2 and Appendix A including a list of the hardware resources. The specific resources' data sheets are available online. The important data sheets are the ones of the *Arduino Mega 2560 Rev3*[6] and the *Arduino Nano*[7] microcontrollers.

The smallest entity of the MechatronicUML HPDM are *atomic resources* [DP]. Among atomic resource, the MechatronicUML HPDM distinguishes between *memory resources* and *computing resources*. Memory resources represent physical resources able to store data using a particular technology, e.g., Random Access Memory (RAM), Read-only Memory (ROM), flash or hard disks [DP]. Memory resources which use these technologies can be modeled using the concrete syntax of atomic memory resources visible in Figure C.6. Figure 3.9 shows the memory resource `Flash`. Memory resources are further described by an *access policy* and their *volatility*. The depicted memory resource `Flash` supports read and write access and is not volatile.

Computing resources are the second kind of atomic resources in the MechatronicUML HPDM resource modeling [DP]. Computing resources can execute program code, and can be either a Programmable Logic Controller (PLC) or a processor. Figure 3.10 shows the concrete syntax for

[6] https://docs.arduino.cc/hardware/mega-2560
[7] https://docs.arduino.cc/hardware/nano

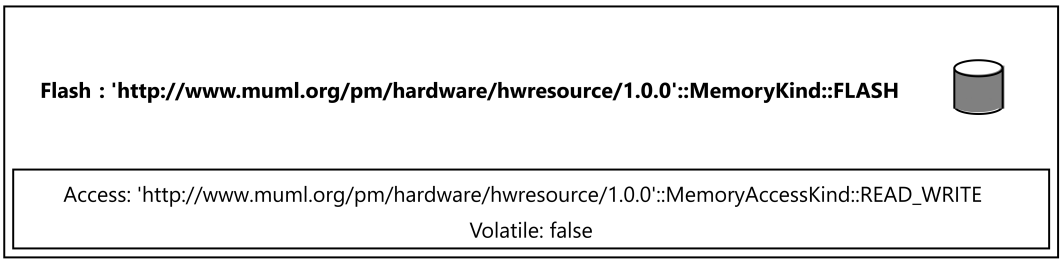


Figure 3.9: A memory resource of kind `FLASH` named `Flash`.

the processor `ATmega`, specifying that it has one core, is based on a RISC architecture and belongs to the ATmega processor family. Atomic resources do not exist on their own, but are embedded into structured resources [DP]. Figure C.6 in Appendix C.2.1 shows the structured resource `Arduino` that embeds the previously mentioned atomic resources.

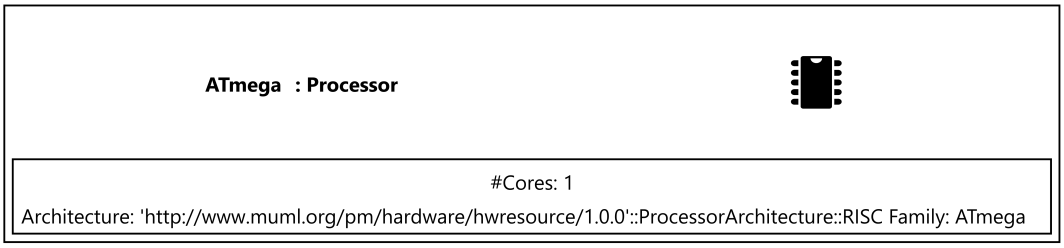


Figure 3.10: A computing resource of kind `Processor` named `ATmega`.

Besides structured and atomic resources, the MechatronicUML HPDM supports the modeling of *devices* [DP]. In contrast to general-purpose resources, devices represent a certain functionality wrapped in a hardware component. A device is viewed as a black box; the internals are not modeled. Based on their functionality, the MechatronicUML HPDM categorizes devices as sensors and actuators. The device types `UltrasonicDistanceSensor` and the actuator `DCMotor` are not depicted as resources here, but in the resource instance diagram in Figure 3.11. These devices represent the ultrasonic distance sensors and the simple DC motors from the target platform (see Section 2.2). Their specification in a resource diagram, i.e., without instantiation, is depicted in Figure C.7 in Appendix C.2.1.

Modeling the target platform for a distributed CPS requires means to model network connections between hardware resources. The MechatronicUML HPDM provides *hardware ports* for this purpose. Hardware ports are points of interaction of a hardware resource with external entities. This is applicable for structured components and devices. They can be connected to other resources via their ports. On the top level, there are two types of hardware ports: *bus ports* and *link ports*. Link ports represent point-to-point connections, e.g., Ethernet, whereas bus ports model the connection to networks with bus topology, e.g., I2C. Each hardware port must define the protocol it uses; protocols are either *bus protocols* or *link protocols* correspondingly. Additionally, hardware ports have a cardinality to define how many instances of a particular port a resource may have. Figure 3.11 shows several hardware ports, e.g., the `inputSignal` port which uses the `MotorControl` communication protocol or the `I2CPins` which use the bus protocol `I2C`.

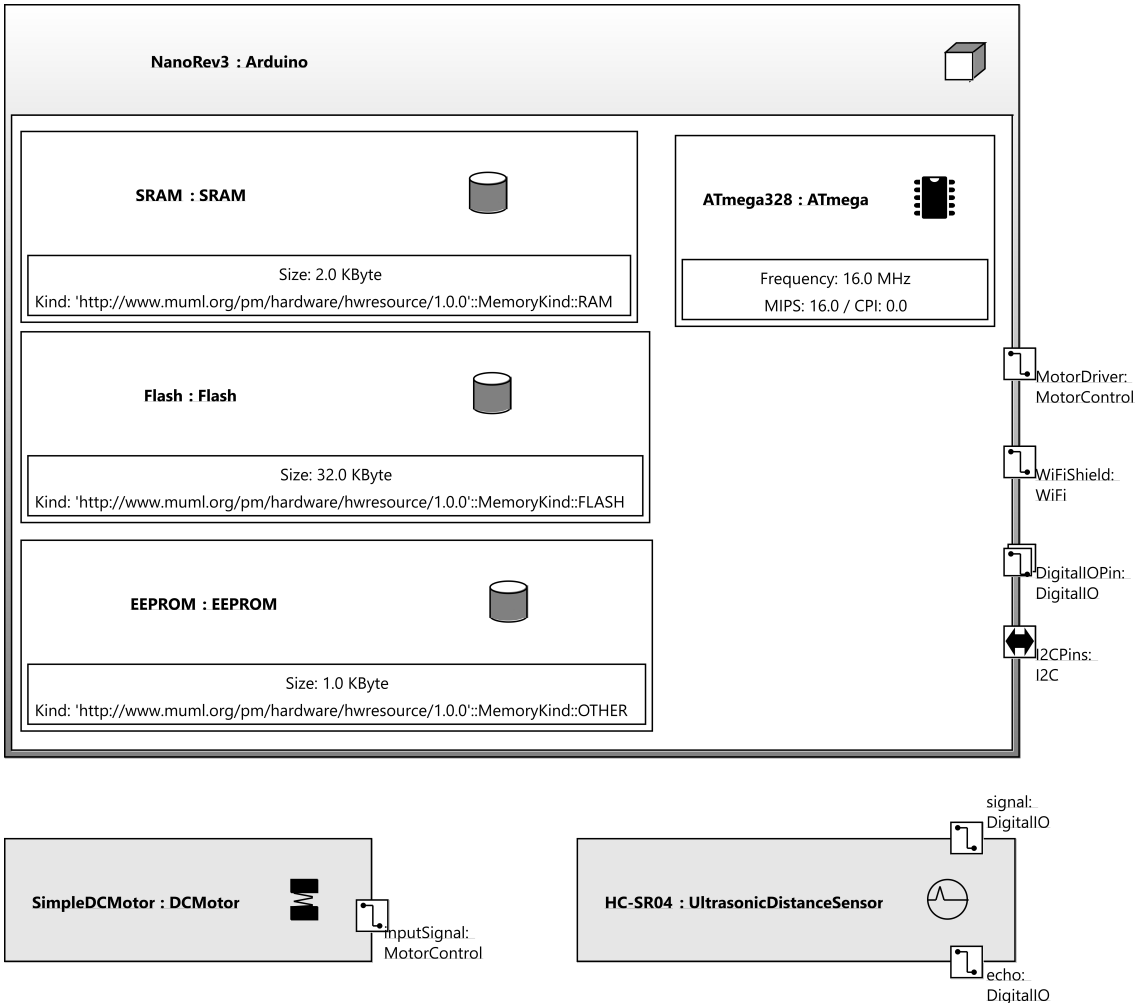


Figure 3.11: The resource instances `NanoRev3`, `SimpleDCMotor` and `HC-SR04`

The overall resource diagram that contains the aforementioned resources is shown in Figure C.6 in Appendix C.2.1. In addition to the resources, it defines four communication protocols. These are the link protocols `DigitalIO`, `MotorControl` and `WiFi` and the bus protocol `I2C`.

In the final step of the resource modeling, the resource types are instantiated adding instance-specific information such as the processor frequency or the memory size [DP]. The resource instances diagram in Figure 3.11 shows the `NanoRev3` Arduino microcontroller with the respective properties retrieved from its data sheet. Similarly, the devices are instantiated as `SimpleDCMotor` and `HC-SR04`; but as they are treated as black box, no internals are specified on instance level.

3.5.2 Hardware Platform Modeling

In order to model a hardware platform composed of different resource instances, the MECHATRONIC-UML HPDM introduces *platform types* [DP]. A platform type consists of *platform parts* that it embeds and that define a platform type's inner structure. Using platform parts, a platform type can

either embed resource instances or other platform types. However, it cannot embed itself (i.e., its own resource type) as a platform part. Regarding resource instances, only the instances of structured resources or devices can be embedded as platform parts; atomic components are excluded.

Furthermore, the embedded platform parts can be connected via their hardware ports by *communication media* [DP]. Like for ports and protocols, the MechatronicUML HPDM distinguishes bus and link connections. Consequently, only hardware ports that make use of the same communication protocol may be connected. Figure 3.12 shows the platform type `DriveControlUnit` that represents the Arduino Mega ECU from the running example together with its devices. The device instances are connected to the Arduino microcontroller via their hardware ports: all of them use link protocols. To keep the diagram a little smaller, the `SimpleDCMotor` is only modeled twice instead of four times, as that still reflects the structure and concept of the robot car target platform.

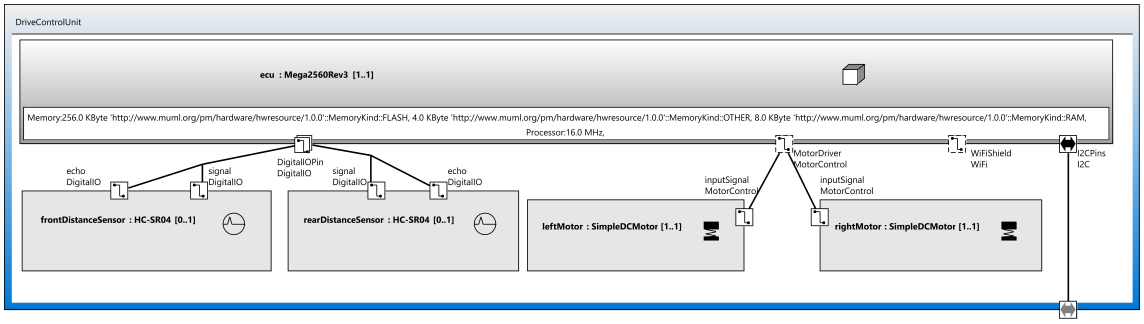


Figure 3.12: The `DriveControlUnit` platform type.

In addition, the platform diagram of the `RoboCar` in Figure 8.4 shows an instance of the `DriveControlUnit` platform type embedded as a platform part. This diagram also visualizes the concrete syntax for bus connections, in this example an I2C bus that both platform parts connect to for communication. Lastly, both example diagrams also show the delegation of hardware ports in hardware platforms: Equivalently to hardware ports, *delegation ports* are either bus ports or link ports [DP]. They expose a hardware port to the outside of a platform type and are connected to the respective hardware port using *delegation connectors*. Figure 8.4 specifically shows the usage of the `DriveControlUnit` type's `I2C` delegation port that is depicted here in the respective platform type diagram in Figure 3.12: The delegation port delegates to the `I2CPins` of the `ecu` resource instance.

Lastly, the MechatronicUML HPDM allows to model a concrete platform as a platform instance [DP]. A *Hardware Platform Instance Configuration (HPIC)* specifies the number and name of hardware instances, and may also add additional communication media between delegation ports of these instances in order to model the concrete hardware and network topology. Such an HPIC represents a concrete target platform instance and may be used as a foundation to model a system's allocation. An HPIC diagram is depicted for modeling the application scenario in Figure C.9 in Appendix C.2.2.

3.6 MATLAB/Simulink and Stateflow

Finally, the commercial tool MATLAB as well as its extensions Simulink and Stateflow are foundational for this thesis. They play a key role in previous model transformation approaches using the MechatronicUML, and thus some foundations help to understand one of the related approaches that is introduced and analyzed in this work.

MATLAB is a software package for numeric calculations with special strengths regarding vector and matrix calculations as well as visualizations [ABRW20]. It is developed and licensed by the Mathworks, Inc. MATLAB comes along with a tool set commonly referred to as MATLAB editor or MATLAB-Desktop, which provides access to the MATLAB programming language and its basic tools. Additionally, there are many so-called *tool boxes* that may be installed to add specific functionality to the MATLAB environment.

Simulink is such a toolbox, and it enables graphical modeling as well as simulation of dynamic systems [ABRW20]. It comes along with a large library of *function blocks* for linear, non-linear, discrete or hybrid systems and thus relieves developers of manually programming such complex functions. Simulink allows the modeling of *signal flow plans* by connecting these ready-to-use function blocks. These blocks have a graphical representation in the Simulink model editor which shows the blocks inputs (one ore several) and outputs (one or several) as well as optionally the block's name. Depending on the function that the block represents there may be additional parameters that can be edited. Furthermore, multiple blocks can also be combined to build so-called *subsystems*, and the blocks can be connected (all connections are directional). In addition to function blocks, there are *signal blocks* and *sink blocks*, which create or consume signals. These are the major elements to model signal flow plans with Simulink, as well as creating own block libraries. Lastly, there are also *inport blocks* and *outport blocks* which represent inputs and outputs of entire Simulink models and may be used to connect those.

Stateflow is an extension of Simulink tailored for modeling finite state machines [ABRW20]. While the state of a block in Simulink is represented by the value of continuous state variables, the states in Stateflow can only be active or inactive. Stateflow models are triggered by *events*, representing event-discrete behavior, while Simulink models are calculated using integration steps that are independent of any event, hence representing time-continuous behavior. A Stateflow model is called *chart* and must be embedded into Simulink models as subsystem. A Stateflow chart basically consists of *states* and *transitions* that connect these states. Simulink also comes along with an *action language* to specify conditions for transitions as well as actions that are executed when entering, leaving or residing in a certain state. Finally, by using *embedded functions* in Stateflow charts, the user is also able to access the MATLAB environment [HRB+14]. The MATLAB programming language allows to express complex functionality beyond the features of the action language. For a detailed introduction to MATLAB, Simulink and Stateflow consult, e.g., [ABRW20].

4 Related Work

The contributions of this thesis are a taxonomy for model-based code generation (see Chapter 5) and a concept and implementation for a MechatronicUML-based code generator for distributed deployments (see Chapter 6 and Chapter 7). Consequently, this chapter also comprises related work in the area of code generation taxonomies in Section 4.1 and related code generation approaches in Section 4.2.

4.1 Code Generation Taxonomies

The related work regarding the classification of code generation approaches was searched by the use of the following search terms and combinations of contained keywords:

- model-based code generation classification
- code generator taxonomy
- code generator classification
- taxonomy for model to code transformation
- model transformation taxonomy
- evaluating model transformation approaches

The search was performed using the search engine Google Scholar Germany[1], and backtracking of references was applied to find more results. To the best of the author's knowledge, there is no existing taxonomy for model-based code generation approaches as defined in Section 3.1. However, there are a few taxonomies or classifications from the domain of MDSE focusing on model transformations and metaprogramming which treat related aspects.

Firstly, Mens et al. propose a taxonomy of model transformation with the goal of assisting developers of model transformations in choosing the most suitable transformation language and tool for their needs [MCG05]. In order to do so, they raise four questions to be answered:

- "What needs to be transformed into what?
- What are the important characteristics of a model transformation?
- What are the success criteria for a transformation language or tool?
- Which mechanisms can be used for model transformations?"[MCG05]

[1] https://scholar.google.de

Mens and Van Gorp later add another dimension to their taxonomy called “Quality requirements for a transformation language or tool”[MV06]. Together with the four previous questions, this fifth dimension increases their focus on the tools for implementing model transformations.

In contrast to this approach of characterizing model transformations regarding different dimensions, Czarnecki and Helsen develop and refine a hierarchical feature model with the goal of describing possible design choices for model transformation approaches [CH03; CH06]. The authors focus on the transformation approach itself rather than tooling and implementation aspects. According to Czarnecki and Helsen, the top-level features of a model transformation approach are the *specification*, *transformation rules*, r*ule application control*, *rule organization*, *source-target relationship*, *incrementality*, *directionality* and *tracing* [CH06].

Moreover, besides these characteristic features that serve to classify model transformations, Czarnecki and Helsen further distinguish between to main kinds of model transformation approaches: model-to-model and model-to-text transformations. This categorization is well suited to characterize tools, and is therefore used in Section 3.2 to describe the foundations of model transformations.

Similarly, Bajovs et al. describe a code generator taxonomy building up on the work by Czarnecki and Helsen in [CH06], focusing on just model-to-text transformation [BNS13]. They apply their taxonomy to code generation tools such as IBM Rational Software Architect or Sparx Enterprise Architect; so by the term *code generator* they refer to what was defined as model-to-text transformation tool in this work (cf. Section 3.2.2). They do not, however, target code generators for a specific context, platform or DSL in whose context the tool might be used to implement a model-based code generator.

The aforementioned taxonomies categorize model transformations focusing either on the tooling or the conceptual transformation approach. Model transformations are a core part of model-based code generation. Therefore, some aspects of these categories inspire and influence the taxonomy introduced in Chapter 5. However, additional aspects such as the specification of the target platform, the implementation of the code generator or properties of the generated code cannot be described by means of these taxonomies.

Looking at the related domain of metaprogramming, Damaševičius and Štuikys introduce a *taxonomy of metaprogramming concepts* [DŠ08]. The authors describe metaprogramming as writing programs that produce, read or change other programs – and producing programs is the essence of code generation. The concepts of metaprogramming are important in the domain of program processors, interpreters and compilers. But these concepts are also applicable to higher levels of abstraction such as model transformations and model-based code generation. Damaševičius and Štuikys define code generation as “the process by which a code generator converts a syntactically-correct high-level program into a series of lower-level instructions”[DŠ08]. The proposed taxonomy describes concepts like *levels of abstraction* and *annotation of metadata* as well as *transformation* and *generation*. The tools which are used to implement model-based code generators may be based upon these concepts. However, the taxonomy of metaprogramming concepts does not have the breadth required to describe model-based code generation approaches as a whole.

Lastly, the taxonomy of Kahani and Cordy is closest to the taxonomy this thesis defines in Chapter 5. The authors firstly classify tools in the two main categories of model-to-model and model-to-text transformation, and additionally, they develop a classification scheme for the comparison of code

generation tools. With their first categorization of tools, they follow the approach of Czarnecki and Helsen in [CH06] very closely. However, with the second classification scheme targeting to compare tools from different perspectives, they introduce some additional ideas.

For the comparison of tools, they define six categories with a number of aspects for comparison in each category. For instance, the *general category* includes the aspect *operating system* describing on which client operating system the tool can be run and used [KC15]. Then, for each of these aspects, Kahani and Cordy define a number of attributes that can be assigned to a specific tool, e.g., "a: The tool has been run on Windows [...] b: The tool has been run on Linux/Unix [...]"[KC15]. These are the categories and some representative aspects:

General Category This category sums up aspects regarding the tool usage in general, such as the operating system and technological platform (e.g., a certain runtime environment) required to operate the tool, the tool licensing, resources like documentation or user guides, or whether the tool is a stand-alone solution or embedded in some other tool or Integrated Development Environment (IDE).

Model-level Category Aspects of the model-level category are the modeling language the tool uses, whether it provides a graphical or textual concrete syntax, how it implements the abstract syntax, the levels of abstraction and the MDA model levels it supports.

Transformation Category This category is all about the transformations the tool supports. It includes the levels of abstraction between which transformations are possible, the direction, the scope and the cardinality. But also the scheduling of transformation rules as well as the control over the rule application are regarded. This category is very similar to the previously introduced feature model for transformation approaches (cf. [CH06]).

Capability Category The capability category refers to the engineering capabilities that the tool provides, such as verification and validation, which kinds of model manipulations the tool supports, or if reverse or round-trip engineering are possible.

Implementation Category Anything related to the tool's usability and feature maturity falls into the implementation category. Regarded aspects are the editor, if it's a graphical or textual editor, to options provided for workspace and project management, or whether the syntax and semantics are automatically checked and highlighted at edit time. Notably, even though the name of the category suggests it, the actual implementation regarding code base and software structure is not regarded here.

Quality Category The aspects evaluated in the quality category include, e.g., the tool's maturity and its maintenance support, but also whether the tool is capable of executing several transformations concurrently, whether it supports traceability between source and target model as well as the security features regarding limiting access rights to models via the tool's user management functionality.

This taxonomy targets model transformation tools specifically. Model-based code generation approaches comprise more aspects, and the transformation tools are only a part of them. Nonetheless, the categories are generally applicable to model-based code generation as well. However, there are a lot of similar aspects in a number of categories (e.g., between the general and the quality category) that could be deemed to be duplicates. Moreover, aspects regarding the actual implementation are

not regarded in the taxonomy at all. Finally, the model-level category is also obsolete for this work, because this thesis assesses code generation approaches regarding one common source modeling language, the MechatronicUML, specifically.

4.2 Code Generation Approaches

The second contribution of this thesis is the design and implementation of a code generator that is able to produce software from a MechatronicUML PIM instance. More specifically, the aim is to generate code for a distributed deployment of a CPS. Accordingly, related work in the area of model-based code generation focusing on MechatronicUML is described in this section. The related code generation approaches were mainly found starting at the website of the MechatronicUML project[2]. In addition, the academic search engine Google Scholar Germany[3] was used as well to find additional literature and related approaches. Following search terms were used, as well as some combinations of keywords contained in these search terms:

- mechatronicuml
- generating code for cyber physical systems
- code generation from mechatronicuml
- model-driven cyber physical systems
- generating code from software models
- model to text transformation
- generating software for mechatronic systems
- model driven software construction
- model driven software generation
- model driven code generation
- model code generator

Additionally, backtracking of references was used to discover additional literature. The following subsections introduce two approaches that support code generation for distributed deployments using MechatronicUML models as input. These existing approaches are introduced in Section 4.2.1 and Section 4.2.2. As they are closely related to the aim of this thesis, they are introduced as related work in this section. However, they are also analyzed in Section 6.2 and Section 6.3 to explore whether they can be reused or extended for the aims of this thesis, instead of creating a new code generation approach from scratch. Lastly, Section 4.2.3 describes a few related approaches that do not use the MechatronicUML, but different modeling languages in the domain of distributed CPS.

[2] http://www.mechatronicuml.org
[3] https://scholar.google.de

4.2.1 The Platform-Modeling Approach

The first related approach to be introduced is the *platform-modeling approach*. It originates from the project group Cybertron [BCD+14; Hei14a] at the University of Paderborn, and was later continued and extended by Pohlmann [Poh18]. Therefore, all joint effort leading to the latest state of this code generation approach is summed up with the term *platform-modeling approach* within this thesis.

The characteristic feature of the platform-modeling approach is that it makes use of the MechatronicUML HPDM language (see Section 3.5). In this approach, a platform model is combined with a MechatronicUML PIM instance to derive a PSM and finally generate code [BCD+14]. The PSM specifically allows modeling a distributed deployment based on a distributed hardware platform that is modeled with the MechatronicUML HPDM. Figure 4.1 visualizes the code generation process by means of the BPMN. The three major tasks are the *platform modeling* using the MechatronicUML HPDM (T1), the *allocation engineering* (T2) and the *software construction* (T3). The platform-modeling approach is implemented in the MechatronicUML Tool Suite.[4] In summary, the input of the code generation process is a MechatronicUML PIM instance, and the output is a source code folder as well as corresponding executable software. These artifacts are also visible in Figure 4.1, where, for reasons of simplicity, the data flow is only depicted when it deviates from the process flow.

The first task *T1: Platform Modeling* consists of two subtasks: First, the hardware resources are modeled (T1.1) and then these resources are used to model the hardware platform (T1.2). These tasks essentially represent the MechatronicUML HPDM process (cf. Section 3.5). Both tasks have to be performed by the user, but they are directly supported by the MechatronicUML Tool Suite. The resulting MechatronicUML Platform Model is then used by the consecutive task.

The goal of the second task, *T2: Allocation Engineering*, is specifying the allocation of the software components to the hardware platform. [Poh18] introduces the concepts and process for allocation engineering separately from the software construction [Poh18]. Together with *T3: Software Construction*, it implements the fourth step of the overall MechatronicUML development process: designing platform-specific software (cf. Figure 3.2). The first step of the allocation engineering is the specification of software resource requirements (T2.1)[5]. For this purpose, there are four extensions to the MechatronicUML PIM metamodel, that allow to annotate a CIC with specific resource requirements: for component instances, the required memory, Worst Case Execution Time (WCET) and scheduling information can be supplied, and for port instances there is an extension to annotate the message frame size. The second subtask serves to specify constraints for the allocation of software components to resources instances of the hardware platform (T2.2). The idea is to limit the complexity of finding a suitable allocation plan to specifying allocation constraints. These constraints may be specified by means of an *allocation constraint specification language*. Then, this

[4] The MechatronicUML Tool Suite contains several plugins for C code generation. These plugins support different target platforms out-of-the-box, but they do not support platform modeling or the specification of a distributed deployment and are thus not regarded here. They produce source code that can mainly be used for the simulation of the platform-independent model without regarding a specific deployment scenario. Here, the focus is on the software construction for distributed deployments, which is also based on a different C code generator embedded in the MechatronicUML Tool Suite.

[5] This step is technically optional with regards to code generation.

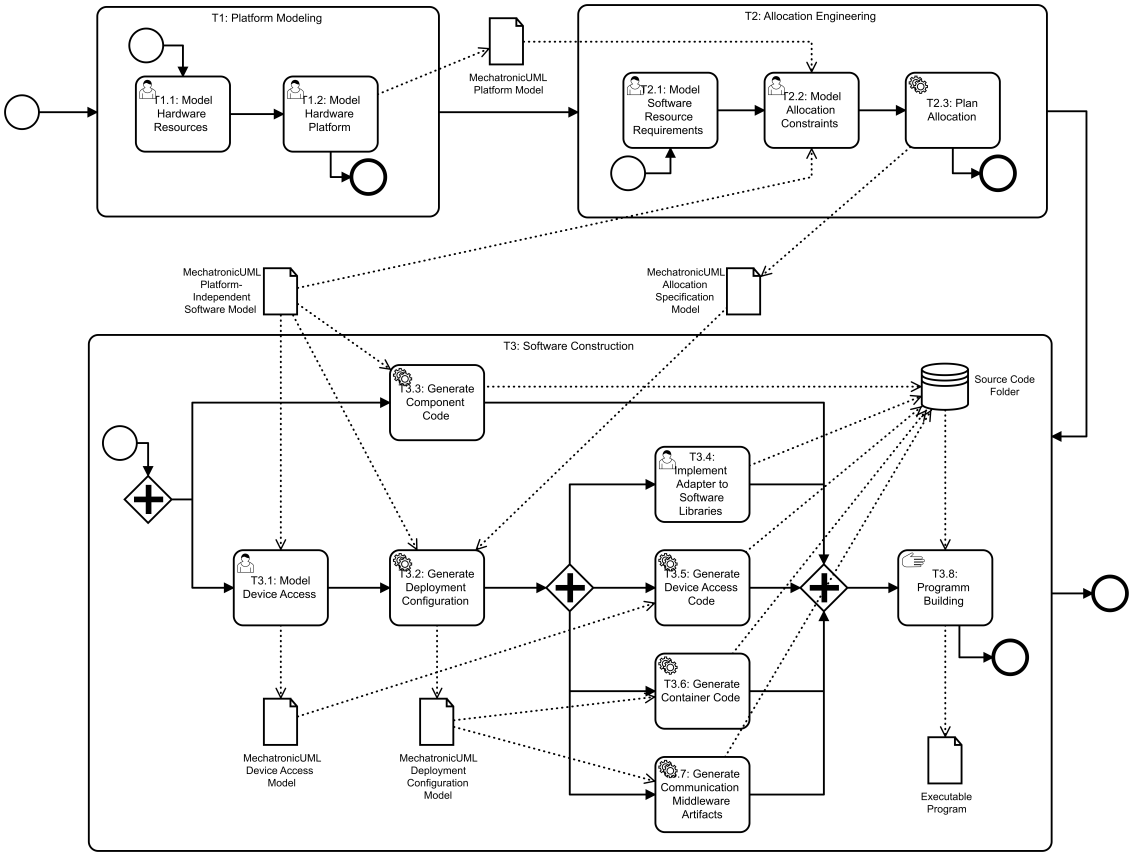


Figure 4.1: The code generation process of the platform-modeling approach based on Pohlmann [Poh18].

specification is used to to automatically calculate an optimal allocation plan (T2.3). The allocation specification language as well as a corresponding solver for the allocation planning are included in the MechatronicUML Tool Suite.

Finally, the last but also largest task of the code generation process is *T3: Software Construction*. Here, the MechatronicUML PIM instance and the previously created MechatronicUML Allocation Specification Model instance are used by several process steps to eventually generate source code. These two input artifacts are required to generate the functional code as well as the structural code and, most notably, the communication between software components that are deployed in a distributed manner.

Overall, the software construction includes the platform-specific modeling, the platform-specific implementation and the platform-independent implementation. From a metamodel perspective, the MechatronicUML Deployment Configuration metamodel fills the role of a PSM. Its most important classes are shown in Figure 4.2. From a conceptual perspective, a `DeploymentConfiguration` is associated to one `SystemAllocation`. A `DeploymentConfiguration` consists of a set of `ecuConfigurations` that are each associated to a `StructuredResourceInstance` of the MechatronicUML HPDM. And each `ECUConfiguration` is equipped with a set of `containers` which are tailored to one specific component type of the MechatronicUML PIM. The containers represent all the platform-specific configuration

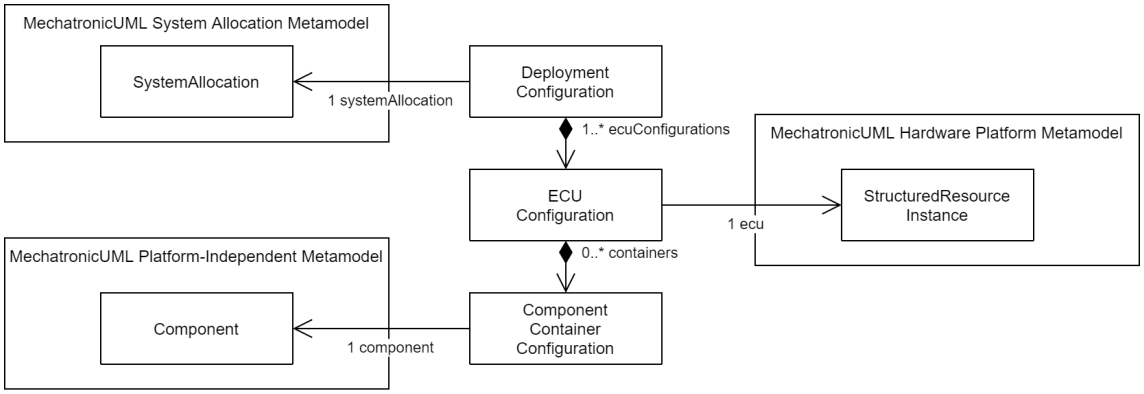


Figure 4.2: The core of the MechatronicUML Deployment Configuration metamodel based on Pohlmann [Poh18].

that is required for the execution of a platform-independent component. Then, this configuration is used to generate the corresponding source code. The following subtasks are performed for the platform-specific modeling and the code generation:

T3.1: Model Device Access This optional task allows the user to specify the API of a concrete device in use [Poh18]. Such a device may be used to realize the behavior of a continuous component in the MechatronicUML PIM. The device API is specified using the so called API Modeling Language (ApiML) [Poh18] and then, the continuous port instances are mapped to these API calls using the API Mapping Modeling Language (ApiMappingML) [Poh18]. This task is part of the platform-specific modeling.

T3.2: Generate Deployment Configuration This task encapsulates the required work to combine the MechatronicUML PIM instance with the MechatronicUML Allocation Specification Model instance: the MechatronicUML Deployment Configuration Model contains the configuration information required for each component instance of the MechatronicUML PIM for its allocated computing resource. This model transformation is automated and is the core of the platform-specific modeling.

T3.3: Generate Component Code The component code is the source code implementing the platform-independent, functional behavior of the software. It is generated by a fully automated process step without further user interaction using the MechatronicUML Platform-Independent Software Model as input.

T3.4: Implement Adapter to Software Libraries In order to reuse existing software libraries and they are made available as operations to be called via the MechatronicUML Action Language. Users have to provide actual implementations of these libraries manually, but stubs for the implementation are provided by the component code generation.

T3.5: Generate Device Access Code This is a fully automated task that uses the previously created MechatronicUML Device Access Model to generate the source code for accessing the device. Hence, it is part of the platform-specific implementation. If no MechatronicUML Device Model is created (as T3.1 is optional), then the mappings may be implemented manually; respective methods stubs are generated as part of the container code generation.

T3.6: Generate Container Code Another fully automated task uses a MechatronicUML Deployment Configuration Model to create the source code for the container environment. These containers are the core of the platform-specific implementation, allowing the platform-independent component code to be executed.

T3.7: Generate Communication Middleware Artifacts This final code generation task also uses the MechatronicUML Deployment Configuration Model and generates the source code for communication between component instances via their port instances. Additionally, for distributed deployments, it generates some of the source code artifacts for the middleware implementation using a commercial code generator of the company RTI [Poh18]. For this purpose, T3.2 also generates an Data Distribution Service (DDS) metamodel instance that contains the configuration for using DDS to implement the communication middleware. This model supports the export of an interface description file that the RTI code generator uses as input for its C code generation. The remainder of the communication middleware code is generated together with the container code.

T3.8: Program Building Finally, the building of the entire program has to be performed in order to obtain executable software. This task is however not included into the MechatronicUML Tool Suite, but can be executed using a *makefile* for compilation and linking [Poh18], which is generated together with the source code.

The users of this code generation process are hardware experts, allocation engineers and software engineers that have the required expert knowledge for the individual tasks [Poh18]. They are supported by different perspectives in the MechatronicUML Tool Suite for performing these tasks (besides T3.4 and T3.8). These perspectives that are available in the MechatronicUML Tool Suite implement the described code generation process. There is no wizard for the overall process, but the user has to perform or initiate the individual steps via the MechatronicUML Tool Suite. As mentioned, not all steps are mandatory for the code generation. Pohlmann describes the state of implementation of the MechatronicUML Tool Suite plugins as “prototypical”[Poh18]. The author mentions limited automated testing, but does not elaborate on the implementations’ maturity otherwise. Nonetheless, the features are included in the “latest version of the MechatronicUML Tool Suite”[Poh18], which corresponds to version 1.0 at the time of their writing (cf. Section 3.3).

All in all, the platform-modeling approach introduces a sophisticated process for software generation for distributed CPS, starting at the MechatronicUML PIM level, planning an allocation and narrowing it down to a platform-specific model and implementation by using existing and also adding new MechatronicUML modeling features. This approach allows the users, who are experts from different areas, to specify their platform entirely and with all details required for code generation. Moreover, the code generator generates platform-independent as well as platform-specific code based on concepts that may be used for additional programming languages and communication technologies. In the prototypical implementation of the code generator, the generated component code and container code uses the C programming language, and the MechatronicUML Deployment Configuration metamodel configures the ports’ communication for the DDS [Poh18]. The communication middleware for DDS is also generated for the C programming language, matching the C container code.

4.2.2 The MATLAB/Simulink Approach

The second related approach does not target code generation in the first place. Instead, the researchers aim to leverage the strengths of the MechatronicUML by transforming MechatronicUML PIM instances to MATLAB/Simulink models [HRB+14]. The system's "discrete behavior is developed using MechatronicUML and then translated to MATLAB/Simulink and Stateflow, where it is combined with the continuous behavior specification" [HRB+14] which cannot be modeled using the MechatronicUML PIM. This allows to use the simulation features of MATLAB/Simulink for software that is modeled with the MechatronicUML, but also to use the MATLAB features for code generation. Therefore, this subsection describes the steps necessary to generate code from the MechatronicUML with MATLAB/Simulink as an intermediate modeling layer, and this approach is called the *the MATLAB/Simulink approach* within this thesis.

The code generation process of the MATLAB/Simulink approach is displayed in Figure 4.3. This process consists of the transformation from the MechatronicUML to MATLAB/Simulink that Heinzemann et al. describe [HRB+14]. This transformation is visualized as task T1. The process also consists of the steps required to generate code from a MATLAB/Simulink model instance [Mat22d], which is visualized as task T2. Heinzemann later extend the first process step, which is referred to as the *MechatronicUML-to-MATLAB/Simulink transformation*, to support reconfigurable components [Hei15]. Additionally, the transformation was also reimplemented due to lack of performance and due to maintenance difficulties: "The transformation [...] was hard to debug and slow." [Hei21]. The new version was a complete reimplementation which was better to maintain [Hei21]. However, there is no documentation of the newer transformation approach. Nonetheless, "both transformations basically follow the same structure" [Hei21]. Therefore, this subsection is based on the initial documentation (see [Hei15; HRB+14]) that reflects the important concepts, and the concepts are the main focus of this subsection.

The subtasks *T1.2: Model-to-Model Transformation* and *T1.5: Model-to-Text Transformation* are the major steps of the MechatronicUML-to-MATLAB/Simulink transformation [HRB+14]. The model-to-model transformation does the logical work to map MechatronicUML PIM features to corresponding MATLAB/Simulink and Stateflow concepts. The model-to-text transformation step afterwards generates the actual target model in a text format defined by MATLAB/Simulink. The other three steps enable these transformations. In more detail, the process is as follows:

T1.1: Analysis The analysis is a preprocessing step for the transformation [HRB+14]. It consists of two parts: The first part of the analysis is responsible for detecting names (e.g., component or role names) that are not valid in MATLAB/Simulink. It also suggests an alternative name to the user. Consequently, some user interaction may be necessary here. Secondly, the analysis generates MATLAB/Simulink expressions for all occasions where the MechatronicUML Action Language is used. All analysis results are annotated to the initial MechatronicUML PIM instance.

T1.2: Model-to-Model Transformation The target of this transformation step is the Simulink EMF model [HRB+14]. The Simulink EMF model represents the modeling language concepts of MATLAB/Simulink and Stateflow (cf. Section 3.6) using the EMF. Thus, the model-to-model transformation can be implemented using EMF tools for model transformations, such as Triple Graph Grammars that were used for the initial implementation of this transformation. From the conceptual perspective, this process step applies a kind of model-to-model transformation to an annotated MechatronicUML PIM instance and creates an instance of a meta model

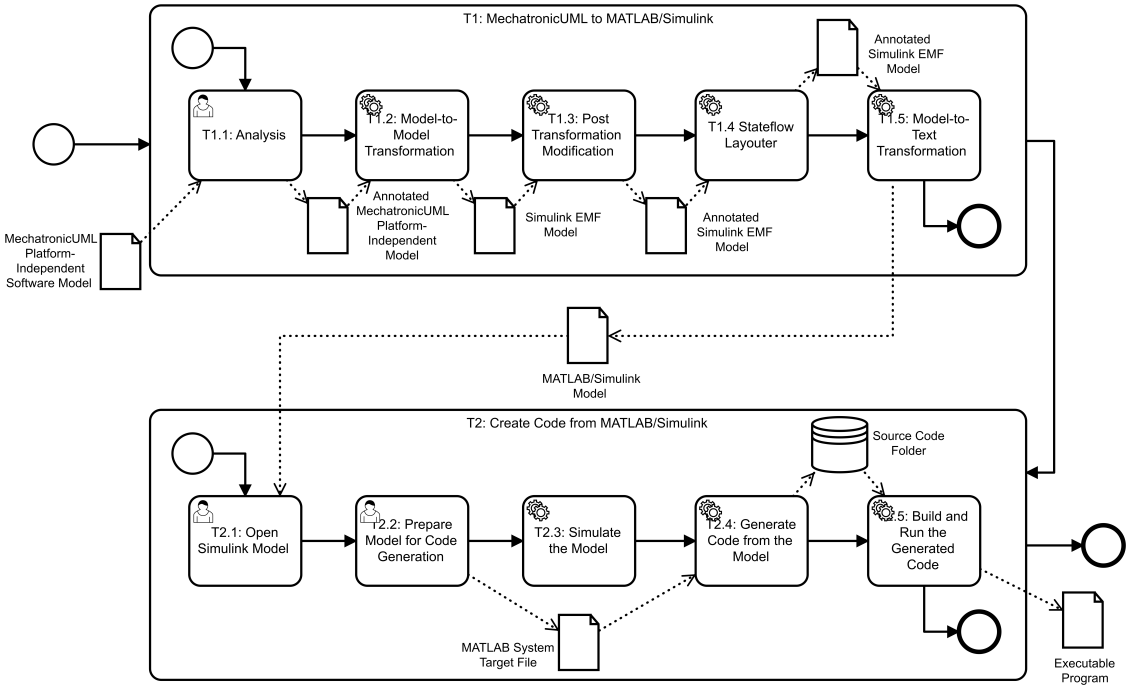


Figure 4.3: The code generation process of the MATLAB/Simulink approach based on Heinzemann et al. [HRB+14] and the Simulink Coder [Mat22d].

representing the MATLAB/Simulink language features. Notably, the behavior encapsulated in RTSCs is transformed to Stateflow charts. The key concept for modeling distributed deployments is the introduction of a link layer subsystem in Simulink that serves as a communication middleware between the Stateflow charts and the network infrastructure [Hei15].

T1.3: Post Transformation Modification The first supporting step after the model-to-model transformation adds information that is required for MATLAB/Simulink, but cannot be directly derived from the MechatronicUML PIM instance or cannot be ensured by the model-to-model transformation tool in use [HRB+14]. These are for example the size and position values for several kinds of Simulink model elements, and also the reordering of some MATLAB/Simulink expressions for correct semantics.

T1.4: Stateflow Layouter The second supporting step after the model-to-model transformation adds additional size and position parameters that are required for Stateflow, a toolbox for Simulink (cf. Section 3.6) [HRB+14]. This diagram information is not available in the MechatronicUML PIM instance, because the MechatronicUML Tool Suite strictly separates model content and diagram layout, and does not require a diagram to be present for a valid model. Therefore, a new layout is simply created automatically, so that all layout parameters are present and the files to be created can actually be opened using the MATLAB/Simulink tools.

T1.5: Model-to-Text Transformation The fifth and final step produces a MATLAB/Simulink model in the Simulink file format, which is a simple structured textual format [HRB+14]. The transformation takes the annotated Simulink EMF model as input and produces the corresponding text files. “Since the Simulink EMF-Model [...] has been designed to directly

represent the internal structure of the Simulink file format, the code generation uses an straight forward approach. Basically, for every element of the source EMF-Model, the code generation creates a block section in the Simulink file." [HRB+14] The same applies to the Stateflow features, that are also part of the Simulink EMF model, hence, corresponding files are generated for the Stateflow elements.

Thus far, the described process covers the transformation from the MechatronicUML PIM to MATLAB/Simulink. The generated Simulink files were originally evaluated using the MATLAB Release R2009b [Hei15]. The focus of this thesis is code generation, which is in fact a feature supported by the MATLAB/Simulink tooling. This second part of the code generation process is not regarded as part of the MechatronicUML -to-MATLAB/Simulink transformation. While all subtasks of task *T1: MechatronicUML to MATLAB/Simulink* were implemented in the MechatronicUML Tool Suite, and are also fully automated besides some user interaction in T1.1, the task *T2: Create Code from MATLAB/Simulink* uses MATLAB tools and features exclusively [Mat22d]. The remainder of this subsection describes the subtasks for the code generation using the Simulink Coder[6], which is capable of generating C as well as C++ code:

T2.1: Open Simulink Model Existing Simulink models can be opened via the MATLAB command prompt or via the user interface, which is the first thing to be done in order to access the code generation features [Mat22d]. Thus, this requires sound Simulink model files, according to the supported versions of the MATLAB/Simulink tooling in use.

T2.2: Prepare Model for Code Generation In the second step, the user selects different options for the code generation in a configuration window [Mat22e]. These options include the selection of the target language (C or C++) as well as model checking options, but most notably also the selection of a *system target file* [Mat22c]. The system target file "defines the run-time environment and the code generation features" [Mat22c] and the MATLAB environment provides several of these files. The file to be used with the Simulink Coder is designed for "generic real-time targets" [Mat22c]; using different files (e.g., for embedded real-time targets) requires additional MATLAB tools to be installed.[7] Moreover, developers may create custom system target files, which is a complex process of its own (cf. [Mat22b]). Lastly, the Simulink Coder documentation assumes an already complete MATLAB/Simulink model. When using the Simulink Coder in combination with the MechatronicUML-to-MATLAB/Simulink transformation, the model will typically have to be adjusted or completed: The behavior of continuous components can be implemented using MATLAB/Simulink. Only then, the model originating from the MechatronicUML is prepared for code generation.

[6] The Simulink Coder was called Real-Time Workshop at the time of MATLAB Release R2009b. Based on a rough hands-on comparison of the MATLAB Tooling of both R2009b and R2021b, the concepts are still very similar. Therefore, this thesis refers the documentation of the most recent MATLAB Release R2021b.

[7] For instance, the Embedded Coder. As these tools are all similar to the Simulink Coder and follow the same concept, this thesis does not distinguish between them. The MATLAB environment also suggests the installation of the required tooling when the user chooses a specific system target file that cannot be handled by the Simulink Coder alone.

T2.3: Simulate the Model According to [Mat22d], the simulation is the third step; however, it is not required for code generation, so it is simply mentioned here for the sake of completeness, and to point out that simulation is a key capability of MATLAB/Simulink. Using these simulation features is one reason why the transformation from the MechatronicUML PIM to MATLAB/Simulink was developed in the first place [Hei15; HRB+14].

T2.4: Generate Code from the Model The fourth step executes the actual code generation, meaning the source code files are created [Mat22d]. The configurations and the selected system target file are applied for the code generation. A new directory is created that contains all source files. The MATLAB/Simulink tooling also provides a code generation report to inspect the source code.

