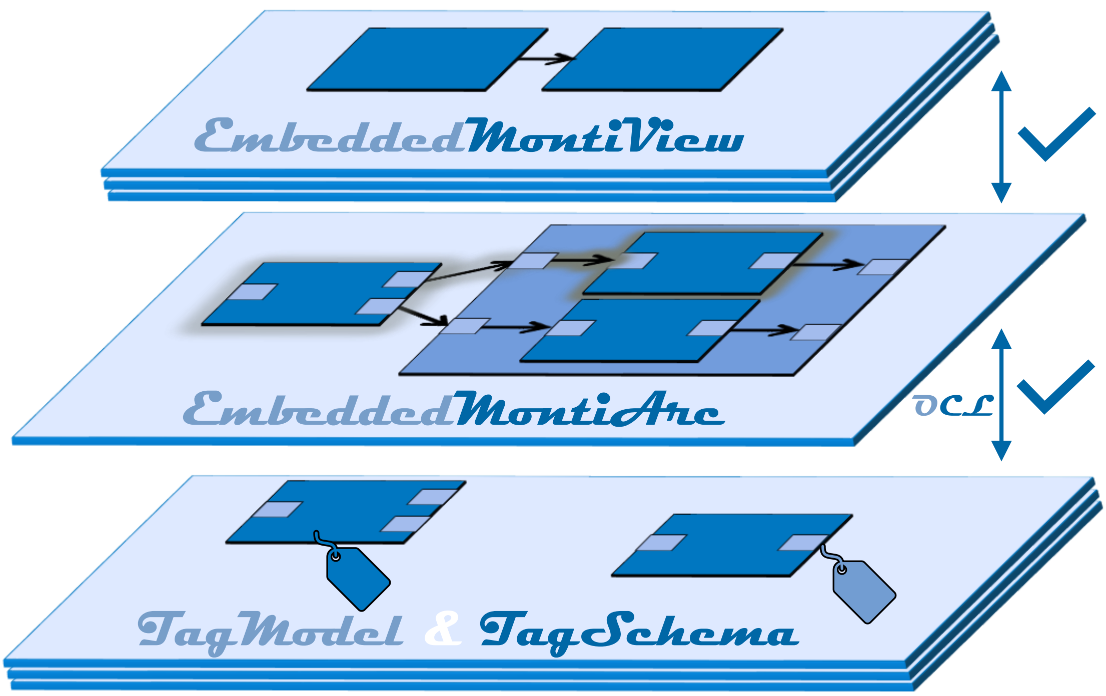
Michael von Wenckstern

Verification of Structural and Extra-Functional Properties in Component and Connector Models for Embedded and Cyber-Physical Systems



Aachener Informatik Berichte,
Software Engineering

Band 44

Hrsg.: Prof. Dr. B. Rumpe

[vW20] M. von Wenckstern:
Verification of Structural and Extra Functional Properties in Component and Connector Models for Embedded and Cyber Physical Systems.
Shaker Verlag, ISBN 978-3-8440-7239-6. Aachener Informatik-Berichte, Software Engineering, Band 44. March 2018.
www.se-rwth.de/publications/

Verification of Structural and Extra-Functional Properties in Component and Connector Models for Embedded and Cyber-Physical Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Math. Michael von Wenckstern
aus Karl-Marx-Stadt

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
University Professor Shahar Maoz, PhD

Tag der mündlichen Prüfung: 31.10.2019

Abstract

The industry area of embedded and cyber-physical systems is one of the largest and it influences our daily life. The global embedded systems marked was valued at about 160 billion US dollar in 2015 and it is getting up to 225 billion US dollar by end of 2021 [Zio17]. Example domains of embedded and cyber-physical systems are: automotive [DHJ+08], avionics [FLV03], robotics [WICE03], railway [DNCH10], production industry [EWSG94], telecommunication [ZSM11], healthcare [ERA09], defense [BNP+04], and consumer electronics [VOVDLKM00].

Model-based engineering, esp. component and connector (C&C) models to describe logical architectures, are one common approach to handle the large complexity of embedded and cyber-physical systems [FR07, MBNJ09, OMG15, EJL+03]. Components encapsulate software features; the hierarchical decomposition of components enables formulating logical architectures in a top-down approach. Connectors in C&C models describe the information exchange via typed ports; they model black-box communication between software features.

The current development of complex C&C-based embedded systems in industry mostly involves the following steps [BMR+17a, DGH+19]: (1) formulating functional and extra-functional requirements as text in *IBM Rational DOORS*; (2) creating a design model of the software architecture including its environment interactions in *SysML*; (3) developing a complete functional/logical model to simulate the embedded system in *Simulink*; and (4) system implementation based on available hardware in C/C++ satisfying all extra-functional properties.

This current development process has the following disadvantages [KBFS12, HKK+18, BMR+17a]: (a) *SysML* models do not follow a formalized approach; i.e., engineers may interpret these models differently due to missing semantics; (b) the check between the informal *SysML* architecture design against the *Simulink* model is done manually, and thus, error-prone and very time-consuming; (c) refactoring of *Simulink* models (e.g., dividing a subsystem) needs manual effort in updating the design model, and therefore, due to timing constraints this step is often skipped resulting in inconsistencies; and (d) most tools do not support a generic approach for different extra-functional property kinds, and thus, extra-functional properties are mostly modeled as comments or stereotypes and consistencies between these properties are checked manually.

This thesis aims to improve the software development process of large and complex C&C models for embedded and cyber-physical systems by providing model-based methodologies to develop, understand, validate and maintain these C&C models. Concrete, this thesis presents concepts to support the embedded software engineer with: (i) automatic consistency checks of C&C models; (ii) automatic verification of logical C&C models against their design decisions; (iii) automatic addition of traceability links between design and implementation models; (iv) finding structural inconsistencies during model evolution; (v) providing a flexible framework to define different extra-functional property types; (vi) presenting an *OCL* framework to specify (company-specific) constraints about structural or extra-functional properties for C&C models; and (vii) generation of positive or negative witnesses to explain why a C&C model satisfies or violates its extra-functional or structural constraints or its design decisions.

Prototype implementations of above mentioned concepts and an industrial case study in cooperation with Daimler AG show promising results in improving the model-based development process of embedded and cyber-physical systems in industry.

Acknowledgments

Special thanks goes to my doctor father Prof. Dr. Bernhard Rumpe for giving me the opportunity to research in the area of model-driven software engineering and for providing many connections to other research groups and industrial partners; participating in several research and automotive industry projects gave me access to many different views on this research topic. Furthermore, I want to express great laud to Prof. Dr. Bernhard Rumpe for the fruitful discussions, ideas and feedback rounds - especially during the 3-day long research workshops - on this thesis.

I would like to thank Prof. Shahar Maoz for being the second reporter of this thesis. Additionally, I want to express gratitude to Prof. Shahar Maoz, Dr. Jan Oliver Ringert, and Prof. Dr. Bernhard Rumpe for the successful teamwork on the GIF research project I-1235-407.6/2014, for the published papers together, and for the time together in Tel Aviv and in Aachen. I also want to thank the Deutsche Forschungsgemeinschaft (DFG) for funding this research and thus supporting this thesis.

Moreover, I would like to thank Prof. Dr. Stefan Kowalewski for heading my PhD exam committee, and Prof. Dr. Erika Ábrahám for being part of this committee.

In addition, my thank goes to Prof. Dr. Manfred Nagl for many interesting discussions about computer science, GI, and architecture.

"If everyone is moving forward together, then success takes care of itself." - Henry Ford about teamwork. Therefore, special thanks goes to Evgeny Kusmenko; together we developed the EmbeddedMontiArc modeling family and had good times on several conferences and workshops presenting our papers. Moreover, I want to thank Dr. Pedram Mir Seyed Nazari for the interesting discussions about language engineering and digitalization; for the first level MontiCore support thanks goes to Pedram, Marita Breuer, and Galina Volkova. Additionally, I want to thank Dr. Alexander Roth, Dr. Christoph Schulz, and Vincent Bertram for the many papers we wrote together. I want to thank Deni Raco for organizing all the extra activities such as poker, or soccer events on our chair.

I also would like to thank the students whose bachelor or master theses supported my research significantly; without them the large evaluation part would not be possible. Thanks goes to "Web(Assembly) expert" Jean-Marc Ronck, "OCL specialist" Ferdinand Mehlan, "C++ generator enthusiast" Sascha Schneiders, and "tooling and gaming fan" Malte Heithoff for supporting me more than two years.

I want to extend my thanks to the other students: Luca Tabone, Stefan Brunecker, Alexander Kogaj, Nicolai Strodthoff, Severin Tolksdorf, Vladimir Parashin, David Ernst, Christian Bajana, Igor Shumeiko, Dinh-An Ho, Fabian Kahlert, Manuel Schrick, Ievgen Strepkov, Ahmet Tayfun Özen, and all seminar and lab students.

Special thanks for reviewing this PhD thesis goes to Sebastian von Wenckstern, Pedram Mir Seyed Nazari, Evgeny Kusmenko, Kai Adam, Simon Varga, and Oliver Kautz.

Furthermore, I would like to thank all colleges of the Software Engineering chair for the nice time together: Lennart Bucher, Arvid Butting, Manuela Dalibor, Anabel Derlam, Florian Donath, Imke Drave, Robert Eikermann, Dr. Timo Greifenberg, Sylvia Gunder, Dr. Arne Haber, Dr. Lars Hermerschmidt, Dr. Christoph Herrmann, Gabriele Heuschen, Steffen Hillemacher, Dr. Katrin Hölldobler, Carsten Kolassa, Thomas Kurpick, Achim Lindt, Dr. Markus Look, Matthias

Markthaler, Dr. Judith Michael, Dr. Klaus Müller, Sonja Müßigbrodt, Lukas Netz, Jerome Pfeiffer, Nina Pichler, Dimitri Plotnikov, Manuel Putzer, Dr. Dirk Reiß, David Schmalzing, Stephanie Schrader, Brian Sinkovec, Max Voß, Louis Wachtmeister, and Dr. Andreas Wortmann.

My largest thanks goes to my family - Bettina, Bodo, and Sebastian von Wenckstern - who supported me all these years; and their belief that education is the best investment for my future.

Aschaffenburg, 13.12.2019
Michael von Wenckstern

Contents

Chapter 1.

Introduction

Modern software systems are becoming more and more complex in many areas [FR07, PBKS07, FB11, Bro05]. One prominent area for complex software and software-intensive systems is the field of embedded and cyber-physical systems [Lee08]. Engineering cyber-physical systems rises specific challenges that are rarely present in other software engineering disciplines due to the systems' steady interactions with their environment [KRRvW17]. A common approach, in current research and in industry, is to describe embedded systems and their real-world interactions as component and connector models [Rin14, Hab16, BS12, MT10, CGL$^+$03]. Component and connector models describe the logical architecture of cyber-physical systems by focusing on software features and their logical communications [KRRvW17, Rin14]. In component and connector (C&C) models, hierarchical decomposed components encapsulate software features, and connectors model the data flow between components via typed ports [HRR12, The18k, DvdHT01, Mod05, Nat98, FMS11]. As extra-functional properties, e.g., worst-case-execution-time, memory and power/fuel consumption, safety, and security, are also key features for the success of embedded systems, component and connector models are often enriched with many of these properties [MRRvW16, Gru07, SSCC09, SCS11a, SSCS16, CM78, Rom85, RM06].

But, the process to develop, understand, validate, and maintain large component and connector models (with extra-functional properties) for complex embedded software is onerous, time and cost intensive [BMR$^+$17a].

Hence, the aim of this thesis is to support the automotive software engineer (cf. Chapter 2) with:

(i) automatic consistency checks of large C&C models,
(ii) automatic verification of C&C models against design decisions,
(iii) tracing and navigating between design and implementation models,
(iv) finding structural inconsistencies during model evolution,
(v) presenting a flexible approach to define different extra-functional properties for C&C models, and
(vi) providing a declarative specification framework to formalize constraints on C&C models for extra-functional properties in order to execute automatic consistency checks.

The remainder of this chapter is structured as follows: Section 1.1 introduces the context of and some preliminaries for this thesis; i.e., component and connector models and their views for specifying design decisions; model based (systems) engineering and domain specific languages; and *MontiCore*, a tool for creating domain specific languages, used to engineer the language family presented in this thesis. Section 1.2 presents the requirements on this PhD thesis; these are based on the working packages of the proposal of the GIF grant I-1235-407.6/2014, that founded the research leading to these results. Section 1.3 states the research question and describes main contributions of this thesis. Section 1.4 outlines the chapter structure of this document. Finally, Section 1.5 gives an overview of related publications created in context of this thesis.

1.1. Context and Foundations

The foundations for the developed methodologies, concepts, algorithms, and tools are mostly formed by previous research of Software Engineering at RWTH Aachen University in Germany and School of Computer Science at Tel Aviv University in Israel.

In more detail, the *EmbeddedMontiArc* language family to model and simulate cyber-physical and embedded systems is based on *MontiArc* [HRR12, Rin14, Wor16, Hab16], whereby the brain domain specific language *NestML* [Plo18, BEP$^+$18] inspired the unit concept used in *EmbeddedMontiArc*. The *EmbeddedMontiView* language, to specify component and connector design decisions, uses concepts of the textual *MontiArc* derivate for C&C views [Rin14, MRR13, MRR14]. The symbol table based tagging mechanism, presented in this thesis to enrich component and connector models with extra-functional properties, is an extension of the tagging mechanism for DSLs (domain specific languages) [GLRR15, Loo17]. The object constraint language to formalize the semantic relationship of extra-functional properties between *EmbeddedMontiArc* and *EmbeddedMontiView* models added units and advanced type resolving features to the existing UML/P *OCL* language [Rum16].

The algorithms of C&C views verification as well as the representation of positive satisfaction and negative non-satisfaction witnesses are based on previous work [Rin14, MRR13, MRR14]; this thesis adapted and extended these previous algorithms and witnesses to fit better in the area of embedded and cyber-physical systems.

The architectural modeling concepts, esp. C&C models and C&C views, of Haber, Maoz, Ringert, and Rumpe are general and domain agnostic. This thesis extends their work with modeling concepts used very much in embedded systems; esp. new port type system with units, matrices, and ranges; static typed arrays of components and ports; as well as component interfaces for product-lines. *EmbeddedMontiArc* and *EmbeddedMontiView*, developed during this PhD thesis, introduce many new language features to facilitate an easier integration of C&C modeling concepts into current development processes of automotive[1] companies. To show the benefits of both languages, this thesis presents many code snippets based on real-world examples in the area of embedded systems.

The next subsections explain in more detail:

- *How do component and connector models and views look like?*
- *What is model based (systems) engineering in the context of this thesis?*
- *What are the important aspects of designing domain specific languages?*
- *How does MontiCore help to create domain specific languages in an efficient and easy way?*

[1] Most likely, the findings of *EmbeddedMontiArc* and *EmbeddedMontiView* according to the development process of automotive systems engineering can also be transferred to other embedded system's domains, e.g., aerospace or robotics. However, this thesis evaluated these two languages according to development processes and examples provided by automotive companies (cf. Section 2.1, Section 2.2, and Chapter 8).

1.1.1. Component and Connector Models and their Specification Language

Main sources: [MR13, Section 1], [KRRvW17, Section 1], [BMR+17a, Section II]

Model-based (systems) engineering together with domain-specific languages (cf. Subsection 1.1.3) help to address the problem-implementation gap by providing a language focusing on the domain rather than on the solution. This subsection describes component and connector (C&C) models and C&C views, which are two DSLs for the logical layer used in component-based software engineering. Today, C&C models are mostly used in embedded or cyber-physical systems, e.g., in avionics, robotics, railway, (production) industry, and automotive. Typical applications developed with C&C models in automotive industry are, among others, trajectory planning, lane correction, battery management, engine control, clutch lock-up, anti-lock braking system, transmission system, automotive suspension, climate control system, power window control, electronic stability control, electronic power steering, adaptive cruise control, adaptive forward lighting, and automatic park assistance systems [KRRvW17, The18j, BMR+17a].

As the name component and connector model suggests, the structure of a C&C model consists of components at different containment levels and connectors connecting components via their typed input and output ports [MR13, Hab16, HRR12, KRRvW17, MRR13, MRR14, MRRvW16, MMR+17, BMR+17a]. Due to the many applications for C&C models, there exist already several tools and methodologies for creating, analyzing, maintaining, and synthesizing them [MR13, KRRvW17], in industry and academia; e.g., MathWork *Simulink* [The18k], *Acme-Studio* [SG04], *AutoFOCUS* [AVT+15], IBM *Rational Rhapsody Architect* [IBM18], *Modelica* [Mod05, EMO99], *MARTE* [OMG11], *LabView* [Nat98], *SysADL* [OLB16], *GALS* [MVF00] (Globally Asynchronous, Locally Synchronous), and *ASCET* [DSW+03] (Advanced Simulation and Control Engineering Tool).

The main advantage of C&C models is their hierarchical decomposition, which enables decomposing complex functions in smaller ones. This way large systems can be implemented by different teams or even different stakeholders in a divide and conquer manner [KRRvW17].

However, the strict hierarchical decomposition of C&C models (showing only one layer at a time) limits the overall design process of a system where different groups or stakeholders participate by providing partial knowledge about the system [MR13]. In contrast to the implementation process that is based on an existing architecture, the design process to create this architecture focuses on multiple user stories, or requirements. Therefore, the design process needs to deal with concern-specific interests resulting in models crosscutting hierarchical boundaries [MR13, Rin14, MRR13].

C&C views - introduced by Rumpe, Ringert, and Maoz [MRR13, MRR14, Rin14] - are invented to describe (abstract) relations between components ignoring hierarchical boundaries. Since C&C views' syntax is an extension of the well-known syntax of C&C models, C&C views describe structural properties of C&C models in an intuitive and formal way [MRR13, BMR+17a]. C&C views enable to abstract from direct hierarchy, direct connectivity, port names and types [MR13, MRR13, BMR+17a, Rin14]. The abstraction of direct hierarchy enables to omit intermediate components in C&C views. The abstraction of direct connectivity enables to connect components in C&C views which are only indirect connected in the C&C model.

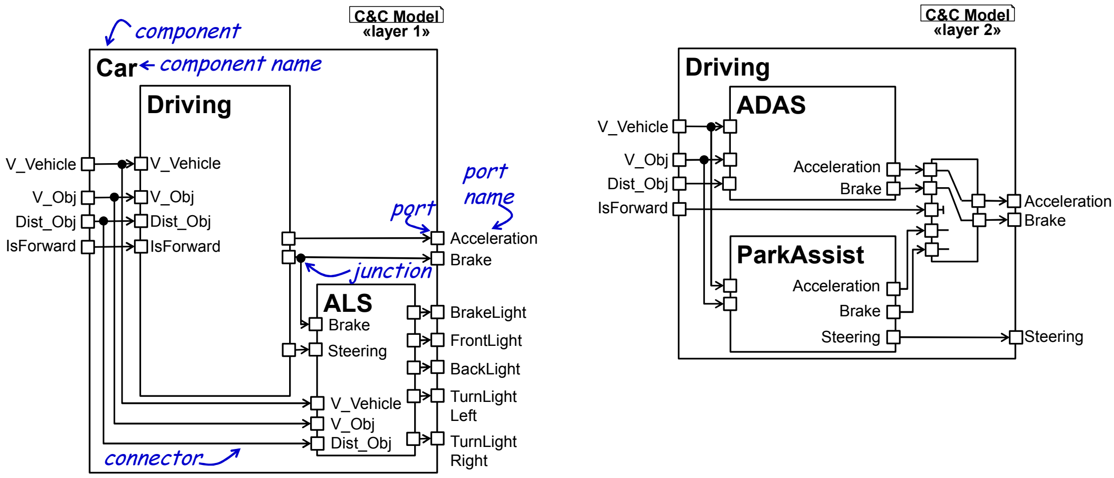


Figure 1.1.: C&C Model of a simple car software (inspired by [BMR+17a]).

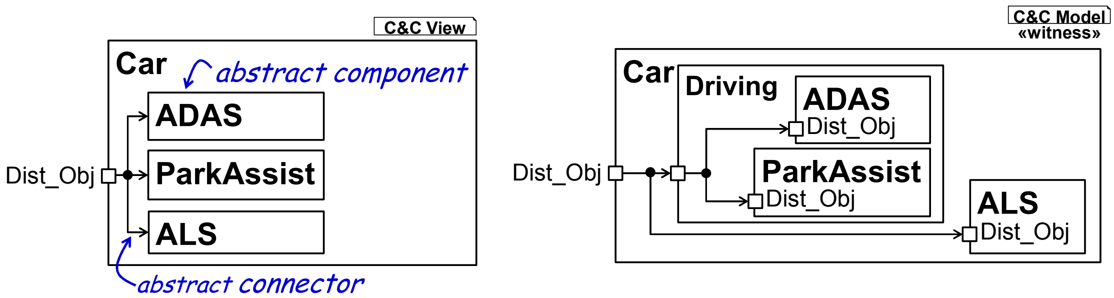


Figure 1.2.: C&C view (left) and C&C witness (right) generated based on C&C model in Figure 1.1 and C&C View (inspired by [BMR+17a]).

Specifically, a C&C view should focus on the design decision relating to one concern, user story, or requirement; and thus, a C&C view typically contains only a small subset of components and connectors belonging to a system.

Recent work [MRR13, Rin14] on C&C views already investigated (1) on C&C view synthesis to create the complete structural C&C model based on multiple structural design decisions; as well as (2) on C&C view verification to create satisfaction and non-satisfaction witnesses explaining why an implementation is (not) conform to a design decision.

The next paragraphs present the difference and the relationship between C&C models and C&C views on small examples.

Figure 1.1 shows an example C&C model of a simplified car software component. Similar to all existing C&C modeling tools, the figure shows two separate hierarchy levels. The `Car` component (left part in Figure 1.1) controls acceleration, brake, and light signals of a car based on the current velocity and drive direction (`isForwad`) of the vehicle as well as based on the distance and speed of an obstacle in front of the car. To handle these tasks, the `Car` component is decomposed into the `Driving` and `ALS` (Adaptive Light System) subcomponents. Since the

`Driving` component (right part in Figure 1.1) is responsible for parking and a superior driver experience on highways; it is further decomposed into three components: `ADAS` (Advanced Driver Assistance System), `ParkAssist`, and a *Switch* merging signals.

The C&C view in Figure 1.2 represents the architectural design decisions dealing with sensor data measuring the distance to the obstacle in front of the car. The C&C view states that the input port `Dist_Obj` has impact on `ADAS` (car needs to hold distance), `ParkAssist` (car must fit in the parking slot), and `ALS` (car should not blind pedestrians) subcomponents. The crosscutting nature of C&C views enables to connect the input port with components being defined in two hierarchical different layers (cf. left and right part of Figure 1.1). Due to the hierarchical abstraction of C&C views, the left part in Figure 1.2 omits the `Driving` component.

The C&C witness, right part in Figure 1.2, reasons why the C&C model in Figure 1.1 satisfies the C&C View, left part in Figure 1.2. The witness contains the complete hierarchy, regarding to the C&C model, of components being addressed in the C&C view; thus, the witness contains the `Driving` component. Additionally, the witness contains all ports of the C&C model being addressed in the C&C view directly or being necessary for resolving an abstract connector in a C&C view to a connector chain in a C&C model. Therefore, the witness contains the `Dist_Obj` port for the components `Car`, `Driving`, `ADAS`, `ParkAssist`, and `ALS`. The C&C view's abstract connector from `Car`'s `Dist_Obj` to the component `ADAS` is resolved to a connector chain of two connectors: (1) connector from `Car`'s `Dist_Obj` to `Driving`'s `Dist_Obj`, and (2) connector from `Driving`'s `Dist_Obj` to `ADAS`'s `Dist_Obj`. As a result, the witness shows these two connectors; similar holds for the other connectors shown in the C&C witness.

1.1.2. Model Based Systems Engineering

Main sources: [HR17, Section 1.1], [MSN17, Section 2.1], [CBCR15, Section 2]

Model based (systems) engineering uses models to speed up the overall software (systems) engineering process. This thesis uses the following definition for a model:
"A **model** is an abstraction of a (real) [software] system allowing predictions or inferences to be made." [Küh06]
This means a model abstracts unnecessary details [Rum16] from the original by showing specific, for the system or application interesting, viewpoints/aspects [MSN17] (cf. [Küh06, Sta73, HBB$^+$94, BG01, Sei03, Sch12]).

There exists several development processes using models in different intensities. These processes are called **MBE** (model-based engineering), **MDE** (model-driven engineering), and **MDD** (model-driven development). Now, these processes are put into a relationship to see how they differently work with models.

MBE is a softer version of MDE; since in MBE software models play an important role (e.g., models as documentation on which developers create manually code), but they may not be first-level artifacts of the process (i.e., they may not "drive" the engineering process) [Cab14].

"Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing." [MCF03]. MDD uses models as first-level artifacts of the development process to generate source code, or to synthesize a larger artifact based on many smaller first-level models.

Examples of MDD at the Software Engineering chair at RWTH Aachen University are:

- *MontiCore* [HR17]: It uses grammar models to generate parser, visitor, and AST.
- *MontiDEx* [Rot17]: It generates a GUI and CRUDS (create, read, update, delete, and search) application logic based on class diagram models. Similar applications are MaCoCo [ANV+18], MontiWIS [Gül14], and WebDEx [Rei16].
- *NESTML* [Plo18, BEP+18]: It uses *NESTML* models to generate C++ code for the *NEST* (Neural Simulation Tool) [GD07] infrastructure. This C++ code is used to simulate neural activities in the brain.
- *MontiArcAutomaton* [Wor16]: It uses component and connector models (similar to *SysML*'s internal block diagrams) and automata as primary modeling artifacts to generate Java or Python source code.
- Facility Models [KLPR12]: They describe the energy flow inside buildings, so that positions and states of hot water circuits and central air conditions can be optimized to develop energy efficient buildings.
- Besides only generating code based on models, model artifacts are also used to synthesize one complete behavior model based on multiple LTL [MR15] or automata [Rin14] specification models. Another example for model-based synthesis is the creation of timetables for TV channels based on broadcast license permissions for movies, or series.

MDE is more general than MDD, since development is only one activity within engineering. Activities of MDE, which are not part of MDD, are, e.g., model-based evolution [RSvW+15, KSRvW18], variability modeling or extraction [KRR+16, RRS+16, HRR+11], and maintaining legacy systems [BRR+10]. Thus, the relationship (based on Jordi Cabot's blog [Cab14]) can be summarized as follows: MBE $\supset$ MDE $\supset$ MDD

In the following this section continues to explain the wordings MBSE, CBSE, and DSL.