T2.5: Build and Run the Generated Code Before building an executable, two things have to be done: firstly, a valid compiler has to be made known to the MATLAB/Simulink environment, and secondly, the building has to be enabled via the code generation options in the MATLAB/Simulink tooling [Mat22d]. Depending on the selected system target file, the executable is built and can also be executed directly via the MATLAB command prompt.

All in all, this process describes the code generation starting with a MechatronicUML PIM instance. The process is distributed between independent tools, namely the MechatronicUML Tool Suite and the MATLAB/Simulink tools including the Simulink Coder. The described MechatronicUML-to-MATLAB/Simulink transformation was implemented for the MechatronicUML Tool Suite in version 0.5 [Hei15]. The actual code generation is enabled by this transformation, but otherwise, it is entirely dependent on the capabilities of the MATLAB/Simulink tools. Furthermore, the abstract link layer subsystem, which is introduced by the MechatronicUML-to-MATLAB/Simulink transformation, must be refined or replaced by a middleware model or implementation for the target hardware platform [Hei15]. Using this code generation requires expert knowledge about the MechatronicUML and its tool suite as well as knowledge about implementing the continuous components' behavior with MATLAB/Simulink and using its tools including the code generation features.

It is important to note that the MechatronicUML-to-MATLAB/Simulink transformation was not designed and implemented for code generation specifically. Consequently, the implementation of the overall code generation approach requires the compatibility of the MATLAB/Simulink file, which the transformation creates, to the MATLAB/Simulink tooling in use for the code generation. The loss of compatibility of the Simulink file was one of the major reasons for reimplementing the transformation [Hei21]. However, with the new implementation, the resulting Simulink model was in fact used to generate C code with the MATLAB/Simulink tools.

4.2.3 Other Related Approaches

Lastly, this subsection briefly introduces related MDSE approaches for CPS without a specific focus on the MechatronicUML. Still, the focus is on approaches that actually target code generation and not just modeling, model verification or similar features. Mohamed et al. conduct a systematic literature review of model-driven tools and languages for CPS. They analyze that only a tenth of the examined MDSE approaches publications or reports mention code generation features [MKC21]. Most of the related approaches introduced in this subsection are also part of this systematic literature review. Again, the focus is on *distributed* CPS.

Ataíde et al. develop a code generation for communication between independent nodes of a CPS [ABBG17]. Input-Output Place-Transition (IOPT) nets are used as a base modeling language: this graphical modeling language is based on petri nets and allows the modeling of the systems' behavior. The IOPT-Tools are a tool set for designing systems using IOPT nets, and the tools include basic code generation features for C code. The modeling language also allows to model distribution by means of so-called *time domains* and special places to be used for synchronization between different time domains. The code generator effectively separates the behavior of each time domain and generates independent code. The researchers use this code and additionally generate a communication layer based on the I2C-bus specification. Thus, they enable the modeling and code generation for a distributed CPS scenario. However, this approach has some limitations: Firstly, the static structure of the system cannot be modeled explicitly. Additionally, the allocation of software to hardware and also the hardware platform itself cannot be described or modeled. The approach currently only supports generating the communication layer for Arduino using I2C. Consequently, the code generator also assumes an assignment of device pins for the communication, which requires either the wiring or the code to be adapted afterwards [ABBG17]. Furthermore, the researchers do not state whether or how their approach can be ported to other target platforms as well.

Gritzner and Greenyer use a graphical modeling language called the *Scenario Modeling Language (SML)* to model the behavior of CPS in a simple, textual manner [GG18a]. Application scenarios can be used to describe how the system must and must not behave using the SML. Such a formal scenario specification can be transformed to *Structured Text*, which is a standardized format for programming PLCs [GG18a; GG18b]. The approach generates Structured Text for the behavior of the systems' coordination, and only atomic actions regarding sensors or actuators have to be implemented manually. In this regard, the SML operates somewhat similar to the MechatronicUML, where the behavior of continuous components must also be implemented outside of the MechatronicUML. However, the SML does not provide means to directly describe the structure of the software or the platform. The scenario specifications are transformed into state machines including one primary state machine that represents the global communication and synchronization behavior that is responsible for the coordination [GG18a]. But the approach is limited to generating code for a single controllable object with a single software controller [GG18b]. Another code generation attempt using the SML targets scenario-based programming in Java [GGK+17]. This approach does also not cover generating code for a distributed deployment and execution of the software. However, the researchers describe that replicating the program for all components and manually implementing communication behavior for synchronization was possible.

Ringert et al. introduce code generators for several target languages using the architecture description language MontiArch [RRW14]. The researchers extend this graphical, component-and-connector-based modeling language with features for behavior description. Similar to the usage of RTSCs in the MechatronicUML, they use so-called I/O$^{\omega}$ automata, which comprise states, transitions, variables and guards. This extension of MontiArc is called the MontiArcAutomaton language [RRW15]. The MontiArcAutomaton language targets the specification of platform-independent software models including the structure and the behavior of the software. It uses constructs like components, ports and connectors and describes the specification of messages. In addition, the researchers developed code generators for several target languages such as Java or Python [RRW14; RRW15]. The code generators use a template-based model transformation approach. The code generators were used to realize different application scenarios with distributed CPS. However, the coordination between different autonomous systems within such a distributed scenario can apparently not be modeled using the modeling features of the MontiArcAutomaton language [RRW15]. Instead, the

researchers add respective information to the platform-independent model manually. With respect to code generation, the researchers state they used to work on a modeling language "to specify the deployment of MontiArcAutomaton components to hardware" [RRW14]. To the best of our knowledge, there is no specification of this language available at the time of this writing. For the previously mentioned code generation approaches, there is a runtime library per target platform that abstracts from underlying APIs. Any code for platform-specific implementations "has to be manually created for every target platform" [RRW15].

Buckl et al. address the challenge of adaptivity in embedded systems engineering by developing a runtime environment for embedded real-time systems [BGG+14]. The so-called CHROMOSOME runtime environment abstracts from the hardware platform and provides a uniform communication interface for the application component running in this environment. In order to configure the environment, a domain model is used to specify the communication topics. These topics are used to enable the communication via a publish-subscribe mechanism. The domain modeling is enabled by the CHROMOSOME Modeling Tools, that further include the modeling of communication patterns and real-time constraints. This tooling also has the capability of code generation in order to configure the runtime environment. However, this approach does not focus on generating the source code for an application in its entirety. It rather provides middleware and runtime environment components and functions in order to deploy application components whose requirements can be modeled for configuration of this runtime environment.

The Progress Component Model (ProCom) provides a graphical modeling language targeted to the domain of embedded software systems [BCC+08]. The ProCom language for platform-independent modeling consists of two modeling layers: The ProSys and ProSave. The ProSys allows the modeling of subsystems with input and output message ports. These subsystems can be connected using their ports as well as message channels. There are composite subsystems containing multiple other subsystems and message channels. Primitive subsystems on the other hand include components of the second modeling layer: the ProSave. ProSave components are passive functions that can be activated and composed by the ProSys subsystem layer that embeds them. Thus, a ProSys subsystem specifies its behavior by embedding ProSave components. These components can again be composite. Primitive components contain the actual behavior and are simply realized by source code [BCC+08]. Carlson et al. develop an synthesis approach for deployment modeling by introducing two additional modeling layers: virtual nodes and physical nodes [CFMS10]. ProSys subsystems are allocated to virtual nodes that define resource budgets for the execution. Virtual nodes themselves are allocated to physical nodes. The modeling of physical nodes defines the hardware platform of the system. This comprises individual node instances with their respective computing capacities (processor type, memory) and network connections with their respective properties (network type, throughput). [CFMS10] also define the executable representations of each modeling artifact [CFMS10]. ProSave components are represented by a function in the C programming language per component. ProSys subsystems are reflected by a set of tasks and parameters configuring the execution of the embedded C functions. The virtual nodes include the aforementioned as well as a real-time scheduler. And the physical nodes are eventually represented by a compiled binary that can be executed on the target. By design, their approach is limited to generating C code as the systems' behavior is specified using source code within the primitive ProSave components. Nonetheless, the approach demonstrates the process from a PIM model level (ProSys and ProSave) to a PSM, and is in that regard similar to the MechatronicUML. But the MechatronicUML PIMoffers significantly more modeling features, e.g., by using RTSCs for the behavior specification and also by clearly distinguishing between types and instances for both software and hardware components. With respect to deploying

distributed systems, there also have been some approaches for ProCom: Inam and Sjödin develop a communication strategy among virtual nodes using a separate communication server on the virtual node level that contains specific software components fo the communication [IS12]. However, the initial implementation was limited to Ethernet-based communication. Lednicki presents a framework for automatic generation of the communication between physical nodes [Led15]. This process includes to extract a communication model defining which application components require to communicate across physical nodes, detecting available media in the physical node model and respectively choosing communication protocols, and finally creating the components required for the communication. The generation of the communication components contains generic as well as protocol-specific parts. For the latter, some implementations were realized for an example scenario.

5 A Taxonomy for Model-Based Code Generation

This chapter introduces a taxonomy for model-based code generation approaches. The purpose of this taxonomy is to classify and characterize code generation approaches. Specifically, model-based code generation approaches that use the same modeling language as input are compared. With respect to this thesis, where the source modeling language is the MechatronicUML, the proposed taxonomy is applied to the previous code generation approaches introduced in Section 4.2. Thus, the taxonomy represents the answer to **RQ2.1:** What criteria are applied to assess the previous MechatronicUML-based code generation approaches? The taxonomy is also used to identify whether these approaches are suitable to be reused or extended for the code generation this thesis seeks to realize (see Chapter 6 and Chapter 7).

In order to describe the proposed taxonomy, this chapter starts by explaining the design of the taxonomy in Section 5.1. Next, the four perspectives of the taxonomy are defined in detail: The *Code Generation Concept* in Section 5.2, the *Usability of the Code Generator* in Section 5.3, the *Implementation of the Code Generator* in Section 5.4 and the *Generated Code* in Section 5.5.

5.1 Taxonomy Design

The design of the taxonomy follows the concept by Usman et al. The authors analyze the usage of taxonomies in software engineering and identify important aspects for the definition of a taxonomy [UBBM17]. At it's core, a taxonomy is "a classification system, [so] one of the main purposes to develop a taxonomy should be to classify something" [UBBM17]. Consequently, the classification concept introduced in this chapter is also called a taxonomy: A taxonomy for model-based code generation.

Most importantly, in order to particularize the specific purpose of the taxonomy, the object to be classified has to be defined. Model-based code generation describes the process of producing source code from a model instance. The unit of classification for the proposed taxonomy is an individual *model-based code generation approach* as defined in Section 3.1.

The taxonomy is designed with the following intentions in mind: the taxonomy is used to (i) define an ideal candidate approach, (ii) analyze possible candidates, and (iii) decide whether available candidates fulfill the previously defined ideal properties well enough. This threefold process is applied to identify whether one of the previous MechatronicUML-based code generation approaches (see Section 4.2) is suitable to be reused or extended in order to realize the code generation for the scenario described in Chapter 2. Thus, this thesis seeks to validate the proposed taxonomy by demonstrating its utility trough application to concrete examples [UBBM17].

In addition to defining the unit of classification, it is beneficial to explicitly define the classification structure of the taxonomy [UBBM17]. This contributes to specifying the semantics of the taxonomy and its intended way of use. Typically, taxonomies in software engineering are designed (i) in a

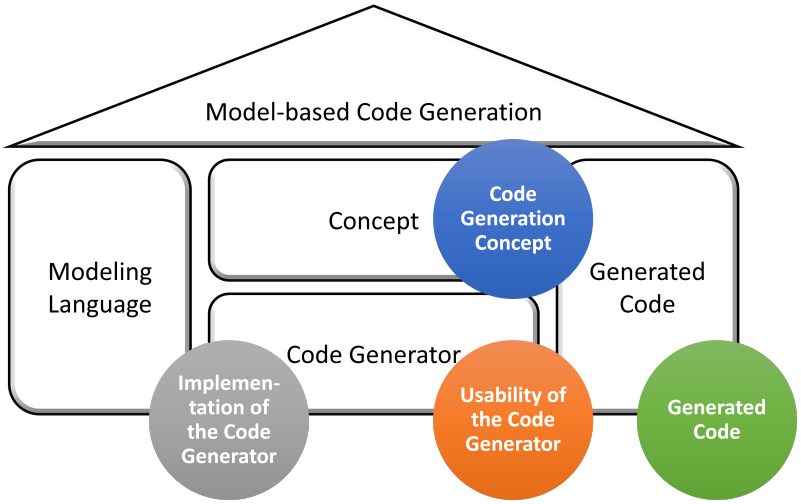


Figure 5.1: The perspectives of the taxonomy for model-based code generation approaches

hierarchical structure, (ii) in a tree-based structure, (iii) following paradigms, or (iv) a faceted analysis [UBBM17]. The particular taxonomy introduced in this work is not designed to be interpreted in a hierarchical or tree-based way. It does not serve the purpose to rank or rate code generation approaches, or put individual approaches into a specific relationship with each other. There are no scales or measurements involved in the classification procedure. Instead, the taxonomy serves the purpose to identify similarities and differences between two approaches, or between an assessed approach and an imaginary, ideal candidate approach. The taxonomy's purpose is to assist software engineering researches and professionals to assess and compare code generation approaches in a structured manner. Hence, the proposed taxonomy follows the concept of a faceted analysis.[1] The taxonomy classifies model-based code generation approaches by describing different facets of an approach. The classification procedure is intended to be a qualitative classification that describes the facets of a code generation approach from different perspectives.

The proposed taxonomy defines four perspectives on model-based code generation approaches as the unit of classification. These perspectives are visualized in Figure 5.1. Each perspective has a specific focus and a set of facets that are used to characterize an individual approach. These perspectives are visualized in Figure 5.1: One perspective is the *code generation concept*, another perspective is the *generated code*, and the code generator is examined from the perspectives of its *usability* and its *implementation*. While all four perspectives are generally applicable to model-based code generation approaches, it is entirely valid to prioritize certain perspectives or facets when applying this taxonomy in a concrete case. The taxonomy does not put a general weight on specific perspectives or facets, so this is purposefully left open to be decided based on the scenario the taxonomy may be applied to.

Notably, the *modeling language* is not part of the proposed taxonomy's perspectives, even though it is a characteristic part of model-based code generation (cf. Section 3.1 and Figure 5.1). This decision was made on purpose, because the focus of this taxonomy is to compare several code generation approaches which use the same modeling language as input, specifically with the MECHATRONIC-

[1] For details about the differences between the analysis types of taxonomies, see [UBBM17, p. 45].

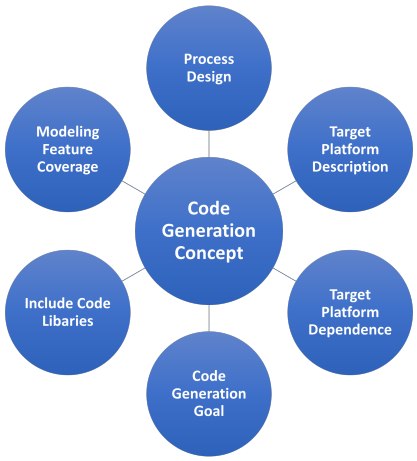


Figure 5.2: The facets of the code generation concept

UML in mind. If entirely different modeling languages are compared, the comparison is extended by a whole different dimension which is not deemed beneficial for this work. All in all, the proposed taxonomy is designed for the comparison of such code generation approaches that are based on the same input modeling language. For comparing modeling languages in the context of model transformations, consult the taxonomy introduced by Kahani and Cordy in [KC15], especially to the model-level category (see Section 4.1). The four perspectives of the proposed taxonomy and all their facets are described in the following subsections.

5.2 Code Generation Concept

The *code generation concept* perspective focuses on the logical concept that the code generation approach is based on. The concept describes the logical transformation steps which are undertaken to generate code from a model instance. This logical concept may be explicitly documented or implicitly defined by the code generator's implementation. The following facets, which are also visualized in Figure 5.2, aim at characterizing a model-based code generation approach from the conceptual perspective:

Process Design A characteristic facet of a code generation concept is the overall process design: The number of steps that are required, the degree of user interaction and automation, the kinds of model transformation technologies being used (see Section 3.2) and also which metamodel and model instance is used to start the code generation.

Target Platform Description Code generation typically requires the description of the target platform, because a PIM as source model does not provide this information. The target platform description may be implemented using modeling features that the source modeling language already provides, by reusing a related modeling language, or by annotating the given PIM instance.

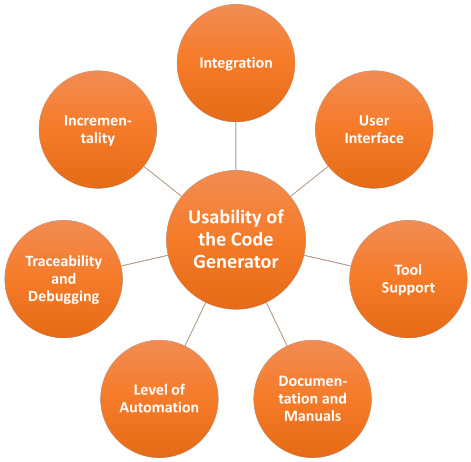


Figure 5.3: The facets of the usability of the code generator

Target Platform Dependence The code generation approach might purposefully allow the user to specify their target platform, or the platform description may be fixed for the code generation. The degree of dependence on the target platform limits the concept's applicability to only one or a few specific target platforms.

Code Generation Goal The primary input for model-based code generation is a model instance. Depending on what the modeling language provides, the goal of the code generation may be to produce the static structure of the code (i.e., code skeletons) or the entire behavior (i.e., the functional code). In addition, the goal of the code generation could also be to create build or deployment scripts for a specific platform that uses the generated source code.

Include Code Libraries Another important facet is whether the code generation concept provides the possibility to include code libraries [Sel03]. This becomes especially important if the software behavior is specified using the modeling language in order to generate functional code. Using libraries in the code generation process allows the usage of existing, legacy code and also to reuse such code within the modeled system.

Modeling Feature Coverage With respect to complex meta models and modeling features a modeling language may provide, it is possible that the code generation is limited to a subset of these modeling features. Therefore, a characteristic facet of a code generation approach is which degree of the feature set is supported, or which specific features are applicable for code generation.

5.3 Usability of the Code Generator

The second perspective of the taxonomy focuses on the implemented code generator as a black box. It describes the usability of the code generator, thus describing the meaningfulness for actual users. These users are expected to have good knowledge of the modeling language and domain expertise, hence, no aspects of the modeling language and its tooling are regarded here. The usability of the code generator is described with the following facets, that are also visualized in Figure 5.3:

Integration The code generator may be a standalone tool, but it may also be integrated into existing tools. Possibly, it is directly integrated into the editing and verification tools which already exist for a specific modeling language [KC15].

User Interface The code generator may have a command-line or a graphical user interface. The general usability is influenced a lot by the quality of the user interface. The user interface may also play a key role in defining which parameters and configuration options are exposed to a user of the code generation.

Tool Support Tool support refers to the availability of a stable release of the code generator, and whether this release is actually still working. Furthermore, the tool support facet includes whether maintenance activities are still ongoing and the code generator software is still supported (e.g., via a bug tracking or ticketing system) [KC15].

Documentation and Manuals Especially if the code generation approach employs a complex transformation process that requires several steps of user interaction, a user documentation or user manual are key in making the code generator usable. The existence and quality of such resources are described in this facet.

Level of Automation In a complex transformation process, there might be the necessity for user interaction between individual transformation steps, e.g., to provide additional information for the next step [KC15; MV06]. The level of automation describes this degree of user interaction versus automation, e.g., by describing how often the user interaction is required, and how much has to be done manually, or can be configured manually according to the user's preferences.

Traceability and Debugging Model transformation tools often provide traceability links between the source and target model instance as persisted records, so that a user can trace which model elements from the source model correspond to those elements in the target model instance [CH06; KC15; MV06]. This is also helpful for a tool user if they want to find out where they made a modeling mistake, or when they want to change their model. In the case of code generation, they may be able to track a problem in the code back to a modeling error in the first place. Therefore, traceability is as important for model-based code generation approaches as it is for the individual model transformation technologies the code generation might employ.

Incrementality A code generator benefits from incremental code generation, meaning that upon changing the model, only the affected parts of the produced code have to be generated again [CH06; Sel03]. Such update mechanisms play a key role for the scalability of the code generation for large applications that are under constant development and change, and for the later stages in an application's life cycle where there are many small changes [Sel03].

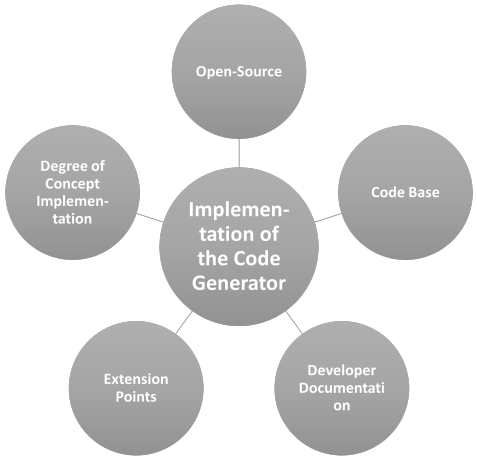


Figure 5.4: The facets of the implementation of the code generator

5.4 Implementation of the Code Generator

The third perspective also focuses on the implementation of the code generator, but not from the user perspective: The code generator is assessed from a white box perspective, i.e., from the perspective of a researcher or developer who implements, maintains or adapts the code generator. The aim is to describe the state and quality of the implementation in order to evaluate is maturity and usefulness for the desired purpose. The implementation perspective has the following facets (see Figure 5.4):

Open-Source A first question to be answered is whether the source code is openly accessible, or not accessible at all. If developers actually want to make changes to an existing code generation approach, access to the source code may be required, unless specific extension points are defined (see below).

Code Base Regarding the code-base, developers may want to know whether it is still maintained, whether all external dependencies are still supported, and whether the code bases uses a modern programming language. Another question may be if build scripts are available. Moreover, the code quality may be analyzed.

Developer Documenation Another important facet is the availability and quality of developer documentation. The developer documentation could also include a specification of the build process to create an executable from the source code in case the code base does not contain build scripts.

Extension Points Another facet is the availability of extension points: the code generator's implementation may define extension points, e.g., for developing own plugins in order to adapt the behavior of the code generator to individual needs.

Degree of Concept Implementation Lastly, there may also be a discrepancy between the code generation concept and what parts of it have already been implemented. Thus, the degree of concept implementation describes the maturity of the implementation with respect to the code generation concept.

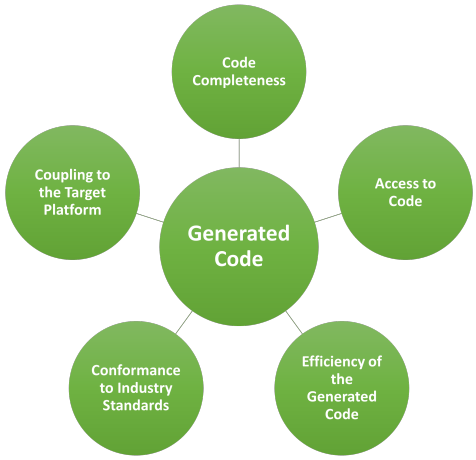


Figure 5.5: The facets of the generated code

5.5 Generated Code

The fourth and final perspective of the taxonomy deals with the output product of the code generator: the generated code. The code and potentially build or deployment scripts are evaluated with the following facets, which are also visualized in Figure 5.5:

Code Completeness The first facet of the generated code is its completeness: the code may be entirely complete, deployable and executable after the code generation. Or it may require manual adaptions or additions, e.g., if only skeletons are created for some parts of the application. Depending on the modeling language's features, functional code might not be generated at all.

Access to Code Secondly, it might also be beneficial to be able to change the code manually – independently of its completeness. So the question is whether the produced source code is accessible by the user of the code generator, or if the source code is entirely hidden and only binaries for a specific target platform are produced.

Efficiency of Generated Code According to Selic, one of the first questions to ask is "how the automatically generated code's efficiency compares to handcrafted code" [Sel03]. While a precise evaluation is difficult regarding this facet because handcrafted code might not be available for comparison, it might be helpful to evaluate whether there are measures undertaken in order to ensure or improve efficiency of generated code. More importantly though, there may be QoS requirements originating from the application domain. Such requirements may even be modeled as QoS policies and used for model verification, depending on the capabilities of the modeling language in use. If such QoS policies are in place, the code may be assessed by means of WCET analysis and schedulability analysis [BCD+14] to verify whether the requirements can actually be met by the generated code.

Conformance to Industry Standards Depending on the application domain for the code generation, there might be industry standards or best practices that the source code should or must adhere to. If this is the case, the question may be raised how many of these standards the code generation adheres to, and to what degree.

Coupling to the Target Platform Lastly, the coupling to the target platform is another characteristic facet: it may depend on the availability of compilers and build tools whether the source code can be built for several target platforms, but also the source code itself may pose restrictions, i.e., by using underlying APIs functions. Then, it may be also interesting to assess how much would have to be changed in order to reuse the generated code on a different platform.

6 Concept for the Code Generation

The MechatronicUML PIM is capable of describing all structural and behavioral aspects of a software system. As such, it is well suited to be used for model-based code generation. The goal of this thesis is to obtain a code generator for the robot cars described in Section 2.2. Previous approaches to generate code for a distributed deployment from a MechatronicUML model are described in Section 4.2. These approaches may be used as the foundation of the envisioned code generator if they are suitable to be extended or reused. Therefore, Section 6.1 defines the properties of an ideal candidate to be extended or reused. Afterwards, in Section 6.2 and Section 6.3, these previous approaches are analyzed and evaluated with respect to this ideal candidate definition. The results of the analysis are summarized in Section 6.4, before Section 6.5 subsequently describes the design choices for the new code generation concept.

6.1 Definition of an Ideal Candidate Approach

Ideally, an existing code generation approach is extended or reused to create a code generator for the robot cars. This section defines the properties of an ideal candidate approach for this purpose. The previously proposed taxonomy (see Chapter 5) is used to define said candidate in a structured way. As suggested in the taxonomy design, some perspectives are prioritized over others to reflect the aim for which the taxonomy is applied. Thus, the *code generation concept* is the most important perspective: The concept describes the flexibility of the concept and whether multiple target platforms may be supported. Consequently, the concept largely influences whether the application scenario described in Chapter 2 fits within the scope of an existing code generator, or can be included by extension. Then, if straight reuse of a previous approach is not possible, the *implementation of the code generator* is the second most important perspective: it defines the properties required to extend an existing candidate.

As motivated beforehand, the exact capabilities of previous code generation approaches with the MechatronicUML and their state of implementation are not precisely known. Therefore, the focus is on finding an approach that can provide a convincing logical concept and a promising implementation state. Consequently, the *usability of the code generator* and the *generated code* perspectives are less critical in this case. Accordingly, a number of facets of the usability and generated code perspectives are described as *not critical for the intended purpose* of identifying a suitable approach for reuse or extension.

6.1.1 Code Generation Concept

The ideal *code generation concept* for the sake of reuse or extensibility is characterized as follows:

Process Design The starting point of the code generation must be a MechatronicUML PIM model instance as it describes all structural and behavioral aspects of the software (cf. Section 3.4). Other than that, the intention of the concept should be that all actual transformations (i.e. model-to-model or model-to-text transformation steps) are automated, and manual user interaction is limited to provide additional information. The concept should be designed in a way that individual steps of the transformation process may be exchanged to support different platforms or technologies; thus, the steps of the process should be as independent as possible. The usage of specific model transformation tools is not required.

Target Platform Description The robot cars are an Arduino-based target platform. This platform should either be directly supported by the code generation concept, or the concept must allow to model this platform, e.g., by employing the MechatronicUML HPDM. Most importantly, the target platform description must allow the specification of a distributed deployment scenario.

Target Platform Dependence The application scenario (see Chapter 2) is intended to represent a scenario from the automotive industry. Deploying code to mini robots only serves as an example, so being able to potentially reuse or extend the concept for other platforms that are actually used in the automotive industry is crucial. Consequently, the code generation approach must not be tailored to only one possible target platform, but must provide the freedom to support new platforms in the future.

Code Generation Goal The goal of the code generation must be to generate the application structure and behavior, and ideally also provide build scripts with the code. Moreover, the concept must support to model a distributed deployment scenario in a way that is suitable for code generation, i.e., by introducing a middleware layer or similar abstraction that can then be implemented for the specific target platform.

Include Code Libraries The inclusion of external source code by the means of libraries must be considered in the logical concept.

Modeling Feature Coverage Finally, Table 6.1 lists the top-level features of MechatronicUML PIM and indicates, which modeling features the code generation must be able to handle. This is only a high-level definition of required features, as each feature itself consists of many modeling elements some of which may be supported for code generation and others not. In

MechatronicUML PIM Feature	Relevance
Real-Time Coordination Protocols (RTCPs)	mandatory
Real-Time Statecharts (RTSCs)	mandatory
Atomic and Structured Components	mandatory
Component Instance Configurations (CICs)	mandatory
Reconfigurable Components	optional

The relevance is categorized with *mandatory* and *optional*. For an explanation of the MechatronicUML PIM features, see Section 3.4.

Table 6.1: An overview about the MechatronicUML PIM features and their relevance for the code generation.

general, a mandatory feature should be supported regarding all modeling elements; specific shortcomings will likely exist and have to be examined in each particular case. As the goal of the code generation is to generate structural and functional code, all features are mandatory besides Reconfigurable Components, which are a special extension of the MechatronicUML PIM language.

In summary, all aspects of the conceptual perspective are very important for the ideal candidate. Figure 6.1 shows this and underlines the importance of the code generation concept for reusing or extending it.

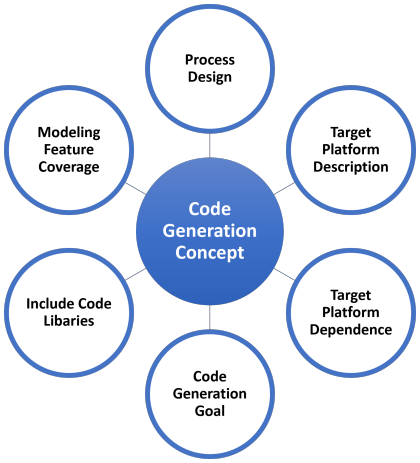


Figure 6.1: All facets of the code generation concept are important for the ideal candidate.

6.1.2 Implementation of the Code Generator

The state of *implementation of the code generator* is especially relevant in order to ensure that the code generator can be extended, both for this thesis as well as in the future. The following properties describe an ideal candidate for this purpose.

Open-Source The source code of the code generator must be accessible in order to extend and adapt it. Even if the code generator supported the desired application scenario without modification, there would have to be the possibility that it can be improved and extended by this thesis or future work. If the source code is not accessible, the code generator may compensate for that by clearly defined extension points (see below).

Code Base Active maintenance of the code base is not required, but a working build process or launcher must exist. Furthermore, all external dependencies must still be supported and there must be a way to build or launch the code generator. In case issues have to be fixed here to reach this desired sate, a limit of roughly five person days of effort is set. Otherwise, the code base is considered unfit.

Developer Documenation Ideally, there is an up-to-date developer documentation that describes the structure of the code base and how to set up the development environment to contribute to the implementation. It should also mention the languages and tools that were used to develop the software.

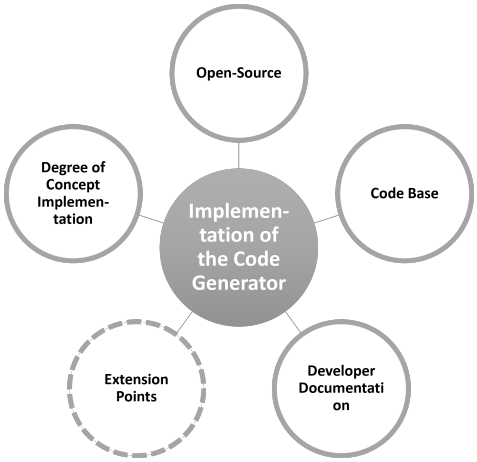


Figure 6.2: All facets of the implementation of the code generator are important for an ideal candidate, with extension points being desirable but not mandatory.

Extension Points Well-defined extension points for the code generator are desirable, but are not a mandatory requirement. An implementation may still be deemed sufficient without extension points.

Degree of Concept Implementation As the concept is the main decision criterion, this concept must be implemented to a very high degree. If the logical concept is more mature or extensive than what has been implemented, reasons for this discrepancy may be examined. An approach may still be deemed fit for reuse even if it is not implemented entirely. However, this may result in initial implementation effort to overcome this gap before the implementation may reach a state suitable for extension.

In conclusion, all facets of the implementation of the code generator are important for the ideal candidate with the extension points being optionally relevant in case there is no source code access. This is also depicted in Figure 6.2 and shows the relevance of the implementation for extending an existing code generation approach.

6.1.3 Usability of the Code Generator

The *usability of the code generator* is a less important criterion for the ideal candidate's properties. Therefore, the description of ideal properties is limited to facets that are deemed relevant for extension and reuse.

Integration The integration of the code generator is not critical for the intended purpose.

User Interface The user interface of the code generator is not critical for the intended purpose.

Tool Support There must be an executable version of the code generator. It may either be downloadable as binary or it may be built or launched otherwise. This is also the basis for an in-depth analysis of a candidate approach. However, active maintenance of the code generator tooling is not required.

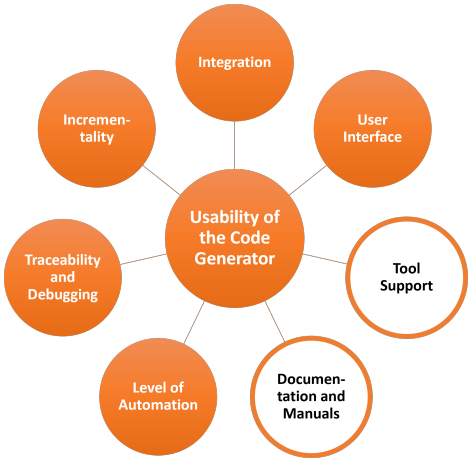


Figure 6.3: Documentation and manuals and tool support are important for an ideal candidate.

Documentation and Manuals There must be some kind of documentation or example available that allows a user to execute the code generation without having to explore the tooling and modeling language features from scratch.

Level of Automation The level of automation is not critical for the intended purpose.

Traceability and Debugging The traceability is not critical for the intended purpose.

Incrementality The incrementality is not critical for the intended purpose.

All in all, as depicted in Figure 6.3, many facets of the usability perspective are not critical for the ideal candidate. This underlines the secondary importance of the usability of the code generator for reusing and extending an existing candidate. Having a documentation and manuals as well as a working tool are the important facets.

6.1.4 Generated Code

Lastly, the *generated code* should ideally fulfill the following properties:

Code Completeness As the MechatronicUML PIM describes all functionality of the software besides the behavior of continuous components, the generated code should have a high degree of completeness. Ideally, relevant technical code for a target platform can also be included in the code generation process directly, so the code is actually complete at output time. Moreover, the required platform-specific code should be included. Ideally, it is also generated automatically, but must at least define interfaces to be implemented for a specific platform. Additionally, the code generator should also produce build scripts.

Access to Code The source code must be accessible.

Efficiency of Generated Code The efficiency of the generated code is not critical for the intended purpose.

Conformance to Industry Standards The conformance to industry standards is not critical for the intended purpose.

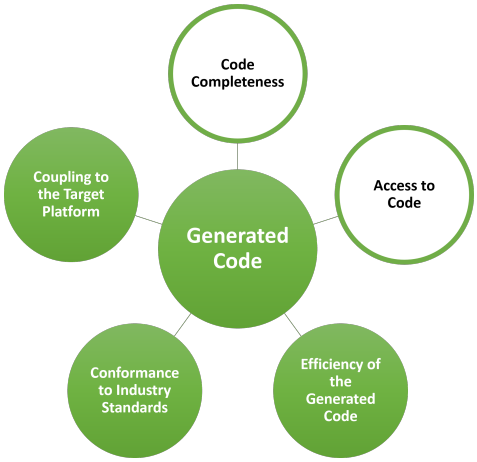


Figure 6.4: Code completeness and access to code are important for the ideal candidate

Coupling to the Target Platform The coupling to the target platform is not critical for the intended purpose.

Thus, the aspects of the *generated code* perspective are only partially important for the ideal candidate. Nonetheless, the completeness of the code and being able to access the code are critical aspects as depicted in Figure 6.4.

6.2 Analysis of the Platform-Modeling Approach

The platform-modeling approach is introduced in Section 4.2.1. Here, it is analyzed using the taxonomy introduced in Chapter 5. This section contains the analysis from all four perspectives of the taxonomy, focusing on the perspectives with a higher importance according to the definition of an ideal candidate in Section 6.1. The perspectives are sorted by importance and thus do not follow the order the taxonomy introduces, because the order does not have a particular meaning in the taxonomy itself.

In order to add some formalism to the analysis, and to allow comparison with the analysis of other approaches, the fulfillment of the ideal properties are summed up by a rating. This rating expresses to what degree the assessed approach meets the ideal properties for a specific facet. Consequently, only those facets that the ideal candidate describes as important are rated, even though the other facets of the analyzed approach are examined as well. The rating is called *fitness rating* and is defined as follows:

3 points The assessed approach **fulfills** the properties of the ideal candidate.

2 points The assessed approach **partially fulfills** the properties of the ideal candidate.

1 point The assessed approach **does not fulfill** the properties of the ideal candidate.

0 points The facet is **not applicable** to the assessed approach.

6.2.1 Code Generation Concept

The code generation concept of the platform-modeling approach is explicitly documented in [BCD+14] and [Poh18] (cf. Section 4.2.1). The overall concept and the code generation process are visualized in Figure 4.1.

Process Design The platform-modeling approach comes along with a sophisticated process leading the user through the steps required for code generation. These steps are the platform modeling, allocation engineering and software construction. The input is a MechatronicUML PIM instance, and the outputs are a source code folder as well as an executable. A total of 13 steps build the bridge between input and output, some of which are independent of each other and can be parallelized. The concept intends automation of all transformations and only required user interaction to add information or integrate source code that is external to the code generator. **Fitness rating: 3 points.**

Target Platform Description The platform-modeling approach allows the user to specify the target platform using the MechatronicUML HPDM language. This enables a high degree of freedom regarding possible target platforms, but also requires the knowledge and expertise about the platform and its modeling using the MechatronicUML HPDM. The target platform described in Section 2.2 is not supported out-of-the-box, but can be modeled with all its specifics, including modeling the particular platform instances. **Fitness rating: 3 points.**

Target Platform Dependence The code generation concept is not dependent on a particular type of platform. As mentioned above, the MechatronicUML HPDM allows to flexibly model the platform from scratch. Additionally, the concepts applied for the code generation are generally applicable to different programming languages [Poh18]. Furthermore, the usage of other middleware implementations can also be included into the concept. **Fitness rating: 3 points.**

Code Generation Goal The intention of the platform-modeling approach is to collect all information necessary in order to generate all structural and functional code as well as a platform-specific container implementation during the process. It also offers the opportunity to include other technical platform code in the code generation process instead of manually adding it later on. The goal is to create the entire source code for the target platform in one go, as well as producing build scripts. **Fitness rating: 3 points.**

Include Code Libraries The platform-modeling approach explicitly includes a concept for the integration of existing software libraries [Poh18]. The interfaces to these libraries have to be modeled in order to be used as part of the MechatronicUML PIM, and then they can be integrated with the generated code. **Fitness rating: 3 points.**

Modeling Feature Coverage The platform-modeling approach supports most features of the MechatronicUML PIM in version 1.0 [Poh18]. Limitations are: Urgent transitions in RTSCs, no optional ports or multi-ports for components, and no support of Reconfigurable Components. Overall, that means that most MechatronicUML PIM features are supported; Reconfigurable Components are not required by the ideal candidate definition. **Fitness rating: 2 points.**

6.2.2 Implementation of the Code Generator

The *implementation of the code generator* perspective does not only refer to the actual process steps that generate source code artifacts, but to the implementation of the overall code generation concept with its entire process and transformation chain. All three main tasks (platform modeling, allocation engineering and software construction) have been integrated into the MechatronicUML Tool Suite [Poh18]. This implementation is analyzed looking at the following five facets:

Open-Source The source code of the platform-modeling approach is available on GitHub and openly accessible. It is distributed over several repositories (see Table 6.2). All repositories indicate that they are published under the terms of the Eclipse Public License (EPL), and so does the documentation [Poh18]. **Fitness rating: 3 points.**

Code Base As depicted in Table 6.2, the code base of the platform-modeling approach is distributed over several repositories. None of these repositories is still actively maintained; the latest commits originate from 2017 and 2018. All repositories are setup as Maven[1] projects and configured to be built with Maven. This build can be used to publish the plugins to be installed in an Eclipse application. The *Allocation* repository's maven configuration references an external repository that does not exist anymore; the external repository is however not required and used in the code, to the best of the authors knowledge. Additionally, the version 2.10.0 of the xtend-maven-plugin which is used in the *PM* and *PSM* projects appears to be bugged. Upgrading the version of the xtend-maven-plugin to version 2.15.0 and removing the unused repository makes the local builds using maven succeed again for all repositories. The unit testing has to be skipped though as there are failing tests that prohibits the execution of the build. The individual repositories expose their functionality as a plugins for the Eclipse-based MechatronicUML Tool Suite, and all plugins can be launched using the MechatronicUML Tool Suite. The repositories include code for the user interface wizards next to the actual plugin functionality. The repository *PM* contains the functionality for the MechatronicUML HPDM, according to the task T1 in the code generation process (see Figure 4.1). The functionality for the allocation engineering of T2 can be found in the repository *PSM* which also holds the functionality for the platform-specific modeling prior to the actual code generation (subtasks T3.1 and T3.2) as well as the code generation of the device access code (subtask T3.5). The *Allocation* repository contains a table-based view for inspecting the result of the allocation engineering. The *C Components* repository contains the functionality for the C code generation of the platform-independent component behavior (subtask T3.3). Lastly, the *C Containers* repository contains the functionality for generating the platform-specific C container code (subtasks T3.6 and T3.7). Through all source code repositories, the Java programming language is used to implement Eclipse user interface wizards and to integrate the model transformations into the plugins. Moreover, the metamodels are created using the EMF. The model transformations of T3.2 are implemented with QVTo (see Section 3.2.1) and the C code generation for the components and containers is realized with Acceleo (see Section 3.2.2). **Fitness rating: 3 points.**

[1] https://maven.apache.org/

Short Name	Repository URL
PM	https://github.com/fraunhofer-iem/mechatronicuml-pm
Allocation	https://github.com/fraunhofer-iem/mechatronicuml-codegen-allocation
PSM	https://github.com/fraunhofer-iem/mechatronicuml-psm
C Components	https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-type
C Containers	https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-container

Table 6.2: The source code repositories of the platform-modeling approach

Developer Documenation First of all, the code base does not include a reference to any kind of documentation, nor does it include a README file or a similar documentation for quick start. The development-related documentation in the publications is either outdated ([BCD+14]) or treats the implementation only very briefly and on a high level of abstraction ([Poh18]). There is a developer documentation for the overall MechatronicUML Tool Suite available online[2], but it is also outdated; the latest meeting notes are from May 2016. The latest version that is described in in the documentation is the version 0.4, while version 1.0 which is available for download is only mentioned (cf. Section 3.3). Furthermore, references to the code do not point to the GitHub repositories (see Table 6.2), but to an old versioning system, which does not contain the latest state of the source code. Lastly, the only page in the developer documentation's "Code Generation" section is an empty placeholder since December 2015[3]. There is some documentation on the build process and used tools regarding the overall MechatronicUML Tool Suite, but no information regarding the platform-specific modeling or the code generation explicitly. The build servers are also not active any more, nor is the Eclipse update site. **Fitness rating: 1 point.**

Extension Points The concept already distinguishes between the platform modeling, the platform-specific modeling, and the code generation. For the code generation, there are defined extension points that are currently implemented by the *C Components* project for the platform-independent C code generation and by *C Containers* for the platform-specific code generation. These extension points can be used to add additional programming languages. For adding additional platform-specific configurations, there are no explicit extension points but the implementation is generally extensible by its structure as the metamodels and transformations may be altered. **Fitness rating: 3 points.**

Degree of Concept Implementation While the authors of the platform-modeling approach call their implementation "prototypical" [Poh18], referring to the fact that they have not put a lot of effort in testing the platform-specific transformation and code generation yet, the overall process and the underlying concepts are implemented as available as plugins for the MechatronicUML Tool Suite. **Fitness rating: 3 points.**

[2] https://trac.cs.upb.de/mechatronicuml/wiki/FujabaDevelopment
[3] https://trac.cs.upb.de/mechatronicuml/wiki/CodeGenDeveloperCreateNewGenerator

6.2.3 Usability of the Code Generator

The analysis of the usability is performed using the MechatronicUML Tool Suite 1.0 that includes the implementation of the platform-modeling approach. The MechatronicUML Tool Suite contains different code generators, most of which generate simulation code or code for single-ECU deployments. The platform-modeling approach on the other hand targets code generation for a distributed deployment of a CPS, and the respective code generator is target of this analysis. In the user interface, this is *Component Type ANSI C99* export option.

Integration The entire implementation of the platform-modeling approach is integrated into the MechatronicUML Tool Suite in version 1.0. Thus, it is included in the natural environment of MechatronicUML users that does not only provide access to the code generation, but also to all the additional modeling and verification features. Only the very last step of the code generation concept, T3.8: Program Building, is not integrated because it intends to use existing third party tools.

User Interface As the code generator is integrated into the Eclipse-based MechatronicUML Tool Suite, it is also accessible via its graphical user interface, including the perspectives for platform and allocation engineering and device access modeling. As the code generation concept itself is quite complex, it does require some expertise to operate the code generator.

Tool Support There is a release of the code generator as part of the MechatronicUML Tool Suite version 1.0. The Eclipse Marketplace entry[4] still exists and indicates maintenance until 2020, but the linked build server is not available anymore and there is no indication of ongoing maintenance activities. Nonetheless, the code generator is still working if the pre-packaged MechatronicUML Tool Suite is used and set-up correctly. **Fitness rating: 3 points.**

Documentation and Manuals A lot of the user documentation is published in the form of scientific publications such as [BCD+14] and [Poh18]. Additionally, there is a user guide for the entire MechatronicUML Tool Suite available online[5]. However, the documentation regarding the code generation capabilities is very brief, not up-to-date and refers to different versions of the MechatronicUML Tool Suite; there is no unified documentation. The publication by Pohlmann appears to be the most recent documentation, but does not focus on the user perspective but rather the technical concepts [Poh18]. There are examples that can be used for code generation, but no documentation on how to use them. There is no specific user manual. **Fitness rating: 2 points.**

Level of Automation The level of automation is very high: user interaction is only necessary to provide the information that should be transformed or combined, i.e., selecting a CIC to start the code generation process with. The actual transformation steps are executed entirely automatically. However, the entire process chain is not integrated by one, large wizard, but it requires knowledge and user interaction to trigger the steps in the right order.

[4] https://marketplace.eclipse.org/content/mechatronicuml

[5] https://trac.cs.upb.de/mechatronicuml/wiki/FujabaUserGuide

Traceability and Debugging There is no indication that traceability links between source code artifacts and model elements (or vice versa) are stored. The trace between model and code can only be reconstructed manually based on the names of components, ports, states, operations and other modeling elements that are then represented in the code.

Incrementality The source code is generated using the Acceleo, and the homepage of this tool advertises incremental code generation as one of its features [Ecl22]. It does support incremental code generation in the sense of not overriding protected regions with user-added code. However, Acceleo does regenerate files even if they are unaffected by the changes made to the input model.

6.2.4 Generated Code

The generated code is analyzed by applying the code generator to some example models[6], as well as custom models that were created for the purpose of this analysis. The analysis was done manually by inspecting the generated code as well as comparing it after modeling changes.

Code Completeness The code implements the structure and the behavior of the software as modeled using the MechatronicUML . It is complete in the sense that all information that the models collect is represented in the source code. Furthermore, the behavior of continuous components which cannot be modeled using the MechatronicUML PIM (cf. Section 3.4) can be added manually or modeled as part of the platform-specific modeling (see T3.1 in Figure 4.1). Furthermore, the platform-modeling approach enables the generation of container code that implements a middleware layer to abstract from the implementation of the communication. The required communication middleware artifacts can also be generated. However, generating the communication artifacts for the DDS-based middleware implementation was not tested as it relies on commercial tools that have not been available to the author of this thesis (see Section 4.2.1). All in all, the platform-modeling approach, if used to its full extent, generates the entire code necessary for a distributed deployment. **Fitness rating: 3 points.**

Access to Code The generated source code is stored in a directory inside the modeling projects working directory. Thus, it is entirely accessible. **Fitness rating: 3 points.**

Efficiency of Generated Code The code is not analyzed with respect to efficiency as part of this thesis. However, Pohlmann states: “The generated code is not optimized considering the memory footprint on the target platform. This is a technical limitation and can be optimized in the future.” [Poh18]

Conformance to Industry Standards Neither the documentation nor the code base of the code generator name the usage of specific coding standards.

Coupling to the Target Platform Already from the conceptual perspectives, the researchers who developed the platform-modeling approach set their focus on separating the source code into platform-independent code representing the components’ behavior, and platform-specific code for device access as well as the implementation of the containers and communication middleware [Poh18]. This is actually visible in the source code. The behavior implementation

[6] https://github.com/fraunhofer-iem/mechatronicuml-examples

is portable between different platforms as long as there is a compiler that supports the C programming language for the specific target platform. Consequently, only the platform-specific implementation has to be adapted to port the code to another deployment target, as long as the C component code can be reused.

6.3 Analysis of the MATLAB/Simulink Approach

The MATLAB/Simulink approach is introduced in Section 4.2.2. It is analyzed using the taxonomy introduced in Chapter 5 in this section. The section contains the analysis from all four perspectives the taxonomy proposes, focusing on the perspectives with a higher importance according to the definition of an ideal candidate in Section 6.1. Again, like in the previous section, the perspectives are sorted by importance and thus do not follow the order the taxonomy introduces, because the order of the perspectives in the taxonomy has no semantics.

The analysis of the MATLAB/Simulink approach is formalized in the same way as the analysis of the platform-modeling approach in Section 6.2, using the *fitness rating* to allow comparison of the results later-on.

6.3.1 Code Generation Concept

The concept of the MATLAB/Simulink approach is visualized in Figure 4.3. It is documented in the publications [HRB+14] and [Hei15] with respect to the MechatronicUML-to-MATLAB/Simulink transformation. Even though this documentation does not reflect the latest state of this approach, it is used here as it still reflects the overall concepts (cf. Section 4.2.2). Additionally, the code generation with MATLAB/Simulink tools is documented in [Mat22d] and [Mat22e] (cf. Section 4.2.2).

Process Design The overall code generation concept of the MATLAB/Simulink approach consists of ten steps in total (see Figure 4.3). The source for the approach is a MechatronicUML PIM instance. The degree of automation is very high: there are only three tasks that require user interaction. The purpose of the user interaction is to provide the information required for the following automatic tasks to be executed. **Fitness rating: 3 points.**

Target Platform Description Detailed modeling of the target platform is not part of the MATLAB/Simulink approach. However, there are different *hardware support packages* that can be installed into the MATLAB environment. These packages include support for Arduino and Raspberry Pi among many others. These hardware support packages provide the required information about the device vendor and type as well as information about data types and a system target file. Additionally, custom system target files may be created to describe custom target platforms [Mat22b]. Creating system target files is a technical opportunity, but due to its own complexity, creating such files is not part of the presented code generation process. Moreover, by introducing a link layer subsystem, the approach abstracts from the implementation of specific communication technologies (cf. Section 4.2.2) **Fitness rating: 3 points.**

Target Platform Dependence The description of the target platform is limited to the options of the Simulink Coder. Overall, the concept is therefore only applicable to platforms that are supported by the MATLAB environment or its extension by hardware support packages. There is the opportunity to create custom system target files, but this is not intended for the code generation concept and its flexibility cannot be reliably estimated without further analysis, due to its complexity. **Fitness rating: 2 points.**

Code Generation Goal The goal is to produce code that contains the structural and behavioral specification of the software originating from a MechatronicUML PIM instance, possibly complemented by the implementation of continuous component behavior using MATLAB/Simulink. Additionally, the link layer provides an abstraction to be realized by a concrete middleware implementation depending on the target platform, thus, supporting distributed deployments **Fitness rating: 3 points.**

Include Code Libraries While the MechatronicUML PIM already supported modeling features to include external operations in version 0.4 [BDG+14], these features were not yet tailored to include entire libraries. Even though there might be some limitations in the MechatronicUML, MATLAB/Simulink also allows to include existing C or C++ libraries [Mat22a]. So overall, there is the opportunity to reuse legacy library code. **Fitness rating: 3 points.**

Modeling Feature Coverage The MATLAB/Simulink approach was originally designed for version 0.4 of the MechatronicUML PIM [HRB+14]. There is no information whether the newer implementations of the approach targeted version 1.0. But regarding the MechatronicUML PIM in version 0.4, the researchers provide a detailed list of supported features [Hei15; HRB+14]. Features of the MechatronicUML PIM that are not supported include: multi-ports, reconfigurable components, and some specifics of RTSCs like constants, do-actions, and if statements and loops in actions or transitions. Arrays are also only partially supported. Additionally, with respect to MATLAB/Simulink features, the version in use for the first implementation was MATLAB Release R2009b [Hei15]. Overall, this means that not all MechatronicUML PIM features may be used. However most significant features are supported, and these features are sufficient to model a vast majority of scenarios, potentially using workarounds to mimic missing features with those features that are supported. **Fitness rating: 2 points.**

6.3.2 Implementation of the Code Generator

For the implementation of the MATLAB/Simulink approach, there are two major tasks of the code generation process to be distinguished. *T2: Create Code from MATLAB/Simulink* is a black box from the implementation perspective, as it is based on the Simulink Coder, which is a commercial tool. Therefore, this subsection mainly focuses on *T1: MechatronicUML to MATLAB/Simulink*.

Open-Source In the same way as there is only a documentation of the initial transformation concept, there is also only open-source access to the corresponding initial implementation of the MechatronicUML-to-MATLAB/Simulink transformation. It is available on GitHub, and

the repository is called mechatronicuml-simulinkadapter[7]. There is no open-source access to the aforementioned reimplementations of the approach (cf. Section 4.2.2). **Fitness rating: 2 points.**

Code Base This openly accessible code base has not been updated since 2016 and does not contain any build scripts. Also, the code base does not match the documentation from 2015 in all details (see [Hei15]): for instance, the package structure was changed from `de.uni_paderbonr_fujaba.muml.*` to `org.muml.*` and some package names were also changed completely. The project exposes its functionality via a plugin for the MechatronicUML Tool Suite and is implemented using Java 6. It can be launched as an Eclipse plugin. The repository contains all functionality for the model-to-model and model-to-text transformation steps, as well as the other auxiliary steps. The model-to-model transformation is implemented using Triple Graph Grammars and the generation of the Simulink files is implemented using Xpand, a model-to-text tool specialized on EMF models, which are used for metamodeling [HRB+14]. Additionally, it includes a user interface wizard for integration into the MechatronicUML Tool Suite. Lastly, the repository contains a meaningless README file that does not contain any information about the usage, development or set-up of the Eclipse-plugin. The transformation could not be executed during the analysis as part of this thesis. **Fitness rating: 1 point.**

Developer Documenation The publications technically serve as developer documentation, but they neither describe the most recent state of the code base, nor do they go into detail about development tools. Yet, they describe the initial implementation of the MechatronicUML-to-MATLAB/Simulink transformation with a lot of detail. The code base does not link to any additional documentation, and the developer documentation of the MechatronicUML Tool Suite[8] does also not contain a section about the MATLAB/Simulink approach or the Simulink adapter[9]. **Fitness rating: 2 points.**

Extension Points Regarding the MechatronicUML -to-MATLAB/Simulink transformation, there are no documented extension points. Regarding the Simulink Coder, the creation of custom system target files an extension point of the code generation capabilities [Mat22b]. Other than that, there is no indication of possible custom extensions to the Simulink Coder. **Fitness rating: 2 points.**

Degree of Concept Implementation The documentation states that the approach was implemented as part of the MechatronicUML Tool Suite version 0.5 [Hei15]. Regarding *T1: Mechatronic-UML to MATLAB/Simulink*, the code base contains the implementation of a wizard which includes the transformation into the user interface of the MechatronicUML Tool Suite. Even though the source code is not able to be built, nor is the plugin able to be launched without errors, the code base indicates that the overall concept is in fact implemented. Heinzemann confirmed that the Simulink adapter was once used to generate Simulink files which were then applied to generate code with the MATLAB/Simulink tools [Hei21]. **Fitness rating: 3 points.**

[7] https://github.com/fraunhofer-iem/mechatronicuml-simulinkadapter

[8] https://trac.cs.upb.de/mechatronicuml/wiki/FujabaDevelopment

[9] that is what the GitHub repository is called and how the feature is referred to in the developer documentation of the MechatronicUML Tool Suite.

6.3.3 Usability of the Code Generator

For the usability analysis, it is important to note that no working executable of the overall approach is available to be analyzed. Both parts of the code generator are analyzed independently, and the analysis of the MechatronicUML-to-MATLAB/Simulink transformation are limited to the initial wizard in the user interface that was accessed using a development version, and to some user documentation. The MATLAB/Simulink tooling on the contrary is fully available for the analysis.

Integration The code generator is distributed over two tools: The MechatronicUML -to-MATLAB/Simulink transformation is integrated in to the MechatronicUML Tool Suite, and the Simulink Coder is integrated into the MATLAB tooling. So there is no standalone implementation of the code generation approach. Also, the MATLAB tools are commercial tools and not freely available for installation and use.

User Interface Both tools have a graphical user interface into which the transformation and code generation functionalities are integrated.

Tool Support For the MechatronicUML-to-MATLAB/Simulink transformation, there is no executable version available. The most recent version of the MechatronicUML Tool Suite which is version 1.0 does not include this feature. The Simulink Coder is available for purchase and still maintained. **Fitness rating: 2 points.**

Documentation and Manuals There is a user documentation for the MechatronicUML -to-MATLAB/Simulink transformation available online[10], but it is only very brief and states that a more detailed manual was intended to be created. Other than that, the aforementioned publications include some hints on how to use the tool (see [HRB+14] and [Hei15]). For the Simulink Coder, there is some detailed user documentation online which includes the previously cited [Mat22d] and [Mat22e]. **Fitness rating: 2 points.**

Level of Automation The entire process is supported via the MechatronicUML Tool Suite and the MATLAB tools. The part of the MechatronicUML Tool Suite is only be analyzed partly, due to unavailability of a working executable, but the documentation suggests a high level of automation with limited user interaction for providing the model files and additional information. The latter also applies to the MATLAB tooling: the code generation process demonstrates a high level of automation.

Traceability and Debugging The traceability is not be examined directly. But considering that the code generation is distributed over several tools, there is typically no support for end-to-end traceability of MechatronicUML model elements to source code (or vice versa).

Incrementality The incrementality is not examined due to the lack of an executable version. The documentation does not mention incrementality either.

[10] https://trac.cs.upb.de/mechatronicuml/wiki/FujabaUserGuide/TutorialMUMLMatlab

6.3.4 Generated Code

The MechatronicUML Tool Suite includes some examples, that contain MechatronicUML modeling artifacts as well as MATLAB/Simulink files[11]. As the MechatronicUML-to-MATLAB/Simulink transformation could not be used during the analysis, these example files are analyzed. The MATLAB/Simulink files contained in the examples cab be opened using the MATLAB Release R2009b, but some errors were detected which also prohibits the usage for simulation or code generation. Additionally, MATLAB Release R2021b was used to inspect the code created by the Simulink Coder. However, the MATLAB/Simulink files from the MechatronicUML Tool Suite examples cannot be opened correctly in this newer MATLAB environment due to incompatibility. Therefore, examples provided by the MATLAB environment are used to generate code instead for the purpose of this analysis.

Code Completeness The Simulink Coder generates the entire source code for the MATLAB/Simulink model, including structural and behavioral code. There is no need to add manual implementations. However, looking at the overall approach, the Simulink Coder cannot generate the communication middleware source. As mentioned, the MechatronicUML -to-MATLAB/Simulink transformation introduces a link layer as abstraction of the communication implementation (cf. Section 4.2.2). A concrete implementation has to be added, and this can be done using the MATLAB/Simulink tools. So overall, the approach has the potential to generate the entire source code for a distributed deployment. **Fitness rating: 3 points.**

Access to Code The source code is stored in a directory of the MATLAB environment's working directory. Therefore, it is entirely accessible to the user. However, the user cannot customize the target location for the code generation, and new runs overwrite the old source code. Therefore, source code of previous runs has to be copied or tracked otherwise if required. **Fitness rating: 3 points.**

Efficiency of Generated Code The efficiency of the code is not analyzed as part of this work. However, the Simulink Coder configuration supports different configuration options for faster builds or faster runs, thus, indicating different optimization options to choose from.

Conformance to Industry Standards The code is not analyzed specifically to detect the conformance to standards. But the documentation does not reference or mention any industry standards or coding conventions.

Coupling to the Target Platform A comparison of the generated code using the same Simulink model with different system target files indicates that the functional code remains mainly unchanged. However, the code generation does not strictly separate platform-specific from platform-independent implementation, so all files are still updated an regenerated. This shows that the code is specifically designed for one target platform, but may be altered manually for compatibility with another platform with the required expert knowledge.

[11] https://github.com/fraunhofer-iem/mechatronicuml-examples

6.4 Analysis Conclusion

This section concludes the analysis of the previous model-based code generation approaches that use the MechatronicUML as the source modeling language. It compares how well these previous approaches fit to the ideal candidate (see Section 6.1), and sums up strengths and shortcomings. Hence, this section also summarizes the answers to **RQ2.2:** What is the state of the previous MechatronicUML-based code generation approaches?

First of all, the platform-modeling approach introduces a sophisticated concept for the code generation, including the detailed modeling of any desirable, distributed target platform using the MechatronicUML HPDM. It is able to generate source code from a MechatronicUML PIM instance supporting most modeling features of the MechatronicUML PIM version 1.0. Furthermore, it supports platform-specific modeling that enables the generation of container code and a communication middleware. The approach is implemented by several plugins for the MechatronicUML Tool Suite and all source code is publicly accessible. A working executable exists as part of the MechatronicUML Tool Suite version 1.0. Its major shortcomings with respect to the ideal candidate approach (cf. Section 6.1) are the poor developer documentation and the lack of explicit user manuals.

Secondly, the MATLAB/Simulink approach suggests to employ the widely used tool set of MATLAB/Simulink for code generation. The approach is based on a transformation from the MechatronicUML PIM to MATLAB/Simulink and on the code generation features of the MATLAB tooling including the Simulink Coder. Furthermore, the transformation to MATLAB/Simulink allows to make use of additional features and benefits of MATLAB for software that is initially

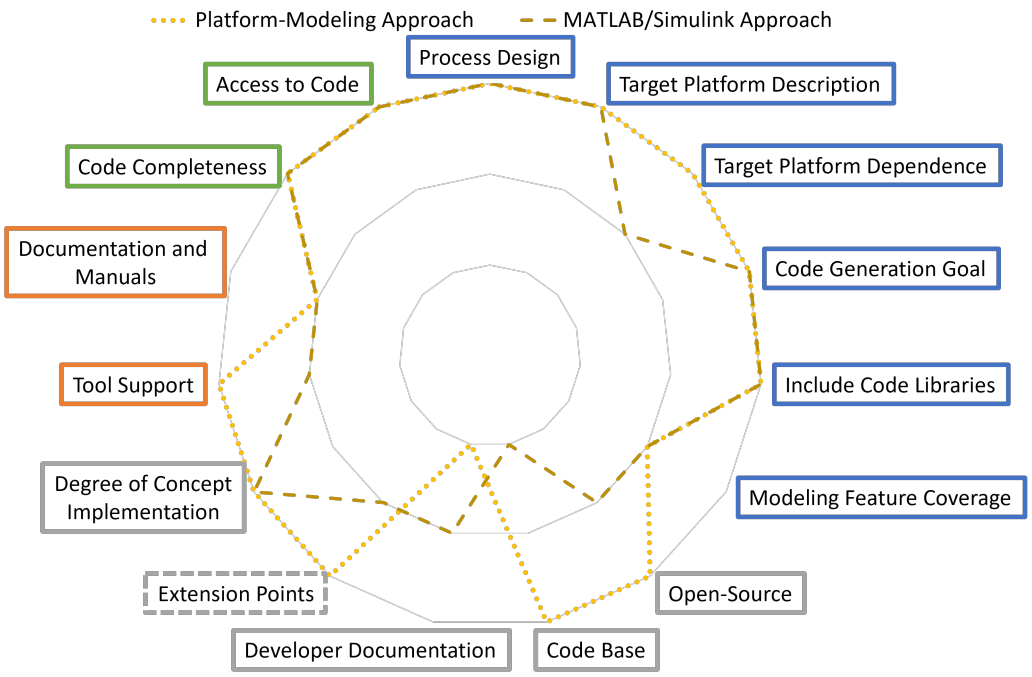


Figure 6.5: The analysis results of the platform-modeling approach and MATLAB/SIMULINK approach in comparison

designed with the MechatronicUML. The MATLAB/Simulink approach supports most features of the MechatronicUML PIM version 0.4 and is implemented using the MechatronicUML Tool Suite and the MATLAB tools. The source code of the MechatronicUML Tool Suite is publicly accessible, however, only the source code and documentation of the first, outdated version of the MechatronicUML -to-MATLAB/Simulink transformation are available. Besides the outdated code base and outdated documentation, another major shortcoming with respect to the ideal candidate approach (cf. Section 6.1) is that there is no executable version of the MechatronicUML -to-MATLAB/Simulink transformation, no working build process, and no up-to-date implementation in the MechatronicUML Tool Suite.

Both approaches do not manage to fulfill all requirements of the ideal candidate approach entirely. The degree of fulfillment of the ideal candidate's properties is depicted in Figure 6.5. The chart sums up and compares the analysis of both approaches w.r.t. the selected facets. The result shows that in general, both approaches fit well to the presented notion model-based code generation and the taxonomy's facets, as a fitness rating of 0 points was never assigned (cf. Section 6.2). Moreover, both approaches achieve the desired code generation goal to conceptually enable code generation for a distributed deployment of a CPS. Similarly, the produced code of both approaches is complete and accessible. The major differences arise in the implementation and usability, where the platform-modeling approach has the more up-to-date code base and better tool support while the MATLAB/Simulink approach cannot supply an executable implementation. Only regarding the developer documentation, the platform-modeling approach falls slightly behind the MATLAB/Simulink approach.

Besides the strengths and shortcomings identified above, there are some more differences between the approaches worth considering:

Required Expertise For using both approaches, expertise about the software and hardware components is required. But a key difference is that for the platform-modeling approach, detailed knowledge about the hardware platform as well as its modeling using the MechatronicUML HPDM are required, while for the MATLAB/Simulink approach, diverse knowledge of MATLAB and its tools is necessary. This includes general knowledge of the MATLAB environment, and also specific knowledge about operating the Simulink Coder.

Dependence on Proprietary Solutions The platform-modeling approach mainly depends on the MechatronicUML and its tool suite, which are open source an publicly available. Only the generation for the DDS specific middleware artifacts is a minor exception. The MATLAB/Simulink approach on the other hand largely depends on proprietary tools like the Simulink Coder, and proprietary formats such as the Simulink file format, which may be subject to change, incompatibility or ceasing tool support.

Initial Goal and Motivation Additionally, the MATLAB/Simulink approach is not originally intended for code generation (cf. Section 4.2.2). While the platform-modeling approach is designed for code generation, the MATLAB/Simulink approach is enabled by the MechatronicUML -to-MATLAB/Simulink transformation. The motivation for creating this transformation is not code generation in the first place.

Secondary Benefits The MATLAB/Simulink approach offers a couple of benefits next to code generation: It allows the user to model a software using the MechatronicUML , and then porting it to MATLAB/Simulink, where all the additional tools and features for simulation may be used. Most notably, the behavior of continuous components may be implemented in

MATLAB, which is a common tool used in control engineering [HRB+14]. In contrast, the platform-modeling approach is limited to the capabilities of the MechatronicUML and its tool suite.

Up-to-Dateness The available documentation and implementation of the MATLAB/Simulink approach are very outdated. While the documentation and code base of the platform-modeling approach are also not entirely up-to-date, the implementation still adheres to the MechatronicUML version 1.0 and an executable version is also contained in the MechatronicUML Tool Suite Release 1.0. This is not the case for the MATLAB/Simulink approach, which was not maintained for the latest MechatronicUML Tool Suite release (cf. Section 3.3).

Platform-Specific Modeling While both approaches introduce a middleware layer to abstract from the communication and thus consider distributed deployments, the platform-modeling approach includes the modeling of platform-specific configurations explicitly. While it does not support the Arduino-based target platform (see Section 2.2) out of the box, i.e. pre-configured and ready to be used, the platform-modeling approach allows to add reusable platform-specific configurations and code generation functionality for a new target platform. This allows the fully automated generation of platform-specific source code including communication middleware artifacts based on a sophisticated PSM.

In conclusion, the platform-modeling approach has the edge over the MATLAB/Simulink approach in direct comparison. Especially as the focus is on code generation, and the secondary benefits of the MATLAB simulation environment play less of a role. Nonetheless, the analysis demonstrated that none of the two code generation approaches fulfills all desired properties of the ideal candidate. Consequently, none of the previous approaches can be used to implement the desired application scenario in its current state. Therefore, in order to obtain a code generator for the application scenario, the most promising approach is to reuse and extend the platform-modeling approach. The next section describes this decision in detail.

6.5 Code Generation Concept and Design Decisions

Based on the previous analysis of existing code generation approaches, this section describes the code generation concept that can realize the application scenario (see Chapter 2). The analysis shows that conceptually, both previous approaches are valid for the distributed deployments and may generally be extended for the Arduino-based target platform. But the platform-modeling approach has a working implementation, and the concepts and its implementation are extensible. Thus, the decision is to reuse and extend the platform-modeling approach instead of building a code generator from scratch. The reasons for this decision are the following: It appears infeasible to create a concept that is similarly sophisticated as the platform-modeling approach which includes the description of the target platform as well as the allocation engineering and platform-specific modeling. While not being a perfect match, the platform-modeling approach still has a high coverage of the ideal candidate's properties. It appears a lot more promising to improve the shortcomings of the platform-modeling approach and extend it where necessary than implementing a new code generator from scratch.

Based on the analysis, it can be concluded that parts of the platform-modeling approach are suitable to be reused entirely. Other steps of the platform-modeling approach have to be adapted or extended in order to support the desired application scenario and to improve the maturity of the approach w.r.t. the weaknesses identified in the previous analysis. Thus, this section answers the research question **RQ3.1:** Which (parts of) previous approaches are reused, and why or why not? As depicted in Figure 6.6, the concepts and implementations of the following tasks are reused:

T1: Platform Modeling The entire platform modeling is reused. The MechatronicUML Tool Suite includes the MechatronicUML Hardware Platform Description metamodel for platform modeling and the editors for the four modeling perspectives (cf. Section 3.5). As briefly demonstrated in Section 3.5, the modeling options are sufficient to model the desired target platform.

T2: Allocation Engineering The allocation engineering is also reused in its current state. The produced allocation specification maps software components to the hardware platform and is directly applicable to the application scenario.

T3.1: Model Device Access and T3.5: Generate Device Access Code Modeling the device access and generating respective code is implemented in a generic way, so it can also be applied to the desired scenario.

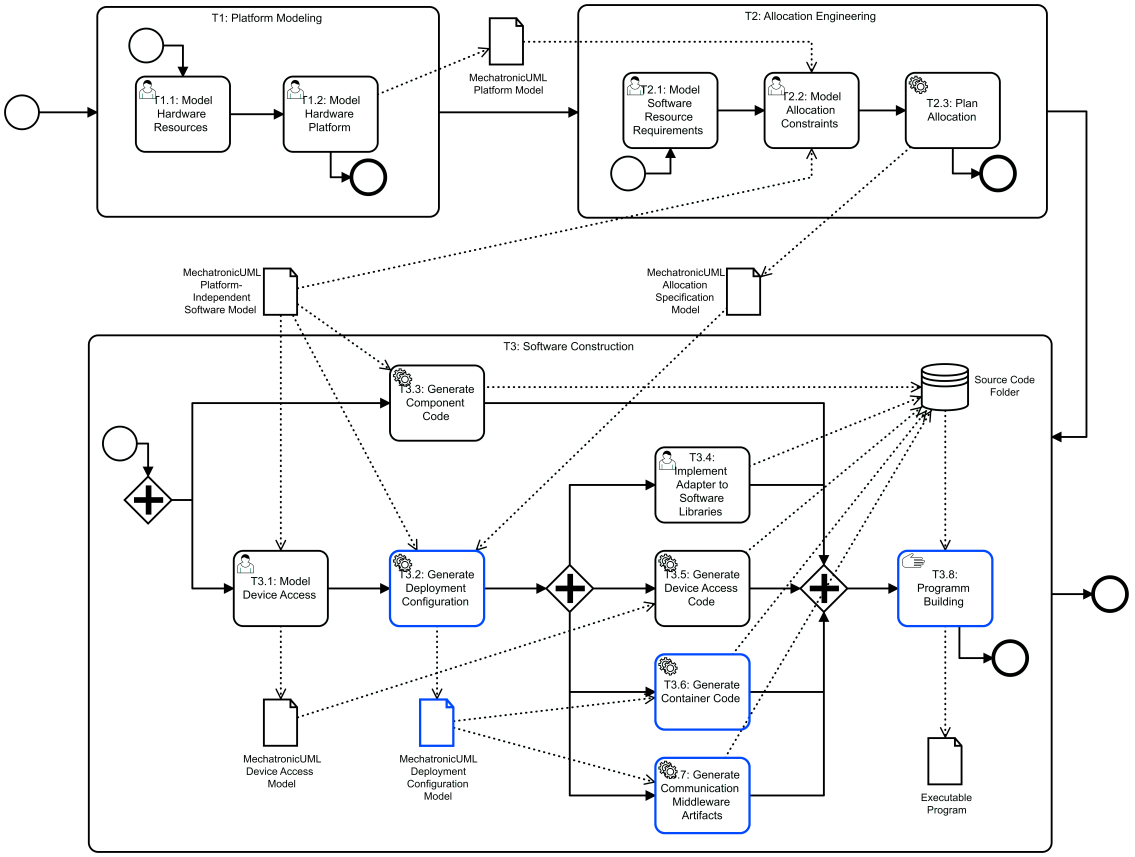


Figure 6.6: The resulting code generation concept with the tasks to be adapted marked with blue border.

T3.3: Generate Component Code A characteristic strength of the platform-modeling approach is the separation of platform-independent code from the platform-specific implementation. Consequently, the generation of the platform-independent component code is reused without modifications. However, the implementation of the clock functionality of RTSCs (cf. Section 3.4.3) is not platform-independent as it depends on system calls to retrieve the processor time. A platform-specific implementation of the clock functionality is added together with the remainder of the platform specific code.

Additionally, as also visualized in Figure 6.6, the following tasks of the platform-modeling approach need an extension to be applicable to the application scenario:

T3.2: Generate Deployment Configuration The platform-modeling approaches MechatronicUML Deployment Configuration metamodel specifies only two configuration types for port instances: A configuration for local communication or a configuration using DDS for remote communication. However, the DDS is not applicable to the Arduino-based hardware platform as it is too resource intensive. Therefore, an alternative for remote communication has to be implemented which includes adapting the metamodel and the corresponding model transformation task. Also, there is no implementation for simple wired communication protocols such as I2C which is used in our target platform.

T3.6: Generate Container Code The container code provides the platform-specific environment for the component code to be executed. This code generation step has to be adapted to the Arduino platform. A new container code generation is implemented to capture the technical particularities of the Arduino-based target environment. This also includes implementing a the aforementioned clock functionality.

T3.7: Generate Communication Middleware Artifacts The generation of the communication middleware code includes the usage of proprietary tools for the DDS in the original platform modeling approach [Poh18]. These tools are not required any longer. Instead, appropriate code for Message Queuing Telemetry Transport (MQTT) and I2C is generated and from an implementation point of view, the generation is effected in one step together with the container code. For the configuration of the middleware code, the MechatronicUML Deployment Configuration metamodel is used. It is adapted to support the configuration for MQTT and I2C, so the communication middleware generation has to be extended to deal with the new metamodel elements as well.

T3.8: Program Building Lastly, the program building has to be adapted for Arduino by employing some platform-specific tools.

While most of these extensions simply require adaptions to support our target platform, replacing the usage of DDS as communication middleware requires some conceptual decisions. As the target platform is equipped with WiFi capabilities, WiFi is the foundation for the wireless remote communication. Three options appear feasible as an alternative to a resource intensive DDS implementation:

1. The first option is **direct communication** between the cars **using TCP**. This requires the configuration of the port instances so that they know the Internet Protocol (IP) address of the respective hardware port's network interface that they communicate with. The RTCP that specifies the interaction may be directly used to implement a protocol of communication.

2. The second option is the usage of **DDS for eXtremely Resource Constrained Environments (DDS-XRCE)** [Obj19]. This OMG specification defines a protocol to connect an environment with constrained resources to a DDS network. Arduino microcontrollers are such a resource constrained environment, and there are suitable implementations for Arduino[12]. The DDS-XRCE specification offloads a lot of the computation to the DDS-XRCE agent which acts as a server to the DDS-XRCE client. The client is lightweight enough to be executed on a microcontroller or similar environment. The DDS-XRCE agent however has to be deployed to a hardware platform with more computing power.
3. Alternatively, **MQTT** may be used as communication middleware [MQT22]. MQTT is a messaging protocol widely used in low resource environments such as microcontrollers. There are different implementations of clients for Arduino[13]. The MQTT also requires additional computing resources to deploy an MQTT server that the clients can connect to and interact with.

While the first option allows a direct implementation of the RTCP and does not require extra hardware resources, it does not provide the benefits of a standardized communication middleware solution. Furthermore, such implementations are cumbersome, error-prone, and difficult to maintain. The second and third option both use a standardized solution, but require compute-intensive middleware components that are deployed to additional hardware resources.[14] Also, both options are based on a publish-subscribe communication model which requires an equivalent effort to map the point-to-point communication of the MechatronicUML PIM to the respective communication pattern. In this regard, the options are quite similar. However, the primary benefit of DDS-XRCE is to connect a resource constraint device to a larger, existing DDS network in order to make us of this network's benefits. This is not the case in our application scenario. Furthermore, the third option using MQTT is more straightforward to implement. Thus, as the focus of this thesis is on the code generation, but not on choosing the best communication middleware for distributed CPS, using MQTT is the preferred option. Some more details of MQTT and the mapping of MechatronicUML ports and messages to the MQTT communication scheme are explained in Section 7.1.3.

In addition, with the implementation presented in this thesis, an option for the wired communication between ECUs is introduced. As described in Section 2.2, the two Arduino microcontrollers on each robot car are connected via an I2C bus. Consequently, the implementation must also consider message exchange using I2C and provide an appropriate middleware implementation. Section 7.1.4 provides some more details about I2C.

All in all, the implementation using MQTT and I2C verifies the overall concept of MechatronicUML component containers and the communication middleware as introduced by Pohlmann. When choosing DDS as communication middleware, they argue that "it is not possible to define one specific, best-fitting middleware for implementing the technical concerns of MechatronicUML " [Poh18]. Hence, the communication middleware may also be exchanged again in the future in a similar fashion as demonstrated in this work, depending on what middleware a specific application scenario

[12] E.g., https://micro.ros.org/docs/concepts/middleware/Micro_XRCE-DDS/

[13] E.g., https://www.arduino.cc/reference/en/libraries/mqtt-client/, https://www.arduino.cc/reference/en/libraries/pubsubclient/ or https://www.arduino.cc/reference/en/libraries/adafruit-mqtt-library/

[14] Both the DDS-XRCE agent or the MQTT server could be deployed on an edge device that is available in the environment. In the concrete application scenario, this could be a road section controller, which may also manage the WiFi network the vehicles connect to. Such road infrastructure may also provide additional services.

may require. Additionally, the user of the code generator should be enabled to choose the type of communication middleware they want to employ; i.e., the usage of DDS should still be available, and MQTT and I2C should be added as an additional option.

Consequently, the overall concept and process for the code generation are reused from the platform-modeling approach. Individual steps are adapted and extended as described above. In addition, the following aspects are improved because the analysis revealed particular shortcomings of the platform-modeling approach.

Usability of the Code Generator: Documentation and Manuals There should be an explicit user manual on how to use the code generation and the examples. The code generation process is quite complex and thus requires some expertise. An improved user documentation decreases the startup time for user of the platform-modeling approaches code generator.

Usability of the Code Generator: User Interface The options in the user interface regarding code generation are limited. As mentioned before, the user should be able to select the kind of communication middleware they want to use for code generation. I.e., next to the already existing and thus far implicitly chosen option to use DDS, the user may also choose MQTT and I2C in the future.

Implementation of the Code Generator: Developer Documentation The lack of developer documentation makes it hard to get an overview about the code base and the components of the MechatronicUML Tool Suite. Therefore, an developer documentation that covers at least the tooling and implementation described in this thesis is prepared.

The additional developer documentation and user manuals that are created as part of this work can be accessed on GitHub[15].

[15] https://github.com/SQA-Robo-Lab/MUML-CodeGen-Wiki

7 Implementation of the Code Generator

This chapter describes the implementation of the code generator based on the previously defined code generation concept (see Section 6.5). The goal is to adapt and extend the existing platform-modeling approach so that its shortcomings are overcome and the code generation may be applied to the application scenario of this thesis. First, the requirements for the code generation are summed up. This list of requirements answers the research question **RQ3.2:** What are the missing capabilities of a MechatronicUML-based code generator for the desired application scenario? The missing capabilities are specified by the following requirements:

REQ1 The MechatronicUML Deployment Configuration metamodel must support configurations for MQTT and I2C. These configurations must reflect the configuration of individual port instances for communication and collect the required information for appropriate communication middleware code to be generated.

REQ2 The task *T3.2: Generate Deployment Configuration* of the original platform modeling approach must be adapted to create the newly specified port instance configurations for MQTT and I2C in the transformation. Furthermore, users must be able to choose between different middleware options via the user interface of the MechatronicUML Tool Suite.

REQ3 The task *T3.6: Generate Container Code* of the original platform-modeling approach must be replaced to make the container code applicable to the Arduino environment. A platform-specific clock implementation must also be provided.

REQ4 The task *T3.7: Generate Communication Middleware Artifacts* of the original platform-modeling approach must be replaced by a code generation that supports MQTT and I2C as communication middleware for Arduino.

REQ5 The task *T3.8: Program Building* of the original platform-modeling approach must be adapted to the Arduino environment.

The implementation of these requirements is explained in this order throughout the following sections. This thesis does not aim to provide an implementation that is able to solve all modeling edge cases the MechatronicUML may allow. Instead, the goal of the implementation is, equivalently to the previous work by Pohlmann, to provide a proof of concept. The implementation is prototypical in the sense that it serves to demonstrate the feasibility of the aforementioned concepts and can, as a minimal feature set, be applied to the application scenario introduced in Chapter 2. Hence, the presented implementation is incomplete regarding the handling of the MechatronicUML modeling features and is only tested manually focusing on the introduced application scenario.

This section explains the implementation of the aforementioned requirements and also explains the underlying concepts of the platform-modeling approach which are reused or adapted in this work. Additionally, new concepts are introduced to implement the aforementioned requirements. The implementation seeks to put the concepts in place so that future endeavors may reuse the concepts, add functionality, and increase the maturity of the implementation.

After the description of the newly implemented functionality in the first five subsections, Section 7.6 briefly explains how the new functionality is integrated into the MechatronicUML Tool Suite and how the new plugins may be used. Overall, the remainder of this chapter provides answers to research question **RQ3.3:** How are these missing capabilities implemented? This is described in the following.

7.1 Deployment Configuration

This section describes how the MechatronicUML Deployment Configuration metamodel is extended in order to support MQTT and I2C. This corresponds to the implementation of **REQ1** and is explained in Section 7.1.1. Furthermore, Section 7.1.2 describes how continuous and hybrid ports are dealt with for code generation. This is required for mapping the message-based communication of the MechatronicUML to MQTT, which is described in Section 7.1.3, and the mapping to I2C in Section 7.1.4.

7.1.1 Metamodel Extensions

The MechatronicUML Deployment Configuration metamodel serves the purpose to describe the deployment of a CPS that is modeled using the MechatronicUML (cf. Section 4.2.1). As Figure 7.1 visualizes, the MechatronicUML Deployment Configuration metamodel is associated to other MechatronicUML metamodels for this purpose. The `DeploymentConfiguration` represents a concrete `SystemAllocation` and consists of `ecuConfigurations` that are each associated to a `StructuredResourceInstance`, i.e., an ECU. These two constructs represent the overall structure for the code to be deployed: each ECU requires configuration and code to be generated. In order to operate according to the MechatronicUML PIM model, each ECU must be execute the functionality that is encapsulated in the components which are allocated to the respective ECU. The `ComponentContainerConfiguration` contains the configuration required for one specific component type. A `ComponentInstance` in the MechatronicUML PIMmetamodel represents a concrete instance of such a component type, fixing the variable parts of a component such as the specific number of ports. Therefore, the MechatronicUML Deployment Configuration metamodel introduces the `ContainerComponentInstanceConfiguration` that represents this refinement. As it is also possible that multiple component instances of the same type are allocated to one ECU, the `ComponentContainerConfiguration` consists of one or many of these `containerComponentInstanceConfigurations`.

For configuring the communication, the platform-independent `PortInstance` entities need an equivalent in the MechatronicUML Deployment Configuration metamodel. Firstly, the abstract class `PortInstanceConfiguration` is associated with a platform-independent `PortInstance` and with a `HWPortInstance` of the platform model. Secondly, its specializations capture the configuration

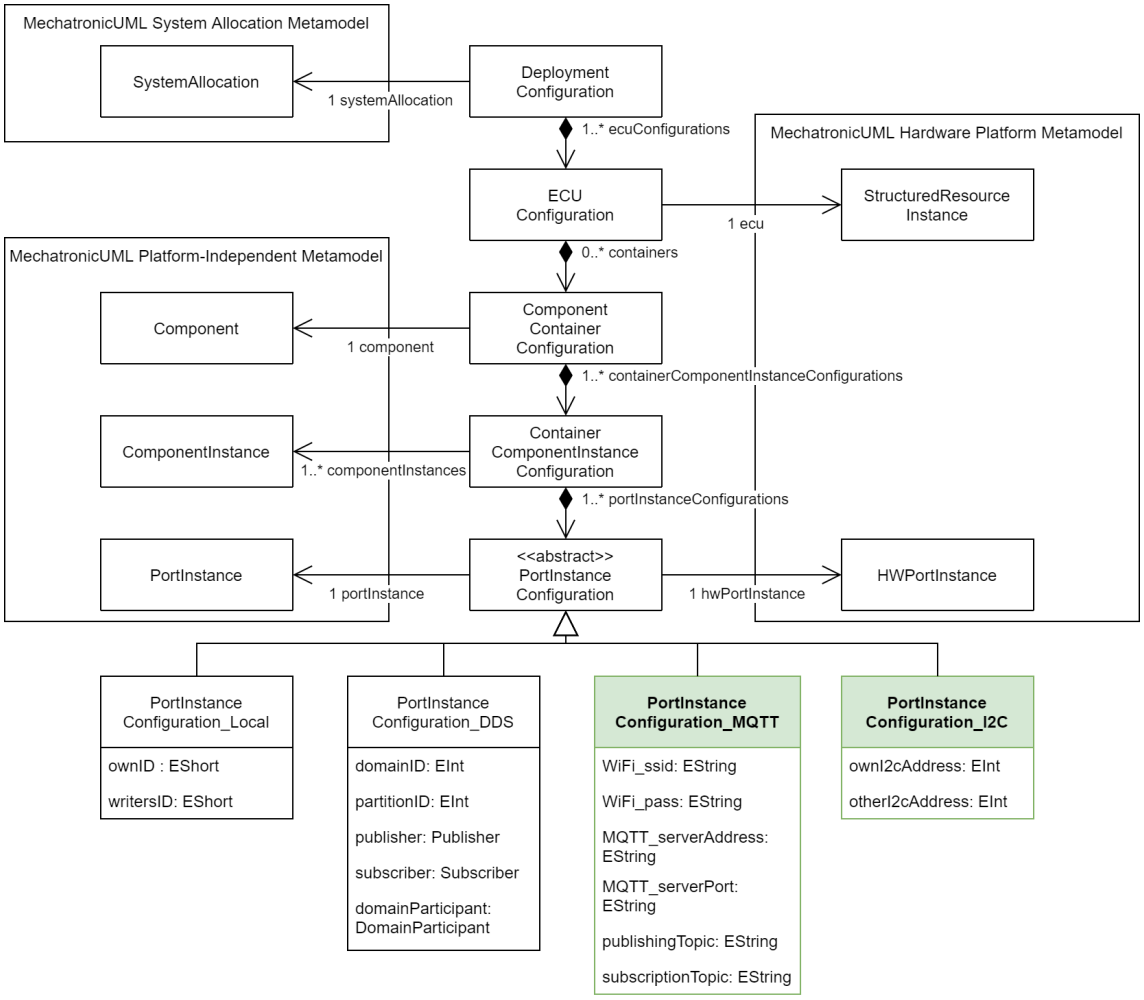


Figure 7.1: The extended MechatronicUML Deployment Configuration metamodel and its associated metamodels, with the metamodel extensions highlighted in green.

properties that are required for the specific communication options. Pohlmann introduce the two kinds `PortInstanceConfiguration_Local` and `PortInstanceConfiguration_DDS` [Poh18]. They represent local communication, i.e., the components are allocated to the same ECU, and communication using the DDS respectively.[1] Multi-ports are not supported by the code generation (cf. Section 6.2). This is visible in the platform-specific modeling: The configurations are explicitly designed for the one-to-one communication of single ports.

One-to-one communication does not prohibit that multiple communication partners use the same communication media, e.g., the same I2C bus. Furthermore, one component can make use of the same communication infrastructure several times: that requires several ports to capture the configuration for each port and communication partner. In order to allow the wireless MQTT and wire-based I2C, two more kinds of `PortInstanceConfiguration` are introduced:

[1] For details about the configuration for local and DDS-based communication, see [Poh18].

PortInstanceConfiguration_MQTT The configuration of a port instance for MQTT requires the configuration of a WiFi network (`WiFi_ssid` and `WiFi_pass`), and the configuration of an MQTT server (`MQTT_serverAdress` for the server's IP address and `MQTT_serverPort` for its port). As MQTT uses a publish-subscribe communication model (cf. Section 7.1.3), the configuration also includes a `publishingTopic` and a `subscriptionTopic`. These topics serve as identification of the sender and receiver to map the port-to-port message exchange of the MechatronicUML to the publish-subscribe model. This mapping is detailed in Section 7.1.3.

PortInstanceConfiguration_I2C In order to make use of I2C, the port instances require their own address on the I2C bus (`owni2cAddress`) as well as their communication partner's address (`otheri2cAddress`). The mapping of the message-based communication of the MechatronicUML to the properties of I2C is described in Section 7.1.4.

The MechatronicUML Deployment Configuration metamodel is implemented in the MechatronicUML Tool Suite. As the containers are the core part of the metamodel, the metamodel is named *muml_container* in the implementation. It is part of the *PSM* repository (cf. Table 6.2). The metamodel is implemented with the EMF tools, and the extension introduced here is also published in the GitHub repository[2].

7.1.2 Continuous Ports in the PSM

Pohlmann introduces a concept for handling continuous ports in the platform-specific modeling [Poh18]. This concept of transforming continuous and hybrid ports to discrete ports is described in Appendix B.1 and it is implemented in the component code generation (see T3.3 in Figure 6.6). This is the generation of the platform-independent code that contains the behavior, i.e., the implementation of the RTSCs. The component code generation is not altered in this work.

But to be compatible with the platform-independent code, the platform-specific implementation, which is the focus of this thesis, must adhere to the same concepts: Hybrid ports and continuous ports are regarded as discrete ports with (i) one message type, (ii) a buffer of size 1 and (iii) the buffer overflow strategy `DISCARD_OLDEST_MESSAGE_IN_BUFFER`. Thus, all communication is modeled in a message-based way, and when mapping this message-based communication behavior of the MechatronicUML to concrete communication implementations, there is no conceptual difference between the discrete, hybrid and continuous ports.

7.1.3 MechatronicUML-based Messages via MQTT

The communication with MQTT follows a publish-subscribe model [MQT22]. Figure 7.2 shows a simplified outline of this communication: all messages pass via an *MQTT server*[3]. When an *MQTT client* publishes a message, it sends the message to a *topic*. The topics in MQTT are specified as character strings, and MQTT allows hierarchical nesting of topics using a forward slash (see `topic/subtopic` in Figure 7.2). MQTT clients can subscribe to one or more topics. Moreover, MQTT clients can publish to several topics, while being subscribed to multiple topics at the same

[2] see https://github.com/fraunhofer-iem/mechatronicuml-psm
[3] also: MQTT broker

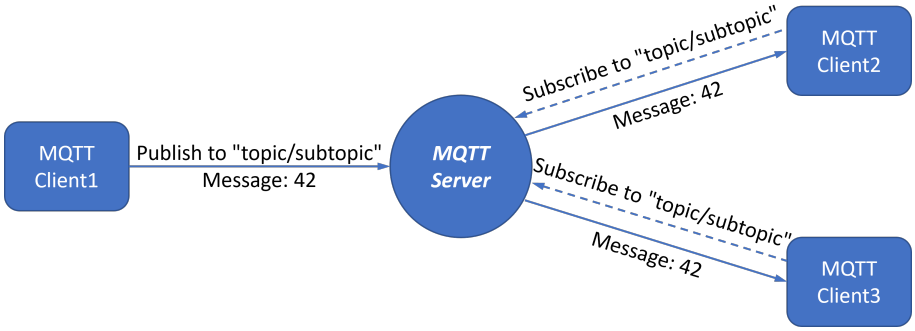


Figure 7.2: The MQTT publish-subscribe communication model based on [MQT22].

time. Whenever the MQTT server receives a message for a particular topic, it forwards it to all subscribers that previously subscribed to this topic. In the example in Figure 7.2, the message *42* is sent by *MQTT Client1* and received by *MQTT Client2* and *MQTT Client3* assuming that the clients are both already subscribed to *topic/subtopic* before the message is published. As clients do not communicate directly but indirectly via topics, the MQTT is not explicitly designed for point-to-point communication.