MBSE (model-based systems engineering) uses models to describe system requirements and system designs as well as support system analysis, and system validation [INC07]. In contrast to model-driven software engineering - mostly focusing on one domain such as financial service systems, insurances, or web applications - system engineering is mostly based on multiple domains - e.g., engineering a car deals with the following domains: embedded software, mechanical, electrical, and safety. The production process to create the system also has influence on the system itself (e.g., the price, or the amount of systems that can be produced). Therefore, model-based production [BKL+18] also belongs to model-based systems engineering. Models, with their cross-cutting nature helps to express relations between different domains in systems engineering.

CBSE (component-based software engineering) is a development paradigm by assembling large software systems from components [Nin97]. One well-known concept to model structural relationships between components in CBSE are C&C models (cf. Subsection 1.1.1). C&C models such as *SysML* IBDs (internal-block diagrams) or *SysML* BDDs (block-definition diagrams) are often used to design the logical layer of embedded software in a systems engineering context.

DSLs (domain-specific languages) are modeling languages formalizing the application structure, behavior, and requirements within a particular domain [Sch06]. DSLs tackle the problem-implementation gap [FR07] - i.e., the conventional gap between the domain problem and the

GPL (general purpose language, e.g., C++, Java, or Swift) based implementation - leading to accidental complexity [CCF$^+$15].

Well-known examples for DSLs, their extensions, and their interaction delivers the world wide web consortium (W3C) as they defined for each problem (webpage structure, layout, graphics, math expressions, or action handling) its own language such as HTML [W3C17b], CSS [W3C17a], SVG [W3C11], MathML [W3C14], JavaScript/ECMAScript [ecm18]; and also extend the HTML language with new keywords for addressing new problems (e.g., Payment API [W3C18b], or HTML Media Capture [W3C18a]).

In contrast to GPLs - mostly programming languages of level 3 - executable DSLs are programming languages of level 4 [HM02, VDKV00] or level 5 (AI-based DSLs [Gri84]), because DSLs provide much higher abstractions due to their tailored problem-specific (and mostly declarative) nature. But not every DSL is executable, e.g., SVG DSL for vector graphics.

DSLs include the following features: concrete syntax, abstract syntax, context conditions (also called static semantics), and (dynamic) semantics (also called meaning) [HR00, HR04]. The **concrete syntax** determines the representation of a DSL. The concrete syntax is the representation how the user of the DSL needs to write down its concrete models. The concrete syntax of a DSL should be as close as possible to existing notations used by domain experts [KKP$^+$09]. The concrete syntax of a tool can be textual or graphical (also includes table-based like Excel) [GKR$^+$07, KKP$^+$09].

DSLs for textual languages mostly use parser generators such as *ANTLR* [Par13], *Yapp* [MMY10], *Rats!* [Gri06], *PEG.js* [Kur16], *Waxeye* [OVM15], or *Mouse* [Red09] to produce parsers for transforming text into a traversable internal data structure.

Graphical tooling such as *MPS* [PSV13], *Gemoc* [BDV$^+$16], or *MetaEdit+* [TR03] are always projectional editors [VS10]. Thus, these tools modify directly the underlying internal structure, avoiding the parsing step, and so they do not need to deal with “token clashes”. The concrete syntax of graphical models are the used graphical elements such as lines, arrows, boxes, stroke width, or color.

The **abstract syntax** of a language describes its essential structure; the abstract syntax does not contain semantically irrelevant words [CBCR15], which make only the concrete syntax better readable. The *ANTLR* parse tree [Par13], the internal structure created by the parser, is not an abstract syntax of a language, because the parse tree contains the complete syntactic sugar of the textual input file. Some tools, e.g., EMF (Eclipse Modeling Framework) [SBMP08], enable to define directly the abstract syntax of a language without using any concrete syntax.

The tree structure after parsing textual input and removing irrelevant concrete sugar is often called abstract syntax tree (AST). This thesis uses the notation of abstract syntax based on Nazari [MSN17]: “the abstract syntax consists of both the AST and the symbol table” (symbol table is introduced in Subsection 1.1.3).

Context conditions, sometimes also called static semantics, are Boolean predicates over the abstract syntax of a DSL [HR00, CBCR15]. Context conditions only constraint the syntax; they do not describe the semantic domain, the meaning of the syntax [HR04]. For example, the concrete syntax `10 + 11` can be interpreted as `21` when applying the semantic domain of natural numbers; or as `01` when applying the Boolean algebra domain with + as *exclusive-or* operator.

Typical context conditions of DSLs are resolving declarations and type checking [Edw00, Car96, Bag10]; e.g., variables must be declared before referenced, a file contains at most one public class in Java, or the type of an assignment's left side must be compatible to the type of the right side. Context conditions can also be used to detect code smells violating conventions, e.g., Java classes should start with a capital letter [Ora99]. A model is well-formed if it fulfills all context conditions [CBCR15, MSN17].

A language's **type system** is mostly part of the essential structure (abstract syntax) and of a language's context conditions. DSLs with a behavior model, e.g., some kind of expression language, mostly define a type system [JSH13]. The type system has inference rules for deriving the type of a composed expression term based on the types of single operators. A type system also has context conditions which check whether an expression based on the (inferred) type information is valid. The type system of a DSL should reflect the problem domain.

For example, a DSL for matrices similar to *MATLAB* can have a matrix-based type system containing matrix dimensions and algebraic matrix properties: One type inference rule is, e.g., how to calculate the matrix dimension after a matrix multiplication. One constraint is, e.g., that matrix addition forces that matrix dimensions of the left and right side are equal.

A type system for an English-like DSL could be based on grammar rules such as singular and plural: An inference rule could be that a list (comma separated, or just with an `and` or `or`) of singular nouns is the same as a plural noun. A constraint could be that after a plural noun no singular verb is allowed (`Evgeny and Michael programs Java` is wrong).

For *xText* [EB10] and its expression framework *Xbase* [EEK+12] exist *XTS* [Voe11], *Xsemantics* [Bet13], and *TS4DSL* [JSH13] as tools for type declaration, type inference, and type checking.

The semantics of a language provides the meaning, in a well-defined and well-understood domain, of each well-formed syntactical element. There are three kinds how to define the semantic of a DSL:

- Denotational semantics: It defines a function mapping of the concrete or abstract syntax to a mathematical domain, e.g., set theory by Scott and Strachey [Ten76, SS76].
- Axiomatic semantics: It defines the semantics and proofs the correctness by using axioms [Hoa69].
- Operational semantics [TP97]: It maps the concrete syntax of a DSL directly to code of a real (or simulated) machine. The weakness is that the machine (simulated, virtual, or real hardware) needs a clear semantic description [Edw00].

EmbeddedMontiArc's semantic is denotational as its syntax is mapped to I/O-EFA (input/output extended finite automata) structures by using the same semantic as *Simulink* models with fixed-step size solvers [RSvW+15]. Later in Chapter 7 this thesis uses denotational semantics to map the meaning of *EmbeddedMontiView* to Boolean mathematical predicates about *EmbeddedMontiArc* models. *OCL/P*'s semantic is operational; its meaning is defined by mapping it to Java code [Rum16, Section 5.3] and the meaning of Java code is defined by its byte code and the Java Virtual Machine [HBL99, Pus98].

1.1.3. *MontiCore*

Main sources: [HR17, Section 1.1, Section 4.2], [MSN17, Subsection 2.2.2, Section 3.8, Section 8.1]

Subsection 1.1.2 shows that models and model-based (systems) engineering are used in many domains, e.g., to speed up development, improve quality, and reduce maintainability costs. A language workbench is a tool to create efficiently new DSLs [Fow10, Gho10, VBD+13, MSN17, Rot17]. With *xText*, *MPS*, *Enso*, *Mas*, *SugarJ*, *Whole Platform*, *MetaEdit+*, *Onion*, *Spoofax*, *Rascal*, and *MontiCore* exist already a number of language workbenches for agile language engineering [PPL14, EVDSV+13]. In contrast to most other existing language workbenches, *MontiCore* is a light-weight, highly customizable, and functional oriented language workbench framework [GKR+08, KRV08, Kra10, KRV10, Völ11, Sch12, Hab16, Rei16, Loo17, MSN17, Rot17, HR17].

MontiCore's main features are [HR17]:

- Modular definition of languages
- Easy definition of large language families via:
  - Independent language development
  - Language extension
  - Language embedding
  - Language aggregation
  - Composition of language tools
- Creation of language specific Eclipse [KRV07] and web editors [KRRvW18, Ron17]
- Assistance for model analysis
- A single source file defines concrete and abstract syntax, as well as parser and internal representation of models

Due to the more than 10-year existence of *MontiCore*, there exists a large grammar repository of many different languages belonging to many different domains, which can be reused to create your own language in a minimum amount of time.

For example, *MontiCore* provides languages for the following domains [HR17]:

- Basic domain: Literals, Lexicals, Numbers, Matrices, Comments, Stereotypes, and Tagging
- *UML*: Class, Object, Activity, and Sequence Diagrams as well as StateCharts and *OCL*
- *SysML*: Units, *MontiArc* (ADL), Automata, Functional nets, *CNNArch*, *MontiMathOpt* (optimization language), and Feature Diagrams as well as *EmbeddedMontiArc* and *EmbeddedMontiView*
- GPLs: Java 7, Ansi-C++, Python, and JavaScript
- Cloud: *MontiSecArc* (security), *MontiClarc* (cloud architecture), *MontiWIZ* (online formular wizard)
- Text-based: Curriculum, Right Restriction for TV movies/series

Important languages for this thesis are class diagrams, *OCL*, and *MontiArc*. Subsection 1.1.1 shortly introduces *MontiArc*.

Class diagrams are part of structural *UML* (Unified Modeling Language) diagrams. For example, class diagrams are used for object oriented modeling or for data modeling as they describe the data structure via attributes of objects and their relations via associations. Another use case for class diagrams is to describe the structure of systems, e.g., as abstraction of Java or C++ code systems, by showing only their classes with their relations (implements/extends and association relation), their attributes, as well as their methods.

start rule

priority

```
grammar PlusMinus {                                                    MC5
  Formula = (Expression ";")+;
  interface Expression;
  PrimaryExpression implements Expression <100> =
    Number | Name;
  AdditiveExpression implements Expression <20> =
    left:Expression op:(["+"] | ["-"]) right:Expression;          }
```

Grammar inheritance

```
grammar Arithmetic extends PlusMinus {                                 MC5
  start Formula;
  MultiplicativeExpression implements Expression <30> =
    left:Expression op:(["*"] | ["/"] | ["%"]) right:Expression;}
```

Grammar slicing

```
grammar SimpleArithmetic extends Arithmetic {                          MC5
  start Formula;
  PrimaryExpression implements Expression <100> =
    Number;                                                              }
```

restrict PrimaryExression, no variable names anymore

Grammar merging

```
component grammar FileContainer {                                      MC5
  File = "file" Name "{" FileContent* "}";
  external FileContent;                                             }
```

```
grammar MathFile extends FileContainer, Arithmetic {                   MC5
  start File;
  FileContent = Expression ";";                                          }
```

Arithmetic's Expression is embedded into FileContent's hole

Figure 1.3.: Grammar Composition in *MontiCore* 5

OCL (Object Constraint Language) is an extension to *UML* models, e.g., class diagrams, to specify precisely detailed aspects of systems. *OCL* is typed, declarative and side-effect free [Cab12]. This thesis uses the *OCL/P* textual notation of Rumpe [Rum16] to specify *OCL* constraints. Chapter 6 explains *OCL* in detail. The paper [BRvW16] illustrates on five constraints the syntactic difference between OMG *OCL* 2.4 and *OCL/P*.

The rest of this subsection explains the concepts how to create a large language family similar to *EmbeddedMontiArc*, the C&C language family introduced in this thesis and described in the next chapters. It shows how eloquent *MontiCore* handles language inheritance, slicing, and merging via grammar files. Section 4.6 describes how the resolving mechanism in *MontiCore* helps to find declared symbols across different DSLs enabling language aggregation and embedding.

Information about the technical architecture of *MontiCore*, or how the here presented concepts of *MontiCore* are implemented, or example code snippets explaining how to use these *MontiCore* features to develop your own DSL are explained in detail in the official "*MontiCore* 5 Language Workbench" book [HR17] and in the thesis "*MontiCore*: Efficient Development of Composed Modeling Language Essentials" [MSN17].

Similar to other DSL tools, e.g., Melange [DCB+15], *MontiCore* also supports language inheritance, slicing, and merging. In contrast to Melange, which specifies these language relations via model types in the abstract syntax, *MontiCore* uses a grammar file to enable inheritance, slicing, and merging not only on the abstract syntax, but also on the concrete one.

The top listing in Figure 1.3 shows the *MontiCore* grammar for a basic expression language. This listing shows that the *MontiCore* grammar format extends EBNF (Extended Backus-Naur form) to specify productions for the lexer and the parser. As shown in lines 2, 4, 6, 10, 14, 17, and 21 most productions are assignments and consist of a left-hand side (LHS) part and a right-hand side (RHS) part.

The LHS of a rule defines a nonterminal. The RHS defines the production's body of the LHS nonterminal. The RHS may consists of any combination of lexicals, terminals, or nonterminals. The elements on the RHS may be annotated with cardinalities describing how often an element appears: `?` for 0 to 1 times; `+` for 1 to infinite times; and `*` for zero to infinite times. The pipe symbol `|` is used to describe alternatives.

In contrast to *ANTLR*, *MontiCore*'s extended grammar format borrows several concepts from object-oriented languages such as Java:

(i) definition of production interfaces,
(ii) definition of abstract productions,
(iii) implementation of production interfaces, and
(iv) extension of production interfaces.

Similar to object oriented languages where interfaces can be (a) marker interfaces without any further function signatures, or (b) "normal" interfaces defining a contract that all classes implementing it should follow [Die17]; *MontiCore* also supports these two kinds of interfaces for productions: (a) the first kind has no RHS meaning it does neither define any concrete nor abstract syntax; (b) the second kind has a RHS defining the signature (abstract syntax) for all nonterminals implementing it [HR17].

Line 3 defines the `Expression` interface. Interfaces enable to decouple the definition of languages as they create open extension points [HR17]. These open extension points enable that the `Expression` interface may not only be implemented in its own grammar `PlusMinus`, but also in the two other grammar files `Arithmetic` and `SimpleArithmetic`. `Formula` in line 2 uses the interface nonterminal `Expression` in its RHS; and thus, it includes all nonterminals implementing `Expression` according to their priority, i.e., first `PrimaryExpression` and then `AdditiveExpression`. If no priority is explicitly defined, the occurrence order of the definitions of the implementing nonterminals is used.

If a grammar contains no start rule as shown in line 9, then the first production rule is a grammar's start rule. Therefore, valid input files regarding to the `PlusMinus` grammar are:

- `-17;`
- `x1;`
- `-17 + B;`
- `3; x1 + 5 - B;`

The `Arithmetic` grammar extends the `PlusMinus` grammar, and, thus, it has access to all `PlusMinus`'s nonterminals. Additionally, the `Arithmetic` grammar adds a new production rule implementing the `Expression` interface. The `Arithmetic`'s inherited `Formula` production using the `PlusMinus' Expression` interface includes all nonterminals of the `Arithmetic` and all nontermals of the `PlusMinus` grammars, which implement the `Expression` interface. Since the priority of the `MultiplicativeExpression` is higher than the one of the `AdditiveExpression`, but lower than the priority of the `PrimaryExpression`; the `Formula` rule includes first `PrimaryExpression`, then `MultiplicativeExpression`, and last `AdditiveExpression`.

A valid input file according to the extended `Arithmetic`, but an invalid input for the `PlusMinus` is:

- `x1 + 5 * 3;`

MontiCore also supports slicing grammars (removing words belonging to a language) by extending a language and overwrite an existing nonterminal to restrict some allowed words. An example is shown in lines 14 and 15 where the inherited `PrimaryExpression` nonterminal is overwritten to accept only numbers and no (variable) names anymore.

Grammar merging, also known as grammar embedding, combines the words defined by both grammars in a controlled way. The `MathFile` grammar embeds the `Arithmetic`'s `Expressions` interface production into the `FileContainer`'s `File` production by binding the external production `Content` to `Expression` (cf. l. 21).

The component grammar `FileContainer` (cf l. 16) is an incomplete grammar, and thus, cannot be used standalone. The definition of the external production `FileContent` (cf. l. 18) creates a slot defining a variation point: So multiple grammars can extend this `FileContainer` grammar and bind `FileContent` differently (e.g., expressions, URL links, automata, etc.).

The later in this thesis presented language *EmbeddedMontiArcBehavior*, extending the pure structural C&C language *EmbeddedMontiArc* with behavior, creates a variation point for behavior to facilitate different behavior implementations: *EmbeddedMontiArcMath* has a *MATLAB*-like behavior implementation, and *EmbeddedMontiArcDeepLearning* defines the behavior of atomic components via a CNN (Convolutional Neural Network) as used in deep learning applications.

Grammar extension and the here mentioned interface mechanisms enable to engineer a large language based on smaller ones. This is one of the most valuable features of *MontiCore*. For example, *ANTLR* does not contain this feature, and so the concrete and abstract syntax of a language must be completely defined in one very large g4-file; also reuse of grammar rules (e.g., names, numbers, or expressions) can only be done via copy and paste in *ANTLR*.

The previous paragraphs shows that *MontiCore* enables to combine concrete syntax as well as ASTs of different languages in an efficient way. But as mentioned in Subsection 1.1.2 symbols belong to the abstract syntax, too. Symbols are, e.g., created when defining variables, and symbols are, e.g., used when resolving previously defined variable names. All symbols of a language family are stored in a symbol table.

According to Nazari [MSN17] is a symbol table a graph-based data structure containing of scopes, where each scope is a local repository for symbols, to fulfil the following tasks:

(i) mapping names to symbols representing essential model information;
(ii) organizing and finding types, declarations, implementation details, etc. of model elements in an efficient way; and
(iii) representing the essence of a language, i.e., of its models by including model interfaces constituted by the language interface [MSN17, CBCR15].

For C&C models, described in Subsection 1.1.1, the essential information is: in- and output ports (i.e., the interface) of a component, subcomponents a component is decomposed of, dataflow between ports, as well as types of ports and components. For example, the symbol table supports finding a component by its name, and then navigate efficiently through the essential data structure.

MontiCore's ability to combine grammars and to exchange symbols between languages enables the development of modular language components and tools which can be completely reused to engineer large language families and powerful modeling tools.

1.2. Requirements on PhD Thesis

To integrate C&C architectural modeling and its C&C views verification techniques into industrial development processes of embedded and cyber-physical systems, this section summarizes the requirements on this PhD thesis in order to improve the existing C&C (views) languages *MontiArc* and *MontiArcView*.

Most results of this thesis are founded by the German Israeli Foundation (GIF Grant No: I-1235-407.6/2014) as a joint work together with Tel-Aviv University. This section contains text fragments of the corresponding proposal [MR13, Section 3].

1.2.1. Enhancing the C&C Views Language

The basic C&C views language based on Maoz et. al. [MRR13] should be enhanced by integrating extensions of *AADL* [Soc06], *SysML* [FMS11, OMG15], and specific application domains such as automotive [SG07] or robotics [BKH+13].

The following extensions of C&C Views should be supported:

(R1) Component instantiation and component/connector types
Since existing architecture description languages already have an instantiation mechanism for component reuse (including types, subtypes and their well-formedness rules), C&C views should also introduce such an instantiation and typing mechanism. Furthermore, advanced language features such as parameter instantiation, or generic component types should be inspected for the C&C views concept.

(R2) An associate predicate language
While C&C views are intuitive and expressive enough to specify abstraction of C&C models - e.g., by omitting complete hierarchy or ports in connections - not all structural properties of C&C models can be expressed by C&C views. Thus, an *OCL*-like constraint language with quantification support over components, connectors, and ports and related operations to its C&C views should be created. The language should be designed that its answer whether the constraint is satisfied or not is decidable; but the language should be able to constraint “the number of ports of a component” or “specify the completeness of a given component hierarchy” in a short compact form.

(R3) Domain-specific language adaptations and extensions
For future application of C&C views in industry, the C&C views language must support domain-specific extensions to become more friendly to engineers. Additionally, C&C views language and its corresponding C&C model language should support a way to add domain or application-specific properties (e.g., extra-functional properties being used in automotive industry).

1.2.2. Advancing C&C Views Analyses

(R4) As the C&C views language is enriched by more and more features, the C&C model language and the formal satisfaction relation between C&C views and C&C models must be updated. Also introducing component types makes the verification problem much harder. In this case also parts of the already existing algorithms might be updated in order to have a good scalability up-to medium-large industry models.

(R5) Also a Boolean answer to the verification problem is mostly not useful enough. Therefore, esp. for negative verification results, a meaningful witness should be generated. The witness should explain the reason (or reasons) why a C&C model does (not) satisfy its C&C design view.

1.2.3. Integrating C&C Views in the Development Process and Environment

Since existing development processes and their tools are all about hierarchical decomposition of systems to sub-systems, and thus, these tools represent only a single hierarchy/sub-system to the engineer at a time; these tools are not suited for the crosscutting nature of C&C views and their verification. Therefore, existing processes and tools must be adopted to take advantage of C&C views together with their abstraction mechanisms and analysis methods.

This requirement consists of the two sub-requirements:

(R6) New design and development processes:
Existing design and development processes based on C&C models should be investigated. Additionally, these mostly hierarchical based processes should be adapted to use C&C views; and usage scenarios for design, development, or maintenance where C&C views support the adapted process should be worked out so that the benefits of integrating C&C views with their verification into existing processes becomes obvious.

(R7) New modes of interaction:
Existing tools represent only one hierarchy of a system or its sub-systems. This means it is not possible to see two components and their interaction between them, if these two components are not on the same hierarchy level. Therefore, a crosscutting visualization of a view as well as a seamless navigation between C&C model and its views in both directions should be developed.

1.2.4. Evaluation

(R8) The new enhanced C&C view language should be evaluated in an industrial context. The evaluated setting should show the benefits and weaknesses of the C&C view approach integrated into existing processes. The evaluation together with an industrial partner should show how efficient this approach is in the daily-life of engineers. Thereby, we will compare the results of C&C view verification against its manual verification, in terms of speed, needed human resources, and correctness.

1.2.5. Further Remarks

This thesis focuses on the C&C view and the C&C model language extension, support of extra-functional properties, *OCL*-like specification language for C&C models, the extended formal satisfaction relation between C&C views and C&C models, as well as how the C&C verification can be integrated in existing development processes by adopting the current modeling methodologies. Additionally, this thesis contains results of an industrial case study together with Daimler AG about C&C views and their verification.

This thesis neither contains C&C views synthesis nor C&C views refinement.

1.3. Objective and Main Results

The following research question summarizes the main research goal of this thesis:

How can domain specific languages support the software systems engineering process for cyber-physical systems by defining structural design decisions and extra-functional properties in an efficient, agile, and intuitive, but also unique and formal way so that industrial-size component and connector models can be validated against them?

This thesis aims to improve software systems engineering by providing a continuous model-based approach from specifying architectural design decisions over defining extra-functional properties up to developing a complete logical architecture by utilizing domain specific languages (DSLs). In particular, we created a language family that consists of DSLs:

(1) to model design decisions via crosscutting structural relations between components;
(2) to specify new kinds, structures and semantics of extra-functional properties, and to define values according to its structure; as well as
(3) to model the complete logical and functional architecture.

Since each domain specific language addresses only one modeling concern in the systems engineering process, the concrete syntax of each textual language focuses on the notation of domain experts for this specific concern. This facilitates domain experts to model in an **efficient** and **intuitive** way. Additionally, the magnitude on different DSLs plus the modular nature of each DSL - building on Java's class, package and import concept - enables to separate information about one model into different artifacts. This provides a more flexible development process as engineers can work on their subset of files, and thus, all features - having an overlapping developing time - must not be integrated at the same time. It is even possible to revert only a set of files representing a specific concern or feature. Therefore, the more flexible process supports different, and thus shorter, development cycles for concerns (e.g., design, functional components, safety, and security) making it more appropriate for **agile** development.

The unique meaning and formal background of these domain specific languages enable formal verification between the logical architecture and its structural design decisions. Additionally, the formal background of these DSLs empowers formal validation of architectures, enriched with extra-functional property values, according to the specified semantics of their extra-functional property kinds. These validations support an incremental and **agile** engineering process, as their automatically generated result witnesses unveil instantly inconsistencies between evolving designs, extra-functional property values, or the frequently modified functional architecture.

The main contributions of this thesis are:

- A number of DSLs with SI unit (Systeme international d'unites) support which has an **intuitive** concrete syntax. This DSL enables engineers to define complex numbers, and numbers with units (and their automatic conversions) in an **intuitive** way as they are written down in daily life, e.g., in textual requirements. In contrast to other DSLs, where units must be encoded between special characters, e.g., `0.8 [m/s]` in Sprat Ecosystem DSL [JH17], this DSL is able to parse numbers with units directly such as `0.8 m/s`. This language is the basis for many other DSL adoptions and extensions addressing (R3).
- *EmbeddedMontiArc*, a DSL for C&C models, with focus on cyber-physical systems. It embeds SI numbers for unit support. *EmbeddedMontiArc* supports generics, component

libraries, configuration parameters, as well as arrays of ports and component instantiations to facilitate modular and reusable functional architectures. The enhancement of the previously used C&C modeling language [Rin14] is necessary so that models satisfying C&C views, being extended to fulfil (R1), do exist.