The MechatronicUML models the communication between ports via messages. For single ports, this fosters a point-to-point communication model. Therefore, a port instance requires the information about the topic to publish a message to so that the intended addressee, i.e., the connected port instance, receives this message. Therefore, topic names must be unique for each port instance, and a port instance must be configured with the topic of its connected port instance. Furthermore, a port instance configuration for MQTT also includes its own topic that it subscribes to in order to receive messages from a connected port instance.

Furthermore, MQTT servers support different methods for authentication. In case authentication is used, the respective information must be added to the configuration as well. CI the current state of implementation, no authentication is used. Similarly, other communication options besides WiFi might be used to employ MQTT, and corresponding configuration information could be added in the future. Additionally, MQTT may be used to implement multi-ports in the future, as its communication style fosters one-to-many or many-to-many message exchange.

7.1.4 MechatronicUML-based Messages via I2C

I2C is a wired synchronous serial bus [eLa18]. From the wiring perspective, the bus consists of a data line (SDA) and a clock line (SCL) for synchronization. Moreover, all nodes share a common ground (GND). This is visualized in Figure 7.3. In normal mode, the transmission speed of an I2C bus is $100 kBit/s$. From a logical perspective, I2C follows a master-slave communication approach, where only master nodes can initiate the communication. Thus, there must be at least one master on an I2C bus. However, a node can be a master and a slave node at the same time, and there may be multiple master nodes.

In order to enable two-way communication between in-out ports, both ECUs must join the I2C bus as master nodes. In I2C, the nodes are identified by 7-Bit addresses [eLa18]. Consequently, up to 128 nodes may be connected to one I2C bus. Port instances are hence configured with their own

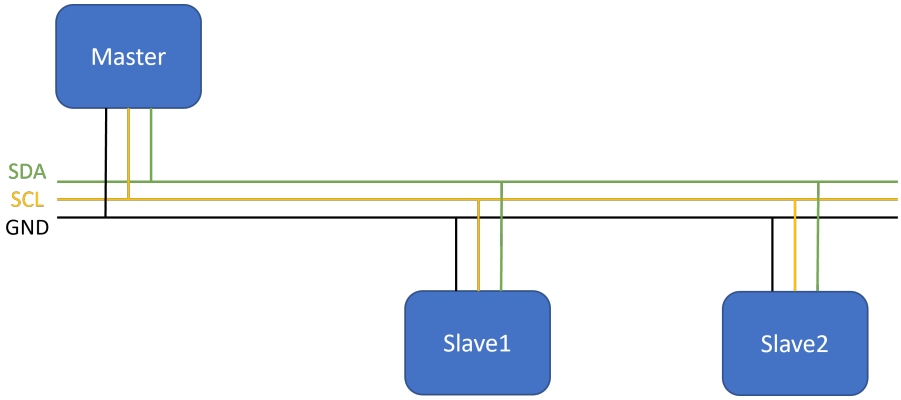


Figure 7.3: The schematics of an I2C bus.

ECU's address, so that they can join the bus as a master, and the address of their connected port's ECU's address to write data to their connected port. Thus, the point-to-point communication of single ports is realized by pair-wise address configuration.

7.2 Container Transformation

The *container transformation* describes the implementation of **REQ2**: the adaption of *T3.2: Generate Deployment Configuration* of the platform-modeling approach (see Figure 6.6). The container transformation uses and instantiates the MechatronicUML Deployment Configuration metamodel while implementing the configurations for MQTT and I2C. First, a model-to-model transformation produces a MechatronicUML Deployment Configuration model. Secondly, the transformation is integrated into the user interface of the Eclipse-based MechatronicUML Tool Suite. Both parts are contained in the MechatronicUML *PSM* project (cf. Table 6.2), and the entire source code is published in the GitHub repository[4].

The model-to-model receive a MechatronicUML Allocation Specification model as input and transforms it into a MechatronicUML Deployment Configuration model. This so-called *container transformation* is implemented using QVTo. In order to implement the requirement **REQ2**, the transformation is extended to (i) incorporate a user choice between DDS or MQTT and I2C as communication options, (ii) create the configuration for port instances using MQTT, and (iii) create the configuration for I2C port instances. The QVTo transformation of the original platform-modeling approach is extended with this functionality.

As an example, Listing 7.1 depicts the creation of a `PortInstanceConfiguration_I2C`. By using the newly introduced intermediate property `i2cAddress` for the type `StructuredResourceInstance`, the address value is retrieved when a `PortInstanceConfiguration_I2C` object is created. Listing 7.1 shows this when setting the object attributes `ownI2cAddress` and `otherI2cAddress` in lines 207 and 208. The `ownECU` and `targetECU` objects are of type `StructuredResourceInstance`. This code snippet is only ever called if (i) two ports are allocated to different ECUs, i.e., they are distributed, (ii) if the user selects I2C for communication, and (iii) if the ports are actually connected via an I2C bus. This is

[4] see https://github.com/fraunhofer-iem/mechatronicuml-psm

Listing 7.1 The creation of `PortInstanceConfiguration_I2C` objects with QVTo.

```
203             portInstanceConfig := object PortInstanceConfiguration_I2C {
204               portInstance:=self;
205               hwportInstance:=hwport;
206               // each I2C port uses the I2C address of its allocated ECU
207               ownI2cAddress:=ownECU.i2cAddress;
208               otherI2cAddress:=targetECU.i2cAddress;
209             };
```

realized in the same way for MQTT and also for DDS. For components which are connected but allocated to the same ECU, a `PortInstanceConfiguration_Local` is created. Local communication is supported no matter which communication middleware option is chosen and the corresponding QVTo transformation is reused from the original platform-modeling approach.

From a schematic point of view, the model-to-model transformation creates the MechatronicUML Deployment Configuration model by creating a respective object for each associated model elements fromt he relate metamodels (see Figure 7.1). A system allocation references a CIC and an HPIC, and the transformation iterates through all structured resource instances that are referenced. The implementation is explained in more detail in Appendix B.2.1.

Furthermore, the selection of a communication middleware option is exposed to the user via the Eclipse-based user interface of the MechatronicUML Tool Suite. This is implemented by an additional wizard page depicted in Figure 7.4. The wizard starts the container transformation and hands over the user choice to the QVTo transformation as a configuration property. The integration of the transformation into the user interface and the new wizard page are described in Appendix B.2.2 with more detail.

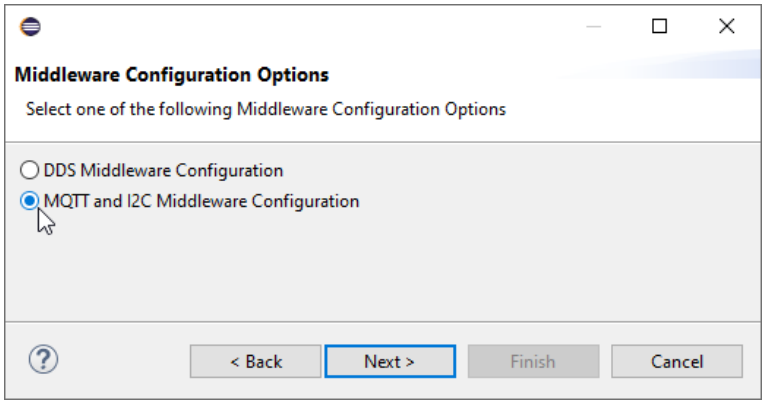


Figure 7.4: The middleware configuration options Eclipse-wizard page.

The configuration properties for creating the port instance configurations, i.e., the I2C addresses or the MQTT server configuration, are not included in the user interface yet. Instead, the I2C addresses are automatically generated in a very simple manner and the configurations for the MQTT server connection including the WiFi configuration are hardcoded in the QVTo transformation at the moment (see Appendix B.2.1 for details). This is sufficient for the purpose of realizing the application scenario and may be improved by future work.

7.3 Container Code Generation

The *container code generation* is started via the MechatronicUML Tool Suite by accessing the context menu on a given MechatronicUML Deployment Configuration model, i.e., a `.muml_container` file. Hence, it is also hooked into the Eclipse-based MechatronicUML Tool Suite as a plugin. All plugins for the container code generation are part of the MechatronicUML *C Containers* project (see Table 6.2). From an implementation perspective, the container code generation and the communication middleware code generation for Arduino are not separated: both is contained in the *C Containers* repository[5]. The repository also contains the integration of the code generation into the user interface. As part of this thesis, two new subprojects are created in the *C Containers* repository: the Acceleo project `org.muml.arduino.adapter.container` and the Acceleo UI Launcher project `org.muml.arduino.adapter.container.ui`. The former contains the implementation of the container code generation for Arduino. Wherever possible, it reuses parts of the `org.muml.c.adapter.container` project, which contains the generation of C container code from the original platform-modeling approach. The Acceleo UI Launcher project `org.muml.arduino.adapter.container.ui` adds a new entry to the context menu of `.muml_container`-files: the *Generate Arduino Container Code* option exposes the code generation functionality to the user (see Appendix D.4). All source code can be accessed in the public GitHub repository[6].

As specified in **REQ3**, the container code must be made applicable to the Arduino environment. This section first describes the overall concept of the source code artifacts and their interfaces in Section 7.3.1. These artifacts are generated for the Arduino platform using the Acceleo model-to-text transformation that is described in Section 7.3.2. This Acceleo model-to-text transformation is contained in the `org.muml.arduino.adapter.container` project. Afterwards, each created file type is explained in detail, starting with the container header in Section 7.3.3, followed by the container implementation in Section 7.3.4, and finally concluding with the Arduino main file in Section 7.3.5. For these files, their functionality and generation procedure are explained.

7.3.1 Source Code Artifacts

Figure 7.5 shows a conceptual view of the generated source code artifacts. The figure includes the mandatory and optional artifacts and their interfaces. The concept is reused from the original platform-modeling approach and tailored to the properties of the Arduino-based application scenario (see Chapter 2)[7] The behavior of the generated software is contained in the component type code. This code is generated from the MechatronicUML PIM and the corresponding code generator is reused from the platform-modeling approach (cf. Section 6.5 and Appendix D.1). The components optionally include user libraries or implementations of continuous devices, that are not

[5] The separation in the code generation concept originates from the initial platform-modeling: An additional code generator was used to generate middleware artifacts from an EMF-based DDS model. This model was also created as part of the model-to-model transformation, but is only applicable to DDS (cf. Section 4.2.1).

[6] see https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-container/tree/stuerner_ma

[7] The main difference to the original platform-modeling approach is, that the Arduino microcontrollers do not have a fully-fledged operating system, but instead the *Arduino Libraries* represent the software layer that provides hardware APIs and additional foundational functionality.

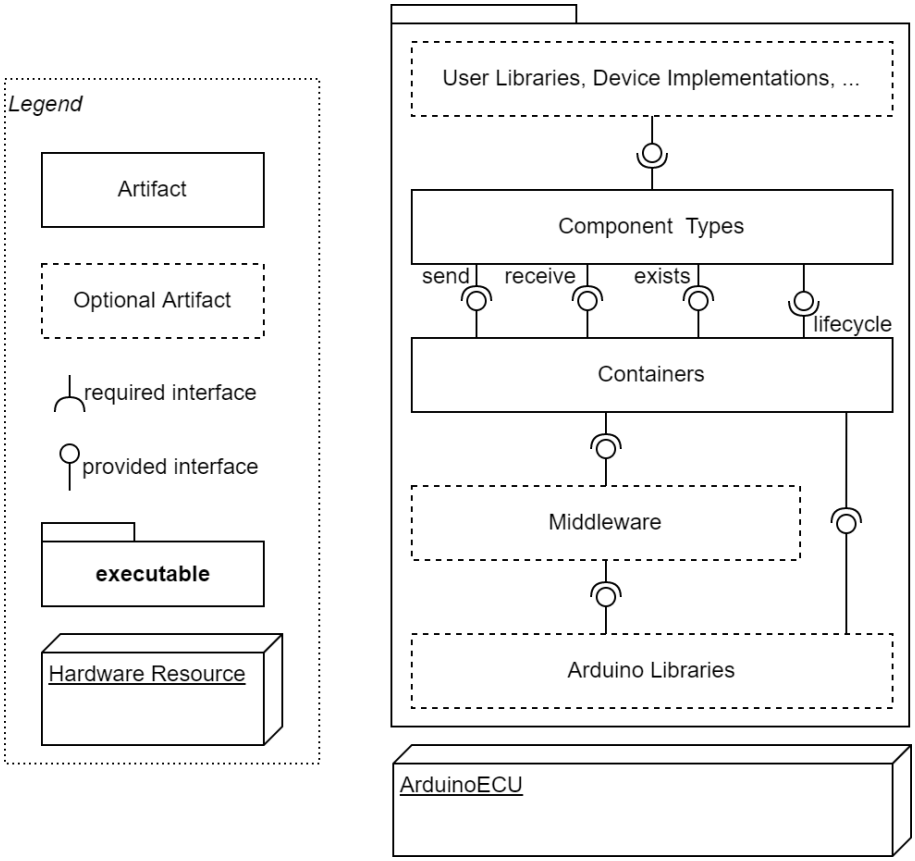


Figure 7.5: The abstract source code artifacts of the the software based on [Poh18].

generated from a model but either already exist, e.g., legacy components. Or they are implemented specifically for the application to be built, e.g., custom controllers. For more details about the user libraries and device implementations, see Appendix D.6.

As visualized in Figure 7.5, the component types always depend on their specific container. On each ECU, there is exactly one container for each allocated component type, and a container may be used by several component instances of that type (cf. Section 7.1.1). The container must provide the interfaces for a component to *send* messages, *receive* messages and check whether a message *exists*. Each component type defines the interface for these operations via forward declaration and the corresponding container supplies the implementation. Additionally, Figure 7.5 also shows that the containers use the lifecycle interface provided by the component types which allows to initialize component instances.

In order to implement the *send*, *receive* and *exist* operations, the container may use the functionality of middleware artifacts, e.g., communication middleware for MQTT, or functionality of the Arduino libraries, that the middleware may also depend on. The Arduino libraries represent common functionalities, similar to operating system-level APIs. Middleware artifacts and Arduino libraries are technically optional, but are employed for all Arduino containers. All in all, these artifacts constitute software for Arduino (see Section 7.5), and the container code is the artifact that is described throughout the following subsections.

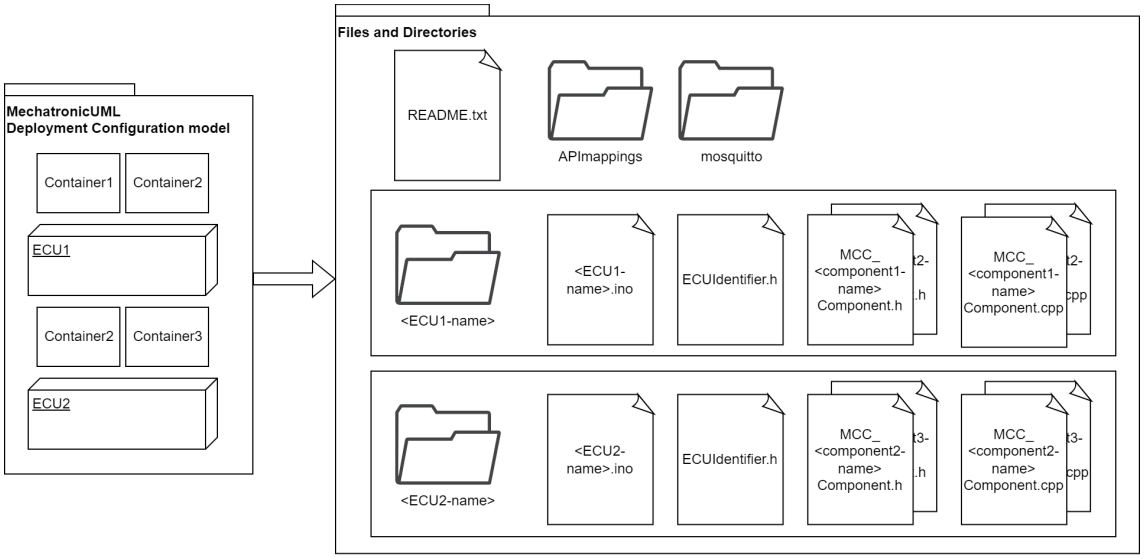


Figure 7.6: The files and directories created by the Acceleo main module of the container code generation.

7.3.2 Model-to-Text Generation Procedure

The container code is generated by a model-to-text transformation that is implemented with Acceleo (see Section 3.2.2). The Acceleo main module defines the overall code generation procedure for a given MechatronicUML Deployment Configuration model. Figure 7.6 shows the files and directories that are generated and form the container code. First of all, a `README.txt` is created to inform the user of the code generator about the most important properties of the generated code. This file is always created when the code generator is used; even if the code generator for Arduino containers is started with a DDS-based deployment configuration model in which case the user is informed via the `README.txt` that DDS is not supported for Arduino[8]. In addition, the `APImappings` directory is created: it contains method stubs for using device APIs that allow accessing continuous components. These method stubs can either be implemented manually or generated using the ApiML and ApiMappingML which are part of T3.5 in Figure 6.6. This task is reused from the original platform-modeling approach (cf. Section 6.5), see Appendix D.6 for more details. Additionally, the `mosquitto` directory is generated if any ECU uses MQTT. It contains the configuration for an MQTT server, see details in Section 7.4.2.

For each ECU, the code generator creates one directory as visualized in Figure 7.6. Each directory contains na Arduino main file. The directory and the Arduino main file, which has a `.ino` file ending, are named like the respective ECU. Additionally, the `ECUIdentifier.h` file is created in each ECU's directory. It specifies identifiers for each component instance and each message type identifiers that are used on a specific ECU; the entities are simply enumerated to be identifiable. The invocation of the Acceleo templates for the Arduino main file and the `ECUIdentifier.h` is also visible in lines 40 and 41 of the Listing 7.2.

[8]For deployment configurations using DDS, the formerly created C container generator is still applicable, and it is still available in the MechatronicUML Tool Suite. There is just no DDS support for the Arduino containers.

Listing 7.2 The template `createArduinoContainers` of the Acceleo main module.

```
31 [**
32  * Generate the containers for one single ECU given as @param ecuConfig.
33 */]
34 [template private createArduinoContainers(ecuConfig: ECUConfiguration)
35   {
36     path : String = ecuConfig.structuredResourceInstance.name+'/' ;
37     useSubDirectories : Boolean = false;
38   }
39 ]
40   [ecuConfig.generateMainFile(path, useSubDirectories)/]
41   [ecuConfig.generateECUIdentifier(useSubDirectories, path) /]
42   [for (container : ComponentContainer | ecuConfig.componentContainers)]
43     [container.generateContainerHeader(path, useSubDirectories) /]
44     [container.generateContainer(useSubDirectories, path) /]
45   [/for]

46 [/template]
```

The example MechatronicUML Deployment configuration model in the left-hand side of Figure 7.6 contains two ECUs with two containers each. Furthermore, the figure shows that for each container on each ECU, a respective header (`MCC_<component-name>Component.h`) and implementation file (`MCC_<component-name>Component.cpp`) are created. The corresponding Acceleo template invocation is shown in lines 43 and 44 of the Listing 7.2. Each container is tailored to one specific component type, and hence, the container is named after the component type. The example also reflects the fact, that component instances of the same type might be allocated to different ECUs, thus, multiple ECUs may have a container for this component type. The Acceleo template in Listing 7.2 is called for each `ECUConfiguration` object in a MechatronicUML Deployment Configuration model and generates the files for each ECU. The container header (Section 7.3.3), the container implementation (Section 7.3.4) and the Arduino main file (Section 7.3.5) are described in the following.

7.3.3 Container Header

The container header file has two tasks: First, to declare a method that creates and initializes a component of the container's associated component type, and secondly, to include the required dependencies for the container implement the container functionality. The container header file is named `MCC_<component-name>Component.h`.

The container header contains a forward declaration for the method to create and initialize a component (in the following: the *create* method). The container implementation implements the *create* method (see Section 7.3.4), and the Arduino main file uses it to create the component instances that are allocated to the corresponding ECU (see Section 7.3.5).

The container header includes several other header files that it depends on for different purposes and reasons:

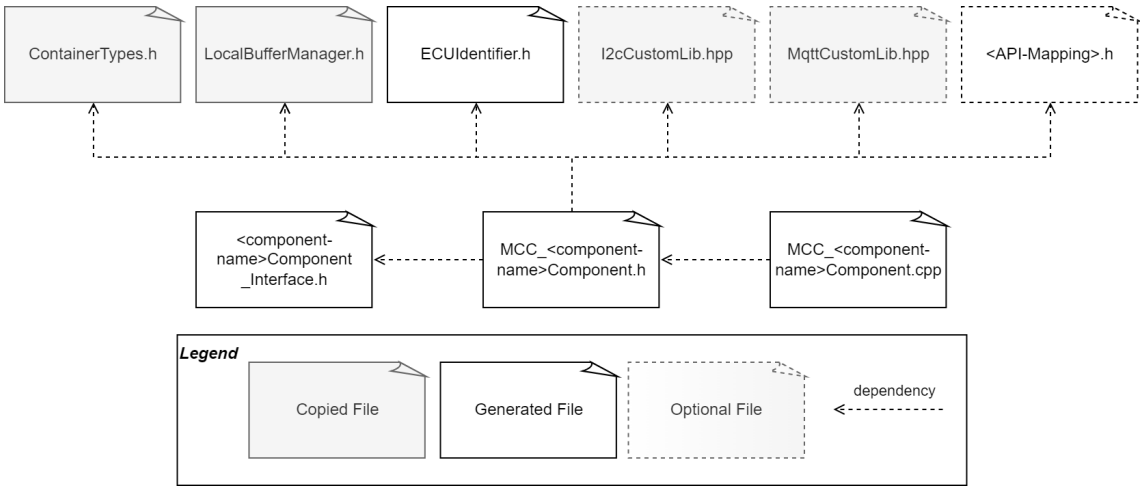


Figure 7.7: The direct dependencies of the container header and container implementation files.

- The `ContainerTypes.h` and `LocalBufferManager.h` are always included. They originate from the platform-modeling approach and contain type definitions for the container implementation as well as an implementation of local communication between ports (cf. Section 7.4). Similarly, the `ECUIdentifier.h` which provides configuration information for the local message exchange is always included.
- If I2C is used, the container requires the custom I2C library (see Section 7.4.1), which is included as `I2cCustomLib.hpp`.
- Equivalently, if MQTT is used, the container requires the custom MQTT library (see Section 7.4.2), which is included as `MqttCustomLib.hpp`.
- In order to implement the *send*, *receive* and *exists* methods that the components require, the container header includes the component type header (`<component-name>Component_Interface.h`). The component type header contains the forward declaration of these methods and is included so that the container implementation (`MCC_<component-name>Component.cpp`) is able to implement these methods (see Section 7.3.4).
- Furthermore, in order to use device APIs that allow accessing continuous components, the respective API mapping header is included. These API mappings can either be implemented manually or generated using the ApiML and ApiMappingML which are part of the platform-specific modeling. These concepts and implementations are entirely reused from the original platform-modeling approach (cf. Section 6.5), so consult [Poh18] for details.

Figure 7.7 sums up these dependencies that the container header includes. The figure also shows that not all files are generated: The copied files are general-purpose implementations for the Arduino platform, and they are simply copied to each ECU's source code directory. They are contained in the `org.muml.arduino.adapter.container` project in the `resources/container_lib` directory, and the copy mechanism is implemented in the `org.muml.arduion.adapter.ui` project. The latter project contains the launcher for the Acceleo model-to-text transformation and thus, the logic for copying is included in the `doGenerate` method of the `org.muml.arduino.adapter.container.ui.common.GenerateAll` class. Even if it is not depicted in Figure 7.7 as is is no dependency of the container code but of the component type code, the copied files also include the `clock.h` specifically for Arduino. It defines the

clock for the RTSC implementation in the component type code using platform-specific functions of the Arduino library (see **REQ3**) and is therefore copied together with all other platform-specific general purpose files. Lastly, Figure 7.7 also shows that the container implementation uses the container header's dependencies by including the container header file.

7.3.4 Container Implementation

The container implementation is the core of the container code. It is generated with Acceleo and partly based on the Acceleo templates of the C container generator. The functionality of the container implementation are listed in the following, specifying whether the respective code generation templates are reused, adapted or newly implemented for the Arduino container generator[9]:

- a builder method including a builder structure definition for the container's associated component type (adapted from the C container generator),
- a builder method for port handles (implemented specifically for the Arduino container generator),
- the implementation of *send*, *receive* and *exists* methods that the component type header declares (implemented specifically for the Arduino container generator),
- the implementation of the *create* method the container header declares (reused from the C container generator),
- and the declaration of a component instance pool to keep track of the component instances in a container (reused from the C container generator).

Thus, the generation of the container implementation code is the core of the container code generation, and it weaves the configuration of the component instances and port instances into the source code according to the MechatronicUML Deployment Configuration model that is used for code generation.

Figure 7.8 shows the contents of a container implementation file in an abstract way. The black frame depicts the component instance pool which is implemented as a simple array. The red frame represents the *create* method that is used in the main Arduino file to create the component instances. The *create* method is generated so that, based on the given component instance, it produces a respective component instance configuration in the source code. For this purpose, the container implementation defines a `ComponentBuilderStruct` (blue frames) that captures the configuration information required for a component instance, including the configuration for the port instances. The *create* method instantiates this `config`. Afterwards, the newly created `config` of type `ComponentBuilderStruct` is handed over to the `ComponentBuilderMethod` which initializes the component using the lifecycle interface of the component type (cf. Section 7.3.1), and adds the component instance to the component instance pool. It also invokes the `PortHandleBuilder` (yellow frame) that initializes the port instances according to the configuration. The created port handles are required for implementing the *send*, *receive* and *exists* methods of the container. As opposed to the simplified representation

[9] Also in cases of new template implementations, the concepts of the original platform-modeling approach are still followed and reused.

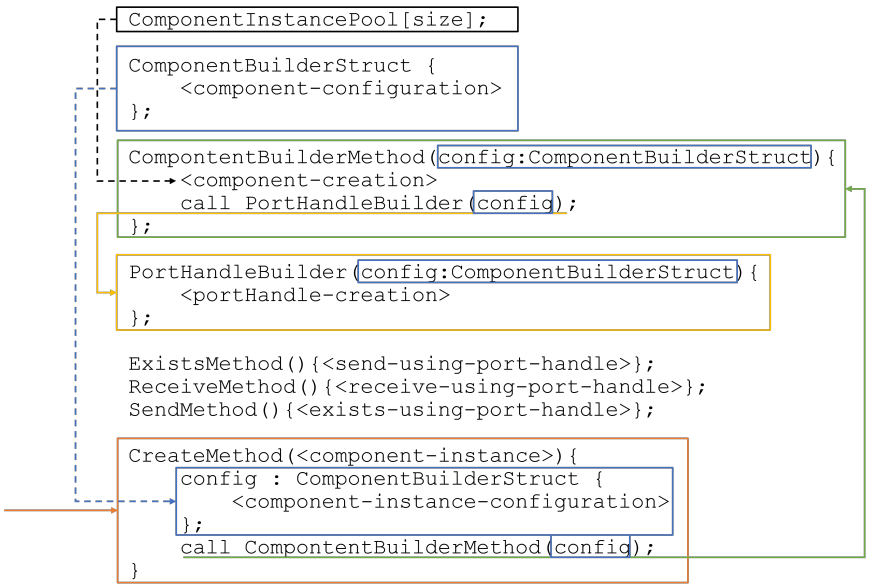


Figure 7.8: An abstract pseudo code description of a container implementation file.

in Figure 7.8, the code generation creates a `PortHandleBuilder` method for each port instance of the component instance, and also the *send*, *receive* and *exists* methods are created for each port and message type. The other parts are generated only once.

These parts of the container implementation must be generated to match the component type code. A key concept for bridging between the platform-independent component implementation and the platform-specific container implementation is the concept of port handles. The component header file contains the forward declaration of the *send*, *receive* and *exists* methods which a component requires in order to communicate via its ports. The component type code implements the ports in a platform-independent way. On the platform-specific side, each container has a set of port handles. Figure 7.9 shows that each `Port` consists of one `PortHandle` and each `PortHandle` is associated to a `Port`. Additionally, there are concrete handles that capture the properties of the communication middleware that is used to implement the port. These concrete handles are modeled as inheritance-based specializations of the `PortHandle` in Figure 7.9, i.e., the `LocalHandle`, `I2cHandle` and `MqttHandle`. The object-oriented modeling is chosen to represent the concept more easily; however, as the component code is entirely implemented with C, the classes are realized as structs and the concrete handles are pointers from the port handle to its concrete handle. The `Port` is declared in the file `port.h` in the component code generation, and the `PortHandle` is defined in the `ContainerTypes.h` (see Section 7.3.3). The concrete handles are defined in the respective communication middleware code (see Section 7.4).

When a container implements the *create* function and creates the `ComponentBuilderStruct`, it has to set the respective configuration for the required port handles as well. Therefore, the `ComponentBuilderStruct` supports different configuration options for the different port types. The code generator retrieves all of the configuration information from the MechatronicUML Deployment

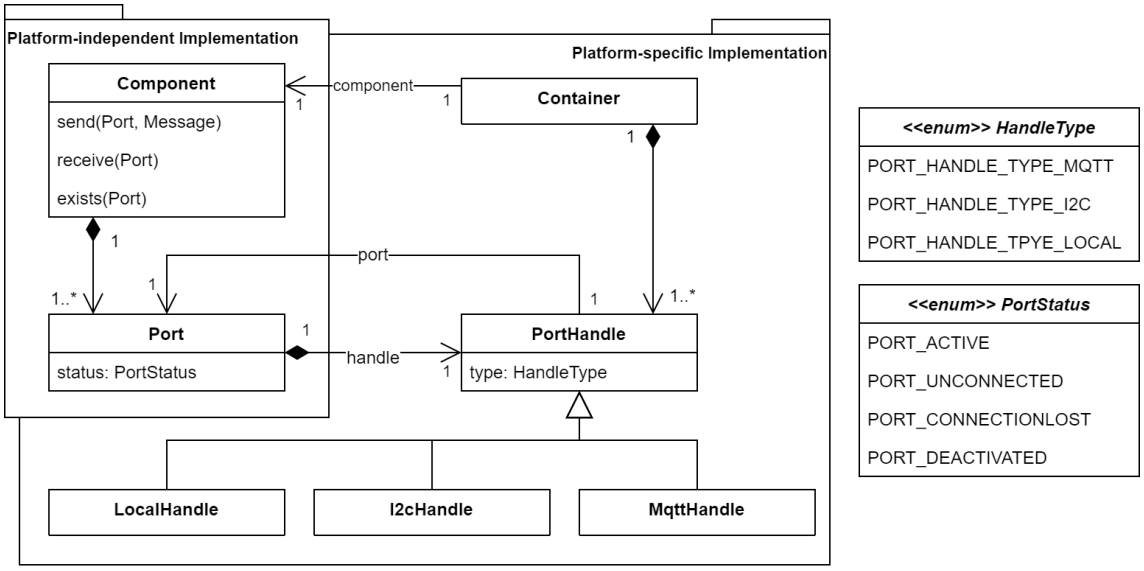


Figure 7.9: The port handle concept to bridge between platform-independent and platform-specific implementation based on [Poh18].

Configuration model that is used for the container code generation, and implements the concepts described in this subsection. More details about the implementation of the code generation templates for the container implementation can be found in Appendix B.3.1.

In order to implement the *send*, *receive* and *exists* methods and the port handles, the container implementation uses the communication middleware libraries. Its generation is implemented together with the generation of the container implementation. The corresponding concepts are code generation templates are described in Section 7.4 with more detail.

The generated container implementation presented in this thesis is tailored to Arduino, and so is the communication middleware code which (see Section 7.4). Arduino supports the usage of the C++ programming language [Ard22]. As the Arduino libraries that are used for MQTT and I2C communication use C++ language features, the container implementation is also stored as C++ file (see Figure 7.6). However, as parts of the C container code generation are reused, barely any C++ language features are used in the container implementation.

7.3.5 Arduino Main File

The Arduino main file is the last file type to be presented as part of the container code. The generation of the main file is adapted from the original platform-modeling approach such that an Arduino main file is generated. This Arduino main file is named like the ECU and thus fulfills the convention that an Arduino main file must be named like the directory it is contained in [Söd22]. The main file is the entry point into the execution of the generated code. Arduino main files consist of a `setup` and a `loop` method. The `setup` is executed once, and after the `setup`, the `loop` method is executed over and over again, i.e., representing an infinite loop.

In the `setup` method, the main file creates each component instance by calling the *create* method of the container implementation (cf. Section 7.3.4). Therefore, it includes the container header that holds the forward declaration of this *create* method, and the container header also includes the dependency to the component type implementation (cf. Section 7.3.3). This dependency is required in the `loop` method, where each component instance is executed using the component type code. These concepts are reused from the original platform-modeling approach and are adapted to the Arduino main file.

Furthermore, the main file includes the setup of the newly added communication middleware libraries for MQTT and I2C. The Acceleo template that generates the Arduino main file determines, which communication middleware is used and writes the respective include statement to the output file. Listing 7.3 shows a snippet of the `generateMainFile` template that contains parts of the setup for the MQTT library. It retrieves the configuration information for WiFi (lines 92-97) and for the MQTT server connection (lines 99-104) and finally, in line 107, calls the setup method of the custom MQTT library. This is done for the I2C library in a similar fashion. In summary, the Arduino main file is generated such that it can be used by the Arduino environment. Thus, it concludes the generation of the container code specifically for Arduino as **REQ3** demands.

Listing 7.3 A snippet of the template `generateMainFile` of the Arduino container code generation.

```
66  void setup(){
      ...
92    //collect the data required for the WiFi configuration
93    struct WiFiConfig wifiConfig = {
94      "[mqttPiConfig.WiFi_ssid.toString()/]",
95      "[mqttPiConfig.WiFi_pass.toString()/]",
96      WL_IDLE_STATUS
97    };
98
99    //collect the data required for the MQTT configuration
100   struct MqttConfig mConf = {
101     "[mqttPiConfig.MQTT_serverAddress.toString()/]",
102     [mqttPiConfig.MQTT_serverPort/],
103     "[ecuConfig.name.toString() /]"
104   };
105   mqttConfig = &mConf;
106
107   mqttCommunication_setup(&wifiConfig, mqttConfig);
108     [/let]
109   [/if]
      ...
114 }
```

7.4 Communication Code

In order to use MQTT and I2C as additional communication technologies, appropriate implementations for the Arduino environment are provided to fulfill requirement **REQ4** (see Chapter 7). The implementation of the code generation using Acceleo distinguishes between the communication code and the remainder of the container code, but the invocation of the templates are all part of a single code generation run (cf. Section 7.3.2). The container implementation and the Arduino main file use the communication middleware libraries to setup the communication (cf. Section 7.3.5) and to create concrete port handles that make use of this communication middleware (cf. Section 7.3.4). Figure 7.5 shows this dependency of the container code on the middleware libraries. This section therefore describes which Arduino libraries are used to implement the communication, how they are wrapped in custom communication middleware libraries which are tailored to be used by the container code, and how the container implementation uses the libraries to implement the *send*, *receive* and *exists* functions.

The custom libraries for I2C and MQTT are static files from the perspective of the code generation: they are created to wrap the communication functionality for an Arduino environment and specify an interface to be used by the container code. Thus, they are not generated but copied to each ECU's directory. Same applies to the library for local communication: If components are allocated to the same ECU and exchange messages, this is realized using the local memory. The `LocalBufferManager.h` middleware library (cf. Figure 7.7) is from the original platform modeling approach and is reused for the Arduino code generation. The newly added custom communication middleware libraries for I2C and MQTT are described in Section 7.4.1 and Section 7.4.2. Afterwards, Section 7.4.3 explains some common concepts of the libraries as well as their usage to generate the *exists* and *receive* methods.

7.4.1 Custom I2C Library

The Arduino *Wire* library[10] is used to implement the I2C communication. It is an Arduino standard library, so it does not have to be installed additionally. It abstract the access to an Arduino microcontroller's I2C hardware interface. The library provides the `Wire.begin(<own-address>)` method to join an I2C bus. The custom I2C library uses a combination of `Wire.beginTransmission(<receiver-address>)`, `Wire.write(<bytes>,<length>)` and `Wire.endTransmission()` to send a message using I2C. Furthermore, `Wire.onReceive(<function-pointer>)` is used to register a callback function that is executed whenever a message is received.

Using these methods of the Wire library, the custom I2C library provides the interface visible in Listing 7.4 to be used by the container code. The `i2cCommunication_setup` method is used by the Arduino main file to setup the communication. The `initAndRegisterI2cReceiver` is used by the port handle builders to make the port ready for receiving messages, and the `sendI2cMessage` method is used to implement the *send* methods. The custom I2C library also maps the concept of MechatronicUML messages to I2C messages: It sends the byte code of the message together with a uniquely identifying byte string of the message type. When it receives a message, the library reads the message type and puts the message into the respective message buffer from where

[10] https://www.arduino.cc/en/reference/wire

Listing 7.4 The interface of the custom I2C library.

```
42 void i2cCommunication_setup(uint8_T ownI2cAddress);

   ...

53 void initAndRegisterI2cReceiver(I2cReceiver* const receiver, char* const
       ↪ messageTypeName, size_t bufferCapacity, size_t messageSize, bool_t bufferMode
       ↪ );

   ...

63 void sendI2cMessage(uint8_T receiverAddress, char* const messageTypeName, byte* const
       ↪ message, size_t messageLength);
```

it can be consumed (see Section 7.4.3). This marshaling and unmarshaling relies on a uniform interpretation of the byte code by the sender and receiver system which is generally given among Arduino microcontrollers.

The custom I2C library furthermore defines the `I2cHandle` (cf. Figure 7.9) and `I2cReceiver` types. The library can generally be used by Arduino microcontrollers and does not imply any assumptions besides the microcontroller having an I2C interface which is physically connected to an I2C bus.

7.4.2 Custom MQTT Library

The *PubSubClient* library for Arduino[11] is used to implement the custom MQTT library. The *PubSubClient* library can be installed using the Arduino library manager. It provides the class `PubSubClient` to manage the connection of an Arduino microcontroller to an MQTT server. Moreover, it provides the methods `PubSubClient.subscribe(<topic-name>)` to subscribe and `PubSubClient.publish (<topic-name>, <message>)` to publish to topics. The method `PubSubClient.setCallback(<function-pointer>)` allows to register a callback function to deal with received messages.

Using these methods of the PubSubClient library, the custom MQTT library implements the interface functions listed in Listing 7.5. In a similar fashion as the custom I2C library, these are the `MqttCommunication_setup` for the library initialization, the `initAndRegisterMqttSubscriber` method to be used by the port handle builder and the `sendMqttMessage` method to implement the *send* methods for the individual ports. However, the *PubSubClient* library also requires a method to be called in the infinite Arduino loop method; this is exposed by the custom MQTT library via the `MqttCommunication_loop` method. Furthermore, the custom MQTT library defines the `MqttHandle` (cf. Figure 7.9) and the `MqttSubscriber` types as well as the configuration structures `WiFiConfig` and `MqttConfig`.

Additionally, the library implementation maps the exchange of specific MechatronicUML message types to corresponding MQTT topics. It uses the hierarchical nesting of topic names for that purpose and attaches the uniquely identifying message type name to the topics for sending and receiving. For instance, if the publishing topic `fastCarCoordinatorECU/communicator.F/overtakingAffiliate1/`

[11] https://www.arduino.cc/reference/en/libraries/pubsubclient/

Listing 7.5 The interface of the custom MQTT library.

```
68 void mqttCommunication_setup(struct WiFiConfig* const wifiConfig, struct MqttConfig*
       ↪ const mqttConfig);

   ...

76 void mqttCommunication_loop(struct MqttConfig* const mqttConfig);

   ...

88 void initAndRegisterMqttSubscriber(MqttSubscriber* const subscriber, char* const
       ↪ subscriptionTopic, char* const messageTypeName, size_t bufferCapacity, size_t
       ↪  messageSize, bool_t bufferMode);

   ...

98 void sendMqttMessage(char* const publishingTopic, char* const messageTypeName, byte*
       ↪ const message, unsigned int messageLength);
```

is used to send a message of type `requestOvertaking`, then the uniquely identifying message name `OvertakingCoordinationMessagesRequestOvertaking` is appended to compose the topic `fastCarCoordinatorECU/communicator.F/overtakingAffiliate1/OvertakingCoordinationMessagesRequestOvertaking`. Thus, the MechatronicUML message type of a received message is represented by the topic it is received from and the message is stored in the respective message buffer for consumption (see Section 7.4.3).

In order to make use of MQTT, the Arduino microcontroller requires access to a TCP/IP network. The robot cars each have one microcontroller with an attached WiFi module (cf. Section 2.2), thus, the WiFi module's capabilities are used to connect to the MQTT server. For simplicity, the custom MQTT library is implemented under the assumption, that microcontrollers that use the library have this hardware available. In detail, the assumptions are the following:

- The attached WiFi module is of the type *ESP8266-01S* and can be operated using the WiFiEsp library for Arduino[12].
- The WiFi module is wired such that it can communicate with the Arduino microcontroller using a software serial connection on the Arduino board pins 2 (receive) and 3 (transmission).
- The WiFi module is configured to operate at a baud rate of 9600.

Furthermore, an MQTT server is required which stores and forwards the messages (cf. Section 7.1.3). It must be addressable via the TCP/IP network and is assumed to be available at `192.168.0.100:1883` (cf. Appendix B.2.1). Moreover, the custom I2C lib does currently not support an authenticated connection to an MQTT server. In order to setup an MQTT server more quickly to use MQTT, the Acceleo model-to-text transformation also generates the configuration for an Eclipse Mosquitto[13] MQTT server. This configuration is contained in the `mosquitto` directory (cf.Section 7.3.2). The

[12] https://www.arduino.cc/reference/en/libraries/wifiesp/
[13] https://mosquitto.org/

MQTT server is configured to listen to port 1883 (cf. Appendix B.2.1). The MQTT server can be started using Docker[14], e.g., with a standard laptop that can be reached by the microcontrollers at the IP address `192.168.0.100`.

These assumptions can be reduced in the future to allow other network connections as well; they are only technical limitations of the current implementation.

7.4.3 Common Concepts

Both custom communication libraries use the message buffer library of the C container generator. This message buffer library provides the functionality to create message buffers according to the MechatronicUML PIM specification ([DPP+16]). These buffers are created for every `I2cReceiver` and `MqttSubscriber` respectively. The message buffers are configured according to the MechatronicUML PIM model that is used for code generation. The current implementation has the limitation that for each message type, a new buffer is created, even though the model might associate several message types to one buffer. This is only a technical limitation and can be improved in the future. The generated *send*, *exists* and *receive* implementation uses the respective methods of the custom communication middleware libraries and the message buffers as shown in Appendix B.3.2.

Furthermore, both custom communication libraries do not implement a platform-independent marshaling of the MechatronicUML messages, i.e., the MechatronicUML messages are not transformed into a special transportation format. They are simply embedded as byte string into the respective communication technologies message format. This only works, when both the sender and receiver have a uniform way of interpreting this byte array; this is given in our application case, as only one type of ECU is used.

7.5 Program Building

Lastly, to complete the software construction process and its adaption for the Arduino-based environment, the source code must be built to be executable by an Arduino microcontroller (see **REQ5** in Chapter 7). The Arduino tools provide the so-called *verify* function [Söd22]. It issues the compilation and linking of an Arduino *sketch*. *Sketch* is the name for the main program file in the Arduino environment, i.e., the `.ino` file. The Arduino IDE is very simple and its target is to provide access to programming microcontrollers in a very simple fashion. Thus, the Arduino IDE does not provide configuration or customization options, but the build is executed in a predefined manner [Ard22]. The same applies for the Arduino *upload* function, i.e., the deployment of the executable on the microcontroller [Ard22; Söd22].

As the configuration options for the build via the Arduino tools are limited, some manual adaptions are defined here to ensure the software is properly built. These adaptions are only executed manually due to the lack of tool expertise and time; they may be automated or replaced in the future. The adaptions to be made are the following:

[14] https://www.docker.com/

- As there is no dependency specification or management tool by Arduino, the relevant third party libraries are installed via the Arduino library manager[15]. These are the WiFiEsp library[16] and the PubSubClient library[17] in case MQTT is used. For I2C, only standard, preinstalled libraries are used.
- The component type code has to be put into the same directory as the container code, i.e., the directory for each ECU, without using subdirectories, and the include paths have to be adjusted accordingly. Exceptions are: The `clock.h` file of the component code is not used for Arduino, but the file that is generated together with the container code, and the `lib/stardardTypes.h` is also not used (the `types/stardardTypes.h` is used instead).
- Similarly, the API mappings and any other manually referenced library code has to be copied into the directory of each ECU (see an example in Section 8.2.4).

The program building for Arduino is realized in the most simple way possible and may be improved in the future. Thus, the aforementioned manual steps may be automated or replaced, e.g., by making the library locations known to the linker. Furthermore, the adaptions to the component code are not implemented in the component code generator because only a working, packaged executable was available at the time of writing of this thesis [18]. The source code of the component code generator is contained in the C components repository (see Table 6.2). The plugins yield errors when being launched and, due to the different focus of this thesis, are not fixed and analyzed more thoroughly in this work. Hence, these manual steps are only technical limitations of the current process definition and implementation.

7.6 Integration into the MechatronicUML Tool Suite

The aforementioned new functionality which implements the extended platform-modeling approach is integrated into the MechatronicUML Tool Suite. Currently, there is no active maintenance of any repository of the MechatronicUML Tool Suite, nor is there a build server or an Eclipse update site (cf. Section 3.3). However, there is a pre-packaged MechatronicUML Tool Suite version 1.0 available for download[19]. The presented implementations were developed and tested based on this latest release of the MechatronicUML Tool Suite. As described in this chapter, all extensions of the platform-modeling approach are implemented within already existing repositories. Some functionality is wrapped in new plugins, other functionality is integrated into existing plugins. Thus, the new implementation is integrated into the MechatronicUML Tool Suite by taking version 1.0 and cloning the aforementioned repositories into the workspace of the MechatronicUML Tool Suite. Then, any plugin can be launched as an Eclipse runtime instance; all plugins available in the workspace are then also available in the runtime instance. In addition, in order to for all transformations to function correctly, the following metamodel plugins must be available in the

[15] see https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries

[16] https://www.arduino.cc/reference/en/libraries/wifiesp/

[17] https://www.arduino.cc/reference/en/libraries/pubsubclient/

[18] The C code generator is available as part of the prepacked MechatronicUML Tool Suite, see Section 3.3.

[19] The download is available at http://www.mechatronicuml.org/de/download.html, but the download has not been working at the time of writing this thesis. Therefore the tool suite was obtained by getting in touch with the researchers working on the platform-modeling approach before, i.e., Pohlmann and Bobolz et al.

runtime instance, e.g., by cloning them into the workspace as well: The `org.muml.core` plugin[20], the `org.muml.pim` plugin[21], the `org.muml.pm.hardware` and `org.muml.pm.software` plugins[22]. This setup is applied to test and use the extended platform-modeling approach as described in the following Chapter 8.

[20] https://github.com/fraunhofer-iem/mechatronicuml-core
[21] https://github.com/fraunhofer-iem/mechatronicuml-pim
[22] https://github.com/fraunhofer-iem/mechatronicuml-pm

8 Implementation of the Application Scenario

This chapter describes the implementation of the application scenario (see Chapter 2) with the MechatronicUML and its tool chain. This exemplary use case seeks to demonstrate and evaluate the usage of the extended platform-modeling approach which is designed and implemented in this thesis. As Figure 8.1 visualizes, the platform-modeling approach consists of several steps prior to the software construction. The implementations presented in this thesis however concern only the software construction. Thus, this chapter briefly summarizes the platform-independent modeling, the platform modeling and the allocation engineering in Section 8.1 before describing the software construction using the extended platform-modeling approach in Section 8.2. All models and diagrams that are used in this chapter as well as the produced source code are publicly available on GitHub[1].

8.1 Modeling the Application Scenario with the MechatronicUML

The structure and behavior of the software are defined as a MechatronicUML Platform-Independent Software Model, and the corresponding model for the cooperative overtaking scenario is presented in Section 8.1.1. The MechatronicUML Platform Model that captures the hardware platform is described in Section 8.1.2, and the MechatronicUML Allocation Specification Model is explained in Section 8.1.3. These models that are depicted in Figure 8.1 lay the foundation for the software construction.

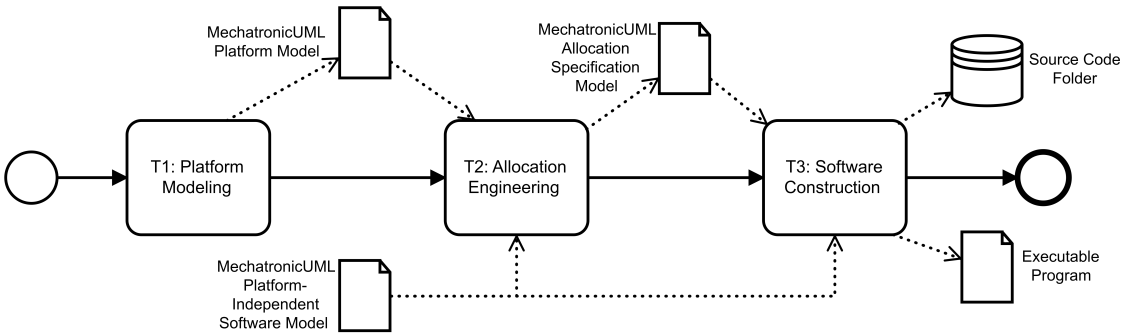


Figure 8.1: A high-level perspective of the process of the platform-modeling approach.

[1] https://github.com/SQA-Robo-Lab/Overtaking-Cars

8.1.1 Platform-Independent Software Model

The components are the core of the MechatronicUML PIM. They each hold a subset of the overall application functionality and model the structure of the software system. The software is designed to implement the cooperative overtaking scenario introduced in Section 2.1. Figure 8.2 shows all components that together compose the software for a robot car. The components encapsulate the following behavior:

DistanceSensor The continuous atomic component `DisctanceSensor` represents the software component to retrieve a distance measurement value from an distance sensor. Thus, it provides a continuous out-port `distance` to expose this functionality.

PowerTrain The continuous atomic component `PowerTrain` represents the software component to control the power train of the robot car and exposes the functionality to set the desired velocity via the continuous in-port `velocity`.

Coordinator The discrete atomic component `Coordinator` encapsulates the functionality to coordinate a cooperative overtaking maneuver with another robot car. Thus, it has two discrete in-out ports that allow it to join a cooperative overtaking maneuver as `overtakingInitiator` or `overtakingAffiliate`. Furthermore, it has a discrete in-out port `communicator` that exposes its functionality to the `DriveControl` component whit which it negotiates the permission to overtake another robot car.

DriveControl The discrete atomic component `DriveControl` controls the driving behavior of the robot car. It has two hybrid in-ports to access two `DisctanceSensor` components, one for the front and one for the rear of the robot car, and another hybrid out-port to instruct the power train by setting a desired `velocity`. It also has a discrete in-out port to communicate with a `Coordinator` component in order to get the permission for a cooperative overtaking maneuver.

RoboCar The `RoboCar` component represents the overall software for the robot cars: it consists of one instance of the `Coorinator`, `DriveControl` and `PowerTrain` component each, and two instances of the `DistanceSensor` component for the front and the rear distance sensors. The two discrete component instances communicate using the `Overtaking Permission` RTCP, and the `communicator`'s ports are delegated to delegation ports and shape the interface of the `RoboCar` to the outside world.

The behavior of the discrete components is also modeled as part of the platform-independent modeling. As detailed in Appendix C.1.3, the `DriveControl` component contains the entire behavior for the driving, including following a lane or changing lanes. Thus, as the infrared sensor are only needed for the purpose of following a line on the ground, which represent a lane of the road, there is no continuous component that represents the infrared sensors' behavior. Thus, the infrared sensors are not modeled as a separate software component, but are part of the `DriveControl`'s functionality. The component behavior is of special interest for the component type code generation. As the component type code generation is unchanged in the extended platform-modeling approach, the component behavior is not detailed here but in Appendix C.1.3.

RTCPs are used to specify the coordination between discrete components. The RTCPs define the coordination as message-based communication between discrete ports (cf. Section 3.4.2). In order to be used within an RTCP, a message has to be specified in a so-called *message repository* first. The message repositories for the cooperative overtaking scenario are depicted in Figure 8.3.

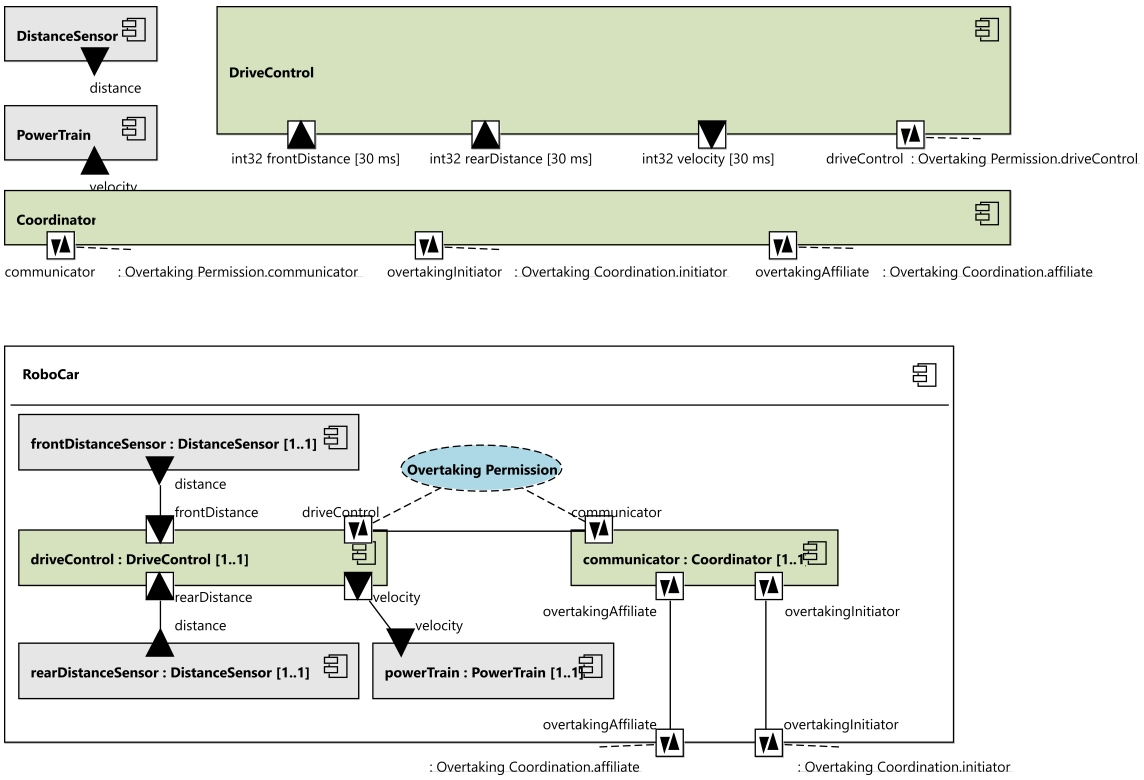


Figure 8.2: The components to implement the robot cars using the MechatronicUML.

There are two RTCPs that model the communication for the cooperative overtaking: the `Overtaking Permission` RTCP and the `Overtaking Coordination` RTCP. The former defines the communication between the software components within a robot car, as depicted in Figure 8.2. It uses the messages of the `OvertakingPermissionMessages` message repository (see Figure 8.3). The RTCP models that the `driveControl` may not execute an overtaking maneuver without the `communicator` having coordinated with another robot car. Appendix C.1.2 explains the `Overtaking Permission` RTCP with more detail.

In the given application scenario (see Chapter 2), there are two robot cars. This is represented by the `fastCar` and `slowCar` component instances, that are depicted in the CIC in Figure C.1. The CIC shows that both cars are of kind `RoboCar`, hence, both cars operate with the same software. Figure C.1 also shows that the two robot cars communicate using the `Overtaking Coordination` RTCP. Thus, both cars can initiate or join a cooperative overtaking maneuver; their roles are not limited or predetermined by their software components. The `Overtaking Coordination` RTCP uses the message types of the

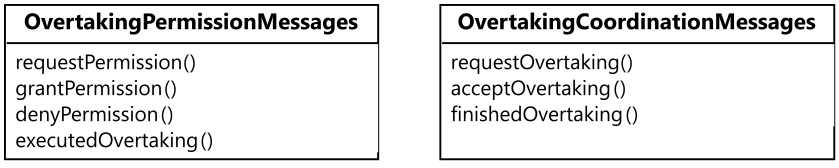


Figure 8.3: The message repositories used to model the robot car.

`OvertakingCoordinationMessages` message repository (see Figure 8.3): The `initiator` requests the overtaking, the `affiliate` accepts[2] and after the overtaking maneuver is executed, the `initiator` sends the `finishedOvertaking` message. Appendix C.1.2 describes the RTCP with all details.

Together with Appendix C.1, this section describes the MechatronicUML PIM model that specifies the software for the cooperative overtaking scenario with the robot cars. Furthermore, the most important design decisions are explained in this section. For different variants of modeling comparable scenarios, see to [BCD+14] and [Poh18].

8.1.2 Platform Model

This section describes the modeling of the robot car platform using the MechatronicUML HPDM. The robot cars are introduced in Section 2.2. This section focuses on the hardware platform and the underlying resources and resource instances are described in Appendix C.2.

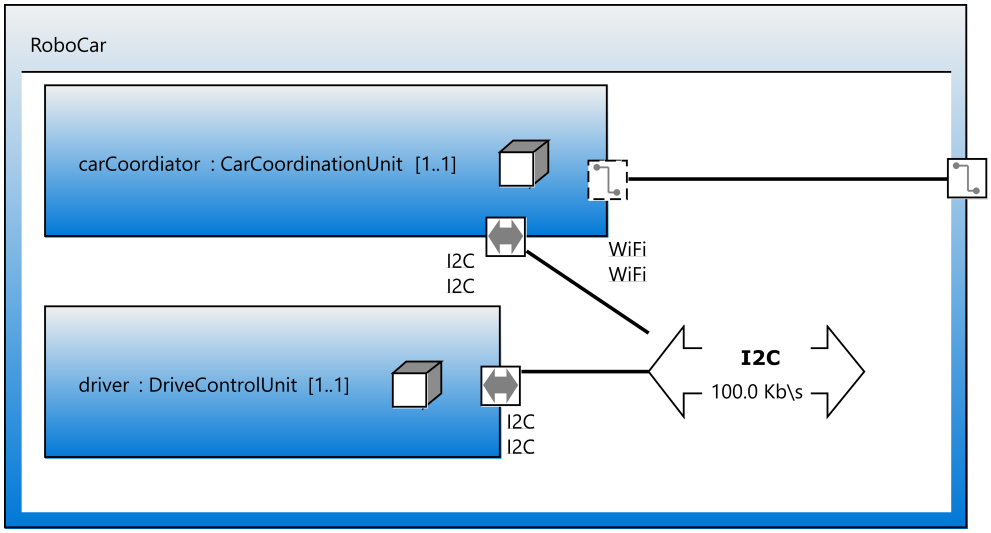


Figure 8.4: The `RoboCar` platform type.

The `RoboCar` platform type shown in Figure 8.4 consists of one `CarCoordinationUnit` instance and one `DriveControlUnit` instance which are connected via an I2C bus. It represents the hardware structure of the robot car (cf. Section 2.2 and Appendix A): The `CarCoordinationUnit` models the Arduino Nano microcontroller with its WiFi module (see Figure C.8) and the `DriveControlUnit` specifies the Arduino Mega with its attached sensors and motors (see Figure 3.12). Appendix C.2.2 provides some more information about the platform models of the robot cars.

Appendix C.2.2 also contains the HPIC diagram, where all of the previous types are instantiated to model a hardware platform instance, i.e., the two robot cars of the application scenario. The HPIC is depicted in Figure C.9 in Appendix C.2.2. Modeling the distributed platform is the foundation for modeling a distributed allocation and a distributed deployment.

[2] In the introduced application scenario, the affiliate always accepts an overtaking request (cf. Section 2.1).

8.1.3 Allocation Specification Model

The allocation engineering (T2 in Figure 8.1) represents the beginning of the platform-specific modeling by defining a relation between the CIC of a MechatronicUML PIM model and the HPIC of a MechatronicUML HPDM model. Figure 8.5 depicts which software components are allocated to the `fastCar` platform instance.

Overall, Figure 8.5 depicts that the embedded component instances of the `fastCar` structured component instance are allocated to the `fastCar` hardware platform instance[3]. The MechatronicUML Allocation Specification allocates atomic components to structured resource instances. The `communicator.F` atomic component instance is allocated to the `fastCarCoordinatorECU` structured resource instance, and the other atomic component instances are allocated to the `fastCarDriverECU`. Thus, as intended for the application scenario, the `fastCarCoordinatorECU` which represents an Arduino Nano with an attached WiFi module, contains the communication capabilities. Furthermore, the components that encapsulate the driving capabilities are allocated to the `fastCarDriverECU` which represents the Arduino Mega with its attached sensors and motors. Equivalently, the embedded component instances of the `slowCar` structured component instance are allocated to the `slowCar` hardware platform instance. The entire allocation is depicted an explained in Appendix C.3.

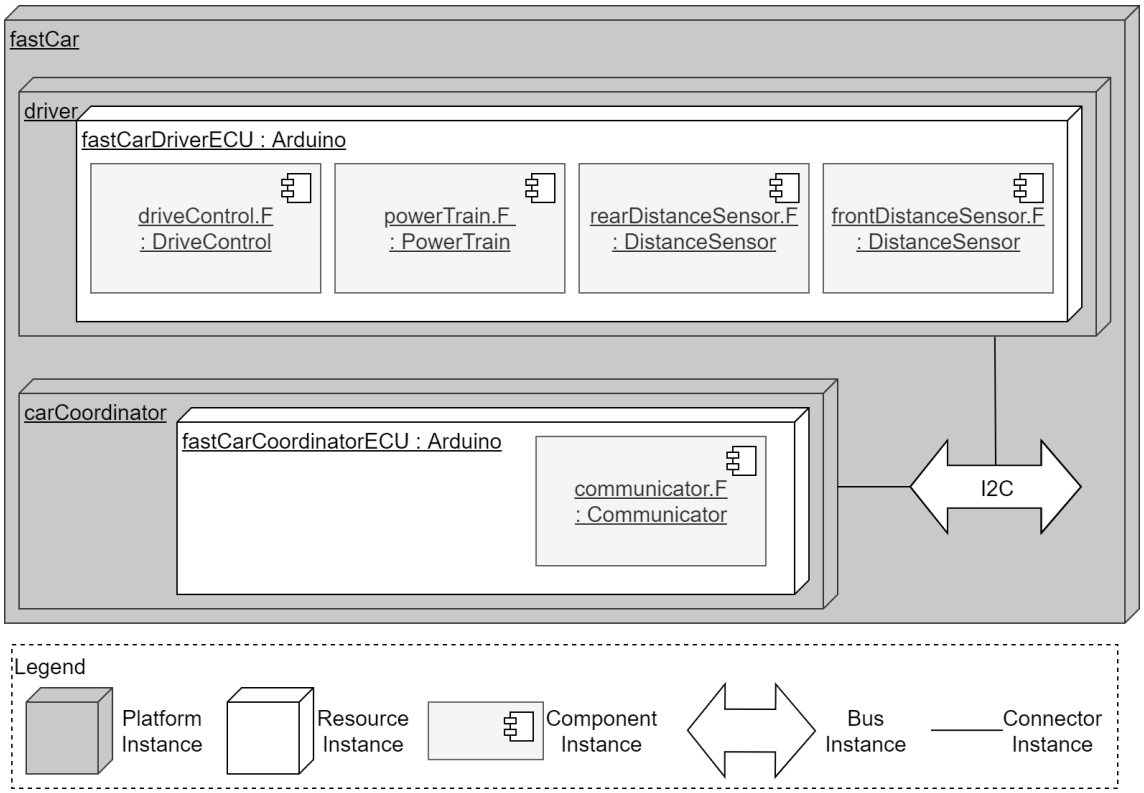


Figure 8.5: The allocation of the robot car software for the `fastCar` platform instance.

[3] All component instances are depicted in Figure C.1, and the HPIC is visualized in Figure C.9.

The MechatronicUML Allocation Specification model is the final asset required for the software construction with the platform-modeling approach. It references a concrete CIC of the MechatronicUML PIM and an HPIC of the MechatronicUML HPDM and thus lays the foundation for a distributed deployment by specifying the distributed allocation. This section concludes the modeling of the application scenario prior to the software construction and thus, altogether, provides the answers to **RQ4.1:** Can the application scenario be modeled with the MechatronicUML in a way that is suitable for code generation? Yes, the application scenario can be modeled using the MechatronicUML up to the allocation specification which is required for the platform-specific modeling and the container code generation.

8.2 Software Construction

The software construction is the final step of the platform-modeling approach, and this is where the presented extensions come into play. The software construction subtasks are visualized in Figure 8.6[4] and comprise the platform-specific modeling and the code generation. The software construction is based on different MechatronicUML metamodels. The previous Section 8.1 presents a concrete model that specifies the software and hardware of the application scenario. This model is used to generate code for the Arduino-based target platform with the extended platform-modeling approach. Figure 8.6 highlights the tasks that are adapted or replaced in order to support the Arduino-based target platform. Therefore, this section focuses on these subtasks and the concepts. Additional details as well as instructions on how to use the platform-modeling approach in the MechatronicUML Tool Suite are described in Appendix D.

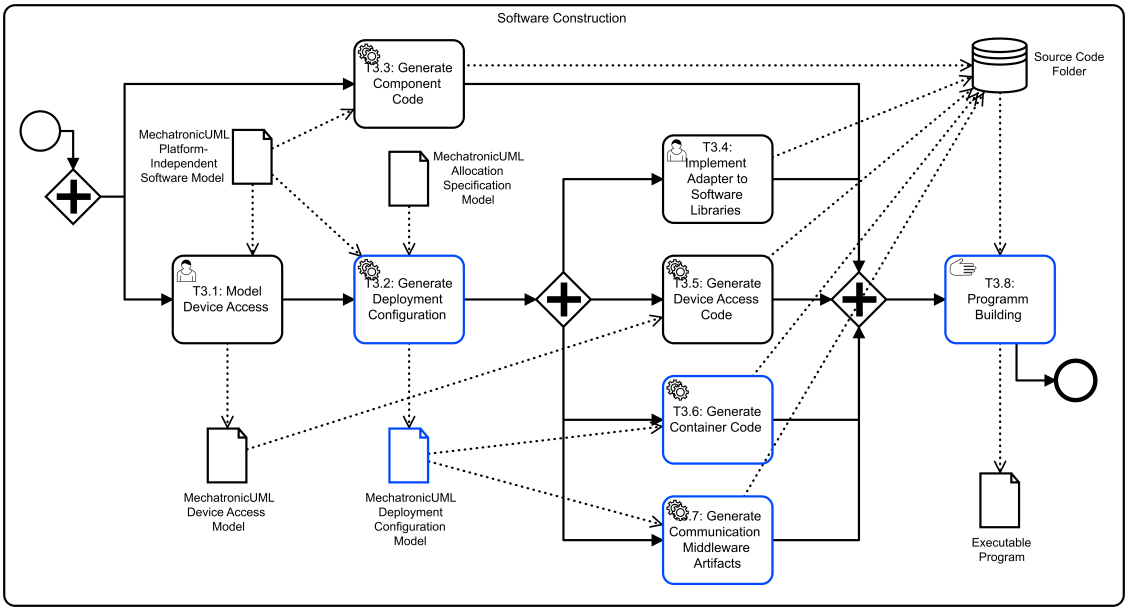


Figure 8.6: The subtasks of the software construction in the platform-modeling approach with the extensions highlighted in blue.

[4]The entire process of the platform-modeling approach highlighting the adapted tasks is shown in Figure 4.1

This section serves the purpose to explain how the extensions of the platform-modeling approach fit together with the reused parts and to highlight characteristic parts of the generated code. First, the component code generation is briefly described in Section 8.2.1, before the deployment configuration is described in Section 8.2.2. The container code generation is explained in Section 8.2.3, and Section 8.2.4 concludes the software construction by describing the results of the code generation.

8.2.1 Component Code Generation

The component code generation is entirely reused from the platform-modeling approach. It corresponds to *T3.3: Generate Component Code* in Figure 8.6. Appendix D.1 describes how the component code is generated using the MechatronicUML Tool Suite. The component code comprises several directories for the components, RTSCs, message types and operations. For details, consult [Poh18]. In this section, the focus is on the code in the `components` subdirectory which contains the header and implementation file of each component type. The header file is relevant for the container code generation: it contains the forward declaration of the *send*, *receive* and *exists* methods. Listing D.1 shows a snippet of the `coordinatorComponent_Interface.h` file. This is the component type header file for the `Coordinator` discrete component type from Figure 8.2, which is also depicted in Figure 8.7.

Figure 8.7: The `Coordinator` component.

Listing D.1 in Appendix D.2 shows the declarations of all communication methods required for the `communicator` and `overtakingInitiator` ports; the methods for the `overtakingAffiliate` port are declared in the same manner, but not shown in the depicted code. There is one *send* method for each sender message type of a port, and for each receiver message type, there is one *exists* and one *receive* method. All methods require a `Port* p` as parameter (cf. Figure 7.9).

Listing 8.1 The *send* method's forward declaration in the component type header file of the `DistanceSensor` component type.

```
19 /**
20  *  Forward Delcaration of Container Functions
21  */
22 void MCC_DistanceSensorComponent_distance_send_value(Port* p, int32_T* value);
```

The message types are also generated as part of the component code. The file `message_types.h` is created in the `message` directory, and the component header file includes it. The component type code generation also deals with continuous components. The continuous ports' behavior is modeled via a standardized RTSC that models the periodic sending or receiving of a message (cf. Section 7.1.2). This is depicted in Listing 8.1: the listing shows a snippet of the component header file for the `DistanceSensor` continuous component type (see Figure 8.2). The `DistanceSensor` has one continuous out-port of type int, which is reflected by the *send* method declaration in line 22 of Listing 8.1.

For the continuous and hybrid ports, no message types are declared, but a message of the port's primitive data type is sent or received, e.g., an `int32_T` value. All these methods are implemented by the generated container code presented in Section 8.2.3.

8.2.2 Deployment Configuration

In order to generate container code for the Arduino-based target platform, an appropriate MechatronicUML Deployment Configuration model is created using the new implementations presented in Section 7.1 and Section 7.2. This corresponds to the task *T3.2: Generate Deployment Configuration* in Figure 8.6. Executing this task is achieved using the MechatronicUML Tool Suite as described in Appendix D.3. Using the system allocation described in Section 8.1.3 with the *MQTT and I2C Middleware Configuration* middleware configuration option produces a deployment configuration that is suitable for the Arduino-based target platform of the application scenario. The resulting MechatronicUML Deployment Configuration model is visualized in Figure 8.8.

Figure 8.8 does not show a formal diagram; a view for deployment configurations has not been defined or implemented in the MechatronicUML Tool Suite, yet. Figure 8.8 depicts all hierarchical relationships between the model entities as well as the most important configuration properties of the `PortInstanceConfiguration` subtypes and is hereby suggested as a possible additional perspective to be added to the MechatronicUML Tool Suite in the future. The deployment configuration is the core of the code generation for a distributed deployment and is therefore described in detail in the following.

Figure 8.8 shows that four `ECUConfiguration` instances are created, one for each of the ECUs in the HPIC (see Figure C.9 in Appendix C.2.2). An `ECUConfiguration` has a container for each component type allocated to the associated ECU. E.g., for the `fastCarDriverECU`, which has three different component types allocated to it (cf. Figure C.10), there are three corresponding containers in the `fastCarDriverECU_config`. Within a container, there is a `ContainerComponentInstanceConfiguration` that captures the component-specific configurations which are the individual port instances. These port instance configurations are also depicted, e.g., the `MCC_DriveControlComponent` container has one instance configuration with four different `PortInstanceConfiguration` subtypes; three of kind `PortInstanceConfiguration_Local` and one `PortInstanceConfiguration_I2C`. The `MCC_DistanceSensorComponent` container of the `fastCarDriverECU_config` has two `ContainerComponentInstanceConfiguration` instances, because two instances of the `DistanceSensor` component type are allocated to the respective ECU.

Furthermore, Figure 8.8 also shows the created I2C addresses and MQTT topics, as well as the identifiers used for local communication. These configurations realize the communication paths using the specific communication middleware: By matching identifiers, I2C addresses or MQTT topic names, communication partners can be identified. What is not depicted in the visualization of Figure 8.8 but still captured in the MechatronicUML Deployment Configuration model is the association of each `PortInstanceConfiguration` instance to both a platform-independent port as well as a hardware port (cf. Figure 7.1). This ensures, that the port instance configurations are each associated to the corresponding elements in the CIC and HPIC. Thus, when the container code generator uses the a deployment configuration, it can resolve these associations to generate the code and configure the ports for communication using the respective middleware.

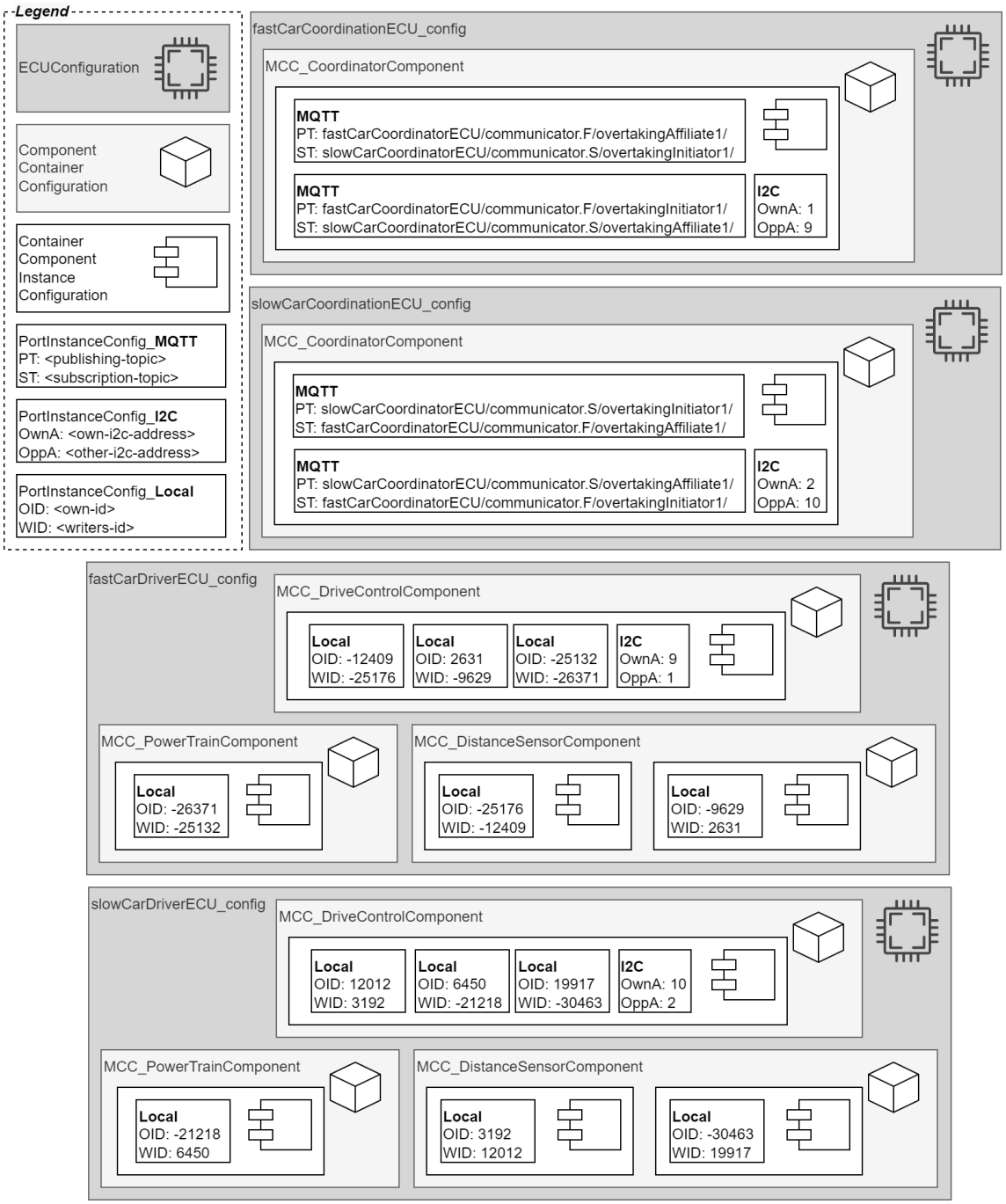


Figure 8.8: A visualization of the MechatronicUML Deployment Configuration model for the robot cars.

8.2.3 Container Code Generation

The previously described MechatronicUML Deployment Configuration model is stored as a `.muml_container` file and is used to start the container code generation. This corresponds to *T3.6* and *T3.7* of the software construction (see Figure 8.6). In the extended platform-modeling approach that does not use DDS, the container code and the communication artifacts are generated all at once (cf. Section 6.5 and Section 7.4). Appendix D.4 describes how the Arduino container code is generated with the MechatronicUML Tool Suite.

For the presented deployment configuration, the following six directories are created. Firstly, the `APImappings` directory which is explained in Appendix D.6, and secondly, the `mosquitto` directory which contains the Docker configuration for an MQTT server (cf. Section 7.4.2). The other four directories contain the container code for the four ECUs of the robot car HPIC (see Figure C.9): the `fastCarCoordinatorECU` and the `fastCarDriverECU` for the `fastCar` platform instance, and the `slowCarCoordinatorECU` and the `slowCarDriverECU` for the `slowCar` platform instance. These directories contain the container code.

Listing 8.2 shows the Arduino main file of the `fastCarDriverECU`. This ECU is configured for four component instances of three different types and has port instance configurations for I2C as well as local communication (cf. Figure 8.8). Lines 5-7 show the inclusion of the three containers for

Listing 8.2 The declarations and `setup` method of the Arduino main file for the `fastCarDriverECU`.

```
4  #include "I2cCustomLib.hpp"
5  #include "MCC_driveControlComponent.h"
6  #include "MCC_distanceSensorComponent.h"
7  #include "MCC_powerTrainComponent.h"
8  //variable for component Instances
9  DistanceSensorComponent* atomic_c1;
10 DistanceSensorComponent* atomic_c2;
11 PowerTrainComponent* atomic_c3;
12 DriveControlComponent* atomic_c4;
13
14 void setup(){
     ...
23   atomic_c1= MCC_create_DistanceSensorComponent(CI_FRONTDISTANCESENSORFDISTANCESENSOR);
24   atomic_c2= MCC_create_DistanceSensorComponent(CI_REARDISTANCESENSORFDISTANCESENSOR);
25   atomic_c3= MCC_create_PowerTrainComponent(CI_POWERTRAINFPOWERTRAIN);
26   atomic_c4= MCC_create_DriveControlComponent(CI_DRIVECONTROLFDRIVECONTROL);
27
28   i2cCommunication_setup(9);
     ...
34 }
```

the three component types, and lines 21-24 show their usage to create the component instances. The custom I2C library's setup method is called in line 28 using the I2C address 9 just as the configuration suggests (see Figure 8.8).

Each ECU's directory also includes container implementations, e.g., the `fastCarCoordinatorECU` directory contains the `MCC_coordinatorComponent.cpp` container implementation (cf. Section 7.3.4). The container implementation implements the component builder and port handle builder as detailed in Appendix D.5.1. Furthermore, Appendix D.5.2 demonstrates that the generated code successfully implements the *create* method including the configuration of the previously described deployment configuration (see Figure 8.8). Furthermore, Appendix D.5.3 shows the generated code of a *send* method to demonstrate the usage of the communication middleware libraries in the generated code.

All in all, these examples and code snippets demonstrate that the container code matches the component type code and implements its required communication in a platform-specific way. From the process for the software construction, the steps *T3.1*, *T3.4* and *T3.5* have not been explained in this chapter yet. They allow the modeling and inclusion of device APIs and software libraries which is important for the implementation of the application scenario because the robot car libraries contain the continuous behavior of the robot car (cf. Appendix A). Their integration into the generated code is required for the physical behavior, i.e., the sensing an moving. Thus, they complement the container code with additional platform-specific implementations. Appendix D.6 describes how the robot car libraries are integrated into the generated code to complete the implementation of the application scenario.

8.2.4 Building the Software

Lastly, the software is built using the simple, manual build process described in Section 7.5. Appendix D.7 provides additional details about required manual adaptions. This is the final step of the software construction (see *T3.8* in Figure 8.6) and provides an answer to the research question **RQ4.2:** Does the code generator produce valid source code for the modeled application scenario? The source code files for all four ECUs can be compiled correctly and deployed on the Arduino microcontrollers. The successful compilation demonstrates the static validity of the generated code[5]. In the course of this thesis, the executable software is tested only briefly due to time limitations. Thus, the dynamic validity of the generated code has not been assessed in detail, yet. The generated software makes the robot car move successfully, connects to the I2C bus, to the WiFi network and to MQTT server. But the coordination behavior has not been tested and verified. More work is required to inspect the software in detail to explore the interaction between the generated and integrated artifacts. The behavior models may also have to be refined. Additionally, the robot car libraries have to be extended with the functionality for changing lanes which has not been implemented thus far (cf. Appendix A).

[5] All source code is available in GitHub, see `arduino-containers_demo` in https://github.com/SQA-Robo-Lab/Overtaking-Cars.

9 Evaluation

This chapter presents the evaluation of what has been explored and achieved in this thesis. First, the taxonomy and case study design are described in Section 9.1. Then, the results are summarized in Section 9.2 and discussed in Section 9.3, before threats to the results' validity are reflected in Section 9.4.

9.1 Study Design

At the beginning of this thesis, a set of research questions are defined (see Section 1.2). They summarize the problem statement and are answered in the course of this work. These answers are compiled by following a case study design. The goal of the thesis is to develop a MechatronicUML-based code generator for distributed deployments. For this purpose, an application scenario is designed which serves as an exemplary use case. Developing this application scenario refines the problem space to be regarded an this thesis and provides answers to the first two research questions. Moreover, the taxonomy for model-based code generation and the new extended code generator are evaluated by applying this use case.

The taxonomy is evaluated by demonstrating its utility. It is applied to formulate ideal properties of a code generator that may be reused or extended for the application scenario. Furthermore, the taxonomy is used to structure the analysis of the previous code generation approaches. The analysis results in the choice of the platform-modeling approach as the foundation for the presented code generation approach.

Furthermore, the application scenario is applied to the presented code generator. The scenario is modeled with the MechatronicUML and the models are used for code generation. This case study setup seeks to evaluate the concepts of the code generator and its implementation and provides answers to the aforementioned research questions.

9.2 Results

The results of the thesis are described by answering the initial research questions. These answers are presented within the thesis and summarized in this section.

RQ1.1: What specific application scenario can be used to demonstrate the MechatronicUML modeling capabilities and the code generation? A variant of the cooperative overtaking scenario is used as the specific application scenario. It represents a distributed CPS example from the automotive domain and has been frequently used to demonstrate the MechatronicUML modeling capabilities (cf. Section 2.1). A variant of the cooperative overtaking scenario has also been used to demonstrate the original platform-modeling approach [BCD+14; Poh18].

RQ1.2: What type of robot car is used as the target platform? The robot cars presented in Section 2.2 are the Arduino-based target platform for the application scenario. They have basic driving, sensing and communication capabilities which makes them suitable to represent autonomous vehicles in a laboratory environment. The robot cars are described in detail in Appendix A.

RQ2.1: What criteria are applied to assess the previous MechatronicUML-based code generation approaches? Model-based code generation approaches may be assessed using the 23 facets and four perspectives of the proposed taxonomy for model based code generation (see Chapter 5). This taxonomy is tailored to analyze different code generators that are based on the same modeling language. Thus, it is applied to the two previous MechatronicUML-based code generation approaches, the platform-modeling approach and the MATLAB/Simulink approach, to describe their state and assess their fitness for the requirements in this thesis.

RQ2.2: What is the state of the previous MechatronicUML-based code generation approaches? The platform-modeling approach presents a sophisticated concept for the software construction for distributed deployments (cf. Section 6.1). It uses the MechatronicUML PIM and the MechatronicUML HPDM and unites them via allocation engineering. It also presents a concept for platform-specific modeling followed up by a C code generator. There is an existing executable version of the code generator, however, it lacks detailed documentation, especially from the user perspective.
The MATLAB/Simulink approach leverages the MATLAB/Simulink tooling for code generation (cf. Section 6.3). A model transformation from the MechatronicUML PIM to MATLAB/Simulink is its foundation. Conceptually, this transformation supports distributed deployments and presents a detailed mapping of MechatronicUML PIM features to MATLAB/Simulink elements. However, there is no working implementation of the approach. It has also been reimplemented due to incompatibilities, but this later reimplementation is neither available nor documented.

RQ3.1: Which (parts of) previous approaches are reused, and why or why not? The platform-modeling approach is suitable to be extended for the desired application scenario (cf. Section 6.4). The concepts for the platform modeling and allocation engineering are reused and the software construction including the platform-specific modeling is adapted to match the properties of the application scenario. Reusing the concepts of the platform-modeling approach allows to infer a similarly sophisticated deployment configuration for Arduino-based target platforms, which would not be feasible in the scope of this thesis if it was implemented from scratch. The deployment configuration is the foundation for reusing the platform-independent component code and generating

a platform-specific implementation that can handle the platform's technical details. The platform-specific modeling and code generation cannot be reused but have to be adapted for the application scenario.

RQ3.2: What are the missing capabilities of a MechatronicUML-based code generator for the desired application scenario? The platform-modeling approach is used as the baseline for the new MechatronicUML-based code generator. The generated code and the DDS-based communication are not applicable to the Arduino-based target environment. Instead, communication with I2C and the WiFi capabilities of the robot car is intended for the implementation of the cooperative overtaking scenario. The platform-specific modeling lacks support for MQTT and I2C communication and corresponding configurations (cf. Section 6.5). Furthermore, some user interface options are missing to select the desired communication middleware. The container code and communication middleware implementations have to be adapted to the robot cars' Arduino environment as well as building the software for Arduino.

RQ3.3: How are these missing capabilities implemented? These missing capabilities are implemented by extending the platform-modeling approach as presented in this thesis. Firstly, the MechatronicUML Deployment Configuration metamodel and its creation via a QVTo model-to-model transformation are adapted, and the new functionality is integrated into the user interface of the MechatronicUML Tool Suite. Additionally, a new pair of plugins for the MechatronicUML Tool Suite are introduced. These plugins implement the container generation for Arduino and integrate the newly created container code generator into the user interface. The code generation is implemented as an Acceleo model-to-text transformation and produces C and C++ source code for the Arduino environment.

RQ4.1: Can the application scenario be modeled with the MechatronicUML in a way that is suitable for code generation? The application scenario can be modeled using the MechatronicUML: the MechatronicUML PIM models the application's logical structure and behavior while the MechatronicUML HPDM supports the specification of the Arduino-based target platform. Both modeling languages allow to appropriately model the application scenario and are combined into a MechatronicUML Allocation Specification model (cf. Section 8.1). These models are used to generate a MechatronicUML Deployment Configuration model which is eventually employed to generate platform-specific container code (cf. Section 8.2). Additionally, the MechatronicUML PIM model is successfully used to generate the platform-independent component code.

RQ4.2: Does the code generator produce valid source code for the modeled application scenario? The extended platform-modeling approach produces source code for the application scenario that implements the platform-independent behavior and the platform-specific technical code. With some limitations, the generated artifacts match and can be compiled for Arduino. These limitations are of technical nature: some minor manual adaptions are required for building the software (cf. Section 7.5). These limitations can be reduced in the future. Moreover, the dynamic validity of the software, i.e., if the modeled behavior is executed correctly, has not been assessed as part of this thesis. The generated software makes the robot car move successfully, connects to

the I2C bus, to the WiFi network and to MQTT server. But the coordination behavior has not been tested and verified. More work is required to inspect the software in detail to explore the interaction between the generated and integrated artifacts.

9.3 Discussion

The goal of this thesis is to explore the code generation opportunities for distributed deployments of CPS based on MechatronicUML models. For this purpose, an application scenario and a taxonomy for model-based code generation are presented. Furthermore, a code generator is envisioned to generate source code for a laboratory environment of Arduino-based robot cars. In the thesis, the code generator is conceptualized and implemented. This section discusses, to what extent the results accomplish the overall goals of the thesis.

The presented work shows that the cooperative overtaking scenario is a suitable application scenario to demonstrate the MechatronicUML modeling and code generation capabilities. Furthermore, the scenario is suitable to be realized with robot cars that represent autonomous vehicles in a laboratory environment.

The taxonomy for model-based code generation allows a structured analysis and comparison of the previous MechatronicUML-based code generators. The usage of the taxonomy for the analysis shows that all 23 facets are applicable to both approaches and provide meaningful information about their characteristics (cf. Section 6.4). The analysis also shows that the platform-modeling approach fits best to the desired properties, based on a fitness rating assigned to 15 of those facets. When defining the ideal properties, the taxonomy's flexibility to prioritize certain perspectives or leave out uninteresting facets for a particular use case is demonstrated. Nonetheless, the analysis conclusion in Section 6.4 shows that there are also notable differences between the platform-modeling approach and the MATLAB/Simulink approach that are not captured by the taxonomy's faceted analysis alone. These differences complete the comparison of the approaches and confirm the outcome of the taxonomy-based analysis: the platform-modeling approach is suitable to be extended for realizing the desired application scenario, and also better suited that the MATLAB/Simulink approach.

The concept of the platform-modeling approach is generic enough to be adaptable to different target platforms. The feasibility of this adaption is demonstrated by extending the approach to support Arduino-based environments as well as different communication middleware implementations. The implementation described in this thesis also reveals which parts of the platform-modeling approach require an adaption when a new target platform is added. Furthermore, the analysis of the previous code generation approaches, the concept for the platform-modeling approach extension and its implementation describe the state of the platform-modeling approach in great detail. The MATLAB/Simulink approach has not been analyzed with the same depth, but only as part of the approaches' initial analysis. Thus, the description of its state is certainly less thorough.

Furthermore, the case study shows how the MechatronicUML models are used by the extended platform-modeling approach to generate source code. The software construction process comprises creating a MechatronicUML Deployment Configuration which is appropriately created for the Arduino-based target platform and includes configurations for MQTT and I2C communication. The case study also shows how the platform-specific and the platform-independent code match and complement each other. Finally, the source code can also be compiled to an executable using the

Arduino tooling and is deployed to the robot cars. However, the dynamic validity of the software, i.e., if the modeled behavior is executed correctly, has not been assessed in detail. Thus, the code generator has not yet shown to be useful for rapid testing in a laboratory environment, but has to be improved with respect to the generated code.

9.4 Threats to Validity

This section discusses what may threaten the validity of the presented results. Furthermore, countermeasures are described if they were applied.

The taxonomy for model-based code generation was inspired by the previously existing code generation approaches related to the MechatronicUML. When analyzing them initially, the need for an assessment scheme was identified and resulted in the initial idea for developing a taxonomy. Thus, the statement that all 23 facets are applicable to the analyzed approaches has to be regarded in this context. The two assessed approaches were already roughly known when the taxonomy was defined, therefore the taxonomy may fit especially well to these particular approaches. However, as countermeasures, the taxonomy is also aligned with existing related work from the domain of model transformation taxonomies. Furthermore, the taxonomy has been discussed independently of the approaches between the author and supervisor of this thesis. And lastly, the taxonomy's facets were fixed and employed to define the ideal candidate properties in advance of the detailed analysis of the two related approaches. So there was only a rough idea of the approaches at this point and no detailed knowledge.

Secondly, regarding the choice of the platform-modeling approach as the foundation for the presented implementation, there may have been some researcher bias: By getting in contact with the researchers involved in the platform-modeling approach first, it has also been analyzed first and appeared promising right from the start. Nonetheless, the researchers who developed the MATLAB/Simulink transformation were also contacted to find out more about their approach.

Moreover, regarding the implementation of the application scenario including the created MechatronicUML models, it is worth mentioning that the author of this thesis has no expertise in the engineering of CPS like control engineers might have. The scenario itself and also the modeling are inspired by previous work regarding the MechatronicUML (cf. Section 2.1). Thus, the presented models are inevitably influenced by these works. Furthermore, the scenario, its modeling, and its implementation are tailored to the robot car laboratory environment and do not precisely reflect a real-world use case.