- Formalization of the abstract syntax of *EmbeddedMontiArc* via class diagrams. This is a necessary prerequisite to define the formal C&C view verification relation between *EmbeddedMontiArc* and *EmbeddedMontiView*; addressing parts of (R4).
- Mathematical framework to specify the consistency constraints of extra-functional properties for C&C models. This is related to (R4).
- Tag schema and a tag model DSL to create new extra-functional property kinds and, later-on, to define extra-functional values for these kinds. This solves completely (R3).
- Extension of the *UML/P OCL* language to specify further constraints in form of context conditions for *EmbeddedMontiArc*. This addresses (R2). The *OCL* language supports defining constraints for extra-functional properties in an **efficient** way, i.e., in a few lines of code.
- Aggregation of tag schema, tag model, and *OCL* DSL to specify context conditions for *EmbeddedMontiArc* regarding to the mathematical framework for extra-functional properties. Also an extension of the verification algorithm, as required in (R4), validates the defined consistency rules for extra-functional properties, and it also generates (non-) consistency witnesses, as wished in (R5).
- Extension of the C&C view language with component types and arrays, abstract effectors, as well as further port abstractions, and abstractions regarding to port arrays. For example, new port abstractions are unit kinds as abstraction between no port type and concrete port type. These extensions address (R1) and (R3).
- Formalization of the *EmbeddedMontiView* DSL, so that the new concrete and abstract syntax has a concrete mathematical meaning. This also belongs to (R1) and (R3).
- Definition of a satisfaction relation between *EmbeddedMontiArc* and *EmbeddedMontiView*; this satisfaction relation extends the existing satisfaction relation between C&C views and C&C models of Maoz, Rumpe and Ringert [MRR13, MRR14, Rin14]. This new extended satisfaction relation fulfills (R4).
- Adaption of satisfaction and non-satisfaction witnesses according to the new satisfaction relation. This accomplishes (R5).
- Integration of C&C view verification with its witnesses into existing methodologies. The tracing witness is added as new additional witness kind for a positive satisfaction relation; tackling (R6). Support of other kinds of user interaction, e.g., by coloring all model elements satisfying a specific view element and adding links between them; addressing (R7).
- An industrial case study together with Daimler AG, solving (R8), evaluated use-cases where C&C views with their corresponding verification can be integrated in an existing development process. Additionally, the case study assess how the development process and our tooling must be adapted; also addressing (R6) and (R7). The process of validating component and connector models against C&C views or verifying extra-functional property consistency is very fast (mostly far below 1 minute). This very fast execution of our implemented tool, that is based on the here presented algorithms, supports **agile** and incremental development.

1.4. Thesis Organization

This thesis is organized in the following way:

Chapter 1 gives an overview of this thesis' context, motivation, requirements, research questions, and achieved research results.

Chapter 2 presents two current development methodologies in automotive industry. This chapter shows how the research findings of this thesis can help to improve these model-based development approaches.

Chapter 3 summarizes the results of our related work study on existing C&C modeling languages. Additionally, this chapter introduces the *EmbeddedMontiArc* language borrowing concepts from established C&C languages.

Chapter 4 presents the abstract syntax of *EmbeddedMontiArc* via class diagrams. This chapter also elucidates the component and connector instance structure representing the statical architecture instantiated by an *EmbeddedMontiArc* model.

Chapter 5 shows the tagging mechanism to enrich C&C models with extra-functional properties in a non-invasive way. This chapter introduces the two DSLs enabling the tagging mechanism: (a) tag schema to define new extra-functional property kinds, and (b) tag model to add extra-functional property values to C&C models.

Chapter 6 gives an overview of the mathematical framework to express consistency constraints of C&C models enriched with extra-functional properties in *OCL*. Additionally, this chapter shows how to specify context conditions via the developed *OCL* framework.

Chapter 7 presents *EmbeddedMontiView*- the C&C view language to specify design decisions for *EmbeddedMontiArc*. The concrete and abstract syntax of the *EmbeddedMontiView* is explained on many concrete listings and use cases. This chapter also defines when a large *EmbeddedMontiArc* architecture satisfies an *EmbeddedMontiView* design specification.

Chapter 8 presents the industrial case study together with Daimler AG. It explains the study design, the results of the preliminary study focusing on finding suitable models, and the results of the main study answering questions about feasibility of C&C views, technical applicability to use existing models, and how helpful the generated witnesses are.

Chapter 9 summarizes the main results and it concludes this thesis.

1.5. Publications

The following list of peer-reviewed research publications, which Michael von Wenckstern authored or co-authored, contribute to the contents of this thesis:

[DGH+19] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, A. Wortmann: SMArDT modeling for automotive software testing.
In: Software: Practice and Experience, 2018.

[KRSvW18a] E. Kusmenko, B. Rumpe, S. Schneiders, M. von Wenckstern:
Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language *EmbeddedMontiArc*.

In: Conference on Model Driven Engineering Languages and Systems (MODELS'18), pg. 447-457, Copenhagen, ACM, Oct. 2018.

[KRRvW18] E. Kusmenko, J. Ronck, B. Rumpe, M. von Wenckstern:
EmbeddedMontiArc: Textual Modeling Alternative to *Simulink*.
In: Proceedings of MODELS 2018. Workshop EXE, Copenhagen, Oct. 2018.

[KRSvW18b] E. Kusmenko, B. Rumpe, I. Strepkov, M. von Wenckstern:
Teaching Playground for C&C Language *EmbeddedMontiArc*.
In: Proceedings of MODELS 2018. Workshop ModComp, Copenhagen, Oct. 2018.

[KKRvW18] S. Kriebel, E. Kusmenko, B. Rumpe, M. von Wenckstern:
Finding Inconsistencies in Design Models and Requirements by Applying the SMARDT Process.
In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18). Univ. Hamburg, Apr. 2018.

[BMR+18] V. Bertram, S. Maoz, J. O. Ringert, B. Rumpe, M. von Wenckstern:
Component and Connector Views in Practice: An Experience Report (extended abstract).
In: Software Engineering und Software Management 2018 (SE'18), pg. 97-99, Ulm, Germany, LNI P-279. Bonner Köllen Verlag, Mar. 2018.

[HKK+18] S. Hillemacher, S. Kriebel, E. Kusmenko, M. Lorang, B. Rumpe, A. Sema, G. Strobl, M. von Wenckstern:
Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology.
In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pg. 163-178. SciTePress, Jan. 2018.

[MMR+17] S. Maoz, F. Mehlan, J. O. Ringert, B. Rumpe, and M. von Wenckstern:
OCL Framework to Verify Extra-Functional Properties in Component and Connector Models.
In: Proceedings of MODELS 2017. Workshop ModComp, Austin, CEUR 2019, Sept. 2017.

[BMR+17a] V. Bertram, S. Maoz, J. O. Ringert, B. Rumpe, M. von Wenckstern:
Component and Connector Views in Practice: An Experience Report.
In: Conference on Model Driven Engineering Languages and Systems (MODELS'17), pg. 167-177, Austin, IEEE, Sept. 2017.

[KRRvW17] E. Kusmenko, A. Roth, B. Rumpe, M. von Wenckstern:
Modeling Architectures of Cyber Physical Systems.
In: European Conference on Modelling Foundations and Applications (ECMFA'17), Marburg, pg. 34-50. LNCS 10376, Springer, July 2017.

[MRRvW16] S. Maoz, J. O. Ringert, B. Rumpe, M. von Wenckstern:
Consistent Extra-Functional Properties Tagging for Component and Connector Models.
In: Workshop on Model-Driven Engineering for Component-Based Software Systems (Mod-

Comp'16), pg. 19-24, Saint Malo, France, Vol. 1723, CEUR Workshop Proceedings, October 2016.

[BMP+16] V. Bertram, P. Manhart, D. Plotnikov, B. Rumpe, C. Schulze, M. von Wenckstern: Infrastructure to Use *OCL* for Runtime Structural Compatibility Checks of *Simulink* Models. In: Modellierung 2016 Conference, LNI P-254, pp. 109-116. Bonner Köllen Verlag, March 2016.

[RSvW+15] B. Rumpe, C. Schulze, M. von Wenckstern, J. O. Ringert, P. Manhart: Behavioral Compatibility of *Simulink* Models for Product Line Maintenance and Evolution. In: Conference on Software Product Lines (SPLC'15), p. 141-150, ACM, 2015.

The contents of these above papers included in this thesis may have been adapted, reorganized, or extended with respect to the published version. For better readability, contents of these above papers are not put in quotation marks when reusing them in this thesis; and also contents of these above papers - when used in this thesis - are not always in detail cited with the reference to the published paper. The following paragraphs explain the papers' contributions, and thus the reused contents, for each chapter.

Chapter 2 is based on the publications about the development process at Daimler AG [BMR+17a, BMR+18] and SMARDT methodology [DGH+19, KKRvW18, HKK+18]. Section 2.1 uses mostly contents of the case-study paper [BMR+17a, Section III] and its supplementary material on the website "Example Process with Focus on Challenges Traceability and Evolution" [BMR+17b]. Section 2.2 describes the SMARDT approach, mostly based on [HKK+18, Section 3] and [KKRvW18, Section 3], and how C&C views can be integrated (mostly based on [KKRvW18, Section 4] and [DGH+19, Subsection 3.1]) in it. Additionally, Subsection 2.2.2 reuses some arguments explaining why the textual *EmbeddedMontiArc* family is better suited then existing modeling tools from the paper presenting the tooling of *EmbeddedMontiArc* [KRRvW18, Section 2].

Chapter 3 is based on the publications about *EmbeddedMontiArc* [KRRvW17] and model examples using *EmbeddedMontiArc* [KRSvW18a, HKK+18]. Section 3.1 and Section 3.2 reuse a lot of contents published in the *EmbeddedMontiArc* language family (in the paper called MontiCAR) paper [KRRvW17] such as industrial-derived requirements, and related work of *EmbeddedMontiArc* to existing C&C modeling languages. Section 3.6 reuses the spectral cluster example [KRSvW18a, Section 2] and some presented models are inspired by the PID controller example [HKK+18, Section 5]. Some *EmbeddedMontiArc* car examples are inspired by *EmbeddedMontiArc* online playground [KRSvW18b].

Chapter 4 reuses nearly no contents of publications. It reuses the flip flop example [RSvW+15, Figure 2] and is inspired by formal definitions [BMR+17a, Section II], which are mostly published in the extra file "Component and Connector Views Definition" being available on the supplementary material website [BMR+17b]. The port type system uses ideas of the published abstract syntax for units [BMP+16, Section3].

Chapter 5 is mostly based on the publication of the two tagging languages and the corresponding tagging mechanism [MRRvW16] for extra-functional properties. Especially, Section 5.5 mostly reuses parts of tag schema and tag model language [MRRvW16, Section IV], and Subsection 5.5.5 reuses the consistency rules between tag model and tag schema [MRRvW16, Section V]. Section 5.4 reuses the turbine controller example [MRRvW16, Section II].

Chapter 6 is mostly based on the publication describing the semantics of extra-functional properties via *OCL* constraints [MMR+17]; especially Section 6.2 and Section 6.3 reuse parts of the already published *OCL* constraints [MMR+17, Section V] based on the mathematical notation of consistency rules [MRRvW16, Section V], [MMR+17, Section III]. The consistency witnesses in Section 6.3 are inspired by the witnesses [MMR+17, Section V] for extra-functional properties.

Chapter 7 reuses nearly no contents of the above publications. It is only encouraged by the abstract effector concept presented in [BMR+17a, Section II] and the corresponding material on the website "Component and Connector Views Definition" [BMR+17b]. Section 7.5 reuses only parts of the examples for a simple car [BMR+17a, Section II] and a parking assistant [KRRvW17, Section II], as well as C&C views and witnesses from the external material website [BMR+17b].

The structure of Chapter 8 is oriented on the case-study paper with Daimler AG [BMR+17a]; the contents of the preliminary and main study in Section 8.2, Section 8.3, and Section 8.4 are mostly the same as in the published Models conference paper [BMR+17a].

Chapter 2.

Underlying Development Methodology

This chapter presents development processes for systems engineering of embedded and cyber-physical systems. Section 2.1 is about the model-based systems engineering process at Daimler AG. Subsection 2.1.1 introduces the current model-based development at the MBC department at Daimler AG[1]. Subsection 2.1.2 shows how to integrate component and connector views and their verification into the previously presented development process at Daimler AG. Furthermore, this subsection explains how C&C views and consistency checks for extra-functional properties improve the work of engineers.

Section 2.2 elucidates the SMARDT systems engineering process developed by BMW Group [HKK+18, KMS+18, DGH+19, KKRvW18, PPS+03, PSAK04, PPW+05]. SMARDT is an abbreviation for Specification Methodology Applicable to Requirements, Design, and Testing; the original German short-form SMArDT is related to the term "Spezifikations-Methode für Anforderung, Design und Test" [HKK+18]. This section also explains how the SMARDT process may benefit from C&C views and its verification.

Section 2.3 compares a small set of tools and methodologies which are kind of similar to the already elucidated development processes at Daimler AG and BMW Group.

2.1. Systems Engineering Process at Daimler AG

This section presents results of the case study with Daimler AG (cf. Chapter 8). The case study contained of several interviews with an employee at Daimler AG [BMR+17a]. One question of this case study was about the current model-based and component-based development process. Another question was about the challenges of this process and where engineers spend a lot of time due to these challenges, and what would help these engineers. In this case study we looked at the process for creating an exterior light system and an advanced driver assistant system. As the case study was about model- and component-based software engineering, we did not inspect other processes that are not satisfying these two software engineering paradigms.

2.1.1. Current Development Process at Daimler AG

ISO 26262 (Road vehicles -Functional safety) is the international standard for functional safety of electrical and/or electronic systems in production and/or automotive industry [Int11]. The

[1] Daimler AG is a large company with 289.321 employees (December 31st, 2017 [Dai18a]). Therefore, insights gained in the industrial case-study [BMR+17a] with the MBC department at Daimler AG may not be representative for other departments at Daimler AG.

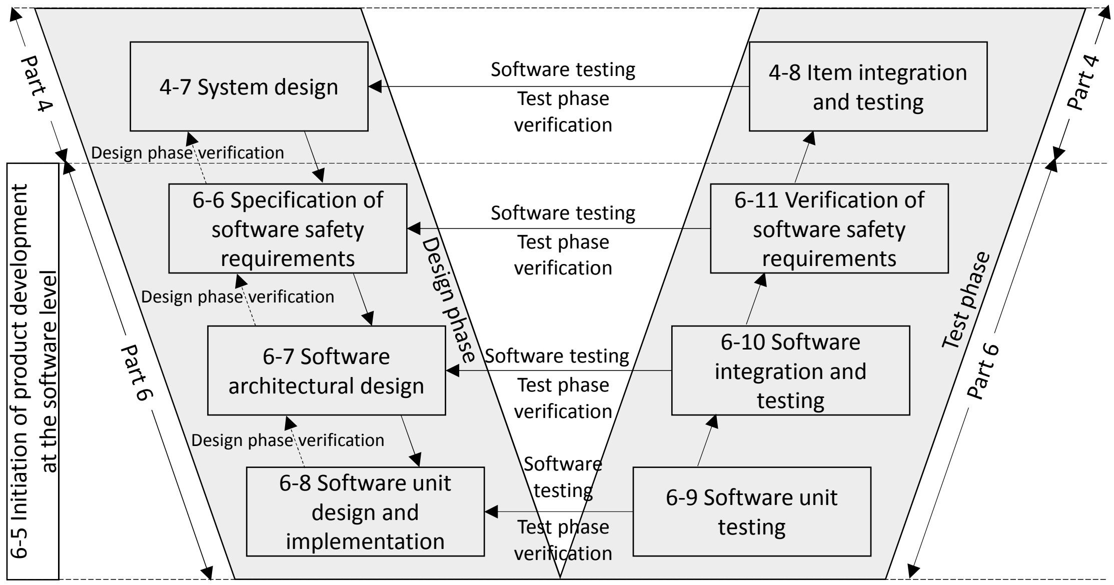


Figure 2.1.: ISO 26262 V-Model (copied from Mentor Graphics).

development process in automotive industry should satisfy the ISO 26262 norm as much as possible, as the competitiveness of a company is measured by the capability of conducting to this standard [JCJ$^+$11]. Therefore, most German automotive industries develop their software for embedded systems based on the ISO 26262 V-Model process as shown in Figure 2.1.

The design of a system is mostly described as textual requirements with links to each other; one famous requirement management tool is *IBM Rational DOORS*. Later extra-functional requirements for safety of a system's design are identified; examples are functional safety, technical safety, system safety, and hardware failures. These extra-functional and stakeholder requirements are integrated into existing requirements of a system's design [IBM13].

The design of a software architecture is mostly modeled in *SysML* block diagram definitions [ECSG09]. Common *SysML* tools in industry are *Enterprise Architect* [RSRB06], *ArchiMate* [Yam15], *Metropolis* [BWH$^+$03], *Cameo Systems Modeler* [HDP14], and *PTC Integrity Modeler* [SHC17]. The requirements are modeled separately in these tools and are linked to the corresponding modeling elements, so that traceability is always given [PMPdK15].

After the design (which defines the interfaces of software components and their interaction with its environment) is modeled in *SysML*, engineers at Daimler AG create manually an executable model in *Simulink* regarding to the previously defined design decisions. To have the traceability between requirements, *SysML* design models, and *Simulink* implementation models, engineers at Daimler AG add to every subsystem in *SysML* and in *Simulink* an information block containing a link to the requirement specification in *IBM Rational DOORS* [BMR$^+$17a]. Adding and maintaining these links manually is time consuming and error prone.

This development process has the following disadvantages:

- The check between the informal *SysML* architecture design against the *Simulink* model is done manually.

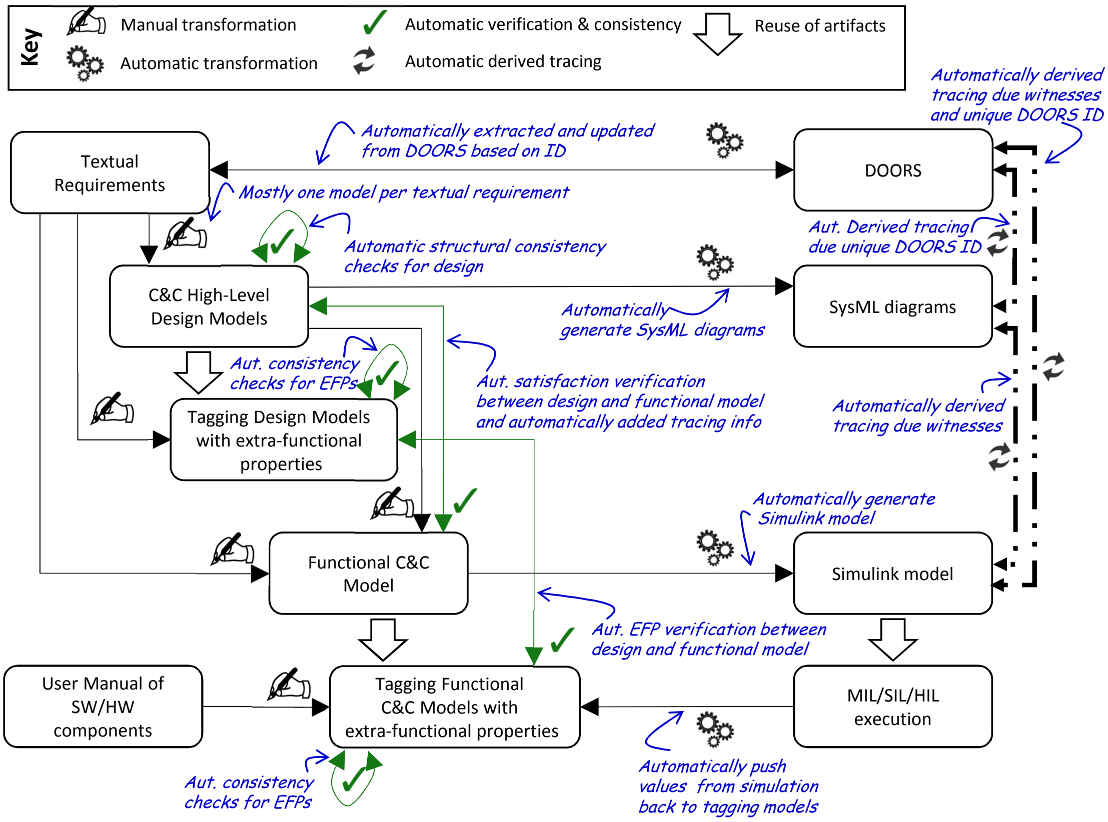


Figure 2.2.: Modified Development Process, compatible to V-Model (only left side of V-Model is shown here)

- The requirement links must be created manually for architectural design model and for the *Simulink* model.
- It exists no automatic check to find outdated *Simulink* subsystems after updating *SysML* design models (e.g., due to model evolution).
- If *Simulink* models are refactored (e.g., subsystem is split into several ones), it may occur that the *SysML* design model is not updated; and then the architecture model becomes obsolete.
- Early inconsistencies in the *SysML* software architecture design, created by different persons or even different teams in large companies, must be detected manually.

2.1.2. Improving the Development Process at Daimler AG

Main sources: [BMR+17a, Section III], "Example Process with Focus on Challenges Traceability and Evolution"[BMR+17b]

To mitigate most of these above mentioned disadvantages, this subsection presents a slightly modified development process and verification tools, as shown in Figure 2.2. The advantage of this new process is that it is completely compatible to existing tools (cf. right side of Figure 2.2). The general workflow of this new process including existing tools is:

1. *IBM Rational DOORS* requirements are automatically extracted to a set of textual requirements.
2. Engineers create manually for each *IBM Rational DOORS* requirement a C&C high-level design model.
3. These C&C high-level design models can be automatically transferred to graphical *SysML* diagrams.
4. The linking between *IBM Rational DOORS* requirement IDs and C&C high-level design models enables to automatically derive tracing information between *IBM Rational DOORS* and *SysML* diagrams.
5. C&C views synthesis algorithms check automatically all defined C&C high-level design level models against structural inconsistencies.
6. Engineers add manually extra-functional properties to the C&C high-level design model based on the textual requirements.
7. The *OCL* (Object Constraint Language) framework checks automatically the consistence of the added extra-functional requirements of the high-level design.
8. Engineers create manually the functional C&C model based on textual requirements and the C&C high-level design models.
9. C&C views verification automatically checks whether the functional C&C model satisfies all C&C high-level design models.
10. In a next step, this functional C&C model can be automatically transformed to a *Simulink* model.
11. The tracing witness of C&C views verification enables to automatically derive tracing information between *SysML* diagrams and the *Simulink* model as well as tracing information between *IBM Rational DOORS* and the *Simulink* model[2].
12. The *Simulink* model is executed. Measured runtime information (e.g., timing, or memory usage) can be used to automatically enrich the C&C model with these extra-functional properties.
13. Engineers enrich manually the C&C model with extra-functional properties based on user-manuals of software or hardware components. Typical information in user-manuals among others are price, latency, working temperature, ASIL (Automotive Safety Integrity Level), and energy usage.
14. The *OCL* framework checks automatically the consistence of the extra-functional properties added to the functional C&C model.
15. The *OCL* framework in combination with C&C views verification validates automatically whether all extra-functional properties in the functional C&C model satisfy all extra-functional requirements defined in all C&C high-level design models.

Even though the new toolchain is larger, there are less manual steps needed due to the higher automation of the steps in this new toolchain. Creating *SysML* diagrams based on textual requirements needs one manual step in the existing approach: *IBM Rational DOORS* $\rightarrow$[3] *SysML*

[2] due to existing tracing information between *SysML* and *IBM Rational DOORS*

[3] $\Rightarrow$: automatic transformation; $\rightarrow$: manual transformation

diagrams. The improved toolchain also needs only one manual step to translate textual requirements to C&C High-Level Design Models as shown in Figure 2.2: *IBM Rational DOORS* $\Rightarrow$ Textual Requirements $\rightarrow$ C&C High-Level Design Models $\Rightarrow$ *SysML* diagrams. The same holds to create *Simulink* models based on *IBM Rational DOORS* requirements and *SysML* diagrams, where the additional manual step in the existing approach is to create *Simulink* models manually, whereas in the new toolchain the functional C&C models are created manually: *IBM Rational DOORS* $\rightarrow$ *SysML* diagrams $\rightarrow$ *Simulink* model $\equiv$ *IBM Rational DOORS* $\Rightarrow$ Textual Requirements $\rightarrow$ C&C High-Level Design Models $\rightarrow$ Functional C&C Model $\Rightarrow$ *Simulink* model.

In the existing approach the tracing between *IBM Rational DOORS* and *SysML* diagrams, between *IBM Rational DOORS* and *Simulink* model, as well as between *SysML* diagrams and *Simulink* model is done manually. In contrast, the new toolchain does the tracing between C&C high-level design models and functional C&C model automatically. Thus, only the tracing between textual requirements and C&C high-level design models is done implicitly manually as each C&C view belongs to one requirement. Based on this implicit relation between textual requirements and C&C high-level design models as well as the automatically generated tracing between C&C high-level design models and functional C&C model, the tracing for textual requirements and functional C&C model can also be done automatically. The two automatic transformations enable to automatically derive the tracing between *IBM Rational DOORS* requirements and the *Simulink* model. This means three manual tracing relations in the old approach are equivalent to only one manual tracing relation in the new toolchain. **Thus, the new toolchain saves a lot of work, especially in agile systems engineering, and it prevents manual tracing errors.**

Furthermore, **the new toolchain adds** due to its unique semantics **many additional automatic verifications** to ensure better model quality and **to prevent modeling errors as early as possible**: step 5, step 7, step 9, step 14, and step 15.

The rest of this subsection explains some of the steps of this new toolchain and the underlying new approach in more detail and it also elucidates what parts of this thesis addresses which steps.

The C&C high-level design contains out of several stand-alone textual C&C view descriptions, which can be merged [MRR13] to one large design model and/or graphically displayed. The advantage of splitting up the design decisions into several textual files (similar as programming languages do it), is the ability to version these files separately. Commercial *SysML* tools such as *PTC Integrity Modeler* use a database approach, which supports to version only the entire (design) model including all *SysML* elements used by different development teams. In *PTC Integrity Modeler* different teams work in one database model, as otherwise (tracing) links between elements - created in different layers or by different teams - are not possible. In contrast to the database linking approach, the here presented C&C view design language uses readable full qualified names (no generated encrypted IDs) to establish the linking process (cf. Section 4.6, [MSN17], and [HR17]).

The synthesis algorithm for C&C views enables to check the C&C high-level design against inconsistencies [KRRvW18]. If this algorithm generates a C&C model based on the specified C&C views, then the design is consistent; otherwise the specified design is inconsistent. For inconsistent designs, the synthesis algorithm generates user-friendly error messages, which include a natural text of the problem description, and a minimal C&C witness containing the involved components causing the conflict. Since these checks are completely automatic, they can be integrated in a commit-based or nightly continuous integration process, e.g., in Jenkins. This thesis does not contain any synthesis algorithm for C&C views; these algorithms are described by Maoz and Ringert [MRR13, Rin14, MPRS17].

The high-level design can also be enriched with extra-functional properties such as safety, performance or security ones. The strong typed tagging mechanism presented in Chapter 5 supports to tag only correct elements reducing human errors (e.g., shifting a line lower). An example of a check for the tagging mechanism is unit correctness: A velocity tag of a car cannot be `9 kg`. Since for each extra-functional property consistency constraints can be defined, our validation framework (cf. Chapter 6) can check full-automatically (no further user action is required) the correctness of the design model with its enriched extra-functional properties. For example, the tool can check whether the price of a component is larger than the sum of the prices of its subcomponents.