Lastly, by the nature of a case study, its results are always somewhat limited to the application scenario that is used for the study. Thus, the cooperative overtaking and its realization with the Arduino-based robot cars influence the results and limit their universal validity.

10 Conclusion

Finally, this chapter concludes the thesis about generating code for distributed deployments of CPS using the MechatronicUML. It summarizes the key aspects in Section 10.1, the benefits in Section 10.2 and limitations in Section 10.3. Afterwards, the lessons learned are reflected in Section 10.4 and a final outlook for future work and research is given in Section 10.5.

10.1 Summary

This thesis introduces a taxonomy for model-based code generation. It serves to describe model-based code generation approaches from four perspectives: the perspective of the code generation concept, the usability of the code generator, the implementation of the code generator, and the generated code. Different facets from each of these perspectives serve to characterize an individual approach holistically.

Furthermore, based on this taxonomy, the thesis presents the assessment of two previous code generation approaches that use the MechatronicUML to generate code for distributed deployments of CPS. The MATLAB/Simulink approach on the one hand has the potential to integrate the MATLAB/Simulink tooling into the MechatronicUML-based development of a CPS. However, the approach is deprecated and unmaintained, hence hard to reuse in its current state. Additionally, the MATLAB/Simulink approach is not primarily tailored to code generation. On the other hand, the platform-modeling approach defines a sophisticated concept for the hardware platform modeling and platform-specific modeling of the software. These intermediate modeling steps lay the foundation for a code generator. This code generator is suitable to be adapted and extended to other target platforms.

The Arduino-based robot cars serve as an exemplary target platform throughout this thesis. The platform-modeling approach is adapted to support Arduino microcontrollers and extended to support I2C and MQTT for communication. A proof of concept implementation of this new code generator is provided and demonstrates that it is feasible to generate source code for a distributed deployment of a CPS which is modeled with the MechatronicUML modeling languages. Only the correct functionality of the generated code is not demonstrated yet and requires further research.

10.2 Benefits

Firstly, researchers as well as software engineers who want to describe or analyze the properties of a model-based code generation approach may use the proposed taxonomy to structure their analysis. The taxonomy is designed as a faceted analysis. It may be used to define properties of an envisioned or desired code generation approach, and also for the comparison of different existing approaches.

Other applications are also feasible. Users of the taxonomy benefit from a predefined structure for the analysis that may be flexibly adapted to the particular focus aspects of a specific analysis subject.

Secondly, the extension of the platform-modeling approach and the corresponding implementation support the code generation for Arduino-based platforms. This proves that the platform-modeling approach is in fact extensible and adaptable for other target platforms. The thesis also demonstrates how new communication technologies may be added to the approach. Both are beneficial for users of the platform-modeling approach or researchers who want to realize an application scenario with a different target platform. Furthermore, with respect to Arduino-based target platforms, the thesis lays the foundation for experimenting with the behavior of the software and evaluating and improving the functionality of the generated code. Thus, the presented code generation is the basis for the envisioned laboratory environment with robot cars operating on MechatronicUML-based software.

10.3 Limitations

The intention of the presented taxonomy for model-based code generation approaches is to compare approaches which use the same modeling language. It is used specifically to compare two MechatronicUML-based approaches in this thesis. Also, it is not suitable to compare different modeling languages; it lacks this perspective. But the four proposed perspectives may be added to a comparison of different modeling languages to characterize the respective code generation capabilities.

Moreover, the described extension of the platform-modeling approach is limited to Arduino-based platforms using MQTT and I2C by design. Other platforms and communication technologies may be added manually. Additionally, for the Arduino-based target platform, the described implementation is limited to produce statically correct code, i.e., the software can be built and deployed. It does not yet demonstrate the functional correctness of the generated code. Lastly, the platform-modeling approach also has limitations regarding the MechatronicUML PIM modeling features which it can use for code generation. The extended platform-modeling approach does not overcome these limitations.

10.4 Lessons Learned

Different things have been learnt in the process of compiling this thesis. First of all, the lack of documentation makes it very cumbersome to get familiar with a software product. Academic publications or reports typically neither adequately describe the implementation, nor how the employed tools have been used to create a software, nor how to use the created software product. The target and focus of academic publications or reports are often the concepts that have been used to overcome the gap between problem and solution space. Thus, it is beneficial if specific developer documentations and user manuals are published in addition, and especially linked to the source code and academic works.

Furthermore, the author of this thesis experienced the variety of contexts that are faced in developing or analyzing model-based code generators. Firstly, there is the modeling language to be understood, including its abstract and concrete syntax and semantics. Additionally, the tools and editors for this modeling language have to be explored, as well as the tools and languages that have been used to define the metamodel and build these tools and editors. The same applies to potential intermediate metamodels and editors. Additionally, the tools and languages for model transformations have to be learned. And finally, when a model-to-text transformation is used to generate source code, knowledge about the target platform and the used programming language and technologies has to be possessed. Thus, acquiring this knowledge and expertise means being exposed to a variety of different contexts, starting at the modeling language and concluding with the generated code. This is characteristic for the upfront investment in a model-based code generator: For a one-time usage or a single scenario, handcrafted code is produced a lot faster with less expertise and knowledge required. The benefits of model-based code generation are only evident once the foundation is laid and a working code generator is created.

10.5 Future Work

This thesis lays the foundation for generating code for distributed Arduino-based deployments of a CPS using the MechatronicUML. Both the code generator and the generated code can be improved and extended by future work.

First of all, with respect to the code generator, more configuration options for the communication middleware could be integrated into the user interface, e.g., the configurations for the MQTT server connection or the WiFi. The configuration options may also be extended, e.g., to support authentication with the MQTT. Additionally, the component type code generation can be adapted for Arduino or even replaced by an entirely new C++ component type code generator. Using C++ language features might also simplify the implementation of the port handle concept and may make it easier to be adapted for further communication technologies.

Moreover, the generated code has to be tested for functional correctness. The coordination behavior has not been tested and verified. More work is required to inspect the software in detail to explore the interaction between the generated and integrated artifacts. The behavior models may also have to be refined. Similarly, more MechatronicUML PIM modeling features such as multi-ports may be enabled for code generation to broaden the set of supported features. Potentially, it may also be beneficial to create a concept for marshaling MechatronicUML messages so that they can be exchanged between different target platforms, i.e., not just between Arduinos. This may done together with improving and unifying the functionality of the custom communication libraries: they have a lot of similarities but are completely independent at the moment. Additionally, technical limitations can be reduced by future work. For instance, the limitation that the message buffers are created for one message type only as well as the limitations for building the software.

The robot cars are also developed in this work including their hardware and software. Both may also be the target of future work, e.g., the robot car may be equipped with different ECUs with more computing power or different sensors such as cameras to realize more advanced automotive scenarios. With respect to the software, the robot car libraries, a next step could be the implementation of further driving mechanisms that may then be used for modeling and implementing different scenarios such as autonomous parking or driving in a convoy.

The extension of the platform-modeling approach has been implemented as part of the existing repository infrastructure of the MechatronicUML Tool Suite. However, as the development of the MechatronicUML Tool Suite was discontinued, there is no active build server or Eclipse update site. If there was ongoing development of the platform-modeling approach and the MechatronicUML Tool Suite, it would be beneficial to setup this infrastructure again.

Finally, with respect to the MechatronicUML, a technical report that sums up the platform-specific modeling and code generation concepts, metamodels and its process may be very useful to create a documentation and specification like there already is for the MechatronicUML PIM and MechatronicUML HPDM. Furthermore, there are two additional views that can be implemented in the MechatronicUML Tool Suite: Firstly, the MechatronicUML Allocation Specification view which was proposed by Pohlmann in [Poh18], and secondly a view for MechatronicUML Deployment Configuration model instances, similar to the suggested visualization in Figure 8.8.

Ultimately, the presented code generator and its implementation demonstrate the feasibility of generating source code for the distributed deployment of a CPS using the MechatronicUML. This work lays the foundation for further research and experimentation with the MechatronicUML in a laboratory environment with robot cars.

Bibliography

[ABBG17] A. Ataíde, J. P. Barros, I. S. Brito, L. Gomes. "Towards automatic code generation for distributed cyber-physical systems: A first prototype for Arduino boards". In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2017, pp. 1–4. DOI: 10.1109/ETFA.2017.8247737 (cit. on p. 43).

[ABRW20] A. Angermann, M. Beuschel, M. Rau, U. Wohlfarth. *Matlab–simulink–stateflow*. De Gruyter Oldenbourg, 2020 (cit. on p. 30).

[Ard22] Arduino SA. *Sketch build process*. Online. 2022. URL: https://arduino.github.io/arduino-cli/0.20/sketch-build-process/ (cit. on pp. 6, 93, 98).

[BCC+08] T. Bureš, J. Carlson, I. Crnkovic, S. Sentilles, A. Vulgarakis. "ProCom–the Progress Component Model Reference Manual". In: *Mälardalen University, Västerås, Sweden* (2008) (cit. on p. 44).

[BCD+14] J. Bobolz, M. Czech, A. Dann, J. Geismann, M. Huwe, A. Krieger, G. Piskachev, D. Schubert, R. Wohlrab. *Final Document*. Tech. rep. Project Group Cyberton, Heinz Nixdorf Institute, University of Paderborn, 2014 (cit. on pp. 5, 16, 35, 53, 61, 63, 64, 99, 104, 114, 157, 172).

[BDG+14] S. Becker, S. Dziwok, C. Gerking, W. Schäfer, C. Heinzemann, S. Thiele, M. Meyer, C. Priesterjahn, U. Pohlmann, M. Tichy. *The MechatronicUML Design Method - Process and Language for Platform-Independent Modeling*. Tech. rep. tr-ri-14-337. Version 0.4. Zukunftsmeile 1, 33102 Paderborn, Germany: Heinz Nixdorf Institute, University of Paderborn, Mar. 2014 (cit. on p. 67).

[BGG+14] C. Buckl, M. Geisinger, D. Gulati, F. J. Ruiz-Bertol, A. Knoll. "CHROMOSOME: A Run-Time Environment for Plug & Play-Capable Embedded Real-Time Systems". In: *SIGBED Rev.* 11.3 (Nov. 2014), pp. 36–39. DOI: 10.1145/2692385.2692391. URL: https://doi.org/10.1145/2692385.2692391 (cit. on p. 44).

[BNS13] A. Bajovs, O. Nikiforova, J. Sejans. "Code Generation from UML Model: State of the Art and Practical Implications". In: *Applied Computer Systems* 14 (June 2013). DOI: 10.2478/acss-2013-0002 (cit. on p. 32).

[CFMS10] J. Carlson, J. Feljan, J. Mäki-Turja, M. Sjödin. "Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems". In: *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. 2010, pp. 74–82. DOI: 10.1109/SEAA.2010.43 (cit. on p. 44).

[CH03] K. Czarnecki, S. Helsen. "Classification of model transformation approaches". In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. USA. 2003, pp. 1–17 (cit. on pp. 14, 32).

[CH06] K. Czarnecki, S. Helsen. “Feature-based survey of model transformation approaches”. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. DOI: 10.1147/sj.453.0621 (cit. on pp. 9, 11–14, 32, 33, 51).

[DGB+14] S. Dziwok, C. Gerking, S. Becker, S. Thiele, C. Heinzemann, U. Pohlmann. “A Tool Suite for the Model-Driven Software Engineering of Cyber-Physical Systems”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 715–718. ISBN: 9781450330565. DOI: 10.1145/2635868.2661665. URL: https://doi.org/10.1145/2635868.2661665 (cit. on pp. 5, 17).

[DP] A. Dann, U. Pohlmann. *The MechatronicUML Hardware Platform Description Method–Process and Language*. Tech. rep. tr-ri-14-336. Version 0.1. Fraunhofer IPT, Project Group Mechatronic Systems Design, Paderborn, Germany (cit. on pp. 1, 16, 17, 25–29, 138, 153).

[DPP+16] S. Dziwok, U. Pohlmann, G. Piskachev, D. Schubert, S. Thiele, C. Gerking. *The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling*. Tech. rep. tr-ri-16-352. Version 1.0. Zukunftsmeile 1, 33102 Paderborn, Germany: Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Dec. 2016 (cit. on pp. 1, 15, 16, 18–25, 98).

[DŠ08] R. Damaševičius, V. Štuikys. “Taxonomy of the fundamental concepts of metaprogramming”. In: *Information Technology and Control* 37.2 (2008) (cit. on p. 32).

[Ecl22] Eclipse Foundation, Inc. *Acceleo | Home*. Online. 2022. URL: https://www.eclipse.org/acceleo/ (cit. on p. 65).

[eLa18] eLab, Individual Network Berlin e.V. *I2C*. Online. 2018. URL: https://robotfreak.de/elab-wiki/index.php?title=I2C (cit. on p. 83).

[Fra22a] Fraunhofer Institute for Mechatronic Systems Design IEM. *Downloads – MachatronicUML*. Online. 2022. URL: http://www.mechatronicuml.org/en/download.html (cit. on p. 17).

[Fra22b] Fraunhofer Institute for Mechatronic Systems Design IEM. *MechatronicUML*. Online. 2022. URL: http://www.mechatronicuml.org/en/index.html (cit. on p. 1).

[GG18a] D. Gritzner, J. Greenyer. “Generating Correct, Compact, and Efficient PLC Code from Scenario-based Assume-Guarantee Specifications”. In: *Procedia Manufacturing* 24 (2018). 4th International Conference on System-Integrated Intelligence: Intelligent, Flexible and Connected Systems in Products and Production, pp. 153–158. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.06.046. URL: https://www.sciencedirect.com/science/article/pii/S2351978918305651 (cit. on p. 43).

[GG18b] D. Gritzner, J. Greenyer. “Synthesizing Executable PLC Code for Robots from Scenario-Based GR(1) Specifications”. In: *Software Technologies: Applications and Foundations*. Ed. by M. Seidl, S. Zschaler. Cham: Springer International Publishing, 2018, pp. 247–262. ISBN: 978-3-319-74730-9 (cit. on p. 43).

[GGK+17] J. Greenyer, D. Gritzner, F. König, J. Dahlke, J. Shi, E. Wete. "From Scenario Modeling to Scenario Programming for Reactive Systems with Dynamic Topology". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 974–978. ISBN: 9781450351058. DOI: 10.1145/3106237.3122827. URL: https://doi.org/10.1145/3106237.3122827 (cit. on p. 43).

[Hei14a] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *Cybertron User Guide*. Online. 2014. URL: https://trac.cs.upb.de/mechatronicuml/wiki/PGCybertron/userguide (cit. on p. 35).

[Hei14b] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *Features of the MechatronicUML Tool Suite*. Online. 2014. URL: https://trac.cs.upb.de/mechatronicuml/wiki/user/MUMLFeatureDescription (cit. on p. 17).

[Hei14c] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *MechatronicUML Tool Suite Features*. Online. 2014. URL: https://trac.cs.upb.de/mechatronicuml/wiki/ToolFeatures (cit. on p. 17).

[Hei14d] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *Version History*. Online. 2014. URL: https://trac.cs.upb.de/mechatronicuml/wiki/VersionHistory (cit. on p. 17).

[Hei15] C. Heinzemann. "Verification and simulation of self-adaptive mechatronic systems". eng. Tag der Verteidigung: 30.07.2015. PhD thesis. Paderborn University, 2021 2015. URL: https://nbn-resolving.de/urn:nbn:de:hbz:466:2-16778 (cit. on pp. 39–42, 66–69).

[Hei16] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *Release 1.0*. Online. 2016. URL: https://trac.cs.upb.de/mechatronicuml/wiki/user/Release_1_0 (cit. on p. 17).

[Hei17a] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *Installing the MechatronicUML Tool Suite*. Online. 2017. URL: https://trac.cs.upb.de/mechatronicuml/wiki/InstallMechatronicUMLToolSuite (cit. on p. 17).

[Hei17b] Heinz Nixdorf Institut, Universität Paderborn and Fraunhofer IEM. *Installing the MechatronicUML Tool Suite via our Eclipse Update-Site*. Online. 2017. URL: https://trac.cs.upb.de/mechatronicuml/wiki/InstallMechatronicUMLToolSuite/Binary (cit. on p. 17).

[Hei21] C. Heinzemann. *Private Conversation*. Dec. 2021 (cit. on pp. 39, 42, 68).

[HFK+16] J. Holtmann, M. Fockel, T. Koch, D. Schmelter, C. Brenner, R. Bernijazov, M. Sander. *The MechatronicUML Requirements Engineering Method: Process and Language*. Tech. rep. tr-ri-16-351. Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Dec. 2016 (cit. on pp. 1, 16).

[HRB+14] C. Heinzemann, J. Rieke, J. Bröggelwirth, A. Pines, A. Volk. *Translating MechatronicUML Models to MATLAB/Simulink and Stateflow*. Tech. rep. tr-ri-14-338. Version 0.4.1. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2014 (cit. on pp. 30, 39–42, 66–69, 73).

[IS12] R. Inam, M. Sjödin. “Implementing and Evaluating Communication-Strategies in the ProCom Component Technology”. In: *SIGBED Rev.* 9.4 (Nov. 2012), pp. 41–44. DOI: 10.1145/2452537.2452545. URL: https://doi.org/10.1145/2452537.2452545 (cit. on p. 45).

[KC15] N. Kahani, J. R. Cordy. “Comparison and evaluation of model transformation tools”. In: *Queen’s University, Kingston, Tech. Rep.* (2015) (cit. on pp. 9, 11, 12, 14, 32, 33, 49, 51).

[Led15] L. Lednicki. “Software and hardware models in component-based development of embedded systems”. PhD thesis. Mälardalen University, 2015 (cit. on p. 45).

[Lee15] E. Lee. “The past, present and future of cyber-physical systems: a focus on models”. In: *Sensors (Basel)* 15.3 (Feb. 26, 2015), pp. 4837–4869. DOI: 10.3390/s150304837 (cit. on p. 1).

[Mad18a] F. Madiot. *Acceleo - Eclipsepedia*. Online. 2018. URL: https://wiki.eclipse.org/Acceleo (cit. on p. 14).

[Mad18b] F. Madiot. *Acceleo/User Guide*. Online. 2018. URL: https://wiki.eclipse.org/Acceleo/User_Guide (cit. on pp. 14, 15).

[Mat22a] Mathworks, Inc. *C Libraries in MATLAB*. Online. 2022. URL: https://de.mathworks.com/help/matlab/call-c-library-functions.html (cit. on p. 67).

[Mat22b] Mathworks, Inc. *Compare System Target File Support Across Products*. Online. 2022. URL: https://de.mathworks.com/help/rtw/ug/compare-system-target-file-support.html (cit. on pp. 41, 66, 68).

[Mat22c] Mathworks, Inc. *Custom Targets*. Online. 2022. URL: https://de.mathworks.com/help/rtw/ug/overview-of-embedded-target-development.html (cit. on p. 41).

[Mat22d] Mathworks, Inc. *Generate C Code from Simulink Model*. Online. 2022. URL: https://de.mathworks.com/help/rtw/ug/generating-code-using-simulink-coder.html (cit. on pp. 39–42, 66, 69).

[Mat22e] Mathworks, Inc. *Generate Code Using Simulink® Coder™*. Online. 2022. URL: https://de.mathworks.com/help/dsp/ug/generate-c-code-from-simulink-model.html (cit. on pp. 41, 66, 69).

[MCG05] T. Mens, K. Czarnecki, P. V. Gorp. “A Taxonomy of Model Transformations”. In: *Language Engineering for Model-Driven Software Development*. Ed. by J. Bezivin, R. Heckel. Dagstuhl Seminar Proceedings 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: http://drops.dagstuhl.de/opus/volltexte/2005/11 (cit. on p. 31).

[MKC21] M. A. Mohamed, G. Kardas, M. Challenger. “Model-Driven Engineering Tools and Languages for Cyber-Physical Systems–A Systematic Literature Review”. In: *IEEE Access* 9 (2021), pp. 48605–48630. DOI: 10.1109/ACCESS.2021.3068358 (cit. on p. 42).

[MQT22] MQTT.org. *MQTT: The Standard for IoT Messaging*. Online. 2022. URL: https://mqtt.org/ (cit. on pp. 76, 82, 83).

[MV06] T. Mens, P. Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), pp. 125–142. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.10.021. URL: https://www.sciencedirect.com/science/article/pii/S1571066106001435 (cit. on pp. 32, 51).

[Obj08] Object Management Group (OMG). *MOF Model to Text Transformation Language*. Tech. rep. v1.0. Object Management Group (OMG), 2008. URL: http://www.omg.org/spec/MOFM2T/1.0/PDF (cit. on pp. 14, 15).

[Obj14] Object Management Group (OMG). *MDA Guide rev. 2.0*. Tech. rep. Object Management Group (OMG), 2014 (cit. on pp. 1, 9, 11).

[Obj15] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification*. Tech. rep. Version 1.2. Object Management Group (OMG), 2015. URL: http://www.omg.org/spec/QVT/1.2/PDF/ (cit. on pp. 12, 13).

[Obj19] Object Management Group (OMG). *DDS for eXtremely Resource Constrained Environments*. Tech. rep. Object Management Group (OMG), 2019. URL: https://www.omg.org/spec/DDS-XRCE/1.0/PDF (cit. on p. 76).

[Poh18] U. Pohlmann. "A model-driven software construction approach for cyber-physical systems". eng. Tag der Verteidigung: 16.02.2018. PhD thesis. Paderborn: Paderborn Universtiy, 2021 2018, 1 Online–Ressource (XII, 323 Seiten). URL: https://nbn-resolving.de/urn:nbn:de:hbz:466:2-30659 (cit. on pp. 5, 6, 35–38, 61–65, 75, 76, 79, 81, 82, 87, 90, 93, 99, 104, 107, 114, 122, 133, 134, 138, 157, 162, 172–175).

[RRW14] J. O. Ringert, B. Rumpe, A. Wortmann. "From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems". In: *CoRR* abs/1408.5690 (2014). arXiv: 1408.5690. URL: http://arxiv.org/abs/1408.5690 (cit. on pp. 43, 44).

[RRW15] J. O. Ringert, B. Rumpe, A. Wortmann. "Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton". In: *CoRR* abs/1509.04505 (2015). arXiv: 1509.04505. URL: http://arxiv.org/abs/1509.04505 (cit. on pp. 43, 44).

[SCDP07] I. Stuermer, M. Conrad, H. Doerr, P. Pepper. "Systematic Testing of Model-Based Code Generators". In: *IEEE Transactions on Software Engineering* 33.9 (2007), pp. 622–634. DOI: 10.1109/TSE.2007.70708 (cit. on p. 9).

[Sel03] B. Selic. "The pragmatics of model-driven development". In: *IEEE Software* 20.5 (2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146 (cit. on pp. 11, 50, 51, 53).

[Söd22] K. Söderby. *How to upload a sketch with the Arduino IDE 2.0*. Online. 2022. URL: https://docs.arduino.cc/software/ide-v2/tutorials/getting-started/ide-v2-uploading-a-sketch (cit. on pp. 6, 93, 98).

[Sof20] Software Design and Quality Group, Prof. Reussner Karlsruhe Institute of Technology. *QVT*. Online. 2020. URL: https://sdqweb.ipd.kit.edu/wiki/QVT#QVT_Operational (cit. on p. 13).

[UBBM17] M. Usman, R. Britto, J. Börstler, E. Mendes. “Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method”. In: *Information and Software Technology* 85 (2017), pp. 43–59. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.01.006. URL: https://www.sciencedirect.com/science/article/pii/S0950584917300472 (cit. on pp. 47, 48).

[Wil22] E. Willink. *QVTo - Eclipsepedia*. Online. 2022. URL: https://wiki.eclipse.org/QVTo (cit. on pp. 12, 13).

[Wol09] W. Wolf. “Cyber-physical systems”. In: *Computer* 42.03 (2009), pp. 88–89 (cit. on p. 1).

[Zan17] S. Zanero. “Cyber-Physical Systems”. In: *Computer* 50.4 (2017), pp. 14–16. DOI: 10.1109/MC.2017.105 (cit. on p. 1).

All links were last followed on May 6, 2022.

A Supplementary Material for the Robot Cars

This appendix contains additional material for the robot cars. The robot cars are the target platform for the code generation, and are used as a laboratory environment for automotive application scenarios (cf. Section 2.2. They are designed specifically for this thesis based on the hardware that was available in the Software Quality and Architecture Group's *RoboLab* at the University of Stuttgart. Two robot cars were assembled for the laboratory as part of this work. In detail, a robot car employs the following hardware components with their intended capabilities:

- One *Arduino Mega 2560 Rev3*[1] microcontroller: Acts as ECU for controlling the driving behavior of the car by reading the sensors and instructing the motor controller.
- One *Arduino Nano*[2] microcontroller: Acts as ECU for coordination behavior with other cars by using the WiFi module.
- One *L298N* motor controller: Capable of connecting simple DC motors to an external power supply as well as controlling speed and operation direction of the motors. Also capable of providing a 5V output.
- One *ESP8266-01S* WiFi module (including breakout board): Capable of connecting to an existing WiFi network.
- Two *HC-SR04* ultrasonic distance sensors: Capable of measuring the distance to other objects. The sensors are employed at the front of the car (detect slower cars that are approached) and at the rear facing right and backwards (detect whether a car was passed).
- Three *KY-033* infrared sensors: Capable of detecting a line on the ground by reflection, and in union capable of determining the car's position in relation to the line (i.e., whether the car is left of the line, right of the line, or on the line).
- Four simple DC motors with attached wheels: Capable of moving the car if they are supplied with a corresponding voltage (forward and reverse depending on the polarity, and the more electric current they are supplied with, the higher the operation speed).
- A rechargeable 12V battery as power supply, including a power switch and a low voltage alarm for convenience: Capable of supplying electric energy for all sensors, actuators and ECUs.
- And finally, a simple robot car chassis, glue, cables and small breadboards to assemble everything.

[1] https://docs.arduino.cc/hardware/mega-2560
[2] https://docs.arduino.cc/hardware/nano

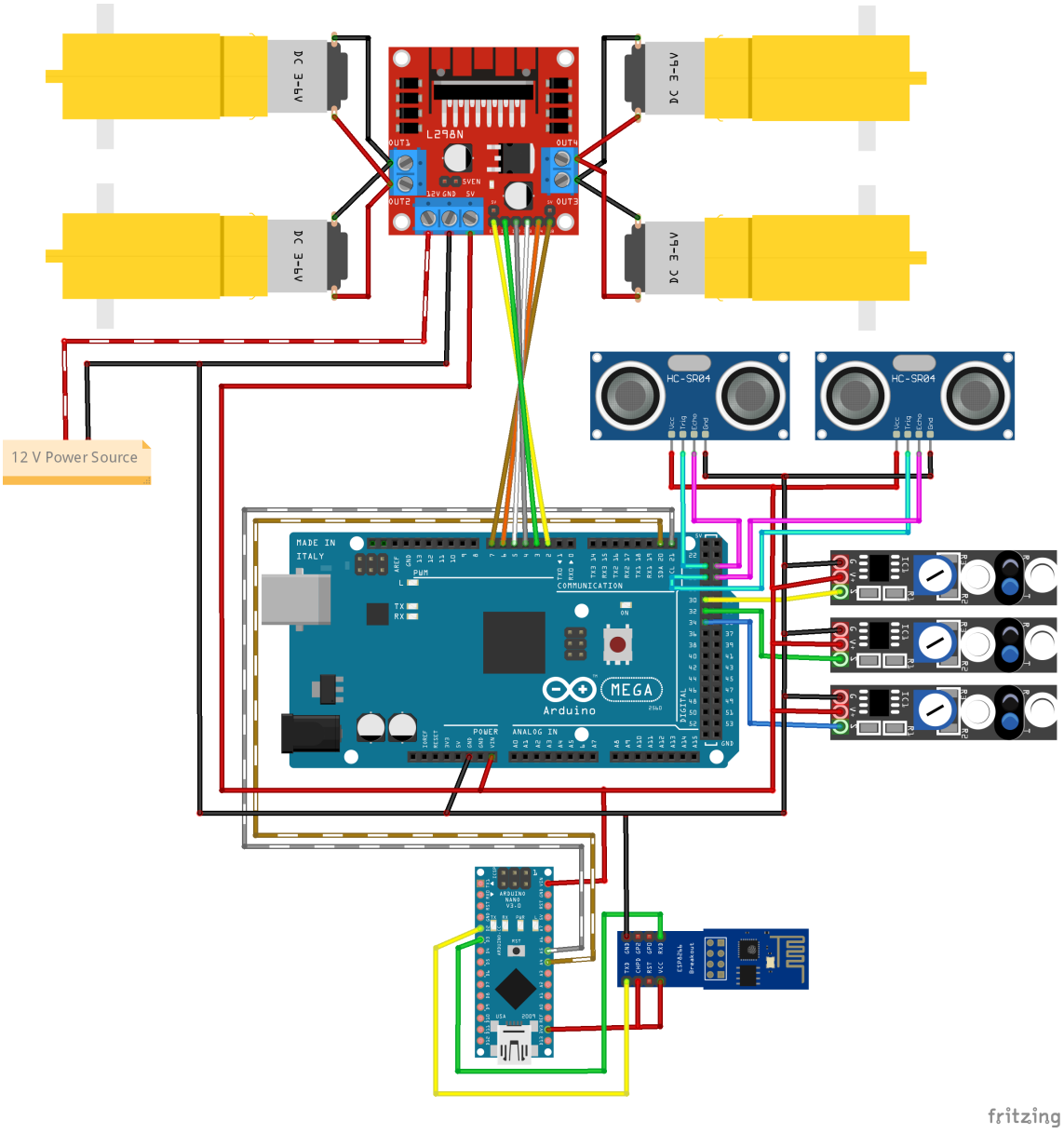


Figure A.1: The wiring sketch of the robot car hardware platform.

The logical assembly of these components is visualized in the wiring sketch in Figure A.1. Breadboards are left out for simplicity, but were applied for the connection to the power supply. Same applies to the battery switch and low voltage alarm that are not depicted. Thus, according to the sketch, the aforementioned hardware components are wired as follows:

- The 12V battery (power source) is connected to the motor controller's 12V input (red-striped wire).
- All components share a common ground connection (black wire).

- The left DC motors are connected to the OUT1 and OUT2 ports of the motor controller, and the right motors to the ports OUT4 and OUT3 respectively.
- The 5V outlet of the motor controller is connected to the respective voltage input ports of all other hardware components (red wire) except the WiFi module as it requires a voltage of 3.3V only.
- The input pins of the motor controller are connected to the Arduino Mega's digital pins as follows: ENA-D2, IN1-D3, IN2-D4, IN3-D5, IN4-D6, ENB-D7. Notably, the pins connected to ENA and ENB need to support Pulse Width Modulation (PWM) in order to regulate the operating speed of the motors.
- The ultrasonic distance sensors are connected to digital pins of the Arduino Mega as follows: Trig-D24 and Echo-D25 for the rear sensor, and Trig-D26 and Echo-D27 for the front sensor.
- The infrared sensors are connected to the Arduino Mega's digital pins D30 (left sensor), D32 (center sensor) and D34 (right sensor).
- The WiFi module is connected to the Arduino Nano as follows: TXD-D2, RXD-D3, VCC-3V3, CHPD-3V3. The ditial pins of the Arduino Nano are used for a software-based serial connection to the WiFi module.
- Finally, the two microcontrollers are connected via an I2C bus using the SDA and SCL pins of the Arduino boards. On the Arduino Mega, these are the D20 (SDA) and D21 (SCL), on the Arduino Nano the I2C pins are A4 (SDA) and A5 (SCL).

Furthermore, in order to access the hardware functionality of the aforementioned sensors and actuators, a set of libraries was implemented to encapsulate the device access. The libraries make use of the capabilities of the Arduino environment. They are designed to abstract from the hardware access and provide a meaningful, functional interface. The library implementation is tailored to the robot car which is described above: The libraries include the wiring as configuration and are designed for this exact setup, e.g., they are tailored to the exact amount of sensors of a certain type. These libraries are the following:

LineDetector The `LineDetector` library defines an interface to use the three infrared sensors. Next to an initialization method, it defines the enumeration type `LinePosition` with the possible values `LEFT_OF_LINE`, `ON_LINE`, and `RIGHT_OF_LINE`. The core method is `LinePosition detectPosition( LinePosition previousPosition)` which allows to determine the relative position of the robot car w.r.t. the line on the ground. The `LineDetector` library is implemented using the Arduino standard library.

MotorDriver The `MotorDriver` library abstracts from accessing the four DC motors via the motor controller. It provides the following interface methods: `void initMotorDriver()` to setup the library, `void driveForward(int speed)` and `void driveReverse(int speed)` for basic forward/backwards movement, `void stop()` to stop all motors and `void turnLeftForward(int speed)` and `void turnRightForward(int speed)` to allow basic steering capabilities while driving forward. The library is implemented using the Arduino standard library.

DistanceSensor This library gives access to the measurements of the two ultrasonic distance sensors. Thus, it provides the following interface methods: one initialization method `void initializeDistanceSensors()` as well as two getter methods `int getFrontDistance()` and `int getRearDistance` that return a centimeter value of the measured distance of the respective sensor. The `DistanceSensor` library is implemented using the Arduino standard library.

LineFollower The `LineFollowerLibrary` uses the `LineDetector` and `MotorDriver` libraries to define the line following behavior, which represents the autonomous driving behavior of the robot cars. It wraps the setup of the required libraries in its own `void initLineFollower()` method and exposes one additional method: `void followLine(int speed)`, which has to be called continuously in the Arduino loop in order to make the robot car follow the line.

All information supplied here as well as additional pictures, examples and the source code of the libraries can be found in a public GitHub repository[3].

[3] https://github.com/SQA-Robo-Lab/Arduino-Car

B Supplementary Material for the Implementation of the Code Generator

This appendix provides implementation details of the extended platform-modeling approach. This thesis presents the design and implementation of a code generator that supports the Arduino-based application scenario. In the following, some concepts and source code snippets of the implementation are shown and explained with more detail.

B.1 Continuous Ports in the PSM

The MechatronicUML PIM does not only introduce message-based communication via discrete ports, but also the communication between hybrid ports and continuous ports of continuous components (cf. Section 3.4). Pohlmann defines the semantics of the communication between hybrid and continuous ports by applying the concepts of discrete ports to them: message types and RTSCs. Essentially, a continuous or hybrid port has a data type and a sampling interval [Poh18]. Furthermore, continuous and hybrid ports are either in-ports or out-ports, i.e., an out-port is connected to an in-port. For instance, Figure B.1 shows some components of the implementation of the cooperative

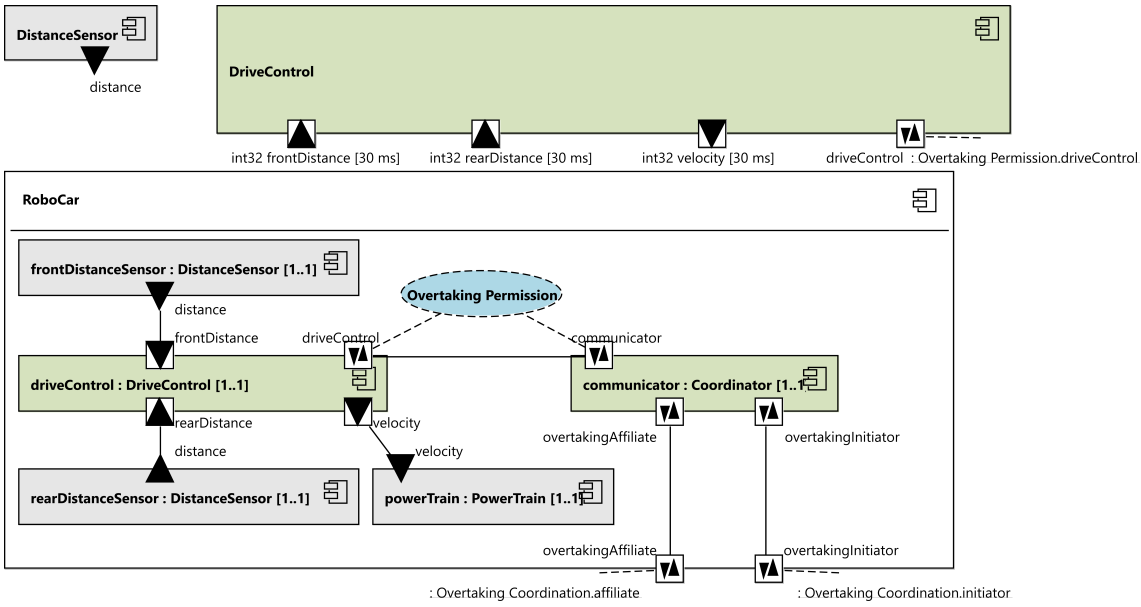


Figure B.1: An exemplary MechatronicUML component diagram with hybrid and continuous ports.

overtaking scenario. In particular, it shows the `DistanceSensor` continuous atomic component with a continuous out-port called `distance`. Even if the concrete syntax of the diagram does not reveal it, the model defines the port with the data type `int32`. The `DriveControl` discrete atomic component has a hybrid in-port `frontDinstance` with the equivalent data type `int32` and a sampling interval of `30ms`. In the structured component `RoboCar`, the embedded CIC shows that instances of both ports are connected.

The semantics of the communication between hybrid and continuous ports is defined as follows: Firstly, the out-port is described as a discrete out-port that has one sender message type of its data type [Poh18]. Secondly, the in-port is described as a discrete port that has one receiver message type of this data type as well as a message buffer for this message type with size 1. Furthermore, the buffer overflow strategy is to always discard and replace old messages (defined as `DISCARD_OLDEST_MESSAGE_IN_BUFFER` in the MechatronicUML PIM metamodel), so that the most recent value is always present in the buffer. Finally, the behavior of a discrete port is defined via a standardized RTSCs for periodic sending and periodic receiving respectively, that effectively realize the aforementioned sampling interval. For details about these RTSCs, consult [Poh18].

The transformation of continuous and hybrid ports to their respective discrete equivalent is implemented as part of the component code generation. The component code generation is reused from the original platform-modeling approach. Thus, the container code has to conform to this concept by generating source code for the message exchange between continuous and hybrid ports as well.

B.2 Container Transformation

This section provides more details about the implementation of the container transformation. It is initially described in Section 7.2, and its functionality is to create a MechatronicUML Deployment Configuration model from a MechatronicUML Allocation Specification model. The implementation of the model-to-model transformation is described in Appendix B.2.1 and its integration into the user interface of the MechatronicUML Tool Suite is explained in Appendix B.2.2.

B.2.1 Model-to-Model Transformation

The model-to-model transformation implements the container transformation by receiving a MechatronicUML Allocation Specification model and transforming it into a MechatronicUML Deployment Configuration model. It is implemented using QVTo. From a schematic point of view, the model-to-model transformation creates the MechatronicUML Deployment Configuration model based on the associated elements of the other metamodels depicted in Figure 7.1. The `SystemAllocation` references a CIC a HPIC. The QVTo transformation iterates through the `StructureResourceInstance` objects that are part of the HPIC referenced by the `SystemAllocation` instance. Listing B.1 and Listing B.2 show important snippets of the implementation. The entire code is available on GitHub[1].

[1] see `Initial_Container_Transformation.qvto` in `org.muml.psm.container.transformation/transforms` of https://github.com/fraunhofer-iem/mechatronicuml-psm

Listing B.1 Relevant snippets of the container transformation with QVTo.

```
1  transformation Initial_Container_Transformation(in mumlModel:PSM, out containerModel:
       ↪ MumlContainer);

   ...

26 modeltype PSM uses psm('http://www.muml.org/psm/1.0.0');

   ...

31 modeltype ECore uses ecore('http://www.eclipse.org/emf/2002/Ecore');

   ...

37 // A global variable to store the selection from the Eclipse UI wizard
38 property selectedMiddleware:String;
39
40 // Each ECU nees one I2C address that all its ports make use of.
41 intermediate property StructuredResourceInstance::i2cAddress:Integer;
42
43 // The properties for MQTT - could also be integrated into the UI wizard in the future
44 property mqttServerAddress:String="192.168.0.100";
45 property mqttServerPort:Integer=1883;
46 property wifiSsid:String="Section Control";
47 property wifiPass:String="********";

   ...

52 main() {

     ...

55   systemallocation:=mumlModel.rootObjects()[SystemAllocation]->any(true);
56   if(systemallocation=null) then{
57     systemallocation:=SystemAllocation.allInstances()->any(true);
58   }endif;
59
60   selectedMiddleware := "selectedMiddleware".getConfigProperty();
61   logToConsole(selectedMiddleware);
62
63   if (selectedMiddleware.equalsIgnoreCase("MQTT_I2C_CONFIG")){
64     // for I2C, every ECU needs an address that all its I2C ports can make use of
65     var nextI2cAddress : Integer = 1;
66     systemallocation.allocations.resourceInstance->forEach(resource | resource.oclIsKindOf(
         ↪ StructuredResourceInstance)){
67       resource.oclAsType(StructuredResourceInstance).i2cAddress := nextI2cAddress;
68       nextI2cAddress := nextI2cAddress + 1;
69     };
70   }endif;

     ...

73   systemallocation.map mapSystemAllocation();

     ...

78 }
```

First of all, line 1 of Listing B.1 shows the declaration of the transformation: An instance of the `PSM` metamodel (referenced in line 26) is transformed into an instance of the `MumlContainer` metamodel (referenced in line 31).

The MechatronicUML Allocation Specification metamodel is a subpackage of the MechatronicUML PSM metamodel. The main operation of the QVTo transformation begins by obtaining an instance of type `SystemAllocation` (lines 55-58). Then, the main operation starts the transformation by calling a mapping operation for the `systemallocation` instance in line 73. In order to implement the requirement **REQ3**, the transformation must (i) incorporate a user choice between DDS or MQTT and I2C as communication options, (ii) create the configuration for port instances using MQTT, and (iii) create the configuration for I2C port instances.

Firstly, to incorporate the user choice, the property `selectedMiddleware` in line 38 serves as global variable to store the chosen middleware configuration option. In line 60, its value is retrieved from the configuration properties[2] that the QVTo script is started with, and it is stored for later usage. It is used again when the port instances are configured, to decide whether the DDS configuration is applied, or whether the port instances are configured for MQTT and I2C (see lines 162 and 185 of Listing B.2).

Secondly, the configuration of I2C port instances requires the definition of I2C addresses for each ECU (cf. Section 7.1.4). Therefore, an intermediate property `i2cAddress` for the type `StructuredResourceInstance` is introduced (line 41 in Listing B.1). This intermediate property is set for each ECU, i.e., each resource of kind `StructuredResourceInstance`, in the given `systemallocation`. The addresses are simply assigned in ascending order (lines 65 to 69 in Listing B.1). For small models, the number of ECUs is typically a lot smaller than 128, hence, offering enough addresses. This is sufficient for the demonstration of the concept. In case larger models have to be handled, it is beneficial to identify those ECUs which are connected to the same I2C bus, thus, the addresses can be reused between independent buses.

The address value is retrieved from `StructuredResourceInstance`'s intermediate property when a `PortInstanceConfiguration_I2C` object is created. This is shown in Listing 7.1 in Section 7.2. The code snippet is taken from Listing B.2 where only the configuration for MQTT is shown.

The configuration for MQTT is the third functionality that is added to the container transformation. Listing B.1 shows four properties that hold the values for the WiFi configuration and the MQTT server connection (lines 44-47 in Listing B.1, cf. Figure 7.1). These configuration values are constant in the application scenario (see Chapter 2) and therefore hardcoded in the implementation. But in the future, they could be handed over from the user interface in the same fashion as for the `selectedMiddleware`.

The configuration properties are used to create the `PortInstanceConfiguration_MQTT` objects in Listing B.2. Most importantly though, the code in this listing demonstrates the functionality of the helper function `createPortIstance2PortInstanceConfiguration` that is used to create a `PortInstanceConfiguration` for each port instance. It does not only encapsulate the creation of the `PortInstanceConfiguration` objects, but also determines which specialization to use:

[2]The function `getConfigProperty` is implemented as so-called *blackbox function* and imported into the QVTo script. The import and its implementation as well as numerous other blackbox functions are not shown or explained here, but they are all visible in the public GitHub repository, see https://github.com/fraunhofer-iem/mechatronicuml-psm.

Listing B.2 The creation of `PortInstanceConfiguration` objects with QVTo.

```
125 helper PortInstance::createPortInstance2PortInstanceConfiguration():PortInstanceConfiguration{
126   var connectedPortInstances:Collection(PortInstance) :=self.getConnectedPortInstances();
127   var portInstanceConfig : PortInstanceConfiguration = null;
128   //get the own ECU
129   var ownECU : StructuredResourceInstance := self.componentInstance.getAllocatedECU(
      ↪ systemallocation);

      ...
136   // we have a one-to-one port
137     var targetPortInstance : PortInstance := connectedPortInstances->any(true);
138     //get the target ECU
139     var targetECU: StructuredResourceInstance:= targetPortInstance.componentInstance.
        ↪ getAllocatedECU(systemallocation);
140     //check if the connected PortInstances are hosted on the same ECU
141     if(ownECU=targetECU) then{

         ...
154     }else {
155       // ownECU and targetECU are different

           ...
160       var hwport: HWPortInstance := getNetworkInterface(ownECU, targetECU, systemallocation.
          ↪ getRootHPIC());
161       switch{ // selected MW switch
162         case (selectedMiddleware.equalsIgnoreCase("DDS_CONFIG")){

            ...
184         }
185         case (selectedMiddleware.equalsIgnoreCase("MQTT_I2C_CONFIG")){
186           if hwport.isMqttApplicable() then { //create the MQTT Port Configuration
187             logToConsole("Create MQTT PortInstance");
188             portInstanceConfig := object PortInstanceConfiguration_MQTT {
189               portInstance:=self;
190               hwportInstance:=hwport;
191               //each MQTT port uses the global configuration for WiFi and the MQTT server
192               WiFi_ssid:=wifiSsid;
193               WiFi_pass:=wifiPass;
194               MQTT_serverAddress:=mqttServerAddress;
195               MQTT_serverPort:=mqttServerPort;
196               //each MQTT port publishes to "their own" topic...
197               publishingTopic:=ownECU.name.toString()+"/"+self.componentInstance.name.toString
                  ↪ ()+"/"+self.name.toString()+"/";
198               //...and subscribes (in 1:1 mode) to their connected port's topic
199               subscriptionTopic:=targetECU.name.toString()+"/"+targetPortInstance.
                  ↪ componentInstance.name.toString()+"/"+targetPortInstance.name.toString()
                  ↪ +"/";
200             };

              ...
220   return portInstanceConfig;
221 }
```

PortInstanceConfiguration_Local, PortInstanceConfiguration_DDS, PortInstanceConfiguration_MQTT or PortInstanceConfiguration_I2C (cf. Section 7.1.1). Firstly, the helper function determines the connectedPortInstances (line 126) to find out if the port is a single-port or multi-port. As described before, only single-ports are supported (cf. Section 7.1.2). Then, the targetPortInstance and the targetECU are computed (lines 137 and 139), and if the targetECU is equal to the ownECU, i.e., there is no distributed allocation, local port instance configurations are created. The relevant QVTo code of the latter is omitted in Listing B.2 as it is entirely reused from the original platform-modeling approach.