Chapter 3 shows a textual modeling language extending *Simulink* with new features such as complete unit support as well as component and port arrays. These extensions facilitate an easier description of functional C&C models: (1) Model references must not be copied to be used multiple times; and (2) stronger types with units prevent inconsistencies when connecting ports. Additionally, our textual approach is based on the modular Java class concept that supports to split one model into several textual files to be modified and versioned by different people.

Furthermore, our layout algorithm (cf. Subsection 8.5.1 and [Sch18]) creates nice graphical representations with boxes and lines of the textual model. These graphical representations enable an easier navigation between different components. Furthermore, our layout algorithm avoids manually (and time consuming) adaptions of the graphical model when adding new ports[4]. Based on the automatically calculated layout of the textual model, it is possible to generate a *MATLAB* script file containing *Simulink* API calls to create a *Simulink* model. Hence, the here presented workflow can be easily integrated into existing ones based on *SysML* and *Simulink* tools.

This thesis also defines formally when a functional model satisfies all its design models in Section 7.4. If the design verification was successful, then the tooling infers automatically all tracing information/links (cf. Subsection 7.5.2). In case the functional model does not satisfy the design model, then non-satisfaction witnesses with user-friendly error messages pointing directly to the error locations are generated (cf. Subsection 7.5.3).

The normal verification algorithm based on Maoz and Ringert [MRR14, Rin14] finds only the shortest path to satisfy the design, thus, not all traces are found. Therefore, this thesis presents besides the "normal" verification witness in Subsection 7.5.1 also a tracing witness in Subsection 7.5.2. The tracing witness contains all matched elements in the C&C model verifying one structural view element. This means the tracing witness highlights/links all structural important elements in C&C models (*Simulink*, or *EmbeddedMontiArc*) belonging to one requirement design view.

Similar to the design model, the functional model can also be enriched with extra-functional properties. For example, measured values - derived by executing the functional model on real hardware - can be added to the C&C model. A simple extension (cf. [Meh17b, Meh17a]) of

[4] *Simulink* does not have a layout algorithm, yet [Gos12]. But other modeling tools such as *Ptolemy* II [Che16] and *LabView* [Nat09] have one.

the mathematical framework, presented in this thesis, enables to check whether extra-functional properties of the functional model satisfy all extra-functional requirements specified in the design model.

2.2. Digitalizing the Systems Engineering Process using SMARDT

First, this section describes how the software systems engineering is done for the electric powertrain at BMW Group before using the SMARDT methodology. Second, this section also introduces the model-based SMARDT approach to improve the software and systems engineering process. Furthermore, this section presents how the structural verification of this thesis can be integrated in the overall SMARDT methodology and it explains the advantages of such an integration.

2.2.1. Current Systems Engineering Process at BMW Group

Main sources: [HKK+18, Section 1, Section 3]

Similar to the current development process at Daimler AG, the process at BMW Group for developing software for powertrains is based on the V-Model [BD95] displayed in Figure 2.1.

A brief summary of the left side (development) of the process is:

(i) Fact sheets describe high-level functionalities in text form.

(ii) General design decisions about the interface to its environment (also external components or user interactions) are informally (PowerPoint or Word documents) collected.

(iii) Large functions (top components) are hierarchically decomposed into smaller functions (subcomponents) so that independent developer teams can work on them; these decisions are only informal documented in Microsoft Office documents.

(iv) Based on these Office documents *Simulink* models or C/C++ code implementing these features are developed.

For the right side of the V-Model, which is the validation and verification part, tests for units, integration and acceptance are manually created representing test steps for each layer on the left side from requirements over design up to implementation.

Since the creation of these tests is done manually most of the time, this leads to several disadvantages [HKK+18]:

- Informal (mostly *SysML*-based) drawn models in Visio, PowerPoint, or other tools lack on a unique semantics [LWL04]. Thus, different teams may interpret the decisions differently.
- Due to the informal nature of *SysML* diagrams, it not possible to detect inconsistencies in one diagram (e.g., contradiction of guard conditions in activity diagrams); so derived tests may contradict each other.
- Only time-consuming and manual checks for completeness and consistencies between different layers (requirements, design, logical architecture, SW+HW implementation) are possible, since only "graphics" with no formal semantics are available. Also variation

handling between different layers, and thus variation handling of test suites, becomes more difficult.

- Ensuring consistency between the tests on the right side and the specifications on the left side becomes difficult, since only vague links between tests and specifications exist.
- Tracing test failures back to the specification is very time consuming as some system tests are for many requirements.
- Updating specifications make it necessary to manually check and update the corresponding handwritten tests. This is especially painful if the specification has not changed for years, and so the test case structure is not well-known anymore.
- Extending a system's functionality is mostly done only on the lowest layer 4 due to time pressure. Requirements and specifications of the higher layers 1 and 2, however, are not updated accordingly. This means that the documentation of the functionality - and thus also for test cases - is inconsistent with its implementation, and this is nearly the worst thing which might happen: "Incorrect documentation is often worse than no documentation." and "Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little." [Mey86]

To overcome these disadvantages, the SMARDT process as it is roughly described in the next subsection has been invented.

2.2.2. Overview of SMARDT process

Main sources: [HKK+18, Section 3], [KRRvW18, Section 3]

The SMARDT approach, as shown in Figure 2.3, does not use informal documentations in form of Office documents anymore. BMW Group decided to use *SysML* to model architecture, use cases, etc. *SysML*'s meaning is not unique as it lacks some formal semantics [LWL04]. Therefore, the here presented SMARDT methodology uses only a formalized subset of *SysML* diagrams [OMG15] with a meaningful and unique semantics [HR04] to specify the functionality of complex systems. This formal background enables to derive consistency checks between different abstraction layers of the V-Model and between productive models (left side of V-Model) and test cases (right side of V-Model). This plus on consistency is especially useful for agile development processes, which are mostly iterative, incremental, and evolutionary [BBVB+01].

The rigorous mathematical theory behind the used *SysML* diagrams enables further validations such as [KRRvW18]:

(i) **backward compatibility checks** [RSvW+15, RRS+16, BMP+16, BRRvW16, KSRvW18] for software maintenance and evolution between different diagram versions of the same layer,

(ii) **behavioral** [Rum96, HRvW17] **and structural** [BMR+17a, KKRvW18] **refinement checks** between diagrams of different layers for detecting inconsistencies in specifications between different layers.

(iii) **extra-functional property checks** on *SysML* diagrams [MRRvW16, MMR+17] to detect timing, memory or safety violations, as well as

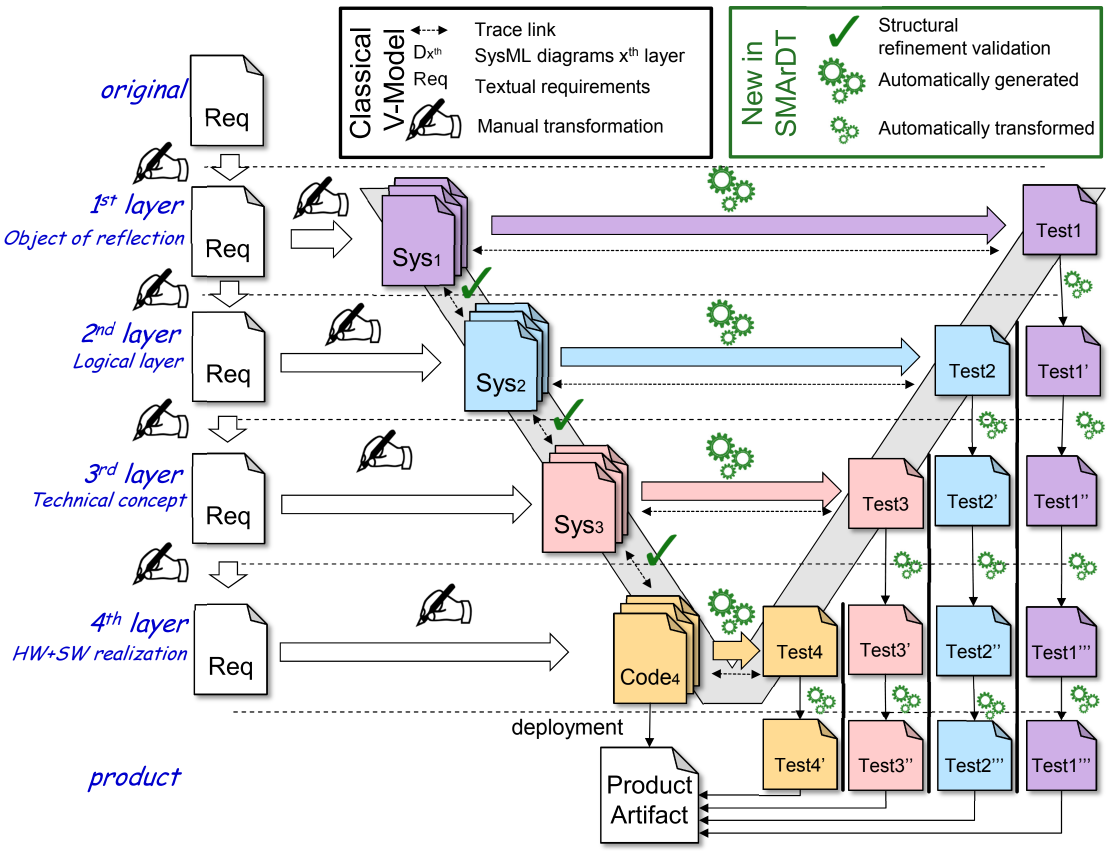


Figure 2.3.: Overview of the SMARDT methodology (copied from [HKK+18]).

(iv) automatic **test-case derivation** based on *SysML* diagrams [KMS+18] and backtracking of failed tests to **effect chains** in models (dt. Wirkkettenanalyse) to identify the cause in an easier way.

In general, SMARDT describes a formal specification for requirements, design, and testing of systems engineering artifacts according to the ISO 26262 specifications, as illustrated in Figure 2.3. Four abstraction layers structure the method [HKK+18, KKRvW18]:

0. The **textual requirement** (it maybe a user feature of a ticket in a ticket system, an exported *IBM Rational DOORS* requirement, or some text from a fact sheet) is the start situation for SMARDT, but it is not part of the actual SMARDT approach.
1. The first layer contains a first **description of the object under consideration** and it shows the object's interaction with other software and/or hardware components. This also unveils the dependencies of the object under consideration. The most common *SysML* diagrams for the first layer are use case diagrams and context diagrams.
2. The second layer contains **functional specifications** and high-level functional decompositions and their relations; e.g., functional effect chains. This layer does not deal with

technical details. The most common *SysML* diagrams for this layer are activity diagrams, state charts, block definition diagrams, and high-level internal block diagrams.

3. The third layer embraces **technical concepts** of the system. This layer contains decisions about the used control systems (e.g., proportional, derivative, bang-bang [BGG56], PID [Sko03] controllers, or lag-lead compensators [WCPL07]) as well as strategies for error handling (detection, isolation/identification and recovery [AUT09]) and diagnosis. The most used *SysML* diagrams are internal block diagrams.
4. The fourth layer represents the **software and hardware artifacts** of the system's implementation. In contrast to the third layer being hardware independent, this layer contains *MATLAB*, C/C++ implementations that are hardware specific as they contain code snippets reacting on chip-dependent low-level behavior such as memory alignments, memory size/cache distribution and I/O interrupts.

The new validation steps (green symbols in Figure 2.3) added to SMARDT, enable higher consistency:

(i) between models and tests [KMS+18, PPS+03] (due to automatically generated tests) inside one layer,
(ii) between models (due to structural and behavioral refinement) of different layers,
(iii) between tests (due to automatically transformed test cases) of different layers, as well as
(iv) between features of different layers (product-line modeling and configuration management) - skipped in Figure 2.3.

The first two layers have a specification character, meaning that these models cannot be directly executed and that multiple implementations (maybe even product-line of them) may satisfy them. Also all signals used in the first two diagrams are abstract ones and they do not correlate with the implementation signals send over FlexRay or CAN bus.

The third layer contains the complete logical architecture of the software component. Thus, a simulator (e.g., MIL, or SIL) can execute the model to detect logical behavior errors. The fourth layer contains additionally technical information such as processor, memory usage derivation. Tests for the fourth layer are mostly PIL or HIL. The models in the third and fourth layer are much larger than the high-level one used in the first two layers as these models also contain complex diagnostics and error handling strategies.

MIL (model in the loop) simulates models and its environment in a modeling framework to detect functional deficiencies at early stages of the development cycle [SPSG14]. The third layer is split sometimes into 3A (generic technical concept) and 3B (concrete technical concept) [HKK+18]. Layer 3A simulates (e.g., interpreting the model, generating code, or using a hybrid approach) the physical model (e.g., implementation model derivation contains integral and derivations in a continuous range) to detect pure logical errors. Whereas layer 3B simulates the implementation model (e.g., numeric calculations are approximated with fix-point numbers) to detect wrong scaling and/or wrong used approximation (e.g., wrong tolerance or wrong algorithm for ordinary differential equations).

SIL (software in the loop) checks the behavior of the generated code with the used compiler to verify the complete generator/compiler toolchain; the environment is simulated again without using special hardware [SPSG14]. Additionally, SIL supports to verify code coverage. PIL (processor in the loop) cross-compiles the code and executes it on a similar target processor to reveal wrong compiler settings (e.g., wrong endian) or faults caused by the processor architecture [SPSG14]. HIL (hardware in the loop) verifies the hardware electronic control unit (ECU) to detect other hardware faults.

Since a car is developed by many teams with different know-how, SMARDT also supports besides the abstraction concept (layers 1- layers 4) a decomposition mechanism which is mostly based on the geometric decomposition of a car. The geometric decomposition (vehicle function such as acceleration, vehicle subsystem such as powertrain, function cluster such as otto powertrain, and function carrier such as fuel injection) can also be interpreted partly as a feature diagram as some features are optional, and thus SMARDT also supports product line modeling. This means in reality is SMARDT a 3D structure as for every decomposition element all four abstraction layers are modelled.

The SMARDT abstraction layers 1-3 focus on different abstraction of the logical level of functions, as these functions and their interplay with the environment (physical laws) do not often change. SMARDT layer 4 is split up into 4SW (software) and 4HW (hardware). Due to different license issues or already existing software architectures of bought-in frameworks, there is no 1:1 mapping from SMARDT layer 3B to layer 4SW as the simplified overview diagram in Figure 2.3 suggests.

As the geometric decomposition and the hardware layers of SMARDT play not an important role in this thesis, the concrete explanation of the hardware layers, the 3D SMARDT structure, and the linking approach between the abstraction layers 1-4 and different decomposition levels would be out of scope for this section.

In contrast to other V-Model extensions mostly focusing on integrating or adapting (management) processes [V-M06, BR05, FKSH09], SMARDT main contribution is the formal *SysML* or *SysML*-like subset used to model artifacts in different layers in such a way that these artifacts are consistent, traceable and testable over the entire development process [HKK+18].

This thesis supports the SMARDT methodology by providing the C&C view language *EmbeddedMontiView* and the C&C architecture language *EmbeddedMontiArc*. *EmbeddedMontiView*, introduced in Chapter 7, is a C&C view language [Rin14] for embedded system designs for the logical layer of SMARDT. *EmbeddedMontiArc*, introduced in Chapter 3, is a C&C architecture language for the technical layer of SMARDT. The unique semantics (cf. [RSvW+15, KRSvW18a] and Section 7.3) of *EmbeddedMontiArc* and *EmbeddedMontiView* prevent different interpretations of the textual or derived graphical models by different developers or even teams. Furthermore, the formal semantics between both languages enables an automatic structural verification whether the technical concept still satisfies (is compliant with) the logical layer.

Additionally, the mathematical implementation [KRRvW17, KRSvW18a, HKK+18, GKR+17] of *EmbeddedMontiArc* enables to execute the technical concept in simulators to validate technical decisions (e.g., used controller kind or image recognition algorithms) earlier. The simulator also has an integrated (simple, but for research purposes sufficient) physics engine and a 3d visualization to inspect the car's interaction with its environment. This way the impact of different technical design decisions can be compared according to specified use cases of the first layer.

The simulation of time (time in simulator must not match the duration of running the simulation - which is also hardware dependent) and the simulation of other extra-functional properties such as noise distribution for different sensors facilitates to enrich the *EmbeddedMontiArc* models with the measured data of the simulation. Section 5.5 shows how to describe extra-functional properties in a model-based manner, and Chapter 6 presents how to verify an existing C&C architecture against self-defined consistency constraints for the previously defined extra-functional properties. For embedded systems interesting extra-functional properties are time, memory, ASIL level, encryption, and communication protocols.

The simulator as well as the mathematical implementation describing the behavior of C&C components is not part of this thesis, but for further information the following publications [KRSvW18a, KRRvW17, HKK+18, GKR+17, KRRvW18, KRSvW18b] and videos [Mok18, vW18, Dal18, Lor17, Ilo18a, Hei18, Hal18, Str18b, Str18c] are available.

EmbeddedMontiArc as well as *EmbeddedMontiView* are textual modeling languages. Textual modeling concept exhibits - compared to existing graphical modeling tools such as *PTC Integrity Modeler*, *Cameo Systems Modeler*, *Enterprise Architect*, and Mathwork *Simulink*- the following advantages [KRRvW18]:

(i) All model information is direct available in files. In contrast, graphical modeling tools hide information behind different dialog boxes and tabs. The graphical layout is often saved in proprietary (binary) formats (e.g., cryptic XML or database format) where accessing and reading information is hard, since the stored data does not contain only syntax but also many customizable layout information. Due to the tree structure of most binary formats and the many additional information, integrated search speed for large models is mostly slow.

(ii) Textual IDEs (e.g., Notepad++ [Ho18], or Eclipse [KPP06a]) can find or replace information via simple or regex search. Additionally, bash scripts can efficiently (in memory and runtime) manipulate many text files; e.g. by calling `sed` or `grep`. In contrast, graphical models must always be updated by using the vendor-specific API with its own functions. These APIs are mostly not very well documented (exception is the *MATLAB* API for manipulating *Simulink* models) and some are even incomplete.

(iii) Text-based versioning tools like SVN [PCSF08], Mercurial [Mer18], Microsoft TFVC (Team Foundation Version Control) [BWHK12], and Git [LM12] support many text differencing, text merging and text branching features. Even most graphical tools have an XML export, reading an XML difference is hard. The exported graph structure uses generated identifies, mostly a cryptic number, for all graphical elements (e.g., boxes). Links between graphical elements connect two of these identifies, and thus, it is hard to understand (in XML diffs) what elements are how connected. Even though some tools have their own Git or SVN plugin, the graphical models can still not be convenient used on version control platforms such as GitHub [DSTH12], GitLab [BHJ16], BitBucket [Leo16], or CloudForge [YGJK16] as they all focus on textual models.

(iv) Similar to all major programming languages (e.g., Java, C, C++, *Ada*, Delphi) different teams collaborate together in large projects via different files, folders, or even repositories according to their responsibilities. *EmbeddedMontiArc* and *EmbeddedMontiView* use this separation of artifacts paradigm as well as a library import concept with version control (based on Maven) to enable modeling in the large. Different files, repositories, or even deployed libraries for complex projects - containing of thousands of components - enable better collaboration, as single component files or single repositories can be easily branched, merged, and independently reverted.

(v) Test driven development increases code quality. *EmbeddedMontiArc* has a textual domain-specific testing language for unit tests of components. This stream testing language is based on the formal semantics of the Focus theory. In contrast to a graphical testing framework where test elements must be copied or modified one by one, textual files enable to copy and modify all tests or only some of them at once. For example, in *Simulink* removing test data (e.g., one time step) for a subsystem is time consuming as every point for every input port data must be removed via mouse clicks in the graphical signal builder editor. Another advantage of *EmbeddedMontiArc*'s stream unit tests is the partial support of underspecification [GKR+17]. Underspecification is needed for test-driven development as the complete behavior specification of a model is not known in higher abstraction levels (e.g., SMARDT level 1 or 2).

(vi) Agile development has short development cycles, e.g., 7-day scrum sprints. Software and models in a sprint are developed for given user stories. For new user stories (due to customer feedback) models are updated. But frequent updates of large graphical models (e.g., by inserting and reconnecting components) is very time consuming, because the existing graphical layout (at least for one visible hierarchy) needs to be manually rearranged to obtain readable models without having overlapping and crossing modeling elements. *EmbeddedMontiArc* integrates a HTML/SVG generator (cf. Subsection 8.5.1, and [Sch18]), which automatically produces a good readable graphical layout based on textual files. This way, modeler can focus on the main task by only adding, changing, or removing textual lines, and still have a graphical C&C architecture for better understanding.

2.3. Similar Existing Methodologies and Model-based Approaches

The first two sections in this chapter introduced the model-based systems engineering process for software components at Daimler AG and at BMW Group. This section presents similar, but not company specific, model-based approaches and their tools; namely *Simulink* Requirements, *Mentor Capital*, *Polarsys Arcadia*, and Vector *PREEvision* at some detail.

In literature exist many other (partly) related approaches. Some of them are:

- VDI V-Model [GM03] approach uses modeling, model analysis and simulation.
- *COLA* (Component Language) automotive approach focuses on three different architecture levels: Feature Architecture, Logical Architecture, and Technical Architecture [Kug12, KTB+07].
- *CAR-CL* (Combined ARchitecutre Description Language) is a seamless model-based development approach using architecture based specification and verification of Embedded Software Systems [Bro08]. This approach uses four abstraction levels: Service Level, Functional Level, Logical Cluster Level, and Platform Level.

- *EAST-ADL* (Electronics Architecture and Software Technology - Architecture Description Language) models automotive systems in four abstraction levels: Vehicle Level, Analysis Level, Design Level, and Implementation Level [CFJ+10].
- *Save-IDE* is an integrated development environment for building predictable component-based dependable embedded systems [SPCH08]. It supports design, analysis, transformations, and verification of models. It uses timed automata as behavior models.
- *Forsoft* Automotive project [BRS00] focuses on requirement analysis and specification of the overall functional development process. It uses three abstraction level: Logical Level (User Requirements - functional network), Abstract Architecture (System Requirements - perfect world assumptions), and Concrete Architecture (Architecture Design - real world assumptions).
- *AutoFOCUS*' [HF10, Kug12] main features are the design and analysis of distributed, reactive and timed systems. It has the three abstraction levels: Functional Architecture, Logical Architecture, and Technical Architecture. It is based on the focus theory.
- *SCADE* Suite [ADS+06] (based on data-flow language Lustre [PHP87]) designs safe and reliable systems. It has failure mode, fault tree analysis and effect analysis to calculate minimal combination failures.
- *MICOBS* framework [PPK+11] transforms high-level components to native component implementations to achieve better abstraction and reusability. It also supports analysis of extra-functional properties to find the best deployment of a system.

2.3.1. *Simulink* Requirements

Simulink Requirements [Urb15, The18m] is a product which focus on the ISO 26262 safety norm and supports to import requirements from well-known existing tools or create its own requirements. Based on the specified textual requirements, they can check whether every requirement is linked to a block and whether every block contains a requirement link. The tool also supports to group requirements hierarchical, and thus it can calculate a percentage number how many requirements of this group are mapped to *Simulink* blocks.

But this tooling (as shown in Figure 2.4) goes directly from textual requirements to executable specifications, and this means it skips the underspecified, and thus not executable, design levels (according to the SMARDT process, it would go from level 0, Requirements, directly to level 3, executable technical model). The tooling only supports to link requirements to blocks, but this way requirements cannot be linked to connectors to express communication between requirements.

The approach presented in this thesis (cf. evaluation on case study in Chapter 8) is more general, it facilitates to create for each requirement a high-level design specification answering the following questions:

(1) *What components/blocks must exist?*
(2) *How are these blocks in relations (siblings, parent, child)?*
(3) *How do these blocks interact with each other (abstract connections between blocks or even their ports)?*
(4) *Are there (direct or indirect) effects between blocks or between in- and output ports inside one block?*

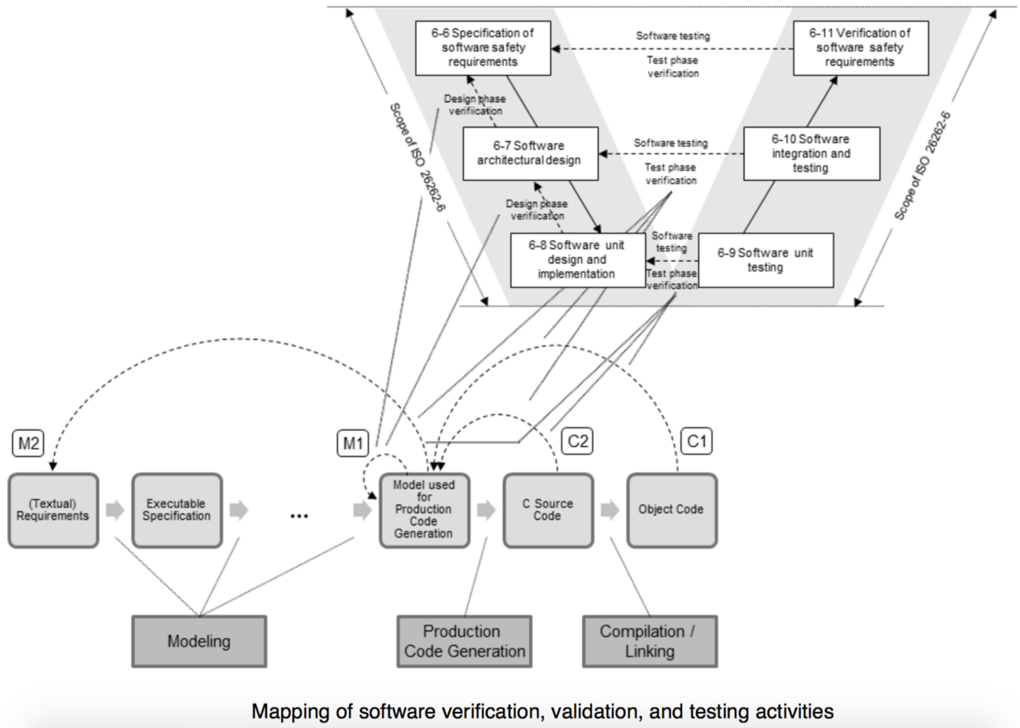


Figure 2.4.: *Simulink* Requirements toolbox for ISO26262 modeling (copied from [Con12]).

Furthermore, the (not-executable, and incomplete) design specifications can be checked against the implementation level. The approach of this thesis also generates tracing between the textual requirement and *Simulink* model, this means all the checks being available in *Simulink* Requirements and other related toolboxes such as *Simulink* Design Verifier can be reused.