In case the ECUs are different, a distributed deployment has to be configured. First of all, the user choice for the middleware option is incorporated: if the DDS option is chosen (line 162 in Listing B.2), PortInstanceConfiguration_DDS objects are created. Again, this is not shown in the listing as it is reused from the original platform-modeling approach. If the option of using MQTT and I2C is selected, then the new PortInstanceConfiguration subtypes are created. For MQTT, Listing B.2 shows that the previously mentioned properties are used for the WiFi and MQTT server configuration. And in order to implement the message-based MechatronicUML communication with MQTT, unique hierarchical topic names are created for the publishingTopic and subscriptionTopic (cf. Section 7.1.3). The topic names are composed as follows: <ecu-name>/<component-instance-name>/<port-instance-name>. These strings are computed in lines 197 and 199. In addition, before creating the portInstanceConfig, a query function is called to determine, depending on the communication protocol of the hardware port, if MQTT is applicable to be used for the particular port instance (see line 186). The implementation of these query functions is not part of the depicted listings.

Finally, one notable mention is the blackbox operation getNetworkInterface that is used in line 160 of Listing B.2 to obtain the hardware port instance. This operation is reused from the platform-modeling approach. During the development of the extensions to the platform-modeling approach, this operation was found to require *networking protocols* in order to function correctly. I.e., the communication protocols specified in the resource model must be defines as *Is Networking Protocol*. Thus, the I2C communication protocol is configured with the property *Is Networking Protocol* set to true, even though I2C is usually not be described as a networking protocol (cf. Appendix C.2.1). Unfortunately, the MechatronicUML HPDM specification [DP] does not define the communication protocol properties such as *Is Networking Protocol*. They are just implemented in the MechatronicUML Tool Suite without further documentation. According to the insights made during this thesis, the specific property *Is Networking Protocol* is required for the getNetworkInterface operation to work. If this property is not set for the I2C protocol, the operation is not be able to resolve the hardware port instance, and thus, the QVTo transformation does not work.

B.2.2 User Interface Integration

Users of the MechatronicUML Tool Suite are able to generate a MechatronicUML Deployment Configuration model via the user interface. This is part of the original platform-modeling approach [Poh18]. In the extended approach, users must be able to use the new model-to-model transformation (see Appendix B.2.1) and additionally, according to **REQ2**, select whether they want to use DDS or MQTT and I2C for communication.

Listing B.3 The plugin configuration of the container transformation wizard.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.4"?>
3  <plugin>
4     <extension
5           point="org.eclipse.ui.exportWizards">
6        <wizard
7              category="org.muml.core.export"
8              class="org.muml.container.transformation.ui.ContainerWizard"
9              id="org.muml.container.transformation.ui.ContainerWizard"
10             name="Container Model and Middleware Configuration">
11       </wizard>
12    </extension>

    ...

38 </plugin>
```

The existing implementation introduces an Eclipse-wizard that is hooked into the export-context menu of the MechatronicUML Tool Suite. The menu entry is adapted in this thesis and is now called *Container Model and Middleware Configuration*. It is added to the `org.muml.core.export`-category. The corresponding plugin configuration is visible in Listing B.3. It references the `ContainerWizard` class in line 8, which is the Eclipse-wizard that leads the user through the steps for starting the container transformation. The container transformation requires a MechatronicUML Allocation Specification model, that the user selects on the first page of the wizard. In this wizard, a new page is added to select the middleware configuration option via the user interface (see Figure 7.4), and finally, the user chooses a destination directory to store the created model instance. The MechatronicUML Deployment Configuration model is stored as a file with the `.muml_container` file ending; it is used to start the container code generation (see Section 7.3 and Appendix D.4).

After collecting the information from the user, the `ContainerWizard` starts the model-to-model transformation on the selected MechatronicUML Allocation specification model. The `MiddlewareOptionsPage` wizard page, which is displayed in Figure 7.4, allows the `ContainerWizard` to retrieve the user choice and hand it over to the `ContainerGenerationJob`. This wizard page is implemented as part of this thesis. The `ContainerGenerationJob` is responsible for configuring and

Listing B.4 Setting the configuration property for the QVTo transformation in the `ContainerGenerationJob.java`.

```
86      ExecutionContextImpl context = new ExecutionContextImpl();
87      context.setConfigProperty("selectedMiddleware", selectedMiddleware.toString());
88      context.setLog(log);
89      ExecuteQvtoTransformationCommand command = new ExecuteQvtoTransformationCommand(
        ↪ transformationExecutor, extentList, context);
90      if (command.canExecute()) {
91        editingDomain.getCommandStack().execute(command);
92      }
```

starting the model-to-model transformation. This QVTo model-to-model transformation requires the selected middleware option as a configuration parameter (cf. Appendix B.2.1). Listing B.4 shows how the configuration property is set (line 87) before the `command`, which wraps the QVTo transformation, is executed (line 91). The retrieval of this configuration property in QVTo is visible in Listing B.1 (line 60).

B.3 Container Code Generation

This section contains additional Acceleo templates and explanations which are used to generate the container code (cf. Section 7.3).

B.3.1 Container Implementation

This section provides additional material for the container code generation. Specifically, the material concerns the container implementation which is presented in Section 7.3.4 and represents the core of the container code.

When a container implements the *create* method and creates the `ComponentBuilderStruct`, it has to create the respective configuration for the required port handles as well. Therefore, the `ComponentBuilderStruct` supports different configuration options for the different port types. Listing B.5 shows the Acceleo template that generates the *create* method for each container. It receives a `ComponentContainer` and a collection of `ContainerComponentInstanceConfiguration` objects. The signature of the *create* method is generated in line 47: the method returns a pointer to a component of the container's associated type. As parameter, it receives an identifier for the component instance to be created. The identifiers are defined in the `ECUIdentifiers.h` header (cf. Section 7.3.2). The method decides which component instance configuration to use with a `switch` statement (line 49), and for each `componentInstanceCfg` in the given collection (line 50), it creates a `case` with the matching identifier (line 51). Thus, each `case` serves to weave the configuration properties of the MechatronicUML Deployment Meta model into the code and sets the corresponding fields in the component builder structure `b`.

Depending on the type of port instance configuration, the corresponding configuration has to be put into the generated code. This is shown for the kind `PortInstanceConfiguration_MQTT` in lines 64-69: Most importantly, the topics for publishing (line 67) and subscription (line 68) are retrieved from the model an written to the generated source code file. Eventually, the *create* method returns the pointer to a newly created component instance by calling the builder method of the respective component type with the newly generated builder structure `b` that holds the configuration. The component builder uses the configuration to initialize all ports before returning the pointer to the newly created component instance.

Listing D.2 shows the generated code of a builder method for the `Coordinator` component. This generated code including a port handle builder method are explained with more detail in Appendix D.5.1 and belong to the implementation of the application scenario that is described in Chapter 8.

Listing B.5 The create method template of the container code generation

```
[**
 * Generates a method to create a component instance via the container.
*/]
[template public generateCreateMethodForComponentInstances(container:ComponentContainer,
      ↪ cicfgs:Collection(ContainerComponentInstanceConfiguration))]
/**
 * @brief Create a component instance with the given id.
 *
 * @details Creates a component instance using the builder and the configuration options, and
      ↪ also configures the port instances.
 *
 * @param ID the identifier of the component instance
 */
[container.componentType.getClassName()/]* [container.componentType.
      ↪ getContainerComponentCreateMethodName()/](uint8_T ID){
  struct [componentType.getBuilderStructName()/] b = INIT_BUILDER;
  switch(ID){
  [for (componentInstanceCfg : ContainerComponentInstanceConfiguration | cicfgs)]
    case [componentInstanceCfg.componentInstance.getIdentifierVariableName()/]:

    ...

      [if (portCfg.oclIsKindOf(PortInstanceConfiguration_MQTT))]
      b.[portCfg.portInstance.portType.name.toUpper()/] = PORT_ACTIVE;
      b.create[portCfg.portInstance.portType.name.toUpper()/]Handle = &[portCfg.portInstance.
       ↪ portType.getMethodNameForMqttPortHandleBuilder()/];
      b.[portCfg.portInstance.portType.name.toUpper() /]_op.mqtt_option.publishingTopic = "[
       ↪ portCfg.oclAsType(PortInstanceConfiguration_MQTT).publishingTopic.toString() /]";
      b.[portCfg.portInstance.portType.name.toUpper() /]_op.mqtt_option.subscriptionTopic = "[
       ↪ portCfg.oclAsType(PortInstanceConfiguration_MQTT).subscriptionTopic.toString() /]";
      [/if]

    ...

    [/for]
    break;
  [/for]
  default:
    break;
  }
  return MCC_[container.componentType.getClassName()/]_Builder(&b);
}
[/template]
```

Listing B.6 The templates to generate the sending of a message using the custom MQTT library.

```
82 [**
83  * Generate the library call for the sending of a message using MQTT for a DiscretePort.
84 */]
85 [template private generateMqttSendingMethod(messageType : MessageType)]
86 sendMqttMessage(mqttHandle->publishingTopic, "[messageType.getName()/]", (byte*) msg, sizeof([
       ↪ messageType.getMessageType()/])); [comment msg is a pointer that is handed to the
       ↪ method as parameter /]
87 [/template]
88
89 [**
90  * Generate the library call for the sending of a message using MQTT for a DirectedTypedPort.
91 */]
92 [template private generateMqttSendingMethod(dataType : DataType)]
93 sendMqttMessage(mqttHandle->publishingTopic, "[dataType.getTypeName()/]", (byte*) msg, sizeof(
       ↪ [dataType.getTypeName()/])); [comment msg is a pointer that is handed to the method
       ↪ as parameter /]
94 [/template]
```

B.3.2 Communication Middleware Library Usage

The communication middleware libraries are used in the generated container implementation to implement the *send* methods (cf. Section 7.4.3). Listing B.6 shows the respective Acceleo template for the MQTT communication[3] Depending on the port type, either the message type of a discrete port or the data type of a continuous or hybrid port[4] is handed over to the template. The template is invoked whenever a *send* method is implemented, and the method signature of the send method, which is not shown here, declares a pointer `*msg` of the corresponding C type (message type or data type, respectively) as parameter. This is implemented equivalently in a template for the I2C communication[5].

The message buffer library is used in the generated container implementation to implement the *exists* and *receive* methods (cf. Section 7.4.3). The methods which receive messages in the custom MQTT and custom I2C library put the messages into the respective message buffer (cf. Section 7.4.3). Thus, the *exists* method is implemented using a the message buffer do determine if it contains messages. This is shown in Listing B.7 in line 114. Similarly, the consumption of a message from the message buffer implements receiving a message as visible in line 146. The Acceleo template snippets show examples for discrete ports. For continuous ports this is implemented in the same fashion (cf. Section 7.1.2).

[3] See `MqttCommunication.mtl` in `org.muml.arduino.adapter.container/src/org/muml/arduino/adapter/container/communication` of the C Containers repository (https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-container).

[4] Both inherit from the the `DirectedTypedPort` in the MechatronicUML PIM metamodel.

[5] See `I2cCommunication.mtl` in `org.muml.arduino.adapter.container/src/org/muml/arduino/adapter/container/communication` of the C Containers repository (https://github.com/fraunhofer-iem/mechatronicuml-cadapter-component-container).

Listing B.7 The templates to generate an *exists receive* method using the message buffer library.

```
116 [**
117  * Generates the loop that iterates through the receivers and finds the right receiver to
         ↪ obtain its buffer.
118 */]
119 [template private generateFindMessageBuffer(messageTypeName : String)]
120 MessageBuffer* buffer = NULL;
121 for (i = 0; i < i2cHandle->numOfReceivers; i++) {
122   if (strcmp(i2cHandle->receivers['['/]i[']'/].messageTypeName, "[messageTypeName/]") == 0){
123     buffer = i2cHandle->receivers['['/]i[']'/].buffer;
124     break;
125   }
126 }
127 [/template]
128
129 [**
130  * Generates the check if an I2C message is available for the corresponding DiscretePort.
131 */]
132 [template public generateSwitchCaseForI2cExists(portInstanceConfigs : Collection(
         ↪ PortInstanceConfiguration_I2C), messageType: MessageType)]
133 [generateCaseAndI2cHandle() /]
134   [generateFindMessageBuffer(messageType.getName()) /]
135   return MessageBuffer_doesMessageExists(buffer);
136   break;
137 [/template]

...

149 [**
150  * Generates the receiving of a DiscretePort's message using I2C.
151 */]
152 [template public generateSwitchCaseForI2cReceiving(portInstanceConfigs : Collection(
         ↪ PortInstanceConfiguration_I2C), messageType : MessageType)]
153 [generateCaseAndI2cHandle() /]
154   [generateFindMessageBuffer(messageType.getName()) /]
155   return MessageBuffer_dequeue(buffer, msg); [comment msg is a pointer that is handed to the
         ↪ method as parameter /]
156   break;
157 [/template]

...

169 [**
170  * Generates the case statement for I2C ports and the corresponding PortHandle.
171 */]
172 [template private generateCaseAndI2cHandle()]
173 case PORT_HANDLE_TYPE_I2C:
174   i2cHandle = (I2cHandle*) port->handle->concreteHandle;
175 [/template]
```

C Supplementary Material for Modeling the Application Scenario

This appendix contains supplementary material for the application scenario's modeling using the MechatronicUML. These models are the foundation for the platform-modeling approaches software construction. The focus of Chapter 8 is the software construction itself, which is also the part of the platform-modeling approach that is extended in this thesis. However, in order to understand the implementation of the application scenario in depth, this appendix depicts and explains all diagrams that are not shown in the thesis itself.

C.1 Platform-Independent Models

This section contains additional models of the MechatronicUML PIM including descriptions and additional information about design choices.

C.1.1 Component Instance Configuration

Figure C.1 shows the root CIC of the cooperative overtaking models. It instantiates the component type `RoboCar` twice and names the instances `fastCar` and `slowCar` to model the overtaker and the affiliate respectively. The CIC diagram shows all component instances of the platform-independent software model. These component instances are allocated to the structured resources of an HPIC (cf. Section 8.1.3). In order to allocate the component instances unambiguously, they have unique names. The `.F` in the instance names represents that the atomic component instances belong to the `fastCar` structured component instance, and the `.S` indicates being embedded in the `slowCar` instance respectively. The CIC also shows that both cars software consists of the same components, thus, their roles are not predetermined by their software components. This is also shown by the `Overtaking Coordination` RTCP: in general, both cars can take either role in the cooperative overtaking scenario. Nonetheless, the `fastCar` is intended to represent the overtaker and may be initialized with a higher velocity, thus, taking the role of the overtaker. Additionally, the coordination between the discrete component instances of either car is specified by the `Overtaking Permission` RTCP.

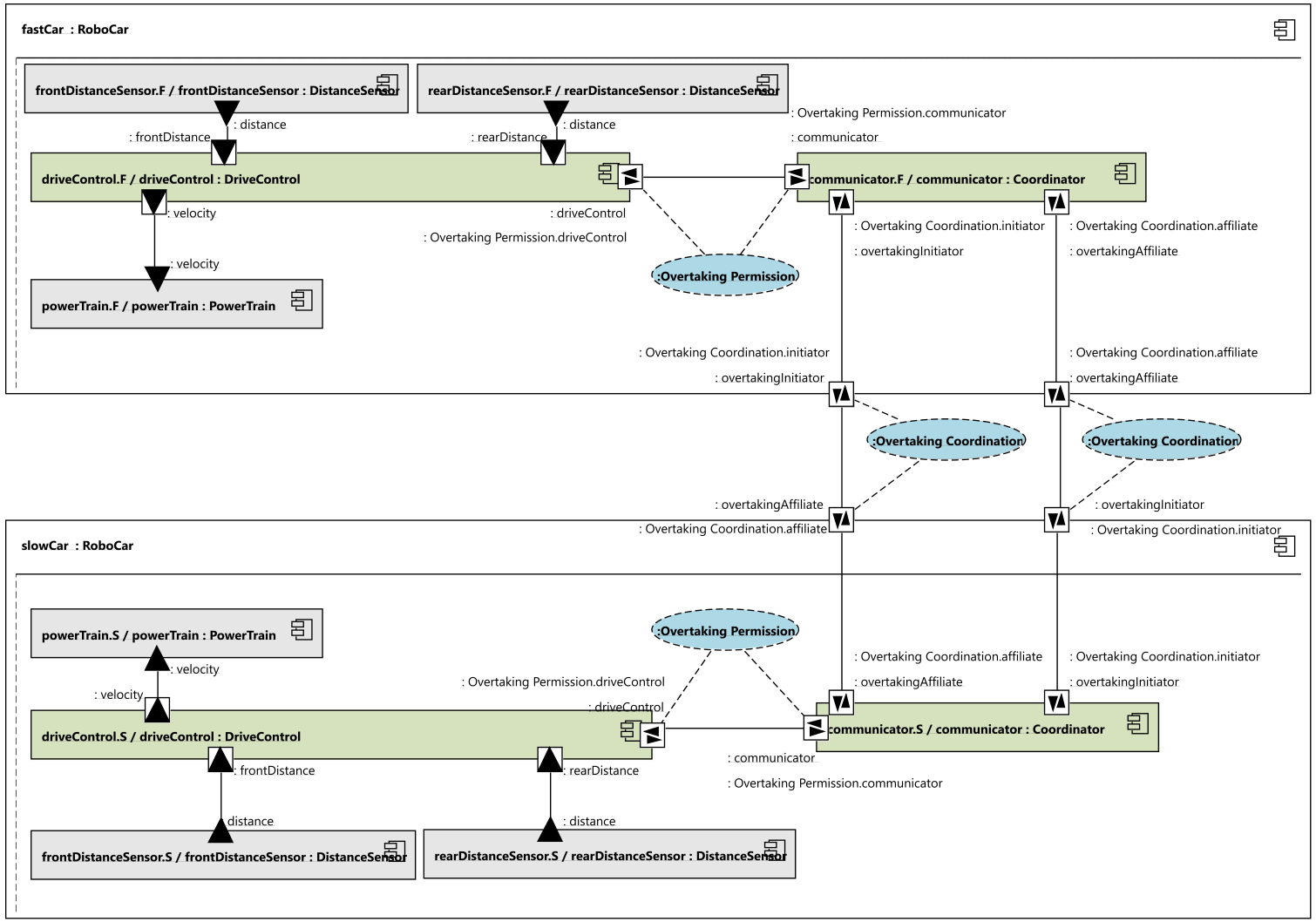


Figure C.1: The root CIC of the cooperative overtaking scenario modeled with the MechatronicUML .

C.1.2 Coordination Behavior

The RTCP `OvertakingPermission` (see Figure 3.5) and its role `driveControl` (see Figure 3.6) are visualized and explained in Section 3.4 as an example of the coordination and behavior modeling with the MechatronicUML PIM modeling language. Figure C.2 shows the missing specification of the `communicator` role. The `Overaking Permission` RTCP defines that the roles communicate using the four message types `grantPermission`, `denyPermission`, `requestPermission` and `executedOvertaking`. Each role has one buffer of size 5 that discards new messages if it is full. As only two cars can coordinate within an instance of the `Overtaking Permission` RTCP, the size of the buffers is not a primary concern; however, the design choice is to make them rather small and discard new messages in order to prevent the accumulation of open coordination attempts. This is also prevented by the role behavior, however. Furthermore, the RTCP models the QoS assumption that the roles communicate via a reliable connection that preserves the message order and has a maximum delay of $500ms$. These assumptions are based on the usage of I2C for communication. As defined by the role's RTSCs, the communication is initiated by the `driveControlRole` that sends a `requestPermission` message if it detects a car, i.e., `distance < distanceLimit`. Then, it waits for permission and only enters the overtaking state if it receives a `grantPermission` message; a `denyPermission` message triggers a transition to the initial state again. The `communicatorRole` consumes the `requestPerimission` message and, after waiting for the coordination, signals the `driveControlRole` whether it may overtake. If permission for overtaking is granted, the `communicatorRole` waits for the `driveControlRol` to finish the overtaking by sending an `executedOvertaking` message. This models the communication that happens within a robot car, i.e., between it two discrete components in order to separate the concerns of controlling the driving and the coordination between robot cars.

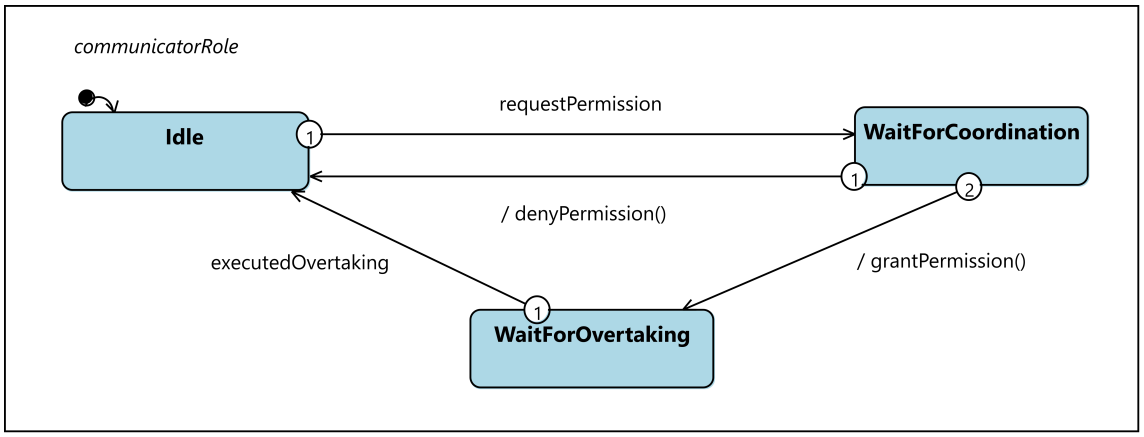
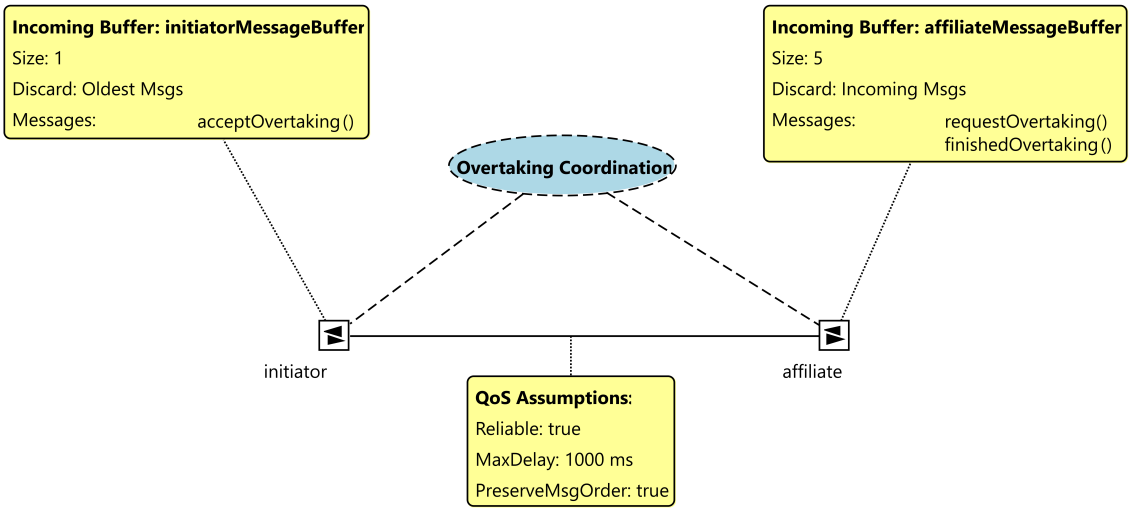
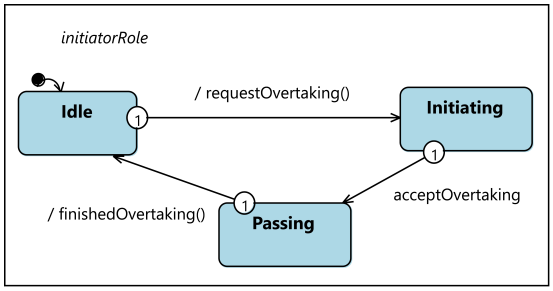


Figure C.2: The role specification of the `communicator`.

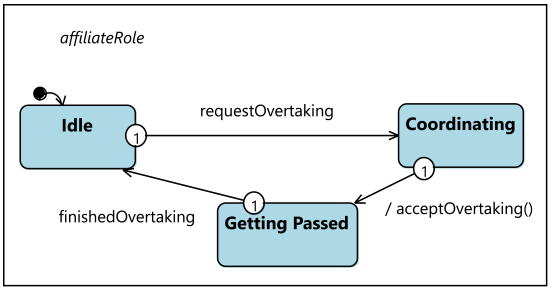
Furthermore, there is an RTCP that defines the communication between different robot cars: the `Overtaking Coordination` RTCP shown in Figure C.3. The roles `initiator` and `affiliate` communicate via a reliable connection, but with a higher maximum message delay of $1000ms$ due to the wireless communication between different cars. The `initiator`, as the name suggests, initiates the communication and thus, the cooperative overtaking coordination. It does so by sending a `requestOvertaking` message. The affiliate responds with an `acceptOvertaking` message, and both roles transition back into the initial state after sending (`initiator`) or receiving (`affiliate`) a `finishedOvertaking` message. This behavior modeling of the `Overtaking Coordination` reflects the fact that in our application scenario, the affiliate does not have the opportunity to reject being passed; the only requirement for the



(a) The `Overtaking Coordination` RTCP.



(b) The role specification of the `initiator`.



(c) The role specification of the `affiliate`.

Figure C.3: The `Overtaking Coordination` RTCP and its roles

the cooperative overtaking this the affiliate being aware that it is part of the overtaking maneuver. This is reflected by the state `Getting Passed`. Furthermore, the `initiator` role is modeled with a message buffer of size 1 and the buffer management strategy to discard old messages. Thus, the `initiator` always has only the most recent `acceptOvertaking` message available in the buffer if it is not empty.

C.1.3 Component Behavior

The behavior description of the roles of an RTCP is an abstract specification that defines the contract which has to be fulfilled by two components communicating via an RTCP (cf. Section 3.4.2). As shown in the component diagram in Figure 8.2, the discrete ports are all assigned a role of an RTCP. Thus, the behavior of the components, which includes the behavior of a component's discrete ports, refines the abstract behavior of the roles.

The behavior specification of the `Coordinator` component contains such a refinement of three roles: the three different discrete ports are each assigned a different role of one of the aforementioned RTCPs. The `Coordinator`'s RTSC is shown in Figure C.4. The `communicatorPort` region refines the `communicator` role of the `Overtaking Permission` RTCP, hence, is the com-

munication interface to the robot car's DriveControl component. The other two regions, overtakingInitiatorPort and overtakingAffiliatPort refine the roles of the Overtaking Coordination RTCP. The CoordinatorComponent RTSC refines both roles, because each car may be the initiator as well as the affiliate of a cooperative overtaking maneuver.

Overall, the behavior of the Coordinator component can be described as follows: When the communicatorPort receives a requestPermission message, it signals this fact via the synchronization channel overtakingRequested!. Then, it waits for the coordination to happen. The overtakingInitiatorPort listens to the synchronization overtakingRequested? and sends a requestOvertaking message to the other robot car. It is configured with a coordination time-out: if the clock coordinationTime reaches this time-out, the overtakingInitiatorPort signals the other car that the overtaking is not attempted any longer (using finishedOvertaking) and synchronizes with the communicatorPort that the coordination was not successful: coordinationSuccessful[false]!. Then, the communicatorPort denies the permission to the DriveControl component that it must not start the overtaking. In case the coordination is successful within the time limit, the overtakingInitiatorPort receives an acceptOvertaking message by the other robot car's Coordinator component. Then, it also synchronizes with the communicatorPort about the coordination outcome: coordinationSuccessful [true] triggers the CommunicatorPort to grant permission to the DriveControl component, which will then execute the overtaking maneuver. Upon its completion, the CommunicatorPort receives an executedOvertaking message by the DriveControl component, and then synchronize with the overtakingInitiatorPort via overtakingSuccessful that it may send a finishedOvertaking message to the other robot car.

The overtakingAffiliatePort region does not contain any critical refinement of the affiliate role behavior. The only addition, in order to prevent a robot car from being an initiator and an affiliate of two cooperative overtaking maneuvers at the same time, is the variable boolean coordinatorIsBusy.[1] It is set to false by an entry action event of the Idle state, and set to true whenever the RTSC leaves this state. Furthermore, the guard coordinatorIsBusy == false prevents the RTSC from leaving the Idle state in case the variable is true. This is implemented for the overtakingInitiatorPort equivalently and ensures, that only one embedded RTSC of overtakingInitiatorPort or overtakingAffiliatePort is active at a time.

The behavior specification of the DriveControl component has already been described as an example in Section 3.4.3 where the behavior modeling features of the MechatronicUML PIM were introduced. The respective RTSC is visible in Figure 3.7 and shows that the component behavior refines the role behavior (driveControlPort region) and also includes its own behavior (driving region). The synchronization and the states were described in Section 3.4.3, and as noted there, the action effects are now also looked at. Firstly, the driveControlPort region uses the effects slowDown and accelerate: When one robot car detects a slower vehicle in front, it has to slow down upon requesting permission to overtake; otherwise, if the coordination took to long, a collision would occur. The slowDown action effect is implemented using the action language with the following statement: {velocity := slowVelocity ;}. This statement sets the velocity variable, which references the hybrid out-port connected to the powerTraing component instance of a RoboCar structured component (cf. Figure 8.2). After being granted the permission to overtake, the car picks-up its initial speed again. This action

[1] Several cooperative overtaking maneuvers at the same time are actually not possible with only two cars. Thus, this can be seen both as a precautionary feature in case messages are somehow extremely delayed, or also as an outlook to design the software for more vehicles.

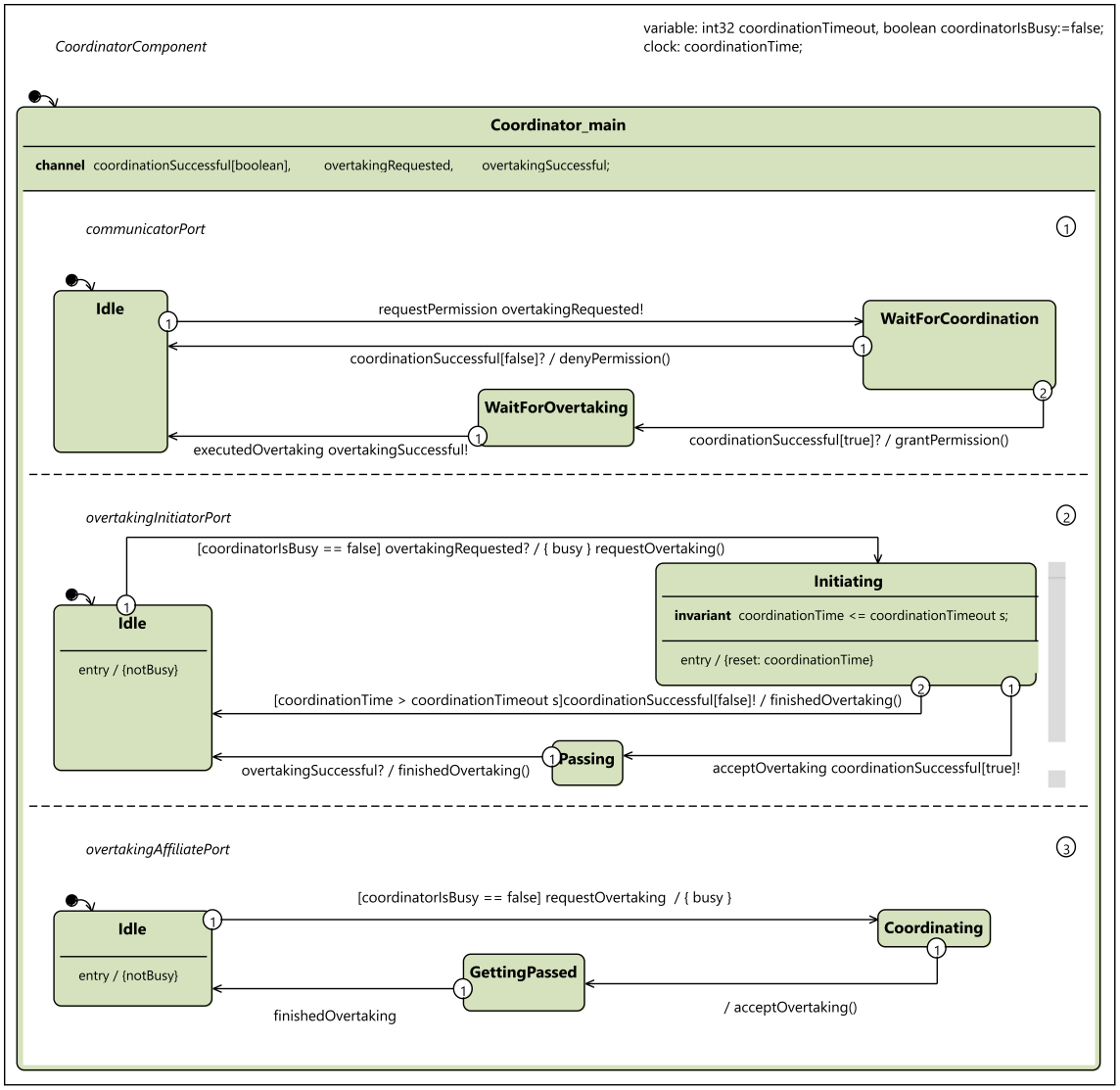


Figure C.4: The RTSC of the `Coordinator` component

`accelerate` is implemented equivalently as `{velocity := desiredVelocity ;}`. In order to have two cars with different values for the `desiredVelocity`, such that one car is faster than the other, the cars would have to be instantiated with different initial variable values to have a fast and a slow car. Ideally, the robot cars would not have to slow down, but the coordination process would be started early enough to avoid this. And additionally, if slowing down, a robot car could ideally adapt to the velocity of the preceding car instead of slowing down to a fixed value. These are technical limitations and could be improved in the future. In the current state of implementation, this is simplified because it is sufficient to demonstrate the overall modeling and code generation process for our application scenario.

Action effects are also used to model the behavior of the `driving` region which contains the behavior for the autonomous driving of a robot car. As is would be very complex and unnecessary to implement such potentially already existing behavior from scratch using the action language,

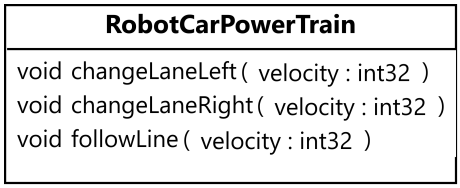


Figure C.5: The operation repository diagram for the robot cars.

the MechatronicUML PIM introduces *operation repositories* which represent external libraries. Figure C.5 shows the operation repository diagram for the robot cars. From a platform-independent viewpoint, these operations are assumed to exist externally. Operations from an operation repository can be used in all RTSCs that reference this operation repository. They can then be called in action effects using the action language. Thus, e.g., the action effect `changeLane` in Figure 3.7 is implemented by the expression `{changeLaneLeft ( velocity := desiredVelocity ) ;}`. This results in a concise specification of the overtaking behavior and reduces the RTSC specification to the most important states in the process. Using operation repositories like this is a design choice; an alternative is to model the overtaking procedure including changing lanes and following a lane with an RTSC. However, this results in a very complex model and does not allow to use external, potentially already existing implementations. This design choice also affects the modeled components: As the infrared sensors are only used for the line following, and this is abstracted by the operations, there is no need to model the sensors as continuous components.

C.2 Hardware Models

This section contains additional material from the MechatronicUML HPDM model, focusing on the underlying resources and resource instances that the platform is composed of and also adding some information on the platform models.

C.2.1 Resources and Resource Instances

This section contains additional material from the MechatronicUML HPDM model: it presents the underlying resources and resource instances that the hardware platform is composed of. The description of the robot car hardware platform in Appendix A provides list of the hardware resources. The specific resources' data sheets are available online. The important data sheets are the ones of the *Arduino Mega 2560 Rev3*[2] and the *Arduino Nano*[3] microcontrollers. Thus, the resources for this Arduino-based hardware platform are modeled in this section.

[2] https://docs.arduino.cc/hardware/mega-2560
[3] https://docs.arduino.cc/hardware/nano

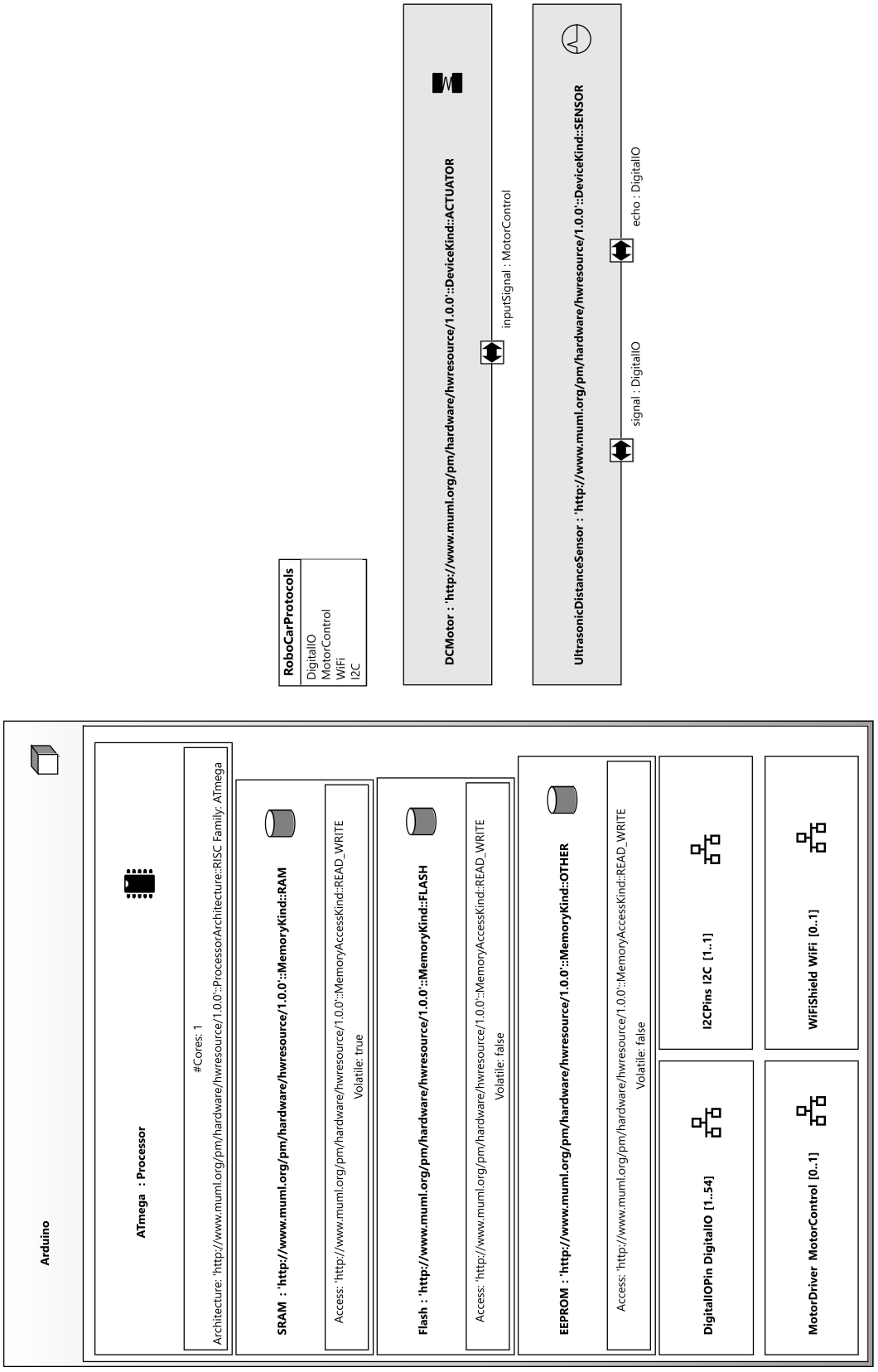


Figure C.6: The resource diagram of the Arduino car modeled with the MECHATRONICUML HPDM.

Figure C.6 depicts the resource diagram for the robot cars. It contains the atomic computing and memory resources, the devices, the structured resource `Arduino` and the hardware ports with their communication protocols (cf. Section 3.5.1). It defines four communication protocols: the link protocols `DigitalIO`, `MotorControl` and `WiFi` and the bus protocol `I2C`. The concrete syntax does not reveal their type, therefore the type is stated here explicitly.

Protocol Name	Type	Properties	Type-specific Protocol Kind
DigitalIO	Link Protocol	-	Other
MotorControl	Link Protocol	-	Other
WiFi	Link Protocol	Is Networking Protocol	WIFI
I2C	Bus Protocol	Is Networking Protocol	I2C

Table C.1: The details of the communication protocols int the robot car's resource model.

The resource diagram in Figure C.6 is not capable of displaying all details about the communication protocols that are relevant for the platform modeling, and also for the platform-specific modeling. Thus, these details are given in Table C.1. The table shows that the `DigitalIO` and `MotorControl` protocols are essentially the same; they are only distinguished to ensure correctly connected hardware ports: only hardware ports that use the same communication protocol may be connected (cf. Section 3.5.1). Additionally, the `I2C` and `WiFi` communication protocols are configured with the property *Is Networking Protocol* set to true; this is required for the container transformation to work correctly (cf. Section 7.2 and Appendix B.2.1]).

Figure C.6 shows that the device `DCMotor` has one port called `inputSignal` of type `MotorControl`, and the device `UltrasonicDistanceSensor` has two `DigitalIO` ports `signal` and `echo`. This example shows the limitations of the MechatronicUML HPDM as technically, both devices operate using digital input/output signals, and digital input/output is no actual communication protocol. However, the specification notes that a main purpose of these protocols is to ensure compatibility of hardware ports [DP], hence, some modeling freedom can be applied in concrete scenarios like the presented one.

Furthermore, the concrete syntax for hardware ports is not implemented by the MechatronicUML resource diagram Eclipse editor in the exact way the specification defines: According to the specification, the concrete syntax of hardware ports is a small squared box on the resource's border. This is the way the MechatronicUML Tool Suite implements the concrete syntax for devices. However, the example resource diagram in Figure C.6 shows that the hardware ports are modeled differently for structured resources: The `Arduino` resource has the port types `DigitalIOPin`, `I2CPins`, `MotorDriver` and `WiFiShield`. Overall, the `Arduino` resource models the variation points of concrete Arduino microcontrollers based on their data sheets. Again, there are some limitations: the `MotorDriver` and `WiFi Shield` are actually kind of devices that provide communication functionality to the Arduino microcontroller; but there are no means to model communication devices. So overall, the given resource diagram models the resources of the running example. Only the infrared sensors are not modeled on purpose; see Section 8.1 for additional details about modeling and design decisions.

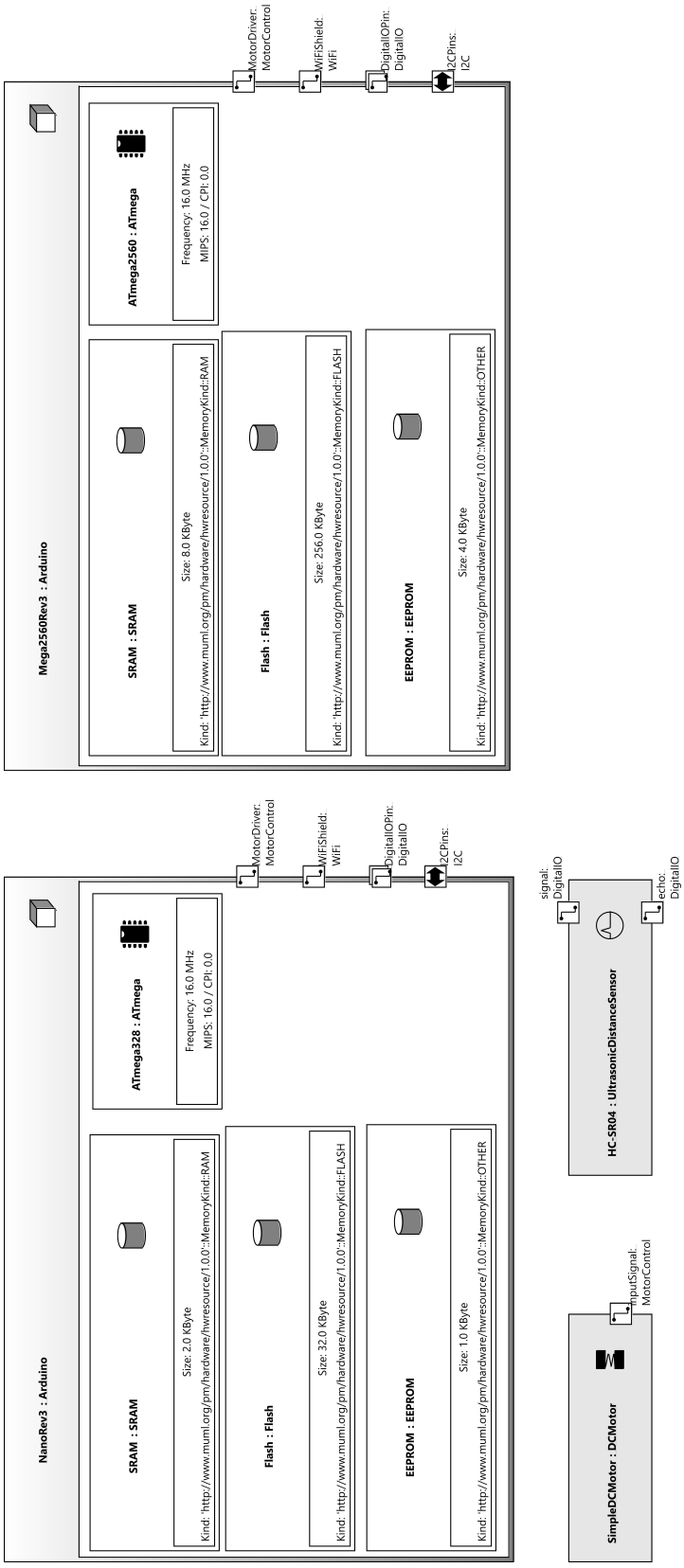


Figure C.7: The resource instance diagram of the Arduino car.

All ECUs that are required for the robot car platform are of type Arduino, and they are instantiated with processing and memory resources according to their data sheets. Furthermore, the resource model is tailored to the application scenario: As the infrared sensors are not treated as an individual software component, they do not need to be modeled by corresponding device type for the allocation. As each robot car is equipped with three infrared sensors (cf. Appendix A), they were left out purposefully to keep the platform modeling diagrams a little smaller.

In contrast to the resource diagram, the MechatronicUML resource instance diagram editor in the MechatronicUML Tool Suite implements the concrete syntax of hardware ports exactly as it is defined in the specification: Here, bus and link ports can be distinguished as visible in Figure C.7. However, the editor does not allow to fix the number of port instances per resource instance, even though the specification suggests that. Consequently, the `Mega2560Rev3` has a `WiFiShield` in the diagram even though it does not have one in reality, and same applies to the the `NanoRev3`'s `MotorDriver`.

These workarounds to the capabilities of the MechatronicUML HPDM and it implementation in the MechatronicUML Tool Suite do not limit is applicability for modeling the desired target platform. Especially with respect to the code generation for a distributed deployment, the important aspects are that the structured resource types are equipped with hardware ports that allow the specification of the concrete hardware platform as a HPIC. This is explained next in Appendix C.2.2.

C.2.2 Hardware Platform Models

In order to model the robot cars using the previously defined resource types and resource instances, three platform types are specified. First, the `DriveControlUnit` platform type which is shown in Figure 3.12 as an example hardware platform and represents the Arduino Mega microcontroller with its attached ultrasonic distance sensors and motors. To keep the diagram a little smaller, the `SimpleDCMotor` is only instantiated twice instead of four times, as that still reflects the structure and concept of the robot car target platform.

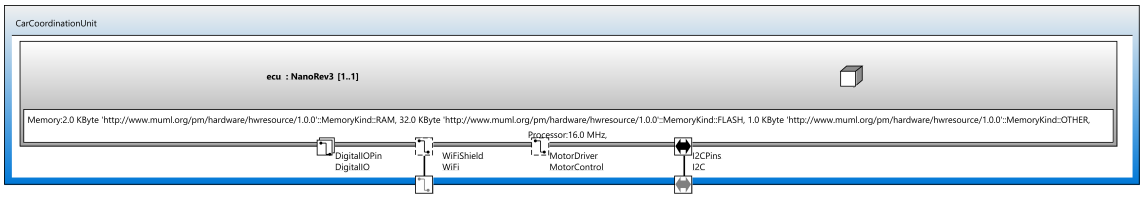


Figure C.8: The `CarCoordinationUnit` platform type.

Secondly, Figure C.8 depicts the `CarCoordinationUnit` platform type which models the Arduino Nano ECU with its attached WiFi module. The WiFi module is not modeled as an own resource or device type because the MechatronicUML HPDM does not support devices that only provide communication functionality (cf. Section 3.5.1). Thus, it is modeled as a communication resource in the `Arduino` resource type (see Figure C.6), and the platform diagram depicts its usage via delegation to a delegation port. Hence, the `CarCoordinationUnit` exposes one hardware port using the WiFi module as well as one hardware port using the Arduino Nano's `I2CPins`.

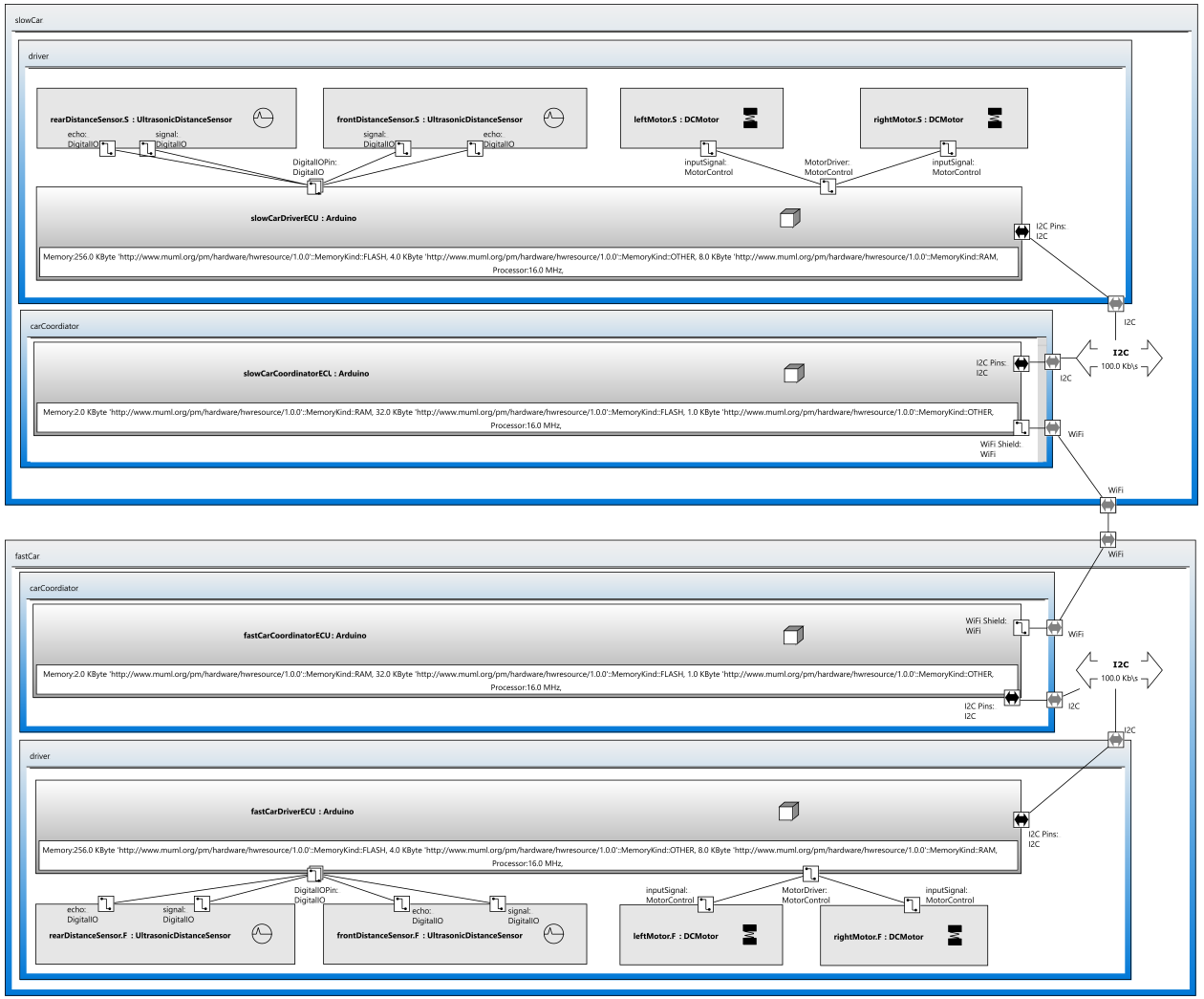


Figure C.9: The HPIC diagram of the robot car.

Thirdly, the former two platform types are used as platform parts to model the `RobotCar` platform type. It is depicted in Figure 8.4 and represents the structure of robot car. The `RoboCar` platform type is depicted in Figure 8.4.

The final diagram of the MechatronicUML HPDM is the HPIC diagram. It is depicted in Figure C.9. The diagram shows two instances of the `RoboCar` platform type with all its structured resource instances and device instances. It also models the communication channels: The `slowCar` platform instance and `fastCar` platform instance are connected via their respective `WiFi` delegation ports and use an I2C bus to connect their embedded platform parts internally, thus, reflecting the target platform described in Section 2.2.

C.3 Allocation Specification Model

The allocation engineering applied for the implementation of the application scenario is limited to the task *T2.2* of Figure 4.1 because *T2.1* is optional with respect to code generation (cf. Section 4.2.1). The allocation engineering is not changed within this thesis, but it is assumed to be functional and reusable. This revealed to be only partially true, however. Pohlmann introduce a so-called *allocation constraint modeling language* that allows to specify constraints for the allocation. The idea is that complex allocation engineering problems are broken down to specifying these constraints that are solved as constraint satisfaction problem [Poh18]. This is especially useful for complex software systems were manual allocation would often be difficult.

The allocation constraint modeling language is also implemented as an Eclipse-plugin and integrated into the MechatronicUML Tool Suite (cf. Section 3.3). However, during this thesis, it was not possible to compile a set of constraints that was handled correctly by the solver. Unfortunately, the modeling language appears to be implemented slightly differently than specified by Pohlmann in [Poh18]: The examples given there yield syntax errors in the editor, and some language features are not supported. Older syntax variations from [BCD+14] are accepted by the editor, but not by the solver. It requires further research to explore how the allocation constraint language is implemented. This is not the focus of this thesis however and therefore left for future work.

The allocation problem for the given application scenario is not as complex as a potential real-world scenario. From the naming of the component types and instances as well as the platform types and instances that was explained thus far, a suitable allocation can already be inferred. The software for small CPS like these Arduino-based robot cars is often developed in close software/hardware co-design; same applies to our application scenario, where the intended allocation is already considered when designing the software and hardware systems. Thus, as the allocation engineering itself is contained in the MechatronicUML Tool Suite and, most importantly, its metamodel is implemented exactly as specified, the metamodel is usable. Therefore, instead of compiling a set of allocation constraints, the MechatronicUML Allocation Specification model is created manually. Figure C.10 depicts the allocation.

The diagram shown in Figure C.10 is based on a graphical view introduced by Pohlmann in [Poh18]. However, to the best of the authors knowledge, this view is not implemented in the MechatronicUML Tool Suite. Hence, this diagram is not a formal representation of the MechatronicUML Allocation Specification model, unlike the diagrams shown before. Overall, Figure C.10 depicts that the embedded component instances of the `fastCar` structured component instance are allocated to

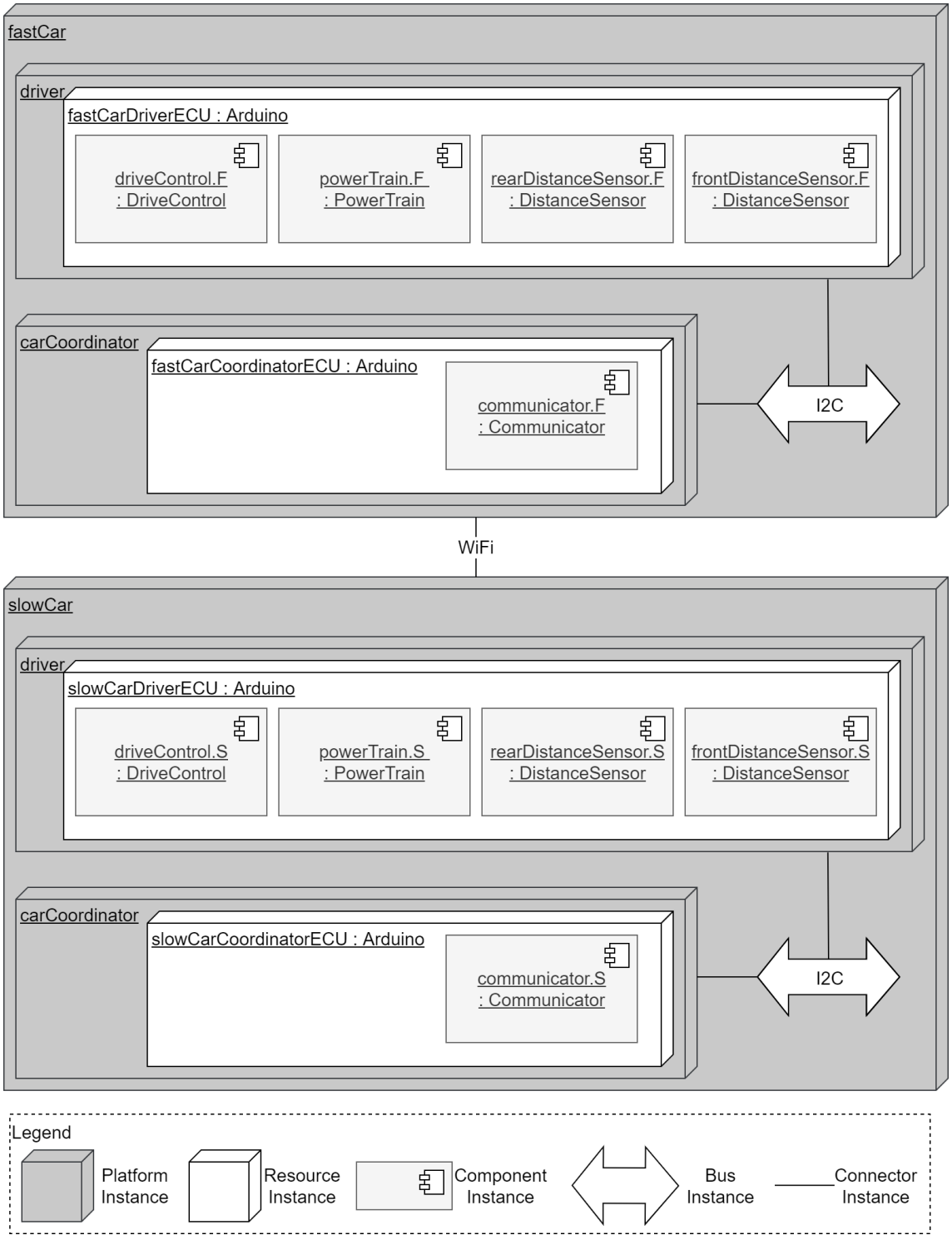


Figure C.10: The allocation of the robot car software.

the `fastCar` hardware platform instance.[4] The `communicator.F` atomic component instance is allocated to the `fastCarCoordinatorECU` structured resource instance, and the other atomic component instances are allocated to the `fastCarDriverECU`. Thus, as intended for the application scenario, the `fastCarCoordinatorECU` which represents an Arduino Nano with attached WiFi module, contains the communication capabilities. Furthermore, the components for the driving capabilities are allocated to the `fastCarDriverECU` which represents the Arduino Mega with its attached distance sensors and motors. Equivalently, the embedded component instances of the `slowCar` structured component instance are allocated to the `slowCar` hardware platform instance. As the hardware platform instances of the `fastCar` and `slowCar` are completely the same, a swapped allocation is also feasible. The MechatronicUML Allocation Specification model is the final asset required for the software construction using the platform-modeling. It references a concrete CIC of the MechatronicUML PIM and a HPIC of the MechatronicUML HPDM and thus lays the foundation for a distributed deployment by specifying the distributed allocation.

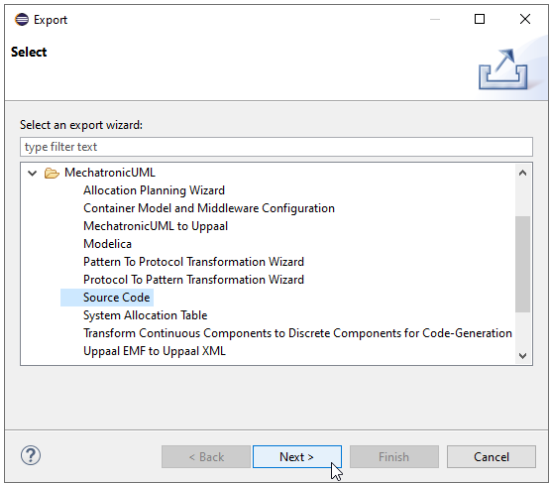
[4] All component instances are depicted in Figure C.1, and the HPIC is visualized in Figure C.9.

D Supplementary Material for the Software Construction

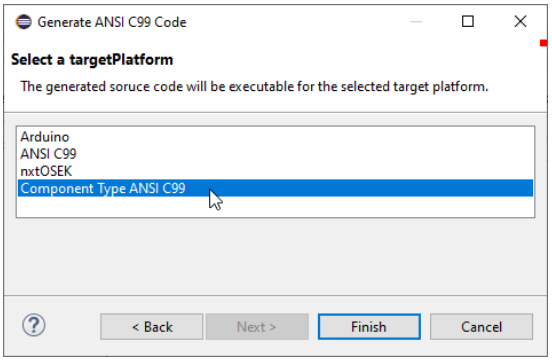
D.1 Generating the Component Code

The component code is generated using the MechatronicUML Tool Suite as shown in Figure D.1. In detail, the following steps are required:

1. The *Export* dialog can be accessed for a project, or directly on a `.muml` model file, via the context-menu of the project or model file.
2. In the *Export* dialog, select the entry *Source Code* in the *MechatronicUML* section as depicted in Figure D.1a.
3. If the *Export* dialog was not opened on a model file directly, select a model file. Otherwise, this step will be omitted.
4. Select the CIC of the model that will be used for code generation.
5. Select a target directory for the source code.
6. Finally, and most importantly, select the option *Component Type ANSI C99* as shown in Figure D.1b.



(a) Select the *Source Code* export option.



(b) Select the *Component Type ANSI C99* as code generation target.

Figure D.1: Generating the component code with the MechatronicUML Tool Suite.

The component code is written to a directory called like the selected CIC. This directory is created inside the target directory that was chosen by the user in step 5. It comprises several directories for the components, RTSCs, message types and operations. See [Poh18] for details.

D.2 Component Header Example

This section shows an exemplary snippet of a component header from the generated component code. Listing D.1 shows a snippet of the `coordinatorComponent_Interface.h` file. This is the component type header file for the `Coordinator` discrete component type from Figure 8.2.

Listing D.1 shows the declarations of all methods which are required for the component's `communicator` (line 35-40) and `overtakingInitiator` ports (line 42-45); the methods for the `overtakingAffiliate` port are declared in the same manner, but not shown in the depicted code. As visible in the listing, there is one *send* method for each sender message type of a port, e.g., line 35 shows the sending of a `grantPermission` message of the `communicator` port. For each receiver message type, there is one *exists* and one *receive* method, e.g., lines 37-38 for the `requestPermission` message type of the `communicator`. All methods require a `Port* p` as parameter (cf. Figure 7.9), and the *send* and *receive* methods take a message pointer `msg` for the message to be sent or to return the received message, respectively.

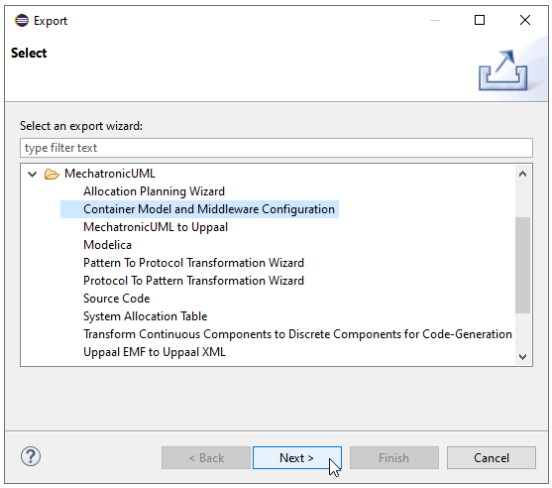
Listing D.1 The communication methods' forward declaration in the component type header file of the `Coordinator` component type.