2.3.2. *Mentor Capital*

Mentor Capital supports three abstraction layers and provides corresponding tooling for them (cf. Figure 2.5): logic layer (similar to SMARDT layer 2), wiring layer (similar to SMARDT layer 3B), and harness (dt. Kabelbaum) layer (similar to SMARDT layer 4).

In contrast to SMARDT focusing on pure functional constraints based on requirements and user stories in the first two layers, *Mentor Capital* is much more focused on electrical wiring. Therefore, *Capital Logic* deals with logical wiring over signals (which is close to SMARDT layer 2) as well as with physical wiring designs (e.g., wires, splices, and multicores) that is related to SMARDT layer 3B or SMARDT layer 4. *Capital Logic* is bound to C&C hierarchy borders in logical and physical designs and, therefore, it is less suited for modeling abstract functional (under-)specification as it is possible in C&C views.

Capital Integrator uses rules and designs for synthesis of wiring systems, so general rules and designs can be established and these rules can be reused (e.g., only a design might be omitted or changed) for different implementations. *Capital Integrator* automatically synthesizes the complete physical implementation [Men18c]. This approach is similar to the C&C views

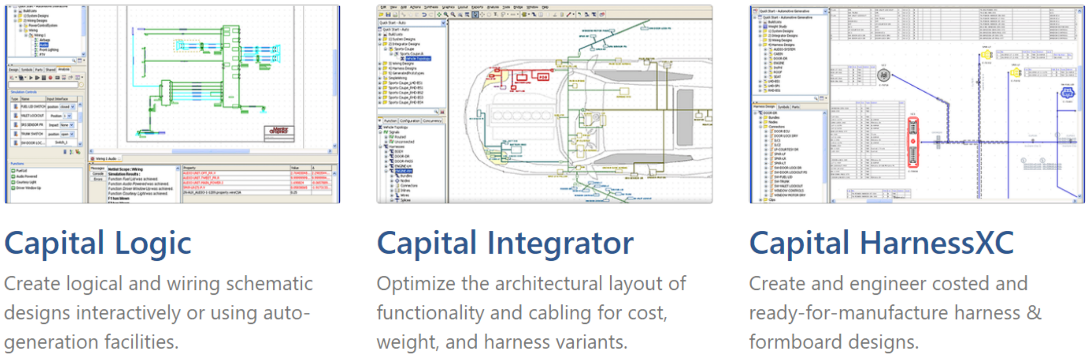


Figure 2.5.: *Mentor Capital*'s tools for modeling at different abstraction levels (copied from [Men18a]).

synthesis [MRR13]. Even though this thesis does not extend the C&C view synthesis algorithm, its tagging mechanism for C&C views enables to model these kinds of wiring constraints, as the views can now be tagged with communication frequencies and delays. Additionally, the design can be verified against the logical architecture.

The overall process for *Mentor Capital* is the following: In *Capital Logic* logical signals of a design and the wanted wiring schemata are defined for small C&C models, and then with *Capital Integrator* all the design models are synthesized by using the specified rules to optimize latency or other properties. So *Capital Integrator* generates from many wiring schemata one large physical wiring architecture. Last *Capital Harness XC* automatically adds, i.e., wires, multicores, terminals, seals, cavity plugs, tapes, tubes, and heat-shrink sleeves to the physical wiring architecture to generate a manufacturing-ready harness design [Men18b].

The generative approach from *Capital Logic*, *Capital Integrator*, and *Capital Harness XC* enables rapid-prototyping as well as the creation of the final product. The synthesis of the physical layer and the generation of harness components enable consistent and fast updates of the other layers when changing signals in the logical layer.

In Mentor's keynote "Systems of Systems - What's the Story?" [Kur17] the validation of designs for new processes, traceability and re-use are very important to integrate systems of systems. *Mentor Capital Publisher* [Men18d] aims to skip documentation and it generates the documentation based on Capital models. This means that Mentor addresses similar challenges (traceability, product line, and evolution) as this thesis in the case study with Daimler AG in Chapter 8.

2.3.3. *Polarsys Arcadia*

Polarsys Arcadia [Pol18], and its corresponding tooling *Capella* [Cla18b] is an overall modeling approach similar to SMARDT. It also consists of four layers (cf. Figure 2.6) which are similar to the four SMARDT layers [Cla18a]:

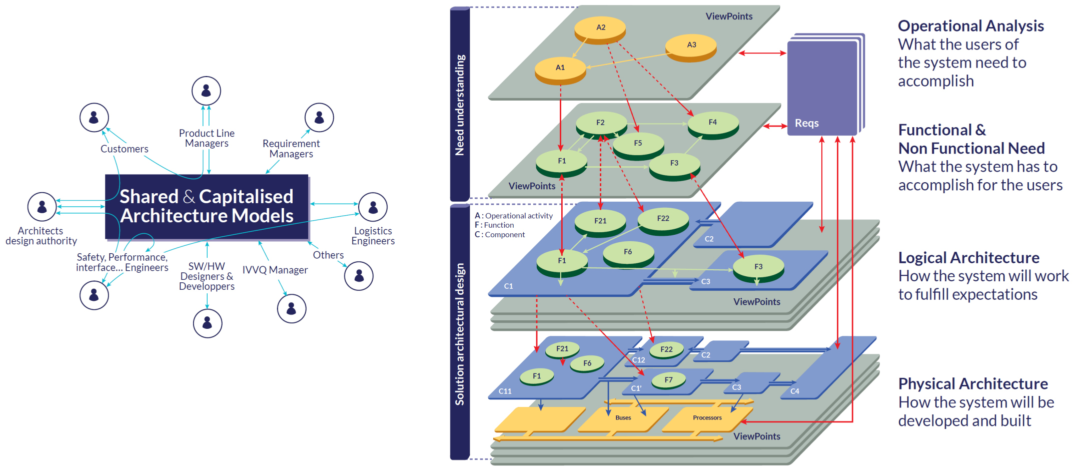


Figure 2.6.: Architectural Layers when modeling with *Polarsys Arcadia* (copied from [Cla18a]).

(1) **Definition of the Problem - Customer Operational Need Analysis:** Analyzing customer needs, expected mission and activities. It is like a case study for the customer what he needs and expects.
(2) **Formalization of system requirements - System Need Analysis:** Focuses on the system itself and how it can satisfy the needs of layer 1. In this phase also extra-functional constraints such as safety, security, performance, etc. are modelled. In this phase also a first architecture is created to check the requirements and extra-functional properties against the architecture and to estimate the total costs for the project.
(3) **Development of System Architectural Design - Logical Architecture (Notional Solution):** Based on the functional and extra-functional requirements of layer 1 and 2, a complete logical system architecture is developed. They use a viewpoint-driven method to formalize all extra-functional properties. To validate the architecture and its viewpoints against layer 1 and 2, the logical architecture contains links to its requirements.
(4) **Development of System Architecture - Physical Architecture:** This layer does the same as the layer above but it finalizes the architecture. The layer introduces design patterns, technical services and framework choices so that the components can be developed by different teams, and that the output can communicate via the technical solutions.

Polarsys Arcadia uses viewpoints on every layer and each viewpoint deals with a specific concern. This approach is similar to our tagging approach, where you can create for each concern (extra-functional property) your own tag file. The viewpoints of the second layer can be compared with the C&C view concept presented in this thesis. Similar to C&C views, in *Polarsys Arcadia* each viewpoint in the second layer deals only with the structural elements being relevant. Similar to calculating the tracing between *EmbeddedMontiView* and *EmbeddedMontiArc*, which are the witnesses, Capella can compute simplified links between the layers [Roq16].

2.3.4. Vector *PREEvision*

Vector PREEvision [Sch16] is a 150% modeling approach with similar purpose of *Polarsys Arcadia* and SMARDT. A more theoretically focused approach similar [Zve08] to Vector *PREEvision* is *COLA*- The component language [KTB+07] - from TU Munich.

As showed in Figure 2.7, the *PREEvision* approach contains eight architecture abstraction levels, whereby each level may belong to different product lines [Sch16]:

Level 1: Requirements, Customer Features, Feature-Functionality-Network
Level 2: Logical Architecture, Activity Chain (from Sense to Actuation), Logical Functions, Block Diagrams
Level 3: System-Software Architecture: Composition of Software Components
Level 4: Implementation: Packages und Files
Level 5: Hardware Component Architecture, Hardware Network Topology
Level 6: Electric Circuit, Power Supply
Level 7: Wiring Harness, Ground, Gateways
Level 8: Geometrical Topology

PREEvision has also a communication layer according to *AUTOSAR* which is orthogonal to the abstraction levels 2 to 5. This layer supports enriching logical communication with extra information such as topology (e.g., CAN, CAN FD, LIN, FlexRay, Ethernet) and then it uses Dijkstra to automatically suggest routing information based on bus loads and data types.

Vector *PREEvision* is an E/E (electric and electronic) architecture design and optimization model-based approach. It supports three groups of optimization targets [Sch16]:

- Global vehicle targets, such as cost, weight, package and geometry (e.g., cable diameters, cable length), and power consumption constraints;
- E/E targets, such as real time requirements, diagnostic and service requirements (e.g., service interface or over the air), and bus load constraints; as well as
- product line targets, such as variants, options, product lines, expected production numbers, and function oriented decomposition vs. component oriented reuse.

Similar to SMARDT supporting abstraction and decomposition, the *PREEvision* approach uses a similar matrix structure: The vertical direction in Figure 2.7 provides abstraction from logical communication over wiring harness details to complete electric circuit to the ECU in network levels. The horizontal direction provides decomposition so that every level can be hierarchical decomposed to support top-down and bottom-up development. SMARDT focuses more on top-down development, but it also supports bottom-up. *PREEvision* level 1 maps to SMARDT layer 1. *PREEvision* level 2 maps to SMARDT layer 2. *PREEvision* level 3 maps to SMARDT layer 3A. *PREEvision* level 4 maps partly to SMARDT layer 3B and also partly to SMARDT layer 4SW. *PREEvision* levels 5 to 7 maps to SMARDT layer 4HW. As SMARDT was mostly developed for the software components of systems engineering, there exists no mapping for *PREEvision* level 8 in SMARDT, yet.

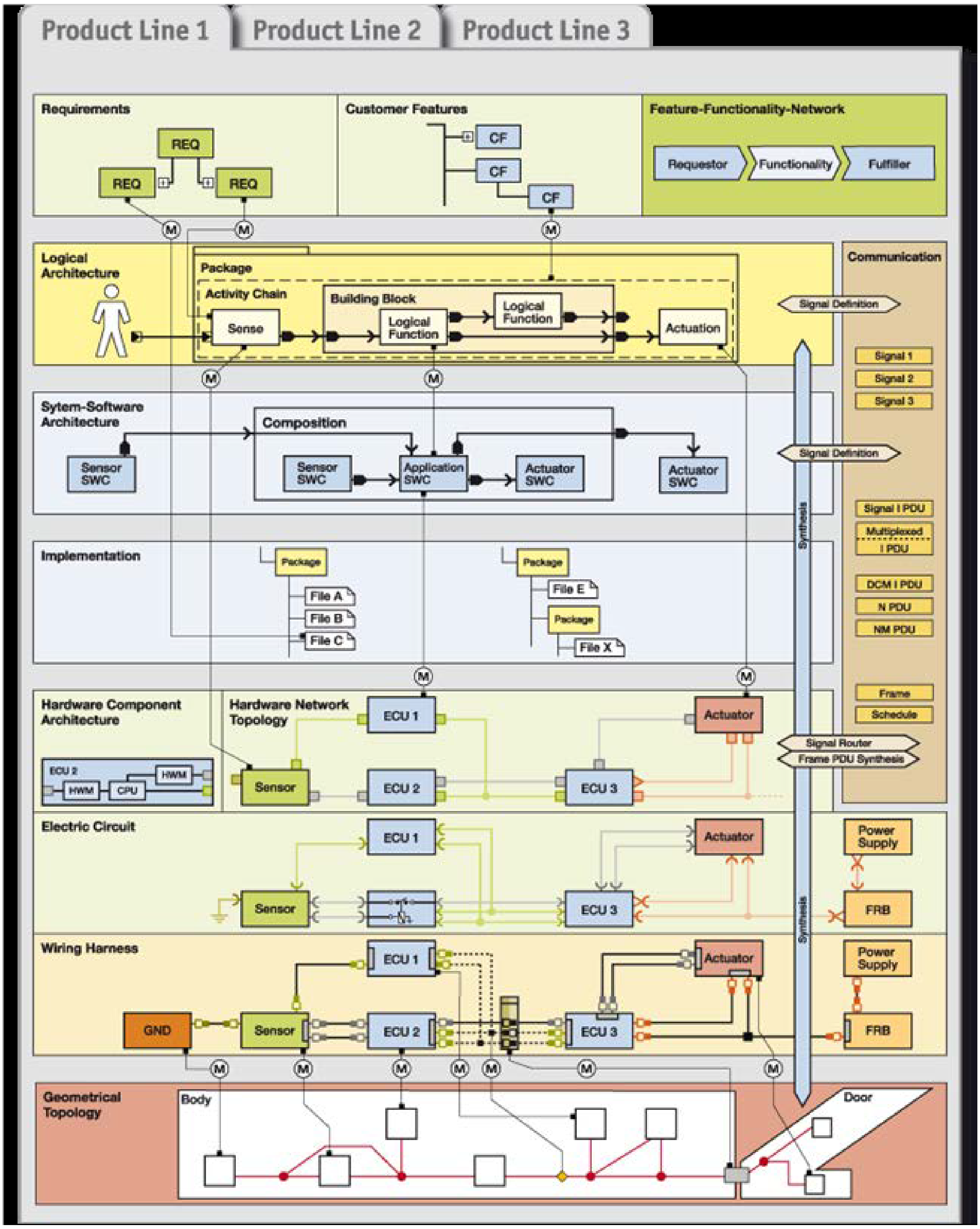


Figure 2.7.: Vector *PREEvision* methodology for modeling embedded systems (copied from [Sch16]).

Chapter 3.

Concrete Syntax of *EmbeddedMontiArc*: A Functional Component and Connector Modeling Language for Cyber-Physical Systems

Chapter 1 and Chapter 2 explained the importance of component and connector (C&C) models for embedded and cyber-physical systems. Chapter 2 elucidates how C&C models can be integrated into the systems engineering process in the automotive industry.

This chapter introduces the functional modeling language family around *EmbeddedMontiArc*. *EmbeddedMontiArc* is a textual domain specific language for the logical layer. This means its main focus is on the functional correctness when modeling features of embedded systems. *EmbeddedMontiArc* does not try to solve the problems of the technical or even software/hardware specific layer with first order language concepts. However, libraries and modeling patterns for *EmbeddedMontiArc* allow to address redundancy, safety, or diagnostics and error recovery strategies due to software or hardware failures in an efficient way.

EmbeddedMontiArc tries to support the functional and logical modeling of embedded systems in an efficient, agile, and intuitive way. Therefore, Section 3.1 starts with a an explicit declaration of requirements. These requirements are derived from many interviews with industrial partners in the automotive domain during project collaborations of them with the software engineering chair at RWTH Aachen University. Section 3.2 continues with a literature overview for a modeling language for the logical layer by presenting a large analysis of existing standards, tools, programming and modeling languages in the field of embedded and cyber-physical systems. We want to investigate how the existing approaches solve some of our requirements. Next, Section 3.4 gives a general overview of the complete *EmbeddedMontiArc* modeling family. Section 3.5, and Section 3.6 present in detail the concrete syntax of the *EmbeddedMontiArc* modeling language, which integrates the best modeling concepts according to our requirements of the investigated existing standards, tools, and languages.

Highlights of the textual C&C modeling language *EmbeddedMontiArc* are:

(i) modular and reusable component types with component interfaces,
(ii) component and connector arrays,
(iii) component libraries due to generics for port types, array dimensions, and components,
(iv) convenient connection patterns, as well as,
(v) a strict type system with unit and accuracy support.

EmbeddedMontiArc's type system together with its configuration and generic parameters facilitates an efficient modeling of large functional C&C software systems, because library components such as PID controllers or image cluster components can be easily reused. Arrays, both port and component instantiation arrays, in combination with generic and configuration parameters support agile and efficient development, as the number of component or port instances can be easily adapted by just changing one number in a model. Thus, time intensive duplicating or removing of component instances (e.g., when changing the number of front or rear park sensors in a car) and reconnecting the other components are avoided. The convenient connection patterns with index- or name-based connection patterns of ports or port arrays facilitate an intuitive modeling of the logical communication between components. The strict type system with its integrated static verifications detects errors (e.g., incompatible matrices or port types) as fast as possible (e.g., during model creation in the IDE), but at the very latest when compiling *EmbeddedMontiArc* models. This prevents cost-intensive runtime failures and long bug-fixing sessions resulting in a more efficient systems engineering process.

After presenting all language features of *EmbeddedMontiArc* (cf. Section 3.5, and Section 3.6), this chapter discusses potential new language concepts for the *EmbeddedMontiArc* modeling family in Section 3.7. Section 3.8 presents an example business use case modeled in *Embedded-MontiArc* to illustrate that *EmbeddedMontiArc* can also be used outside the systems engineering domain. Finally, this chapter finishes by presenting *EmbeddedMontiArcStudio*, the tooling around *EmbeddedMontiArc* language family, with all its user experience features.

3.1. Requirements for a Logical Architecture Modeling Language

According to requirement analysis based on a decade of multiple automotive industry collaborations [KMS+18, KKRvW18, BMR+18, HKK+18, KMS+17, BMR+17a, DDE+17, KRR+16, RRS+16, BMP+16, RSRS15, BBH+15a, BBH+15b, RSvW+15, KRR15, BBH+14b, BBH+14a, CEG+14, BHK+07, HKM+13, BBH+13, KDH+13, GRJA12, HRRW12, BRR+10, BRRW13, RBL+08, MFZ+09, BRS09, BBKR09] at the Software Engineering Chair at RWTH Aachen University, a modeling language for cyber-physical and embedded systems should satisfy the following requirements (points (M1) to (M7) are already discussed in [KRRvW17, Section 3]):

- (M1) **Unit support.** In- and output ports should support metric, imperial, and customized units, such as pixel-per-inch.
- (M2) **Unit conversion.** Units should be convertible to SI units in port connections and in mathematical expressions.
- (M3) **Array support.** Redundancy in models should be avoided by supporting arrays of ports and component instantiations. A convenient mechanism to interconnect and access ports and component instantiations should be supported.
- (M4) **Domain.** There is a need for concepts to model the domain; i.e., minimum, maximum, and resolution, of the values exchanged between components.

- (M5) **Static Analysis.** Tools to support static analysis, i.e., over- and underflow checks, division by zero, detection of components in dead paths, and detection of duplicated components.
- (M6) **Reuse concepts.** A library concept for components and ports configurable over parameters is needed. Advanced reuse concepts such as configuration parameters and generics are required to enable modifications of component interfaces and behavior.
- (M7) **Matrix type supports.** Discrete control systems are often described by matrix-vector expressions. To reduce error-proneness a type system should support static matrix dimension, units, and detection of domain incompatibilities, e.g., multiplying two 3×3 matrices having the domain $[0; 1]^{3\times3}$ (all values of both 3×3 matrices are between 0 and 1) must result in a $[0; 3]^{3\times3}$ matrix (the values of the 3×3 result matrix are between 0 and 3).
- (M8) **Support for test driven development.** The language should have a first level integration for unit tests so that the correctness of a model can be checked. To enable complete test driven modeling, the tests should also support underspecification[1].
- (M9) **Product-line support.** Most embedded systems offer different variants (e.g., cars, coffee machines, stoves, or airplanes) to customers. Therefore, the software for these systems is mostly a large product family.
- (M10) **Multiple behavior languages.** Extendibility to support different languages for behavior implementations for components. Examples are statecharts, differential equations, imperative programming, declarative programming (e.g., as an optimization), and convolutional neural networks as used in artificial intelligence.
- (M11) **Advanced search.** Large embedded software systems are composed of many components (or classes, modules) in different hierarchy/abstraction levels. An advanced search enables to look for relations of components in different hierarchies in an efficient and intuitive way.
- (M12) **Annotation mechanism.** Cyber-physical systems are often enriched with extra-functional properties or hardware information; mostly in form of profiles, tags, or stereotypes.

3.2. Existing C&C Modeling Languages

Main sources: [KRRvW17, Section 4]

This section compares important standards, modeling and programming languages for embedded and cyber-physical systems. Table 3.1 lists an overview of the investigated languages according to the requirements presented in the previous section. The features for *AADL* and *ADML* are the same, because *ADML* tried to standardize concepts of *ACME* in XML; so *ADML* is an XML-version of *AADL* [TMD10, slide 37].

About half of the languages in Table 3.1 support units (M1). *AADL* provides a special `units` keyword to define units and their relation (e.g., `ns => ps * 1000`) [Gre07], [Ins15, slide

[1]This requirement in this chapter only forces underspecification for tests. However, Chapter 7 introduces an architectural specification language supporting architectural underspecification

Table 3.1.: Comparison of standards, modeling languages, and programming languages of cyber-physical and embedded systems $\surd$: yes, p: partially, -: no, ?: unknown

Architectural Description Language	Unit support (M1)	Unit converison (M2)	Component/Port arrays (M3)	Domain (M4)	Static analysis (M5)	Reuse concepts (M6)	Matrix Support (M7)	Test driven development (M8)	Product-line support (M9)	Multiple Behavior implementations (M10)	Agnostic search (M11)	Annotation mechanism (M12)
AADL	p	p	$\surd$	$\surd$	$\surd$	$\surd$	p	$\surd$	$\surd$	p	?	$\surd$
ACME	-	-	-	-	$\surd$	$\surd$	-	-	-	-	?	$\surd$
Ada	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	-	$\surd$	-	-	p	-
ADML	-	-	-	-	$\surd$	$\surd$	-	-	-	-	?	$\surd$
ArchJava	-	-	$\surd$	-	?	$\surd$	-	$\surd$	$\surd$	-	?	-
ASCET/ESDL	-	-	-	$\surd$	p	$\surd$	-	?	p	p	p	-
AutoFOCUS 3	-	-	-	$\surd$	$\surd$	p	-	$\surd$	-	p	p	?
AUTOSAR	$\surd$	$\surd$	-	$\surd$	?	p	-	$\surd$	$\surd$	?	n	$\surd$
Darwin	-	-	$\surd$	?	$\surd$	p	-	-	-	-	-	-
EmbeddedMontiArc	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	p	$\surd$	$\surd$	$\surd$
Koala	-	-	-	-	p	$\surd$	-	-	$\surd$	-	?	-
LabView	$\surd$	$\surd$	-	$\surd$	$\surd$	p	$\surd$	$\surd$	$\surd$	-	p	?
MARTE	$\surd$	$\surd$	-	p	-	$\surd$	-	-	p	p	p	$\surd$
Modelica	$\surd$	$\surd$	p	$\surd$	p	$\surd$	$\surd$	$\surd$	p	p	p	$\surd$
MontiArc	-	-	-	$\surd$	p	$\surd$	-	$\surd$	$\surd$	$\surd$	p	$\surd$
Ptolemy II	-	-	$\surd$	-	$\surd$	$\surd$	-	p	-	$\surd$	p	?
Rapide	-	-	-		$\surd$	p	-	p	?	-	-	-
ROOM	-	-	$\surd$	-	?	p	-	?	-	-	-	-
SCADE	-	-	p	-	$\surd$	$\surd$	$\surd$	$\surd$	-	p	?	$\surd$
Simulink	$\surd$	$\surd$	-	$\surd$	p	p	p	p	$\surd$	p	p	-
SysML	$\surd$	$\surd$	-	p	-	$\surd$	-	-	$\surd$	p	-	$\surd$
SystemC	$\surd$	$\surd$	$\surd$	-	$\surd$	$\surd$	p	$\surd$	p	-	-	-
TECS	-	-	-	-	?	p	-	?	-	-	-	-
UniCon	$\surd$	-	$\surd$	$\surd$	$\surd$	p	-	-	$\surd$	-	?	p
Verilog (AMS)	$\surd$	p	$\surd$	$\surd$	$\surd$	p	-	$\surd$	-	-	?	
VHDL (AMS)	$\surd$	p	$\surd$	$\surd$	$\surd$	p	-	$\surd$	-	-	?	
Weaves	-	-	-	-	p	-	-	-	-	-	-	-
WRIGHT	-	-	$\surd$	p	$\surd$	p	-	-	-	-	-	-
xADL	p	-	-	p	p	p	-	-	p	?	p	$\surd$

17]. But the Cartesian product of units is not really readable in *AADL* [BKU14, slide 15]. The Cartesian product looks like `mph`, `mpsec`, `kmph`, or `kmpsec` due to name conventions and package conflicts in *AADL*. *Ada* does not support units natively. *Ada*'s ability to overload operators (also via generics) and to define UTF-8 unit constants (e.g. ^{2}, ^{3}, etc.), makes it possible to create complete unit libraries [Kaz14]. This way also combined units and measures are possible; e.g., `Entity := 5.0 * A / s` [Kaz14]. *AUTOSAR*, *LabView*, *MARTE*, *Modelica*, and *SysML* fully support SI units according to ISO 31-1992 [KRRvW17]. *Simulink* introduced unit support step-wise since version R2016a [The18l]. *MATLAB* and *Simulink* [The18k, Section 9] version R2018a also support unit consistency checks [The18o, p.2-7], unit conversions [The18o, p.2-31ff.], defining new units [The18o, p.2-35ff.], restriction of unit kinds, as well as units in differential equations [The18o, p.2-9]. *SystemC* extends C++ to enable discrete event simulation via event-driven interfaces [Wik18]. Similar to *Ada*, *SystemC* does not support units natively. C++ preprocessor templates [Lem16, p.299],[Jur15] add full unit support to *SystemC*. C++ templates are executed at compile time and, thus, unit inconsistencies do not cause runtime errors. *Verilog* and *VHDL* integrate units as part of their numbers and support number prefixes such as Nano, or Pico [KRRvW17]. *EmbeddedMontiArc* reuses *SysML*'s unit concept (cf. abstract syntax in *SysML* 1.4 [OMG15, Section 8.6.4]) to be compliant to ISO 31-1992.

All languages with full unit support enable unit conversions (M2). *Simulink* supports to enable or disable whether units are automatically converted [The18k, p. 9-16]. For manual unit conversions, *Simulink* offers the `Simulink-PS Converter` block [The18k, p. 9-14]. *Verilog* and *VHDL* can only convert flow to potential and vice versa by using disciplines [KRRvW17]. Both AMS languages do not support complex conversions (e.g., `km/h` into `mi/h`, or `°C` into `°F`) [KRRvW17]. *EmbeddedMontiArc* converts units automatically when their dimensions are compatible.