```
30 /*****
31 *
32 *  Forward Delcaration of Container Functions
33 *
34 */
35 void MCC_CoordinatorComponent_communicator_send_OvertakingPermissionMessagesGrantPermission_OvertakingPermissionMessages_Message(Port* p,
   OvertakingPermissionMessagesGrantPermission_OvertakingPermissionMessages_Message* msg);
36 void MCC_CoordinatorComponent_communicator_send_OvertakingPermissionMessagesDenyPermission_OvertakingPermissionMessages_Message(Port* p,
   OvertakingPermissionMessagesDenyPermission_OvertakingPermissionMessages_Message* msg);
37 bool_T MCC_CoordinatorComponent_communicator_recv_OvertakingPermissionMessagesRequestPermission_OvertakingPermissionMessages_Message(Port* p,
    OvertakingPermissionMessagesRequestPermission_OvertakingPermissionMessages_Message* msg);
38 bool_T MCC_CoordinatorComponent_communicator_exists_OvertakingPermissionMessagesRequestPermission_OvertakingPermissionMessages_Message(Port*
   p);
39 bool_T MCC_CoordinatorComponent_communicator_recv_OvertakingPermissionMessagesExecutedOvertaking_OvertakingPermissionMessages_Message(Port* p
   , OvertakingPermissionMessagesExecutedOvertaking_OvertakingPermissionMessages_Message* msg);
40 bool_T MCC_CoordinatorComponent_communicator_exists_OvertakingPermissionMessagesExecutedOvertaking_OvertakingPermissionMessages_Message(Port*
    p);
41
42 void MCC_CoordinatorComponent_overtakingInitiator_send_OvertakingCoordinationMessagesRequestOvertaking_OvertakingCoordinationMessages_Message
   (Port* p, OvertakingCoordinationMessagesRequestOvertaking_OvertakingCoordinationMessages_Message* msg);
43 void
   MCC_CoordinatorComponent_overtakingInitiator_send_OvertakingCoordinationMessagesFinishedOvertaking_OvertakingCoordinationMessages_Message(
   Port* p, OvertakingCoordinationMessagesFinishedOvertaking_OvertakingCoordinationMessages_Message* msg);
44 bool_T
   MCC_CoordinatorComponent_overtakingInitiator_recv_OvertakingCoordinationMessagesAcceptOvertaking_OvertakingCoordinationMessages_Message(Port*
    p, OvertakingCoordinationMessagesAcceptOvertaking_OvertakingCoordinationMessages_Message* msg);
45 bool_T
   MCC_CoordinatorComponent_overtakingInitiator_exists_OvertakingCoordinationMessagesAcceptOvertaking_OvertakingCoordinationMessages_Message(
   Port* p);
```

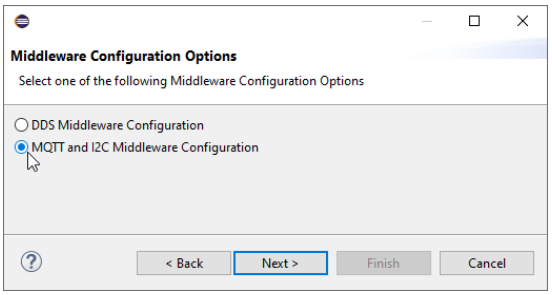
D.3 Generating a Deployment Configuration

In order to generate container code for the Arduino-based target platform, an appropriate MechatronicUML Deployment Configuration model is created using the new implementations presented in Section 7.1 and Section 7.2. This corresponds to the task *T3.2: Generate Deployment Configuration* in Figure 8.6. Executing this task is achieved using the MechatronicUML Tool Suite with the new additions to the platform-modeling approach installed (cf. Section 7.6). The container transformation is accessed via the *Export* dialog of the MechatronicUML Tool Suite as depicted in Figure D.2. In detail, the following steps have to be performed:

1. The *Export* dialog can be accessed for a project, or directly on a `.muml` model file, via the context-menu of the project or model file.
2. In the *Export* dialog, select the entry *Container Model and Middleware Configuration* in the *MechatronicUML* section as depicted in Figure D.2a.
3. If the *Export* dialog was not opened on a model file directly, select a model file. Otherwise, this step will be omitted.
4. Select the MechatronicUML System Allocation of the model that will be used for code generation.
5. Most importantly, select the option *MQTT and I2C Middleware Configuration* as shown in Figure D.2b.
6. Finally, select a target directory for the source code.



(a) Select the *Container Model and Middleware Configuration* export option.



(b) Select the *MQTT and I2C Middleware Configuration* middleware configuration option.

Figure D.2: Generating the deployment configuration with the MechatronicUML Tool Suite.

Using a MechatronicUML Allocation Specification Model for the container transformation and selecting the *MQTT and I2C Middleware Configuration* middleware configuration option as depicted in Figure D.2b produces a deployment configuration that is suitable for the Arduino-based target platform of the application scenario.

D.4 Generating the Container Code

The container code generation for Arduino-based target environments is started via the context-menu of a `.muml_container` file in the MechatronicUML Tool Suite with the new additions to the platform-modeling approach installed (cf. Section 7.6). Use the *mumlContainer* context menu entry, and then select *Generate Arduino Container Code* as depicted in Figure D.3. This creates the code in the directory `arduino-containers` in the parent project of the deployment configuration model file; if this directory already exists, the files will be overwritten.

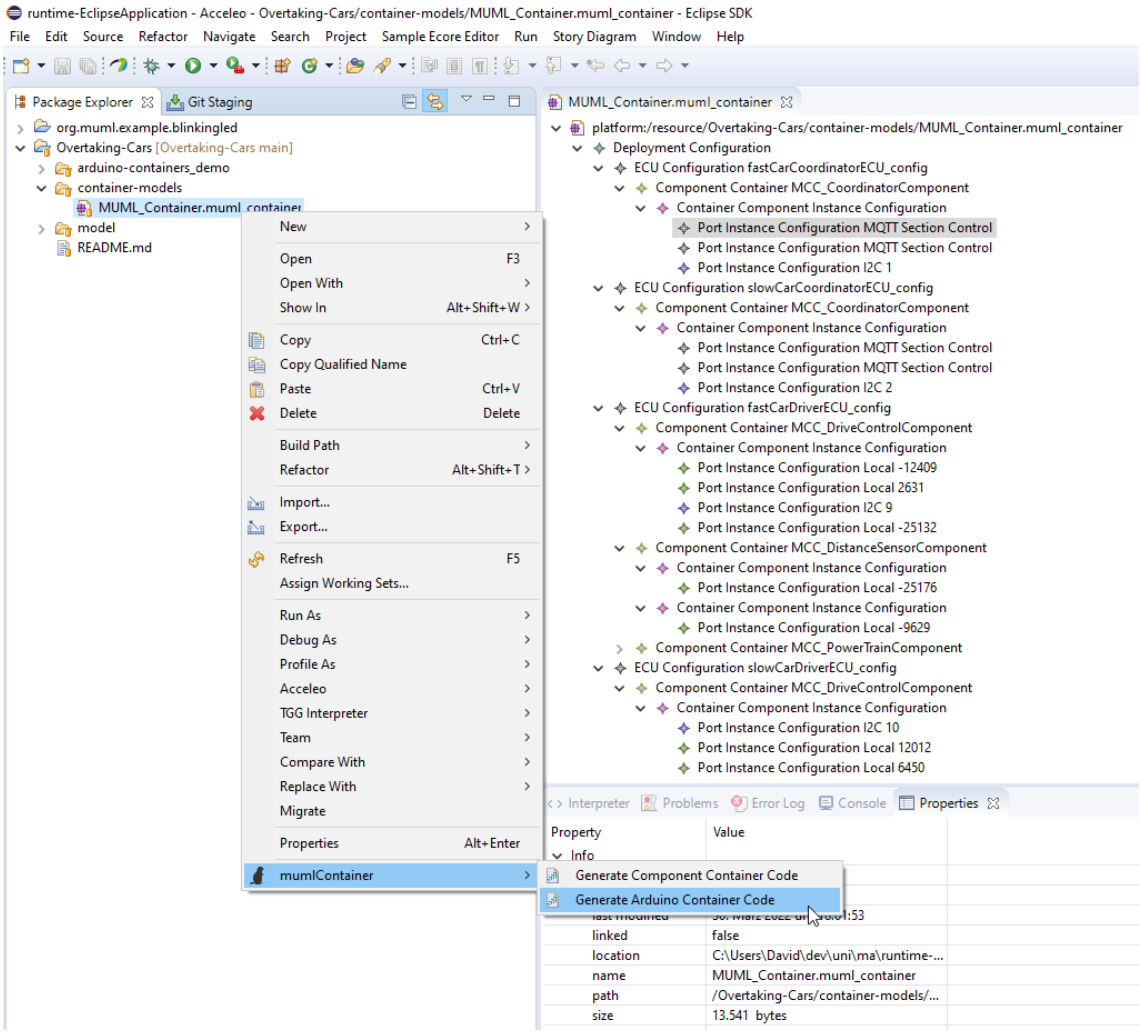


Figure D.3: Generate the Arduino container code via the context menu in the MechatronicUML Tool Suite.

D.5 Characteristic Snippets of the Container Code

This section explains characteristic code snippets the show how the generated container code realizes the concepts presented in Chapter 7. All code snippets are taken from the implementation of the application scenario which is described in Chapter 8. Specifically, the generated container implementation for the `Coordinator` component deployed on the `fastCarECU` is used as an example, i.e., the generated `MCC_coordinatorComponent.cpp` file.

D.5.1 Builder Methods

This subsection shows and explains the generated component builder method for the `Coordinator` component and the port handle builder method for or its `overtakingInitiator` port. These methods are part of the container implementation of the container code. The concepts to generate this code are described in Section 7.3.4 and Appendix B.3.1.

The `Coordinator` is a component modeled with the MechatronicUML PIM (see Section 3.4) and is depicted in Figure 8.7[1] It has three discrete ports. The builder method in Listing D.2 is generated explicitly for this component type and its configuration for deployment on the `fastCarDriverECU`. The `instancePool` and `pool_index` are static variables representing the component instance pool. The builder method retrieves all configuration from the parameter of type `coordinatorComponent_Builder` which is the builder structure. This builder structure contains the configuration for the specific component instance on the `fastCarDriverECU` and is created in the *create method* (cf. Appendix D.5.2). In line 428 of Listing D.2, the usage of the *lifecycle* interface which is provided by the component type (cf. Section 7.3.1) is depicted: the component instance is initialized. Afterwards, all ports are initialized unless they are deactivated (lines 430-447). For each port, the respective port handle builder is called, e.g., for the `overtakingInitiator` port in line 440. This particular port handle builder is explained below.

In order to invoke the correct port handle builder, i.e., the builder for the correct port instance and the correct communication middleware, the component builder structure has the field `create<port-name-in-upper>Handle` for each port. The *create* method is responsible for setting the pointer to the method for the communication middleware according to the deployment configuration. This is visible in Listing D.4 which is explained in Appendix D.5.2: In line 530, the builder structure's general port handle builder field is set to the concrete method for the I2C port handle builder of the `communicator` port handle. Thus, the call `b->createCOMMUNICATORHandle(...);` in line 434 of Listing D.2 resolves to calling the `create_COMMUNICATORI2cHandle` method. In the case of an MQTT configuration, the same applies to line 440 of Listing D.2 where the `b->createOVERTAKINGINITIATORHandle(...);` for the `overtakingInitiator` port instance is resolved to `create_OVERTAKINGINITIATORMqttHandle` as set in line 538 of Listing D.4. This port handle builder method is also generated, and depicted in Listing D.3.

Each port handle builder first sets the type of the port, in Listing D.3 line 479, this type is set to `PORT_HANDLE_TYPE_MQTT` (cf. Figure 7.9). Most importantly, in line 481, memory for the concrete port handle is allocated and a the method `initAndRegisterMqttSubscriber` from the MQTT middleware

[1] More details about the `Coordinator` component in Section 8.2.1.

Listing D.2 The generated builder method for the `Coordinator` component.

```
419 /**
420  *@brief The builder for component instance of Component Type Coordinator
421  *@details This method creates and initializes a component instance properly by using
         ↪ the struct coordinatorComponent_Builder
422  */
423 static CoordinatorComponent* MCC_CoordinatorComponent_Builder(
         ↪ coordinatorComponent_Builder* b){
424   instancePool[pool_index].ID = b->ID;
425   instancePool[pool_index].stateChart =
         ↪ CoordinatorCoordinatorComponentStateChart_create(&instancePool[pool_index]);
426   //call init after RTSC was created
427   CoordinatorComponent_initialize(&instancePool[pool_index]);
428   //For each port initialize it
429   if(b->COMMUNICATOR != PORT_DEACTIVATED) {
430     instancePool[pool_index].communicatorPort.status = b->COMMUNICATOR;
431     instancePool[pool_index].communicatorPort.handle = (PortHandle*) malloc(sizeof(
         ↪ PortHandle));
432     instancePool[pool_index].communicatorPort.handle->port = &(instancePool[pool_index
         ↪ ].communicatorPort);
433     b->createCOMMUNICATORHandle(b, (instancePool[pool_index].communicatorPort.handle))
         ↪ ;
434   }
435   if(b->OVERTAKINGINITIATOR != PORT_DEACTIVATED) {
436     instancePool[pool_index].overtakingInitiatorPort.status = b->OVERTAKINGINITIATOR;
437     instancePool[pool_index].overtakingInitiatorPort.handle = (PortHandle*) malloc(
         ↪ sizeof(PortHandle));
438     instancePool[pool_index].overtakingInitiatorPort.handle->port = &(instancePool[
         ↪ pool_index].overtakingInitiatorPort);
439     b->createOVERTAKINGINITIATORHandle(b, (instancePool[pool_index].
         ↪ overtakingInitiatorPort.handle));
440   }
441   if(b->OVERTAKINGAFFILIATE != PORT_DEACTIVATED) {
442     instancePool[pool_index].overtakingAffiliatePort.status = b->OVERTAKINGAFFILIATE;
443     instancePool[pool_index].overtakingAffiliatePort.handle = (PortHandle*) malloc(
         ↪ sizeof(PortHandle));
444     instancePool[pool_index].overtakingAffiliatePort.handle->port = &(instancePool[
         ↪ pool_index].overtakingAffiliatePort);
445     b->createOVERTAKINGAFFILIATEHandle(b, (instancePool[pool_index].
         ↪ overtakingAffiliatePort.handle));
446   }
447   return &instancePool[pool_index++];
448 }
```

Listing D.3 A port handle builder for MQTT for the `overtakingInitiator` port of the `Coordinator` component.

```
478   ptr->type = PORT_HANDLE_TYPE_MQTT;
479   //create the handle for the discrete port OVERTAKINGINITIATOR with all its message
      ↪ types
480   MqttHandle* handle = (MqttHandle*) malloc(sizeof(MqttHandle)+1*sizeof(MqttSubscriber
      ↪ ));
481   handle->numOfSubs = 1;
482   //register a subscriber for every message type of the port
483   initAndRegisterMqttSubscriber(&(handle->subscribers[0]), b->OVERTAKINGINITIATOR_op.
      ↪ mqtt_option.subscriptionTopic, "OvertakingCoordinationMessagesAcceptOvertaking", 1,
      ↪  sizeof(
      ↪ OvertakingCoordinationMessagesAcceptOvertaking_OvertakingCoordinationMessages_Message
      ↪ ), false );
484   handle->publishingTopic = b->OVERTAKINGINITIATOR_op.mqtt_option.publishingTopic;
485   handle->subscriptionTopic = b->OVERTAKINGINITIATOR_op.mqtt_option.subscriptionTopic;
486   ptr->concreteHandle = handle;
487
488   return ptr;
489 }
490 static PortHandle* create_OVERTAKINGAFFILIATEMqttHandle(coordinatorComponent_Builder*
      ↪ b, PortHandle *ptr){
```

library is called in line 484. Additionally, the MQTT configuration is finally handed over to the concrete handle (line 485-486). This is the point where the container implementation uses the communication middleware library in order to implement the port instances by instantiating concrete port handles (cf. Figure 7.9).

D.5.2 Create Method

The component builder method implemented by the `MCC_coordinatorComponent` container of the `fastCarCoordinatorECU` is shown in Listing D.2, and the port handle builder for the `Coordinator` component's `overtakingInitiator` port is shown in Listing D.3. Listing B.5 shows the Acceleo template to generate the create method for the component instances and how the port instance configuration is used in this template to be weaved into the source code. The builder methods are the same for all component and port instances of a particular type. Using the create method which captures the instance-specific deployment configuration, the component instances are effectively realized. In this section, Listing D.4 shows the generated *create* method for the `coordinator` component instance that is allocated to the `fastCarCoordinatorECU`. The respective configuration including the three different port instance configurations can be seen in lines 528 to 540 of Listing D.4. The configuration is injected by Acceleo based on the deployment configuration presented in Section 8.2.2. If there were additional component instances of type `Coordinator` allocated to the `fastCarCoordinatorECU`, then the *create* method would have additional `case` statements for the respective identifiers which are defined in the `ECUIdentifiers.h` (cf. Section 7.3.2).

Listing D.4 The *create* method implemented by the `MCC_coordinatorComponent` container.

```
517  * @brief Create a component instance with the given id.
518  *
519  * @details Creates a component instance using the builder and the configuration
         ↪ options, and also configures the port instances.
520  *
521  * @param ID the identifier of the component instance
522  */
523 CoordinatorComponent* MCC_create_CoordinatorComponent(uint8_T ID){
524   struct coordinatorComponent_Builder b = INIT_BUILDER;
525   switch(ID){
526     case CI_COMMUNICATORFCOORDINATOR:
527       b.ID = ID;
528       b.COMMUNICATOR = PORT_ACTIVE;
529       b.createCOMMUNICATORHandle = &create_COMMUNICATORI2cHandle;
530       b.COMMUNICATOR_op.i2c_option.ownAddress = 1;
531       b.COMMUNICATOR_op.i2c_option.otherAddress = 9;
532       b.OVERTAKINGAFFILIATE = PORT_ACTIVE;
533       b.createOVERTAKINGAFFILIATEHandle = &create_OVERTAKINGAFFILIATEMqttHandle;
534       b.OVERTAKINGAFFILIATE_op.mqtt_option.publishingTopic = "fastCarCoordinatorECU/
          ↪ communicator.F/overtakingAffiliate1/";
535       b.OVERTAKINGAFFILIATE_op.mqtt_option.subscriptionTopic = "slowCarCoordinatorECU/
          ↪ communicator.S/overtakingInitiator1/";
536       b.OVERTAKINGINITIATOR = PORT_ACTIVE;
537       b.createOVERTAKINGINITIATORHandle = &create_OVERTAKINGINITIATORMqttHandle;
538       b.OVERTAKINGINITIATOR_op.mqtt_option.publishingTopic = "fastCarCoordinatorECU/
          ↪ communicator.F/overtakingInitiator1/";
539       b.OVERTAKINGINITIATOR_op.mqtt_option.subscriptionTopic = "slowCarCoordinatorECU/
          ↪ communicator.S/overtakingAffiliate1/";
540     break;
541   default:
542     break;
543   }
544   return MCC_CoordinatorComponent_Builder(&b);
545 }
546
```

D.5.3 Send Method

The container implementation also realizes the *send*, *receive* and *exists* methods. Listing D.1 shows the forward declaration of these methods in the component type code. Here, Listing D.5 shows the implementation of two of these methods. Firstly, in lines 182-192, the method for sending a `grantPermission` message of the `OvertakingPermission` RTCP via the `communicator` port of the `coordinator` instance of the `fastCar`. This port is configured for I2C communication as visible in lines 530-533 of Listing D.4. Equivalently, Listing D.5 shows the send method for the `requestOvertaking` message via the `overtakingInitiator` port instance of the same `communicator` component instance in lines 268-278. The template for a part of this *send* method is depicted in Listing B.6.

This implementation of the *send* method is contained in the generated container implementation. Thus, the generated container code matches and implements the forward declaration of the communication methods in the component header (cf. Section 8.2.1).

Listing D.5 Two *send* method using I2C and MQTT implemented by the `MCC_coordinatorComponent` container.

```
182 void MCC_CoordinatorComponent_communicator_send_OvertakingPermissionMessagesGrantPermission_OvertakingPermissionMessages_Message(
    ↪ Port* port, OvertakingPermissionMessagesGrantPermission_OvertakingPermissionMessages_Message* msg){
183   I2cHandle* i2cHandle;
184   switch(port->handle->type) {
185     case PORT_HANDLE_TYPE_I2C:
186       i2cHandle = (I2cHandle*) port->handle->concreteHandle;
187       sendI2cMessage(i2cHandle->otherI2cAddress, "OvertakingPermissionMessagesGrantPermission", (byte *) msg, sizeof(
    ↪ OvertakingPermissionMessagesGrantPermission_OvertakingPermissionMessages_Message));
188       break;
189   default:
190     break;
191   }
192 }

    ...

268 void
    ↪ MCC_CoordinatorComponent_overtakingInitiator_send_OvertakingCoordinationMessagesRequestOvertaking_OvertakingCoordinationMessag
    ↪ ...(Port* port, OvertakingCoordinationMessagesRequestOvertaking_OvertakingCoordinationMessages_Message* msg){
269   MqttHandle* mqttHandle;
270   switch(port->handle->type) {
271     case PORT_HANDLE_TYPE_MQTT:
272       mqttHandle = (MqttHandle*) port->handle->concreteHandle;
273       sendMqttMessage(mqttHandle->publishingTopic, "OvertakingCoordinationMessagesRequestOvertaking", (byte*) msg, sizeof(
    ↪ OvertakingCoordinationMessagesRequestOvertaking_OvertakingCoordinationMessages_Message));
274       break;
275   default:
276     break;
277   }
278 }
```

D.6 Device APIs and Software Libraries

From the process for the software construction (see Figure 8.6), the steps *T3.1*, *T3.4* and *T3.5* are not covered in Section 8.2 when the implementation of the application scenario is explained. This is due to the fact that these steps are not are not changed in this thesis, thus, now new implementations are demonstrated. However, the inclusion of device APIs and software libraries is critical for the implementation of the application scenario, as the corresponding concepts enable to include the robot car libraries into the generated code. The robot car libraries contain the continuous behavior of the robot car (cf. Appendix A) and are essential for implementing the physical behavior in the generated code, i.e., the sensing an moving. Therefore, the concept for device APIs and software libraries is the content of this section.

More specifically, the concepts allow the inclusion of external source code into the software construction process, either as device implementation realizing the behavior of continuous component, or as software libraries that are used as operation repositories within RTSCs which specify the discrete components' behavior.

Listing D.6 The robot car libraries modeled with the ApiML.

```
1  import "robocar.muml"
2
3  OperatingSystem: RoboCarLibraries{
4    Device_API_Calls: DistanceSensor {
5      void initializeDistanceSensors();
6      int32 getFrontDistance();
7      int32 getRearDistance();
8    }
9    Device_API_Calls: MotorDriver {
10     void setSpeed(int32 speed);
11   }
12 }
```

First, *T3.1: Model Device Access* allows the modeling of a device's APIs using the ApiML (cf. Section 4.2.1). Listing D.6 shows how the APIs of the robot car libraries that are used to implement the `DistanceSensor` and `PowerTrain` continuous components are modeled using the ApiML[2]. Then, using the ApiMappingML, these API calls can be mapped to continuous port instances. However, the ApiMappingML is not implemented in the MechatronicUML Tool Suite exactly like it is specified by Pohlmann in [Poh18]. The MechatronicUML Tool Suite in version 1.0 contains an implementation based on an older specification [BCD+14], but the identification of the port instances ind the ApiMappingML code does not work as specified and was hence applied to the robot car libraries in this work. Some more investigation is necessary in order to explore the specifics of the ApiMappingML implementation. Then, an `.apimapping` file can be used to generate the device access code, which corresponds to *T3.5: Generate Device Access Code* of Figure 8.6. As mentioned in Section 7.3.2 and Section 8.2.3, the container code generation generates the `APImapping` directory. It contains method stubs for all the methods the port instances require in order to interact with the devices; thus, instead of generating the device access code, it can also be implemented manually.

[2]The ApiML was formerly called *operating system language* [BCD+14] and its file ending is `.osdsl`.

Listing D.7 The header file declaring the port access command for the `frontDistanceSensor.F` of the `fastCar`.

```
#ifdef __cplusplus
extern "C" {
#endif
// Start of user code FORINCLUDES
#include "DistanceSensor.h"
// End of user code
#include "standardTypes.h"
#include "customTypes.h"
void CI_FRONTDISTANCESENSORFDISTANCESENSORdistancePortaccessCommand(int32_T* distance);
// Start of user code API

  // End of user code
#ifdef __cplusplus
}
#endif
```

Listing D.8 The implementation of the port access command for the `frontDistanceSensor.F` of the `fastCar`.

```
#include "CI_FRONTDISTANCESENSORFDISTANCESENSORdistancePortaccessCommand.h"
void CI_FRONTDISTANCESENSORFDISTANCESENSORdistancePortaccessCommand(int32_T* distance){
  // Start of user code API
  *distance = getFrontDistance();
  // End of user code
}
```

This manual implementation is shown in Listing D.7 and Listing D.8 for the continuous component `frontDistanceSensor.F` of the `fastCar` component instance. This is implemented equivalently for all other distance sensors, and similarly for the `PowerTrain` instances as well. The generated method stubs include *protected regions* for Acceleo to not overwrite manually added code when the code is regenerated, e.g., after a model adaption.

As the robot car libraries require that their initialization methods are called, but the initialization of devices is not yet implemented as specified in [Poh18], the initialization is also added manually. Thus, in this work, additional protected regions are introduced in the Acceleo template for the Arduino main file that allow initializing the sensors and actuators. This is depicted in Listing D.9 which shows the Arduino main file of the `fastCarDriverECU`, parts of which are also shown in Listing 8.2.

Similar to the device API integration, existing software libraries can be integrated as operation repositories to implement the behavior of the discrete components (cf. Appendix C.1.3). The component type code generation creates method stubs for all operation repositories. For the robot cars, it hence creates methods stubs for the three operations of the operation repository shown in Figure C.5. These methods stubs can be replaced or completed with the corresponding library code, which is reflected by *T3.4: Implement Adapter to Software Libraries*. In the presented

Listing D.9 The device initialization in the Arduino main file of the `fastCarDriverECU`.

```
14 void setup(){
   ...
19   // Start of user code DEVICEINIT
20   initializeDistanceSensors();
21   initLineFollower();
22   // End of user code
   ...
34 }
```

implementation of the application scenario, the operation repository stubs are implemented manually. This is shown for the `followLine` operation of the `RobotCarPowerTrain` operation repository depicted in Figure C.5. The corresponding source code is depicted in Listing D.10. In this thesis, the changing of lanes has not yet been implemented as part of the robot car libraries (cf. Appendix A). This can be done in the future to complete the continuous functionality required for the overtaking scenario.

Listing D.10 The operation implementation of the `followLine` operation of the `RobotCarPowerTrain` operation repository.

```
29 void RobotCarPowerTrain_robotCarPowerTrainFollowLine(int32_T velocity){
30 /** Start of user code RobotCarPowerTrain_robotCarPowerTrainFollowLine **/
31     followLine(velocity);
32 /**End of user code**/
33 }
```

As a final comment, the `MotorDriver.h` library that is used to implement the the port access commands of the `PowerTrain` component, is initialized as part of the `initLineFollower();` command depicted in line 21 of Listing D.9. This initialization is also required for the implementation of the operation `followLine` to work correctly, as the `LineFollower.h` library is used for that purpose, see Listing D.10. This has been added manually because the operation repository concept does not include a concept for the library initialization as they are assumed to be passive software components (cf. [Poh18]).

D.7 Manual Adaptions

As explained in Section 7.5, a couple of manual adaptions are required for the building of the software. Additionally, manual adaptions are also required for the code to be working. These adaptions are summed up here.

Firstly, the adaptions comprise the list presented in Section 7.5. Secondly, in order to include the robot car libraries, the manual adaptions describe in Appendix D.6 are undertaken. Thirdly, in order to initialize the two `RobotCar` component instances, in particular their respective `DriveControl` instance with the corresponding velocity values, manual adaptions are necessary. The reason is that the

MechatronicUML PIM does not provide a concept to parameterize RTSCs. Pohlmann introduces such a concept, however it is implemented for the platform-specific modeling only and, in its current state of implementation, cannot be used for the component RTSC instantiation [Poh18]. Therefore, in the `driveControlDriveControlComponentStateChart.c` of both ECUs' directories, the values for the `desiredVelocity` are set to reflect the behavior of the `fastCar` with `desiredVelocity=80;` and the `slowCar` with `desiredVelocity=50;`. The `slowVelocity` is set to 50 in both cases. The other RTSC variables also have to be initialized with suitable values.