AADL offers array support (M3) since version 2. *AADL* 2 supports component arrays and connection patterns [Fei10, slide 15]. *Ada* has array support of types (similar to components), and it supports function access arrays (similar to connectors). *ArchJava* offers arrays of component and port types via Java arrays, and it adds connection patterns [ACN02]. *Darwin* supports component arrays and efficiently binds (similar to connectors) components with for loop constructs [TMD10, slide 7]. *Ptolemy* supports `MultiInstanceComposite` components, which defines the number of instances of channels via parameter values, and it also supports array iterations [Pto14, Section 2.7]. *ROOM* (Real-Time Object-Oriented Modeling) supports as one of the first languages port arrays and a way how to connect these [Sel96]. *SystemC*, a C++ extension, supports arrays for ports and signals. *SystemC* supports to declare array sizes of ports and signals using a C-like syntax. Java and ADLs using the Java type system (e.g., *MontiArc*) contain only array dimensions. In contrast, C++ arrays contain the complete array size. *UniCon* (arrays of simple types), *Verilog* (one and two dimensional arrays), *VHDL* (ranged and unconstrained arrays), and *WRIGHT* (multiple instances of pipes) also satisfy the array requirement (M3) [KRRvW17].

AADL uses the same syntax as *Ada* to define the domain of ranges (M4). *AADL* also supports to combine ranges with units (e.g., `1 b .. 1 kb`) [TMD10, slide 26]. *Ada* can define the step-operator with the `delta` keyword [Nag99]. *ASCET*'s *ESDL* language uses similar syntax (e.g., `type c_uint8 is integer 0 .. 255 using c_unsigned_char`) [Rie18]. *AADL* and *ASCET* do not support to define steps in ranges. *Ada* supports besides ranges also modulo types, where no overflow or underflow can occur [Nag99]. Modulo types are especially useful for hash calculations. *AUTOSAR* (using subnode constraint), *LabView* (custom scales), *Modelica* (attributes in reals and intergers variable decleration), *MontiArc* (stereotypes when using *MontiMatcher* [RSvW$^+$15, RSvW16] extension), *Simulink* (slope and bias for fix point data types), *UniCon*, *Verilog* (ranges plus abstol attribute), and *VHDL* (same as *Verilog* plus tolerances) support domains (M4) [KRRvW17]. *MARTE*, *SysML* (stereotype «data type» and *OCL*), *WRIGHT* (ranges for instances [All97, p. 154]), and *xADL* support ranges (M4) partially.

The following languages have tools or provide a theory for static analysis or verification of structure or behavior: *AADL* (OCARINA model analysis framework [LZPH09]), *ACME* (*AcmeStudio*'s verification engines [Rec08, p. 274]), ADA (ADACore's CodePeer Static Analysis Tool [AB14]), *ADML* (see *AADL*), *AutoFOCUS* 3 (model checkers NuSMV/nuXmv [CCD$^+$14] for unreachable states and range checks), *Darwin* (uses pi-calculus for formal analysis [TMD10, slide 5], *LabView* (programs are verified by ACL2 solver [KKR09]), Ptomely (Java static analysis tools such as Cibai [Log07]), *Rapide* (cf. publications of stanford program analysis and verification group [LKA$^+$95]), *SCADE* (model analysis with *SCADE* design verifier [JH05]), *SystemC* (type checking, CFG analysis, and verifying pointers with SCOOT [BKS08]), *UniCon* (translation of ADL to labelled transition systems, computational tree logic, or petri nets [TX00, Figure 1] and then verification with tools such as SPIN [Hol97], or NuSMV2 [CCG$^+$02]), *Verilog* (using VIS [BHSV$^+$96]), *VHDL* (verification with tools [ZTB08] such as PVS [GV99] and Mathematica [ASZT07]), and *WRIGHT* (translation into communicating sequential processes for automated analysis [TMD10, slide 17]). *EmbeddedMontiArc* uses *MontiMatcher* to identify structural and behavioral duplicates [RSvW$^+$15, RSvW16] or inconsistencies [HRvW17] as well as to detect over- and underflow [Tol16].

Reuse concepts (M6) of components/classes can be satisfied by different ways, e.g., using configuration or generic parameters, implementing interfaces, template mechanisms, or via feature modeling. The reuse concepts of the investigated languages are: *AADL*, and thus also *ADML*, have packages to structure large project, as well as *AADL* supports extending and refining components [Ins15, slide 5]. *ACME* has an extensible type system with parameters and templates [TMD10, slide 36]. *Ada* has package structure, modules (to provide and hide information), inheritance, generics, and access (pointers) types for controlled reusability [Nag99]. *ArchJava* reuses all atomic concepts [Aßm08], and inherited features like generics of Java. *ASCET/ESDL* supports classes, variants and features. *Koala* enables reusability via product-lines [TMD10, slide 21]. *MARTE* uses stereotypes to define configuration and generic parameters [KRRvW17]. *Modelica* uses the MBLOCK for generics, it also supports configuration parameters [KRRvW17]. *MontiArc*'s type system is based on the Java one: it supports component inheritance, generics, and configuration parameters for components [Hab16]. *Ptolemy* is a Java extension, and thus it inherits these reusable features. Higher order functions and generics are the basis for *SCADE*'s reusability concepts [Est10]. *SysML* enables product-line modeling, generics, and configuration parameters via stereotypes or profiles [KRRvW17]. *Simulink* supports model references with configuration parameters to reuse components. *Simulink* has a library concept but without generics [The18k]. *SystemC* has the features inherited from C++. *Verilog* and *VHDL* support configuration parameters, but no generics [KRRvW17]. *xADL* does not have native generic support, but its extensible nature enables to add generics in an efficient way [KRRvW17],[TMD10, slide 39]. *EmbeddedMontiArc* uses *MontiArc*'s extension concepts for components.

Most languages shown in Table 3.1 do not have matrix support at all (M7). Languages having only basic matrix support via a library without hardware support, and languages providing no matrix access features, as *MATLAB* provides them, are listed with a dash (no support) in the table; *MARTE* [OMG08, p. 44], providing only integer matrices, or *Ada* are such cases. Some languages support matrices and their operations in a dedicated way (marked with a `p` in the table). For example, some tools have powerful libraries with hardware support to improve calculations or some languages have built-in mechanism for matrix operators without a typing concept. Languages with partially matrix support are: *AADL* (ArcheOpterix uses matrices for network interactions [ABGM09]), *LabView* (it has matrix support, but “you cannot limit the size of a matrix to a fixed number of elements” [Nat17a]), *Simulink* with *MATLAB* [The18k, Chapter 1, Chapter 2, Chapter 4] (provides special matrix operators[2] and easy matrix access, but *MATLAB*[3] has no type system for matrices), and *SystemC* (Mat-Core extension [SAJ09] maps matrix operations to special hardware instructions of chips). *EmbeddedMontiArc*, Modelica, and *SCADE* are the only three languages having full matrix support with a type system and hardware acceleration. *Modelica* has the Matrices library [Wat18]; it offers high-level matrix support (incl. matrix dimensions) mapped to native instructions via *LAPACK* [Uni18b]. Scade also offers matrix operations and, additionally, it provides array data types (e.g., `int^4`) to define matrix dimensions [EA15, slide 51], [Est14]. *EmbeddedMontiArc* has the most powerful matrix type system, because it supports besides matrix dimensions also the specification of domains for matrix values and algebraic matrix properties (e.g., diagonal matrix). *EmbeddedMontiArc* uses all matrix operators of *MATLAB* (e.g., backslash, and element-wise operators). Similar to *Modelica*, *EmbeddedMontiArc* uses the Armadillo library [SC16] (based on *LAPACK*) to get access to native chip instructions. The algebraic matrix types together with the *LAPACK* backend enables faster execution of matrix operations in *EmbeddedMontiArc* than executing them in *Modelica* or *Simulink* [KRSvW18a].

Nearly half of the investigated languages or tools provide (full or partial) mechanisms for testing (M8). The following languages have full support of test-driven development: *AADL* (via the COMPASS project [vS13]), *Ada* (with unit and integration test framework VectorCAST/ADA [Vec18]), *ArchJava* (with Java test frameworks such as ArchUnit [Arc18] or JUnit [MH03]), *AutoFOCUS* 3 (via simulation tests), *AUTOSAR* (supports functional safety tests [AUT16], e.g., core or ram tests), *LabView* (has its own NI *LabView* Unit Test Framework Toolkit [Nat17b]), *Modelica* (commercial UnitTesting library offered by Emmeskay [TK06]), *MontiArc* (blackbox stream unit testing [Hab16]), *SystemC* (SCV - *SystemC* Verification library [BDBK10, acc18]), *Verilog* (*Verilog* Testbenches [CG14]), and *VHDL* (similar to *Verilog*). *EmbeddedMontiArc* uses the stream test mechanisms of *MontiArc* [Hab16, Section 6.4.1], [Sof16]. Partial support for test-driven development have the following tools: *Ptolemy* uses Java to instantiate and thus also test the abstract syntax [Lee13], but “Vergil does not provide means for automated test executions” [Hab16]. “*Rapide* toolset supports testing for interface conformance by both compile time and runtime checking” [Luc96], but not for behavior [Luc96]. *Simulink* enables to create

[2] such as backslash for solving linear equations

[3] the behavior language for atomic *Simulink* subsytems

input signals using the `Signal Builder` block [The18k, p. 61-124]. However, in *Simulink* exists no convenient way to define the expected output, unless you compare each calculated result again against a `Signal Builder` value.

Nearly one third of all investigated languages support product-line (M9) modeling, also called feature or variant modeling, completely : *AADL* (languages has `features` keyword [Fei05]), *ArchJava* (connection patterns enable variance for components and their interactions [PNR04]), *AUTOSAR* (defines a feature model exchange format [AUT17, p. 11]), *Koala* (via the language extension Koalish [ASM04]), *LabView* (built-in variants manager [Gar12]), *MontiArc* (via the language extensions *Delta-MontiArc* [MNR+13, HKR+11a, HKR+11b], or *MontiArc*^{HV} [HRR+11]), *Simulink* (variability bindings over model references [LEK13]), *SysML* (variation points [Wei12a]), and *UniCon* (via variant property [Zel94]). The following languages support partly variance modeling: *ASCET* (has interfaces and binding points to existing product-line modeling tools such as dSpace), *LabView* (can be coupled with *EAST-ADL*'s product line support), *Modelica* (replaceable classes and interfaces serve as a plug-in mechanism for product-lines [Mod17, Chapter 6]), *SystemC* (supports variants via the `#ifdef` C preprocessor mechanism [KAT+09]), and *xADL* (does not support product-line modeling natively, but it can be easily added [FG07]). *EmbeddedMontiArc* supports partial product-line modeling via configuration parameters of component types (cf. Subsection 3.6.5). Additionally, *EmbeddedMontiArc* supports conceptually the delta mechanism of *MontiArc* as presented in Section 3.7, but it is not implemented yet. *EmbeddedMontiView*, the high-level design specification language of *EmbeddedMontiArc*, has no concept how to deal with product-lines; even after this thesis, there is still some research on C&C view language features needed.

Component and Connector models describe the architectural and structural decomposition. Since the tasks of atomic components in embedded or cyber-physical systems vary, the language should provide means to embed different behavior models such as automata, matrix operations, differential equations, or neuronal nets. Languages with such an behavior embedding mechanism (M10) are *MontiArc* (see behavior description extension point [Hab16, requirement LRQ3.2]), *Ptolemy* ("*Ptolemy* II supports several, and can be extended with new models of computation" [Ber18] via extension points of the core infrastructure [Lee04, slide 05:26]). *EmbeddedMontiArc* reuses the language extension mechanism of *MontiArc*. In contrast to *MontiArc* only providing Java (cf. *AJava*) and automata (cf. *MontiArcAutomaton*) for behavior, *EmbeddedMontiArc* provides already a large family to describe behavior. Examples of behavioral languages in *EmbeddedMontiArc* are automata (reused from *MontiArc*), *MontiMath* (typed *MATLAB*), *MontiMathOpt* (math plus non-linear optimization problems), *CNNArch* (convolutional networks for deep learning), and *OCL* (for logical declarative description of components similar). This thesis describes only the basic *EmbeddedMontiArc* language; thus, it does not explain any language containing an implementation. Table 3.1 marks with partial (`p`) all languages that support multiple behavior descriptions without being extendable to new behavior languages.

Requirement (M11), advanced search, is to our best knowledge only supported by *EmbeddedMontiArc*. The *EmbeddedMontiView* design language can also be used to search for components in a very intuitive way. Search examples are: (i) find all components of a given component type that have the flip flop component as parent, (ii) find all components that are connected with the speed control component, or (iii) find all input ports that have effect to the output port acceleration in the tempomat component. The crosscutting nature of C&C views enables to search between different hierarchies without writing scripts using the data structure of *EmbeddedMontiArc*.

Table 3.1 contains a check symbol when the language provides a way to add information (e.g., extra-functional properties) to models (M12). If the language or tool supports only comments, then this requirement is not satisfied. In contrast, if the language provides only untyped stereotypes as a key-value list, then the requirement is satisfied; even though this is not an elegant way how to add information. Languages not using stereotypes to enrich information are *AADL* (valued typed attributes [Ins15, slide 5]), *ACME* (it supports to add new attributes [MT00]), *AUTOSAR* (based on *UML* profiles [AUT06]), *SCADE* (imports information from *SysML* diagrams [LSLGG$^+$11]), and *xADL* (extension of the architecture via the "xADL Way" [Uni18a]). *EmbeddedMontiArc* uses a tagging mechanism (cf. Chapter 5) similar to *AADL*'s valued types.

3.3. Comparison to Other *MontiArc* Derivatives

MontiArc is the base language of *EmbeddedMontiArc*, even though *EmbeddedMontiArc* incorporated also many features of other languages (cf. Section 3.2). This section compares *EmbeddedMontiArc* with the other languages (technologically or conceptually) derived from *MontiArc* [BHH$^+$17], [Hab16, p. 257]. This list is not complete.

- ***MontiArc*** [Hab16]
  Besides the language feature differences shown in the section before, *MontiArc* uses dynamic scheduling so that "different component timing domains can be combined with each other" [Hab16, p. 85]. Both, *MontiArc* and *EmbeddedMontiArc* separate timing slots by abstract ticks. *MontiArc* supports both strong and weak-causality. In contrast, *EmbeddedMontiArc* uses only weak-causality where tick-delays must be explicitly modeled (e.g., via the `UnitDelay` component). In contrast to *MontiArc* using asynchronous communication, *EmbeddedMontiArc* uses a time-synchronous approach processing exactly one value (e.g., number, matrix, or struct object) in one time slot (between two ticks). The result is, that *MontiArc* uses a runtime environment which does the scheduling of components. "To simulate logical distributed and concurrent components in a single thread, an explicit scheduling is needed. The scheduler is responsible for message processing and the simulation of time." [Hab16, p. 96] The *MontiArc* runtime scheduling is needed due to the different simulation modes of component and ports (e.g., tickfree ports, blocked ports, instant components, delayed components, untimed components, and causal synchronous communication).

  In contrast, *EmbeddedMontiArc*'s restrictions with one value for each time slot facilitates the generator to analyze and optimize the complete C&C structure at compile time. Thus, *EmbeddedMontiArc*'s generated C++ code (*MontiArc* generates Java code) contains the complete scheduling information. The *EmbeddedMontiArc* generator works similar to the *Simulink* code generator, also first analyzing the dependencies (cf. `slist` [The18i] and `elist` [The18c] commands), then producing optimized code. The "simple" (compared to *MontiArc*'s scheduling options) nature of *EmbeddedMontiArc*'s scheduling enables mapping the behavior of *EmbeddedMontiArc* (with its *MontiMath* implementation for atomic components) to input output extended finite automata [RSvW+15]. This formal interpretation of *EmbeddedMontiArc* models are the theoretical foundation for many behavioral validations [RSvW+15, RRS+16, HRvW17, Tol16]. Examples of such validations are: component backward compatible checks to its previous version, detect duplicated models, find dead paths, effect chain analysis (how many time steps of an output signal are influenced by a change in an input signal), and detect over or under-flow.

The relatively simple scheduling of *EmbeddedMontiArc*, compared to the one of *MontiArc*, enables to generate optimal multi-threaded C++ code [KRSvW18a]. *EmbeddedMontiArc*'s C++ compiler toolchain is highly optimized by using BLAS libraries [KRSvW18a] to speed up the runtime of *EmbeddedMontiArc* models dramatically. Therefore, the execution of computationally intensive C++ code generated by *EmbeddedMontiArc* runs in seconds; whereas similar code executed by the JVM crashes due to memory problems or needs about 10 minutes of execution time (cf. case study [KRSvW18a, KRSvW18]). In classical embedded domains, simulators or microcontroller processors often execute functional models in loops with different input data; e.g., continuous image detection and steering correction. Therefore, a fast execution of logical models of controllers decreases the time to test or simulate these controllers dramatically. In contrast, simulators for some non-embedded domains may not frequently update the input data of many components (e.g., when depending on user inputs). Therefore, the scheduler may skip the execution of most components in a simulation; in such cases the dynamic scheduling of *MontiArc* might have performance advantages.

EmbeddedMontiArc executes all components in every time step. This means (also in contrast to *MontiArc*'s scheduling mechanism) the worst-case execution for a given hardware can be estimated a priori. This is very important for worst-case execution time analysis of embedded systems.

Another difference between *MontiArc* and *EmbeddedMontiArc* is that *EmbeddedMontiArc* does not support "dirty" (not side-effect-free) components such as the `ACCSystem` model defined in *MontiArc* [Hab16, p. 253]. Prohibiting "dirty" components ensures that also the most high-level component can be black-box tested with the stream language.[4] Many other modeling languages derived from *MontiArc* also have the "dirty component illness".

- ***AJava*** [HRR10]
  "*MontiArc* does not include a language that allows the implementation of behavior within components, the behavior has to be implemented externally in Java" [Hab16, p. 160]. Therefore, the modeler using *MontiArc* must understand how to add handwritten code to an atomic component. For small atomic components such as simple mathematical expressions, this is rather cumbersome. Thus, *AJava* addresses this issues by embedding the *MontiCore* JavaDSL [SE18] language into *MontiArc*. This way the behavior of atomic components can be directly described in the component definition file. *EmbeddedMontiArcMath* uses a similar approach to embed *MontiMath* into *EmbeddedMontiArc* [KRRvW17]. The advantage of *AJava* is that it can use all JVM libraries.

[4]Since the `ACCSystem` component in *MontiArc* has neither input nor output ports, a system test is not possible.

SIStructs

```
// emulation of ArrayList as port type
struct ArrayListBounded<T, N1 maxNbOfElements> {
  T data[maxNbOfElements];
  (1: maxNbOfElements) length;
}
```

Figure 3.2.: Code how to emulate `ArrayList` in *EmbeddedMontiArc*.

EmbeddedMontiArc supports using external libraries, this way via JNI (Java Native Interface) the C++ code can also invoke Java libraries. But due to the nature of *EmbeddedMontiArc*, e.g., that the array size is fixed at runtime, no complete dynamic `ArrayList` can be passed between JVM and *EmbeddedMontiArc*. If the upper bound for the number of elements in an `ArrayList` or any similar object (e.g., collections) is well known, then the modeler can use the data `ArrayListBounded` data type shown in Figure 3.2.

Similar to Java, *AJava* is more suited for object oriented problems; and similar to *MATLAB*, *EmbeddedMontiArcMath* is more suited for mathematical and matrix based problems.

- ***clArc/cloudADL*** [NPR13]
  clArc is designed for model based development of cloud applications. *clArc* uses port groups (see Figure 3.3) to specify that all ports in a group belong semantically together and are executed at the same time. In *EmbeddedMontiArc* data, which belongs semantically together, is encapsulated in data structures (cf. *SIStructs* language in Section 3.4). Since in *EmbeddedMontiArc* all ports receive their values at the same time, no special group semantics for this case is necessary. Similar to *EmbeddedMontiArc* component instances can be replicated, but in *clArc* they have no instance limit and the number of instances can vary (according to the request number or other runtime parameters) during runtime. In contrast to *clArc*, *EmbeddedMontiArc* defines exactly the number of component instances at generate/compile time. In contrast to *clArc*, *EmbeddedMontiArc* does not need routing for messages of newly created components. This is the case, because simulators of *EmbeddedMontiArc* execute the entire system in every time step and these simulators know the execution time of the entire system when compiling the models. In *clArc* "Message channels attached to replicating components guarantee that every message is received by exactly one replica" [NPR13]. This means components can also gain empty input in *clArc*, if more components are present than messages in the message channel. Contexts in *clArc* address ambiguities when connecting replica of different components with each other by defining rules (e.g., based on session IDs) how to connect instances with different component types.

  clArc and *EmbeddedMontiArc* complete each other: *EmbeddedMontiArc* models the behavior of one self-driving car (the logical behavior for one car hardware). *clArc* models how this one car (which is a black-box component in *clArc*) is replicated and how these cars (dynamic number of cars) interact with each other. This enables to model dynamic local traffic systems.

clarc

```
component UserManagement {
  port group UserData in User usr, in UpdateRequest req;
  component UpdateStore store [*];
  connect usr -> store.user;
  connect req -> store.requrest;
  service required clarc.db.NoSQL;
}
```

Figure 3.3.: *clArc* user management system (copied from [BHH+17]).

MontiSecArc

```
component CashDeskLine {
  port out PayMentRequest; // to Bank
  component CashDeskUI ui {
    port out Sale;
  }
  component CardReader reader {
    port out CardHolderData;
  }
  component CashDesk cashDesk {
    port in CardHolderData,
         in Sale,
    port out PaymentRequest;
    trustlevel +1;
    accesscontrol on;
  }
  identity weak ui -> cashDesk;
  connect ui.sale -> cashDesk.sale;
  connect encrypted reader.cardHolderData -> cashDesk.cardHolderData;
  connect encrypted cashDesk.paymentRequest -> paymentRequest;
}
```

Figure 3.4.: *MontiSecArc* architecture of cash desk line in supermarkets (copied from [BHH+17]).

- ***MontiSecArc*** [BHH+17]
  MontiSecArc extends *MontiArc* by adding security information to architectural models. An example is shown in Figure 3.4. *MontiSecArc* enriches the textual syntax directly with security information. *EmbeddedMontiArc* has a powerful tagging mechanism to enrich models with different extra-functional properties (e.g., security) via different tagging schemata (cf. Chapter 5). This supports a better separation of concerns and, additionally, the same logical architecture can be tagged with different security features (e.g., for different deployments).
- ***MontiArcAutomaton*** [RRW12, RRW13a, RRW13b, RRW14, RRRW15, RRW16, HRW16, BRW16, Wor16, HKR+16, BKRW17, BEK+18]
  MontiArcAutomaton embeds input/output automata into *MontiArc* to describe the behavior of atomic components. The *EmbeddedMontiArc* modeling language family also embeds input/output automata for behavior modeling, whereby the input/output automata version in *EmbeddedMontiArc* uses the unit-based type system to specify `velocity < 4 km/h` in guard conditions, whereby *MontiArcAutomaton* uses the Java type system. Similar to

MontiArc^{HV}

```
1  component WindowSystem {
2   port
3     in WinderRequest driverRequest,
4     in WinderRequest coDriverRequest,
5     out WindowStatus;
6
7   component WindowWinder driverWinder,
8                          coDriverWinder;
9
10  component WindowWatchDog {
11    port
12      in WindowStatus driverStat,
13      in WindowStatus coDriverStat,
14      out WindowStatus overallStat;
15
16    variationPoint: MoreWindowsDog [0..1];
17  }
18
19  connect driverRequest ->
20      driverWinder.driverRequest,
21      driverWinder.passengerRequest,
22      coDriverWinder.driverRequest;
23  connect coDriverRequest ->
24      coDriverWinder.passengerRequest;
25  connect driverWinder.WindowStatus ->
26      WindowWatchDog.driverStat;
27  connect coDriverWinder.WindowStatus ->
28      WindowWatchDog.coDriverStat;
29  connect WindowWatchDog.overallStatus ->
30      WindowStatus;
31
32  variationPoint: MoreWindows [0..1];
33 }
```

MontiArc^{HV}

```
34 variant FourWindows realizes
35               WindowSystem.MoreWindows {
36   port
37    in WinderRequest rearLeftRequest,
38    in WinderRequest rearRightRequest;
39
40   component WindowWinder rearLeft,
41                          rearRight;
42
43  connect driverRequest ->
44      rearLeft.driverRequest,
45      rearRight.driverRequest;
46  connect rearLeftRequest ->
47      rearLeft.passengerRequest;
48  connect rearRightRequest ->
49      rearRight.passengerRequest;
50  connect rearLeft.WindowStatus ->
51      WindowWatchDog.rearLeftStat;
52  connect rearRight.WindowStatus ->
53      WindowWatchDog.rearRightStat;
54
55  WindowSystem.WindowWatchDog.MoreWindowsDog
56     realizedBy FourWindowsDog;
57  }
```

MontiArc^{HV}

```
58 variantConfig FourWindowSystem for
59        WindowSystem {
60    WindowSystem.MoreWindows
61        realizedBy FourWindows;
62 }
```

Figure 3.5.: Four window system modelled via *MontiArc* ^{HV} (copied from [HRR+11]).

MontiArcAutomaton's controlled underspecification [RRW16], *EmbeddedMontiArc* language family has two input/output automata versions: One deterministic one for behavior implementations which is directly embedded into *EmbeddedMontiArc*. Another version to specify the behavior of components; this version is non-deterministic. For example, non-determinism enables that output assignments must not have a concrete value as well as that conditions may satisfy multiple guards in an implementation. Bounded model checking between specification and implementation automata can be used to verify the behavioral correctness of an implementation [HRvW17]. Since this thesis focuses on the structural part of the *EmbeddedMontiArc* family, these two automata languages are not part of this thesis.

- ***MontiArc^{HV}*** [HRR+11]
  The HV in *MontiArc^{HV}* stands for hierarchical variability modeling "which supports specifying component variability integrated with the component hierarchy and locally to the components" [HRR+11].

  An example model of *MontiArc^{HV}* is given in Figure 3.5. The code (except of ll. 16, 32) in the left part of Figure 3.5 is identical to the normal *MontiArc* language describing a `WindowSystem` being decomposed of three subcomponents `dirverWinder` (l. 7), `coDriverWinder` (l. 8), and `WindowWatchDog` (l. 10, it is automatically instantiated). Both variation points are optional; thus an empty realization of the `WindowSystem` creates a software component with two electric power windows in front. If the `MoreWindows` variation point is realized with the variant shown on the top right part in Figure 3.5, the `WindowSystem` has four electric power windows: two in the front and two in the back. The solution of *MontiArc*^{HV} destroys the encapsulation nature of C&C models as variants replace (line by line) the variation point. The variant `FourWindows` realizes `MoreWindows`, but it accesses the `driverRequest` port of the `WindowSystem` component (cf. l. 43).

The port and component array concept of *EmbeddedMontiArc* (introduced in Subsection 3.6.2) addresses this product-line problem much more intuitive and in a much more generic way. A generic parameter in *EmbeddedMontiArc*, let's call it `N1 nbElectricalWindows`, for the component type definition `WindowSystem` creates this product-line. Creating a variant is very easy by binding this generic parameter. For example, `instance WindowSystem<2> electricFrontWindows` and `instance WindowSystem<4> electricFrontAndBackWindows` creates this two variants mentioned above. This over 60 lines of *MontiArc*^{HV} code can be modeled in *EmbeddedMontiArc* with about 20 lines (cf. Figure 3.6). For a bus having 15 rows each with an electric power window, the line savings in *EmbeddedMontiArc* is even higher, because `nbElectricalWindows` must be only bound to 30 and there is no need to write an extra line. Since the driver front window plays an important role in this product-line, *EmbeddedMontiArc* also supports partial enumerations as generic types. Therefore, Section 3.5.4 elucidates on a concrete example how partial enumerations increase the readability by keeping the generality of the here presented approach.

More complex product-line modeling in *EmbeddedMontiArc* is possible via component interfaces in combination with configuration parameters and arrays. Subsection 3.6.5 presents a product-line example.

3.4. Overview of *EmbeddedMontiArc* Modeling Family

This section presents the most important languages of the *EmbeddedMontiArc* language family shown in Figure 3.7. *MontiArc* is not part of the *EmbeddedMontiArc* language family, because the port type system of *MontiArc* is based on the Java type system (cf. Section 3.3) and all languages of the *EmbeddedMontiArc* language family have a SI unit based port type system. However, the *EmbeddedMontiArc* language borrows many language concepts from *MontiArc*; e.g., the concrete and abstract syntax to model components and connectors.

This section also contains hints how to realize some of these languages in *MontiCore* 5.

The base language (omitting all *MontiCore* commons languages) for the *EmbeddedMontiArc* family is *SIUnit*. This language defines all kinds of numbers, i.e., complex numbers such as `0.5 + 3i`, numbers with units such as `5 m/s^2` or `-30.4°C`, and normal numbers without units such as `7` or `-0.3`.

Special about the *SIUnit* grammar in contrast to most other *MontiCore* grammars is that it uses semantic predicates to define all alpha-numeric tokens. Line 6 in Figure 3.8 shows the definition of the imaginary sign using the existing `Name` token (defined in the basic *MontiCore* grammars) together with the semantic predicate (the italic text in Figure 3.8). If the *SIUnit* grammar would be used standalone (no other grammar would embed or extend this grammar), then the rule in

EMA

```
component WindowSystem<N1 nbElectricalWindows> {
  ports
    in WinderRequest driverRequest,
    in WinderRequest coDriverRequest[nbElectricalWindows - 1],
    out WindowStatus windowStatus;

  instance WindowWinder winders[nbElectricalWindows];
  instance WindowWatchDog<nbEletricalWindows> watchDog;

  // connects all winders instances
  connect driverRequest -> winders[:].driverRequest;

  connect driverRequest -> winders[1].passengerRequest;
  connect coDriverRequest[1: nbElectricalWindows - 1] ->
      winders[2: nbElectricalWindows].passengerRequest;
  connect winders[:].windowStatus -> watchDog.windowStatus[:];
  connect watchDog.overallStatus -> windowStatus;
}
```

EMA

```
component WindowWatchDog<N1 nbElectricalWindows> {
  ports
    in WindowStatus windowStatus[nbElectricalWindows],
    out WindowStatus overallStatus;
}
```

Figure 3.6.: Four window system of Figure 3.5 modeled in *EmbeddedMontiArc*.

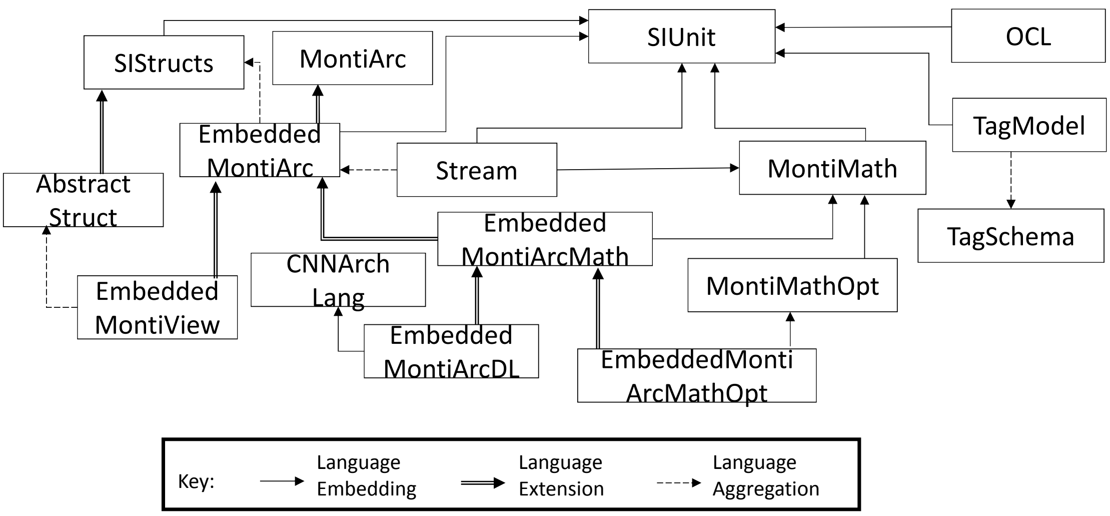


Figure 3.7.: *EmbeddedMontiArc* language family (inspired by [KRRvW17]).

line 6 is equivalent to `I = "i"`. However, the expression `"i"` introduces an extra lexer token resulting that no `Name` token will ever recognize the variable name `i` again. This is similar to

MC5

```
ComplexNumber =
   (negRe:"-")? real:NumericLiteral ("+" | negIm:"-") im:NumericLiteral I
;

// use Name instead of i, otherwise no variable can be named i again
I = { _input.LT(1).getText().equals("i") }? Name;
```

Semantic predicate *Use existing token 'Name'*

Figure 3.8.: Excerpt of *SIUnit* grammar for *MontiCore* 5.

SIStructs

```
struct GPS {  (-90° :0.001°:90°)   latitude;
              (-180°:0.001°:180°)  longitude; }
```

Embedding of SIUnit syntax inside struct grammar

Figure 3.9.: Example model of *SIStructs* language.

most existing programming languages such as Java where variables must differ from keywords such as `for` or `if`.

Since the *SIUnit* grammar contains all units together with their prefixes, it would introduce tokens for nearly every single-letter variable name[5]. Introducing all these tokens would result in many "token clashes" when combining the *SIUnit* grammar with other grammars. Therefore, the complete *SIUnit* grammar introduces no alpha-numeric tokens; it uses for units or unit prefixes the same approach with semantic predicate plus `Name` or `Literal` token as shown in Figure 3.8 for the imaginary sign in complex numbers.

SIStructs is a language similar to C structures to encapsulate data. *SIStructs* embeds the *SIUnit* language to reuse numbers with units in the type definition of single elements in one structure; cf. underlined numbers in Figure 3.9. Figure 3.9 shows the `GPS` structure model of the *SIStructs* language. The `GPS` structure encapsulates the two elements: `latitude` and `longitude`. The `latitude` attribute accepts values from minus 90° up to plus 90° with a resolution of 0.001°. Therefore, `-89.999°` is a valid number for `latitude`. However, `89.9989°` is invalid, violating the resolution, and `100°` is invalid, violating the range. The `longitude` attribute accepts values from minus 180° up to plus 180° having the same resolution as `latitude`. The check whether a value is a valid element of a SI unit type is implemented as a context condition; this check is similar to the type compatibility check defined in Figure 4.12.

MontiMath is a typed matrix language inspired by *MATLAB* to avoid runtime errors due to matrix (numbers are interpreted as 1×1 matrix) incompatibilities. *MontiMath* also embeds *SIUnit* to create matrices with units. For example, `(0m:10m)^{1,10} distance` defines a row

[5]Examples of single-letter units or unit prefixes are: a (are), A (ampere), b (barn), c (centi, unit prefix), d (deci, unit prefix), e (exa, unit prefix), f (femto, unit prefix), g (gram), G (Giga, unit prefix), h (hour and hector, unit prefix), J (joule), k (kilo, unit prefix), K (kelvin), l (liter), m (meter and milli, unit prefix), M (mega), n (nano, unit prefix), N (newton), p (pico, unit prefix), P (peta, unit prefix), R (roentgen), s (second), T (tera, unit prefix), U (rack unit), V (volt), W (watt), y (yocto, unit prefix), Y (yotta, unit prefix), z (zepto, unit prefix), and Z (zetta, unit prefix) [GG18].

result type   left operand type   right operand type

MontiMath

```
1  operator<N1 n> diag Q^{n,n} y = diag Q^{n,n} a + diag Q^{n,n} b
2    y = diag(diag(a) + diag(b));
3  end
```

diag with vector input behaves different than diag with matrix (at least two rows) input

MontiMath

```
4  function<N1 n> diag Q^{n,n} y = diag(Q^{1,n} a)
5    y = zeros(n, n);
6    for i = 1:n
7      y(i,i) = a(i);
8    end
9  end
```

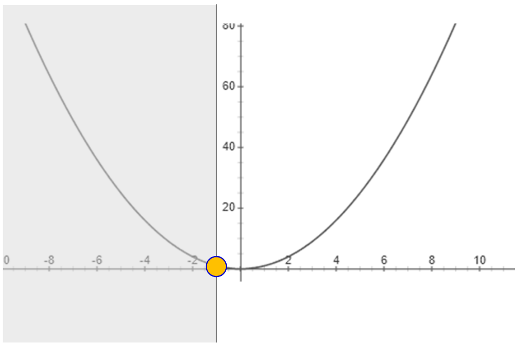
MontiMath

```
10 function<(2:oo) m, N1 n> Q^{1, min(m,n)} y = diag(Q^{m,n} a)
11   for i = 1 : min(m,n)
12     y(i) = a(i,i);
13   end
14 end
```

Figure 3.10.: Example how to overload operators and functions in *MontiMath*.

vector of length 10, where each element of the vector is between 0m and 10m. On the other side, `Q^10`, which represents $\mathbb{Q}^{10\times1}$ and which is a short-form of `(-oo, oo)^{10,1}`, defines a column vector. The expression `diag inv (0:1)^{10, 10} facMatrix` defines a diagonal and invertible 10×10 rational matrix, whose elements are between 0 and 1. Similar to MATLAB, *MontiMath* supports matrix operations such as matrix addition (+), matrix subtraction (−), matrix multiplication (`*`), element-wise multiplication (`.*`), right matrix division (`B/A`, solves $xA = B$ for x), left matrix division (`B\A`, solves $Ax = B$ for x), element-wise division (`./`), element-wise power (`.^`), matrix power (`^`), and matrix modulo (`mod`). The type system of *MontiMath* (cf. symbol table part in Subsection 1.1.3) enables one to overload matrix operators. This provides more efficient calculations for special matrix types.

Figure 3.10 shows an example how to overload matrix functions and operators. Lines 1 to 3 offer a more efficient way to add matrices when both of them are diagonal ones. Using this overloaded operator reduces the algorithmic complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ for adding two $n \times n$ matrices. For practical reasons such as existing documentation and the high prevalence of *MATLAB*, *MontiMath* tries to be compatible as much as possible (modulo type system) to *MATLAB* functions. Therefore, the diagonal function (cf. *MATLAB* documentation [The18b]) behaves different for a vector (cf. ll. 4-9) or a matrix (cf. ll. 10-14) as input parameter. Due to the type system of *MontiMath*, the function signature unveils this difference; in *MATLAB* you need to study the documentation or understand the implementation of a function to detect this difference. Line 2 does not bind any generic value to the `diag` function, because the type inference algorithm can derive the value of the generic parameter. *MontiMath*'s type inference is similar to Java's one. The short-form used in line 2 increases the readability compared to the, also possible, long-form `y = diag<n>( diag<n,n>(a) + diag<n,n>(b) )`.



MontiMathOpt

```
1  script MinimizeXSquare
2      minimize( Q x)
3          Q y = x^2;
4      subject to
5          x <= -1;
6      end
7  end
```

Figure 3.11.: Example model of the *MontiMathOpt* language.

SIStructs

```
1  struct GPS {  (-90° :0.001°:90°)   latitude;
2                (-180°:0.001°:180°) longitude; }
```

Aggregation of languages via Symbols

EMA

```
3  component AutomatedVehicle {
4    ports in   GPS posCar,
                     port type          port name   port array size
5               (0.01m:0.01m:4.2m) distance[10],…
6          out  (0N:1N:200kN) brakeForce[4];
7  }
```

Figure 3.12.: Example model of *EmbeddedMontiArc* language.

MontiMathOpt embeds the *MontiMath* language to reuse all mathematical statements including matrix operations. Additionally, the *MontiMathOpt* language adds support for optimization equations, i.e., minimization and maximization problems. Modern control theory models the behavior of concrete car controllers as minimization problems; e.g., control the steering of a car so that the mean squared error of the calculated trajectory according to the new set steering angle against the optimal (wished) trajectory is minimal. The `subject to` equations in modern control theory model environment restrictions to the steering angle; e.g., the derivation of the steering angle must be in a specific range so that the car does not flip.

Figure 3.11 shows a simple script of the *MontiMathOpt* language. The underlined text lines (cf. ll. 3 and 5) highlight the embedded syntax of *MontiMath*. The gray filled square on the left side represents the valid solution area satisfying the one `subject to` constraint in line 5. The yellow circle on the left side (x = 1) is the solution of this simple minimization script.

Since the rest of this chapter presents the *EmbeddedMontiArc* language in detail, Figure 3.12 is only used to show that the language also embeds the *SIUnit* language (cf. italic numbers `0.01m`, `4.2m`, `6N`, etc. in ll. 5-6) and aggregates the *SIStructs* language via the symbol table (cf. Section 4.6) by resolving port type names (cf. solid underlined `GPS` name).

Figure 3.13 shows an example of the stream language (cf. ll. 6-10). The stream language embeds *MontiMath* (cf. italic text in ll. 7-9) to specify matrices. Line 7 uses the *MATLAB* colon operator to define the two 1×3 matrices `[1,2,3]` and `[4,6,8]`; in contrast to *MontiArc*

"1:3" is short-form of matrix "[1, 2, 3]" and
"4:2:8" is short-form of matrix "[4, 6, 8]"

EMA
```
1  component SumVec<N1 n> {
2    ports in  Z^n summand1,
3          in  Z^n summand2,
4          out Z^n sum;
5  }
```

Stream
```
6  stream SumVec3Test1  for SumVec<3> {
7    summand1:  1:3          tick  4:2:8;
8    summand2:  [11,13,17]   tick  [-19,23,-29];
9    sum:       [12,15,20]   tick  [-15,29,-21];
10 }
```

Aggregation of languages via symbols

```
4  function<N1 n> diag Q^{n,n} y = diag(Q^{1,n} a)
5    y = zeros(n, n);
6    for i = 1:n
7      y(i,i) = a(i);
8    end
9  end
```

Figure 3.13.: Example model of Stream language.

EMAM
```
1  component SensorFusion <N1 n> ( (-90°:90°)^{1,n} tilt ){
2    ports in   (0m:0.2m:10m)  distance[n],
3          out  (0m:0.2m:10m) mergedDistance;
```
component behavior implementation is specified using the MontiMath Language (syntax)
```
4    implementation Math {
5*     (0m:10m)^{1,n} distance;
6*     (0m:10m)^{1,1} mergedDistance;
7      diag (0:1)^{n,n} factorMatrix = diag(cos*(tilt));
8      mergedDistance = min(distance * factorMatrix);
9    }
10 }
```
How ports are visible in Math language
after applying Math-EMA-Adapter

* Lines do not belong to original source code. They are only inserted in this listing to see how port types are handled as matrix types in the embedded MontiMath language.

Figure 3.14.: Example model of *EmbeddedMontiArcMath* language (copied from [KRRvW17]).

and *Focus*, the syntax `1:3` does *not* represent a stream with the two values `1` and `3`. The stream language aggregates the symbols of *EmbeddedMontiArc* (cf. solid underlined names in Figure 3.13). As already explained in Section 3.3, *EmbeddedMontiArc* processes exactly one value (number or matrix) per time slot. The `tick` keyword in lines 7 to 9 separates two time slots (also called execution cycles). The stream language enables model-based black box testing of *EmbeddedMontiArc* models. Model-based testing abstracts all technical details of the C++ compiler toolchain from the modeler.

EmbeddedMontiArcMath embeds *MontiMath* into *EmbeddedMontiArc* to describe the behavior of atomic components in a convenient way. Figure 3.14 shows the *EmbeddedMontiArcMath* code of an atomic `SensorFusion` component. The component receives as input the distances measured by the single sensors (cf. l. 2) of a back park distronic system and produces as output

EMA

```
1  component AutomatedVehicle {
2    ports in   GPS posCar,
3     in   (0.01m:0.01m:4.2m) distance[10],
4     out  (0N:1N:200kN) brakeForce[4];
5  }
```

EMV

```
6  component AutomatedVehicle {
7     ports in  ? posCar,
8               Length ?,
9           out ? ?[4];
10 }
```

overwrite Port rule to add underspecification

EmbeddedMontiArc.mc5

```
11 Port =
12   ( incoming:["in"] |
13     outgoing:["out"] )
14   Type Name
15   ( "[" Number "]" )?
16 ;
```

EmbeddedMontiView.mc5

```
17 Port =
18   ( incoming:["in"] |
19     outgoing:["out"] )
20   (Type | "?") (Name | "?")
21   ( "[" Number "]" )?
22 ;
```

Figure 3.15.: Example model of *EmbeddedMontiArc* and *EmbeddedMontiView* (top), as well as an example excerpt how *EmbeddedMontiView* extends the *EmbeddedMontiArc* grammar (bottom).

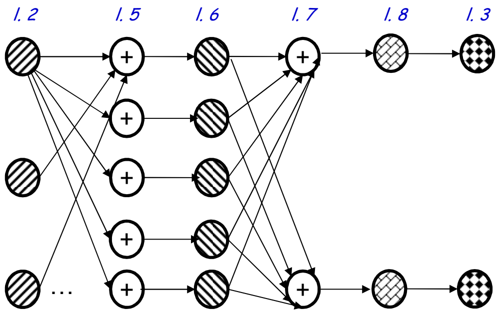
the distance to the obstacle (cf. l. 3). The `tilt` configuration parameter (cf. l. 1) is the tilt of the sensors in the back bumper. Lines 5 and 6 show how the embedded *MontiMath* language sees the variables introduced by the ports in *EmbeddedMontiArc*; both lines do not belong to the *EmbeddedMontiArcMath* model. The *MontiMath* language does not need to know anything about port arrays, because the port array is adapted (cf. Section 4.6) to a matrix variable. This enables reusing all of the type inference and check rules of *MontiMath* in *EmbeddedMontiArcMath*. The star after the cosine function in line 7 states that this vector function applies the normal cosine element-wise on the input vector. This way exists a distinction between functions known from school mathematics (without *) and new vector-based ones (with *).

EmbeddedMontiArcMathOpt extends *EmbeddedMontiArcMath*. *EmbeddedMontiArcMathOpt* does not extend *EmbeddedMontiArc* directly to reuse the context conditions of *EmbeddedMontiArcMath*; e.g., that variables transformed to an output port must be assigned at least once.

The top part in Figure 3.15 shows an *EmbeddedMontiArc* (cf. l. 1-5) and one corresponding *EmbeddedMontiView* (cf. l. 6-10) model. The *EmbeddedMontiView* model contains underspecification (cf. the underlined text parts in ll. 7-9); e.g., the unknown[6] data type of the `posCar` input port, the unknown name of the second input port, and the unknown data type and name of the output port. The bottom part in Figure 3.15 shows an excerpt how *EmbeddedMontiView* language extends the *EmbeddedMontiArc* one, e.g., by overwriting the `Port` rule to specify besides concrete types or concrete names also question mark signs for types or names. Chapter 7 introduces the *EmbeddedMontiView* language to specify incomplete C&C design models in detail.

The right side in Figure 3.16 shows the concrete syntax of the *CNNArch* language. Evgeny Kusmenko designed this language. The left side shows the graphical convolutional neuronal network of the textual syntax. The network contains 6 layers (cf. ll. 4-9) represented as vertical nodes in the graphic. Each layer consists of a different number of nodes: layer 1 has three nodes

[6]The `?` symbol represents not specified data types or port names.



CNNArch

```
1  architecture SimpleNetworkRelu {
2      def input Q^3 in
3      def output (0:oo)^2 out

4      in ->
5      FullyConnected(units=5, no_bias=true) ->
6      Tanh() ->
7      FullyConnected(units=2, no_bias=true) ->
8      Relu() ->
9      out
10 }
```

Figure 3.16.: Example model of the *CNNArch* language.

Aggregation of languages via symbols

TagSchema

```
1  tagschema LatencyTagSchema {
2  tagtype Latency:Duration
3    for PortDefinition,
4        PortInstance;    }
```

TagModel

```
5  conforms to LatencyTagSchema;
6  tags Latency for ECU1 {
7    tag vehicleSpeed with
8            Latency = 100 ms; }
```

Figure 3.17.: Example models of *TagSchema* and *TagModel* language.

(cf. `Q^3 in` in l. 2), layer 2 has five nodes (cf. `units=5` in l. 5), layer 3 has also five nodes as `Tanh` function is applied node-wise (cf. l. 6), layer 4 has two nodes (cf. `units=2` in l. 7), layer 5 has also two nodes as `Relu` (cf. l. 8) is also a node-wise function, and layer 6 has two nodes (cf. `Q(0:oo)^2` out in l. 3). The `FullyConnected` keyword connects all nodes from one layer to all nodes of the next layer whereby the value of the next layer is calculated as sum of the values of the source nodes (represented as incoming edges in Figure 3.16).

EmbeddedMontiArcDL embeds the *CNNArch* language into *EmbeddedMontiArc* to express the behavior of atomic components via convolutional neuronal networks. The mechanism is the same as in *EmbeddedMontiArcMath*.

The tag schema model defines new tag kinds. Tag kinds are similar to typed *UML* stereotypes or *UML* profiles. The tagging schema defines what model elements of a language can be tagged with what kind of information. The tag schema example in Figure 3.17 enables to tag port definitions (cf. l. 3) and port instances (cf. l. 4) with latency information (cf. l. 2). Every concrete tag model is conforming to a given tag schema (cf. l. 5). The tag model in Figure 3.17 enriches the `vehicleSpeed` port of the component `ECU1` with a `Latency` value of `100 ms`. The tag model embeds the *SIUnit* language to reuse numbers with units (cf. italic text in l. 7). It is important to know, that the *TagSchema* and the *TagModel* language are language agnostic as both of them work on the general `Symbol` interface (cf. [MSN17]) provided by *MontiCore*. Therefore, every language exporting symbols via the symbol table (cf. Section 4.6) can be enriched with extra information using these both languages. Chapter 5 explains more information about the tagging mechanism based on these two languages.

The *OCL* language also embeds the *SIUnit* language to express physical constraints, e.g., `context Person: small <=> size < 160 cm`. The *EmbeddedMontiArc* family uses *OCL* to express context conditions for *EmbeddedMontiArc* models (cf. Section 6.1) as well as to describe semantic constraints for extra-functional properties (cf. Section 6.2).

This section gave a high-level overview of the most important languages of the *EmbeddedMontiArc* family, including behavioral languages such as *MontiMath*, *MontiMathOpt*, or *CNNArch*. The rest of this thesis will only focus on the languages needed to express structural and extra-functional properties as well as their constraints; i.e., *EmbeddedMontiArc*, *EmbeddedMontiView*, *TagModel*, *TagSchema*, and *OCL*. The next sections explain the *EmbeddedMontiArc* language on many concrete syntax examples.

3.5. Typing in *EmbeddedMontiArc*

This section shortly explains the typing of ports in *EmbeddedMontiArc*. Section 3.4 already showed that *EmbeddedMontiArc* language family is based on a type system with SI unit and matrix support. Subsection 3.5.1 introduces the port type system focusing on SI unit support. Subsection 3.5.2 explains how algebraic properties of matrices are encoded into the port type system. Subsection 3.5.3 and Subsection 3.5.4 explain how values and types of the port type system can be passed via configuration parameters as well as how generic port type parameters can be used as port types to increase modularity. Chapter 4 presents the abstract syntax of the port type system including its type parameters in detail.

3.5.1. Port Type System

This subsection shortly introduces the abstract syntax of the port type system. The abstract syntax helps to understand parameters and how *EmbeddedMontiArc* binds these parameters in the next (sub)sections. Figure 3.18 shows an excerpt of the port type system. Figure 3.18 does not include the `Boolean` type and any encapsulated types such as (nested) structures.

`Quantity` is the interface describing the dimension of units. The left side of this figure shows some quantities implementing this `Quantity` interface; Appendix B contains all available quantities. The dimension of most quantities is unique; exceptions are, e.g., `Torque` and `Energy` having the same physical dimension. Two quantities are compatible when their dimensions are compatible. The dimension of a quantity is a structure with the following 7 real-valued properties: `length`, `mass`, `time`, `current`, `temperature`, `substance`, and `luminosity`. For example, the singleton `Velocity` object has the values `length = 1`, `time = -1`, and all others are `0`. Each `Quantity` object has one base unit, e.g., `Velocity` has the base unit meter per second. Every `Unit` belongs to exactly one `Quantity`. However, a `Quantity` has multiple units. The `Unit` class has the additional attribute `prefix`. Every base unit always has a `prefix` value of one. The unit object `mile` belonging to quantity `Length` has as `prefix` the value `1 600`, because 1 mile are 1 600 meters.

Every `Number` has exactly one `Unit`. Java numbers such as `2` or `-2.3` have the singleton unit `ONE` with `prefix` equals `1` and quantity `Dimensionless`. Thus, the port type system of *EmbeddedMontiArc* supports all numbers of the common programming languages. The *EmbeddedMontiArc* syntax `2 cm` creates the object of the type `Number` with the following attribute values: `value = 2`, `unit = cm` having `quantity = Length`, `isPlusInf =`

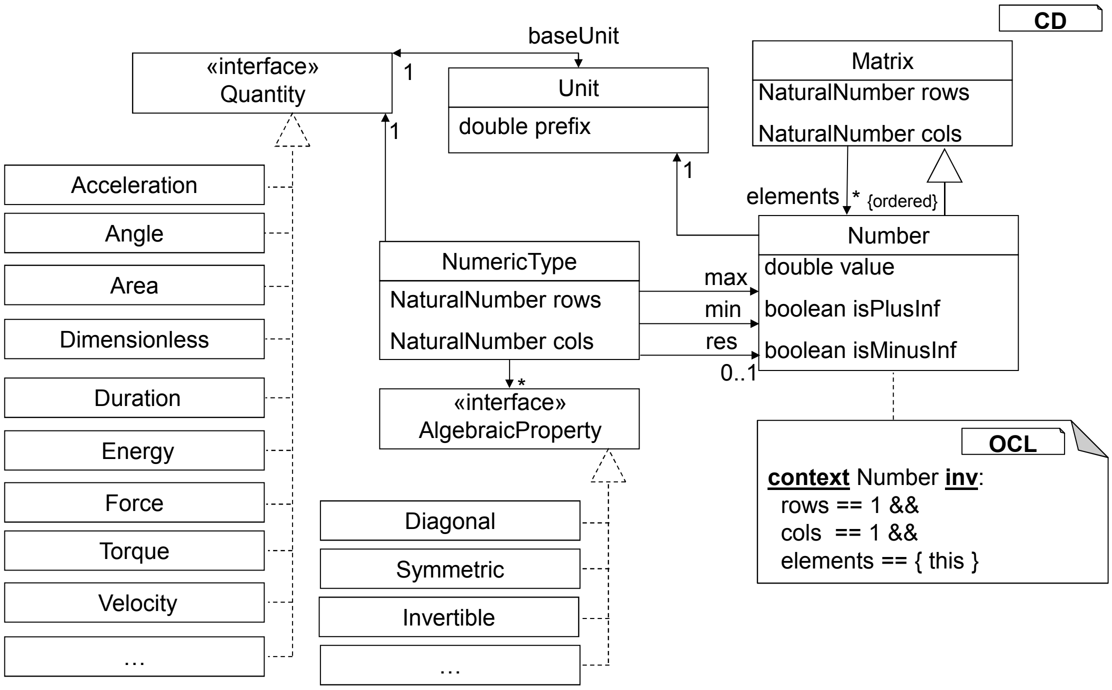


Figure 3.18.: Excerpt of abstract syntax of *EmbeddedMontiArc*'s port type system (Boolean type, enumerations and structures are omitted).

`false`, and `isMinusInf = false`. The *EmbeddedMontiArc* syntax `oo`, `+oo`, and `-oo` represent plus infinity or minus infinity. Since for both infinities the value and the unit prefix is irrelevant, the `value` is set to 0 and the `unit` is set to `Quantity.baseUnit`. Therefore, plus and minus infinity are compatible to every quantity. Mostly, *EmbeddedMontiArc* can infer the quantity of plus and minus infinity automatically, then these quantities can be skipped; otherwise you must specify the quantity explicitly in the concrete syntax; e.g., `-oo<Length>`.

The `NumericType` class represents numeric port types. Numeric port types have the mandatory range attributes minimum (`min`) and maximum (`max`), as well as the optional range attribute resolution (`res`). Every `NumericType` belongs to one quantity, and the quantities of `min`, `max`, and `res` are equals to the quantity of this `NumericType`. The minimum/maximum attributes may have the values minus/plus infinity, the resolution attribute may not have infinity values. Besides the range attributes, the `NumericType` class also has the matrix attributes: number of `rows`, number of columns (`cols`), and a set of algebraic properties (cf. Subsection 3.5.2).

The *EmbeddedMontiArc* syntax `(1 cm^2 : 5 m^2)` creates a `NumericType` object with the following attribute values: `quantity = Area`, `min = 1 cm^2` having `quantity = Area`, `max = 5 m^2` having `quantity = Area`, `res = ⊥`[7], `rows = 1`, `cols = 1`, and `algebraicPorperties = {}`. The *EmbeddedMontiArc* syntax `diag (0s : 1ns : 1h)^{20, 10}` creates a `NumericType` object with the following attribute val-

[7]equals to Java's `Optional.empty()`

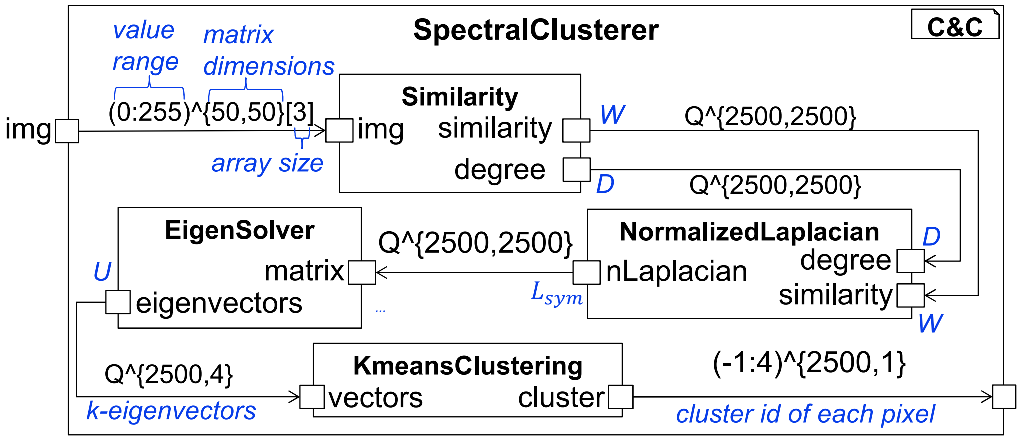


Figure 3.19.: Spectral clustering algorithm as example for needs of matrix properties in *EmbeddedMontiArc* (copied from [KRSvW18a]).

ues: `quantity = Duration`, `min = 0s` having `quantity = Duration`, `max = 1h` having `quantity = Duration`, `res = 1ns` having `quantity = Duration`, `rows = 20`, `cols = 10`, and `algebraicPorperties` is a set with one value - the singleton `Diagonal` object.

EmbeddedMontiArc supports special syntactic sugar:

- Writing down a class name implementing the `Quantity` interface, represents the object of `NumericType` with minimum set to minus infinity of the specified quantity, maximum set to plus infinity of the specified quantity, and resolution is not present. For example, *Velocity* in *EmbeddedMontiArc* means the object with type `NumericType` with quantity `Velocity` having the attribute values `min = -oo m/s`, `max = +oo m/s`, and `res = ⊥`.
- *Z* (inspired by $\mathbb{Z}$) in *EmbeddedMontiArc* means the object of type `NumericType` with quantity `Dimensionless`, `min = -oo`, `max = +oo`, and `res = 1`.
- *Z+* (inspired by $\mathbb{Z}_+$) or *N+* (inspired by $\mathbb{N}_+$) is similar to *Z*, but with `min = 1`.
- *Z0* (inspired by $\mathbb{Z}_0^+$), *Q+* (inspired by $\mathbb{Q}_+$), and *Q* (inspired by $\mathbb{Q}$) work in the same way.

In *EmbeddedMontiArc* each number is a matrix of dimension 1×1. Every matrix in *EmbeddedMontiArc* consists of numbers all having the same quantity.

3.5.2. Matrices as Port Types

Most cyber-physical systems contain object recognition algorithms, e.g., pedestrian detection in self-driving cars, object and position recognition in automated fabrications (industry 4.0). Due to the cheap hardware prices for cameras, more and more systems use image processing, esp., image segmentation, often dealing with image matrices or higher dimensional arrays. Figure 3.19 gives an example of a spectral clustering algorithm used in image segmentation. The `Similarity` component (cf. Figure 3.20) gets as input three (for the channels red, green, blue) 50×50 matrices with pixel values between 0 and 255. Based on these in-

EMA

```
component Similarity {
  ports in   (0:255)^{50,50} rgb[3],
         out symmetric Q(-255:255)^{50,50} W,
             diagonal Q(0:1)^{50,50} D;
}
```

algebraic property

Figure 3.20.: Example defining Matrix properties as part of port types.

puts it calculates a symmetric similarity matrix W and a diagonal degree matrix D. As shown in Figure 3.19, *EmbeddedMontiArc* supports specification of matrix properties, as they are essential properties components rely on. The component NormalizedLaplacian in Figure 3.19 calculates the symmetric Laplacian matrix and this calculation is wrong if the input matrix W is not symmetric. Besides symmetric other, but not limited to, valid algebraic matrix properties are defective, non-defective, invertible, idempotent, Hermitian, Skew-Hermitian, positive-definite, positive-semidefinite, indefinite, negative-definite, negative-semidefinite, normal, diagonal, tridiagonal, upper-triangular, lower-triangular, unitary, non-normal, identity, permutation, singular, non-singular, nilpotent, or unitary. A matrix may have multiple algebraic matrix properties. For example, a 5×5 matrix with only twos on its main diagonal and only ones on its first diagonal below as well as above is symmetric tridiagonal positive-definite. *Matrix Taxonomy & Matrix Properties* paper [Bor06] shows the relationship between these algebraic matrix properties as well as their exact mathematical definitions. Some matrix properties have short-forms, e.g., diag for diagonal.

3.5.3. Configuration Parameters of Port Type System

Configuration parameters of the port type system present configurable holes in the concrete implementation. Configuration parameters enable to reuse component types, and at the component's instantiation the binding of these parameters specify different behavior. *MontiArc* uses squared brackets for configuration parameters [Hab16, Listing 3.4]; whereas *EmbeddedMontiArc* uses parenthesis surrounding configuration parameters, because squared brackets define array definitions or array access operators in *EmbeddedMontiArc* similar to most other languages such as Java or C++. Please note that generic parameters influence component's interface or signature, having impact of any port's type or the number of ports (array size), while configuration parameter will never influence a component's interface. Generic parameters needed for configuration types should be defined at the end, because values of these generic parameters can be mostly inferred due to the passed configuration parameters during component instantiations[8]. Since this thesis deals with structural properties of component and connector models for cyber-physical systems, the focus of this thesis lays on component interfaces and their connections. Our ECMFA

[8]For example, `component X<Z p1, Z p2, Z p3>(Z^p3 p4)` defines the generic parameter p3 at the end, as p3 can be inferred by the type of the bounded configuration parameter p4. The component instantiation `instance X<4,5>([1, 2]) x1` binds p3 to 2, because the bounded type of p4 is Z^2 ($\mathbb{Z}^2 \ni [1 \quad 2]$).

EMA
```
1 component Max<T, N+ n=2> {
2   ports in  T values[n],
3         out T maxValue;
4 }
```

Figure 3.21.: Component type definition `Max` with generic port type and generic array size parameter.

EMA
```
1 component MaxInstancesExamples {
2   instance Max<(0$:150$), 3> maximumOfThreePayments; // T=(0$:150$), n=3
3   instance Max<(0°:0.1°:180°> maximumOfTwoDegrees;
4                                           // T=(0°:0.1°:180°), n=2
5   instance Max<n=5, T=(-5 m/s^2: 5 m/s^2)>; // T=(-5 m/s^2: 5 m/s^2), n=5
6 }
```

Figure 3.22.: Example showing how to instantiate generic component type definitions.

paper [KRRvW17], introducing *EmbeddedMontiArc*, contains a more detailed explanation and an example in [KRRvW17, Figure 5].

3.5.4. Type Parameters

Most Cyber-physical systems contain control components such as filters, error/derivation calculations, and prediction functions. These components are based on a generic mathematical background; e.g., the finite impulse response (FIR) filter is generic [GKR+17] in the number of stored input values it may response to.

Figure 3.21 shows the code of a simple generic component calculating the maximum value (cf. l. 3) of input values (cf. l. 2). The component definition has two generic parameters: `T` for the (yet unknown) port type of the input and output values, and `n` for the array size of the input port (Subsection 3.6.2 explains arrays of ports and component instantiations). Even when the port type is not specified yet, Figure 3.21 forces that the port type of both input and output are the same. Since `T` is not restricted in this context, `T` can be anything such as `(0 Eur:0.01 Eur:oo Eur)`, `(-10 km/h:250 km/h)`, or just plain numbers `(1:100)`, but also structures (cf. *SIStructs* language in Section 3.4). The expression `T is UnitNumber` excludes structures. The expression `T is Money` restricts `T` to the JScience [Dau07] `Money` quantity, whereas `T is Velocity` restricts it to speed values, and `T is Dimensionless` restricts `T` to dimensionless numbers, vectors, matrices, or tensors having units such as percentage or degree. If a generic component type is used often with the same concrete generic parameter, then this value can be specified as default value; e.g. `N+ n = 2` as shown in line 1 in Figure 3.21.

Figure 3.22 illustrates how to instantiate the generic component type `Max` defined in Figure 3.21. *EmbeddedMontiArc* supports binding parameters by order (ll.2-3) or by names (l. 5). The last concept is borrowed from *Ada* where it is also possible to bind generics by names. This is especially useful for generic components having multiple generic parameters with default values.

EMA

```
component PID
    // generic parameters
    < Qt1 as Quantity = Dimensionless,
      Qt2 as Quantity = Qt1,
      Qt1 lower = -oo, (lower : +oo) upper = -1*lower >
    // configuration parameters
    ( Q+ P, Q+ I, Q+ D,       Qt1*T windup = oo ) {
   ports in   Qt2 error,
         out  (lower : upper) output;
}
```

as lower bounds Quantity to Qt1, +oo is +oo<Qt1>

-oo is derived short-form for -oo<Qt1>

is short-form for Type<Qt1> with min = -oo and max = +oo

*increases time dimension by one, e.g., if Qt1 = Velocity, then Qt1*T = Acceleration and Qt1*T is short-form for NumericType<Acceleration> and this Is equals to (-oo m/s^2 : +oo m/s^2)*

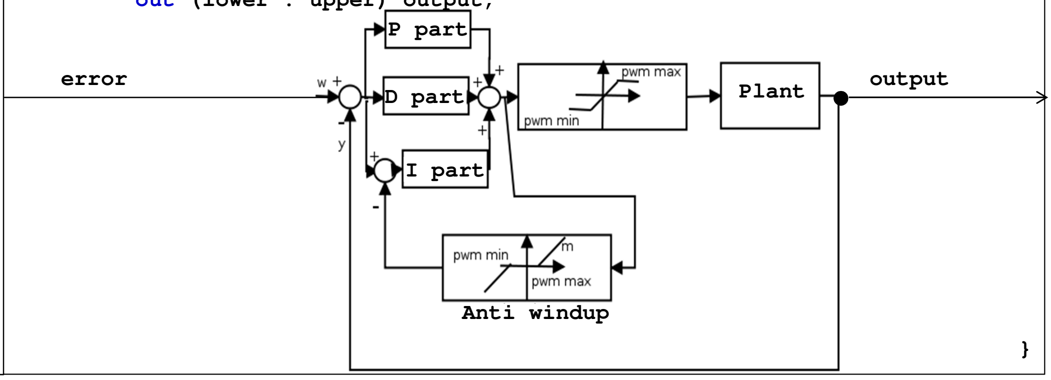


Figure 3.23.: Generic PID controller modelled in *EmbeddedMontiArc*. Textual code for connections are skipped, instead a graphical picture showing how the subcomponents are connected is inserted. The inner picture of the structure of the PID controller is copied from [Kra18, Bild 11.3-7].

Figure 3.23 shows the header of a generic PID controller modeled in *EmbeddedMontiArc*. The `PID` controller has one input port `error` and one output port. The general `PID` controller receives control errors (derivation between wished and actual values), and produces new outputs based on the error history. Since there exist different controllers, e.g., reacting on velocity errors with an acceleration value, or reacting on distance errors by adopting the speed value, the general `PID` controller has two `Quantity` generic parameters for the types of the input port and the two other generic parameters. The `lower` and `upper` generic parameters define the type of the output port, as they limit the output signal to a given range. The first three configuration parameters `P`, `I`, and `D` are the constant factors for the proportional, integral, and derivative part of the `PID` controller; these constants are mandatory. Additionally, this generic controller has a `windup` limiter, which uses the discrete derivation of the output divided by the time. For this reason the `windup` configuration parameter has the type `NumericType` with quantity `Qt1*T` (cf. Subsection 3.5.1), whereby `Qt1*T` is the quantity derived from `Qt1` by increasing the time dimension by one. More information how the generic `PID` controller works in detail is available from the online tutorial “Der Windup-Effekt bei Reglern mit begrenzten Stellgrößen” [Kra18]. This online tutorial also contains C++ code to execute this PID controller.

EMA

```
1   component Controller {
2     // specify all generic parameters:
3     instance PID<Velocity, Acceleration, -10 km/h, 10 km/h>
4                  (12, 0.7, 2.3, 1.5 cm) velAccPid;

5     // in- and output port have the same dimension, no antiwindup protection
6     instance PID<Length, lower=0m, upper=10m>(2.3, 1.3, 0.2) distancePid;

7     // input and output port are both dimensionless
8     instance PID<lower=0°, upper=90°>(2.3, 1.3, 0.2, 0.2°*s) steeringPid;

9     // in- and output port are dimensionless, symmetric limiter, no antiwindup
10    instance PID<lower=-45°>(2.3, 1.3, 0.2) symmSteeringPID;

11    // a very simple PID without limiter and antiwindup protection
12    instance PID(1, 1, 1) simplePid;
13  }
```

Figure 3.24.: Controller component showing how to instantiate the generic PID controller in multiple ways.

Figure 3.24 contains several instantiations of the general `PID` controller to show bindings of generics with and without default parameters. The first instantiation `velAccPid` (cf. ll. 3-4) defines all generic and configuration parameters. The second one `distancePid` (cf. l. 6) skips the second generic parameter, as the controller maps `Length` to `Qt1` and `Qt2` as `Qt2`'s default value is `Qt1`'s bounded value, and it omits the last configuration parameter. The third instantiation (cf. l. 8) skips the first two generic parameters, as the input and output ports are dimensionless. The fourth instantiation (cf. l. 10) has a symmetric limiter, and thus it defines only the `lower` generic parameter plus all required configuration ones. If a modeler does not have so much background knowledge about PIDs (or does not need limiters and antiwindup protection), then the modeler can instantiate a `simplePid` controller only focusing on the important parameters `P`, `I` and `D` as shown in line 12.

The realistic PID controller example shows how to create general reusable library components in *EmbeddedMontiArc*. These library components can be instantiated (used) by modelers with different technical backgrounds (less background: output and input ports are the same and use nearly all default values; much background: do a lot of fine tuning via type adjustments).

Enumerations in Arrays of Component Instances and Ports as well as in Type Parameters

One drawback of port and component instantiation arrays (cf. Subsection 3.6.2) is the reduced readability; `actuators[1]` is not as good understandable as `frontActuator` in a model not using the array concept of *EmbeddedMontiArc*. Using partial enumerations for generic types tackles this problem.

Figure 3.25 presents an example with and without the usage of partial enumerations for generic types. The models on the left and on the right are the identical. But the left model in line 9 is

```
EMA
1  component BumperBot<N+ axes> {

2    instance Sensors sensors;
3    instance Controller
4                      controller;
5    instance Actuators
6                actuators[axes];

7    // special treatment for
8    // front axe
9    connect actuators[1] -> …

10   connect actuators[2:end] -> …
11 }
```

```
EMA
12 component Actuators {
13   instance Motor leftMotor;
14   instance Motor rightMotor;
15 }
```

```
EMA
16 // possibilities how to use it:

17 instance BumberBot<2> bb2;
18 // or


19 instance BumberBot<3> bb3;
```

```
EMA
20 component BumperBot
21             <enum { FRONT | … } axes> {
22   instance Sensors sensors;
23   instance Controller controller;

24   instance Actuators actuators[axes];


25   // special treatment for front axe
26   connect actuators[FRONT] -> …

27   connect actuators[axes\{FRONT}]
28     -> …
30 }
```

```
EMA
31 component Actuators {
32   instance Motor leftMotor;
33   instance Motor rightMotor;
34 }
```

```
EMA
35 // possibilities how to use it:
36 enum TwoAxes { FRONT | BACK }
37 instance BumberBot<TwoAxes> bb2;
38 // or
39 enum ThreeAxes { FRONT | MIDDLE |
40                  BACK }
41 instance BumberBot<ThreeAxes> bb3;
```

Figure 3.25.: Example code of port arrays without (left) and with (right) partial enumerations in generics.

hard to understand without the comment above it. A modeler reading the model in a few years without having the comment has a hard time to figure out if this is a special treating for the first or for the last axe. According to clean code (rule 1 for comments “Always try to explain yourself in code.” [Luk16] in clean code by Robert C. Martin [Mar09]) a comment should not say what you do, it should only say why. Therefore, *EmbeddedMontiArc* has the opportunity to use partial enumerations (cf. l. 21) instead of using subsets of $\mathbb{N}_+$ as data type for type parameters. When normally numbers as indices are used to access array elements (e.g. in l. 9 and l. 10), elements of the enumerations (cf. l. 26) or sets (cf. l. 27) of enumerations' elements are used for partial enumerations. When normally numbers (cf. l. 17 and l. 19) are used to bound this type element, enumerations (cf. l. 36 and l. 39) are used to bind a partial enumeration type parameter (cf. l. 37 and l. 41). Also numbers and enumerations are not the same, as a number is a single element and enumerations are sets of elements, they are interpreted equally by meaning in line 24 actually the cardinality of the enumeration `|axes|`. Only for better readability of the concrete syntax, *EmbeddedMontiArc* uses the short-form `axes`.

EMA

```
1  package safety;

2  component EmergencyBrake {
3    ports in    (0m : 0.5m : 25m) distance,
4                (0 km/h : 0.1 km/h : 250 km/h) speed,
5           out  (0% : 1% : 100%) force;
6  }
```

Figure 3.26.: Component type definition with two inputs and one output ports.

The idea by accessing array elements by names instead of only numbers is borrowed from JavaScript object properties [Dat18]. Another advantage of using partial enumerations instead of numbers is that the developer sees at the `BumperBot` signature that the front axe receives special treatment.

For the semantics of the model and the ability of reusing the component in different scenarios, it makes no difference if the left or the right version is used. Therefore, this thesis uses only the left version as the right model can be transformed to the left one. But for creating models with *EmbeddedMontiArc* the better readable right version should be preferred.

3.6. Components and Ports in *EmbeddedMontiArc*

This section introduces the textual C&C modeling language *EmbeddedMontiArc*. *EmbeddedMontiArc* is a domain specific language for the logical layer in the systems engineering process (cf. Section 2.2). *EmbeddedMontiArc* enables efficient, agile, and intuitive functional modeling by providing component types (cf. Subsection 3.6.1), arrays of ports and component instantiations (cf. Subsection 3.6.2), component interfaces (cf. Subsection 3.6.3), configuration parameters of component types/interfaces for reference architectures (cf. Subsection 3.6.4) and product-line modeling (cf. Subsection 3.6.5), intuitive connection patterns (cf. Subsection 3.6.6), and packaging concept similar to Java (cf. Subsection 3.6.7).

3.6.1. Component Type Definitions and Component Instantiations

In *EmbeddedMontiArc* components communicate only via their interfaces containing of in- and output ports. *EmbeddedMontiArc* uses direct point to point communication as its base language *MontiArc*. In *EmbeddedMontiArc* exists, in contrast to *Simulink*, no data exchange via local/global variables.

Figure 3.26 defines the new component type `safety.EmergencyBrake` (ll. 1-2). The full-qualified name of a component type includes the package (l. 1) and the short component type (l. 2) name. Similar to Java's class definitions, the full-qualified name of a component type definition must be globally unique. The `EmergencyBrake` component type has two in- and one output ports. The first input port `distance` (l. 3) accepts the rational numbers `0.0m`, `0.5m`, `1.0m`, `1.5m`, `...`, `25.0m`. The second one `speed` (l. 3) accepts numbers between `0 km/h` and `250 km/h` as a multiple of `0.1 km/h`. In contrast to its base language *MontiArc*, using the Java type system, *EmbeddedMontiArc* uses the SI type system including domain definitions.

EMA

```
1 package adas;
2 import safety.EmergencyBrake;
3 import safety.BrakeActuator;

4 component ParkingAssistant {
5   ports …;
6   instances EmergencyBrake brakeLeft, brakeRight;
7   instance BrakeActuator brakeActuator;
8 }
```

Figure 3.27.: Hierarchial decomposition of components.

EMA

```
component A {
  instance B b1;
}
```

EMA

```
component B {
  instance A a1;
}
```

Figure 3.28.: Model contains hierarchy cycle on instances, and is therefore invalid.

Therefore, memory-unbounded port datatypes such as `String` or `List` are not available in *EmbeddedMontiArc*.

Other component definitions can instantiate the `EmergencyBrake` component definition multiple times as shown in Figure 3.27. Figure 3.27 also shows the hierarchical decomposition of the complex component `ParkingAssistant`. The `ParkingAssistant` component type is decomposed of three component instantiations `brakeLeft`, `brakeRight` and `brakeActuator`. Please note, that one component type definition can be decomposed with several component instantiations of the same type. In the example in Figure 3.27, the `ParkingAssistant` component type definition contains two instances of the component type `EmergencyBrake`. Resolving component types used in component instantiations (cf. ll. 6-7) is based on the full-qualified name of component types. Therefore, Figure 3.27 imports the two artifact files defining the component types `EmergencyBrake` and `BrakeActuator` (cf. ll. 2-3), because the package `adas` (cf. l. 1) of the `ParkingAssistant` component type definition differs from the package `safety` (cf. l.1 in Figure 3.26) of the referenced component types `EmergencyBrake` (cf. l. 6) and `BrakeActuator` (cf. l. 7).

The package/import mechanism of *EmbeddedMontiArc* is the same one as in Java; importing an entire package [Ora17g] via the asterisk symbol is also supported. Thus, the rest of this thesis omits `package` and `import` statements for better readability reasons.

In *EmbeddedMontiArc* the component types of instantiations, already decomposing a parent component type, may contain other instantiations of component types. The component (type) hierarchy is a tree of all component types starting with the component type of the main component instantiation (cf. Subsection 3.6.7). The component instance hierarchy is always a tree with no cycles: Figure 3.28 is invalid.

Note that in *MontiArc*, which is the base language of *EmbeddedMontiArc*, the keyword `instance` is also `component`. However, this ambiguity, `component` keyword for both

EMA

```
component OuterComponentTypeDefinition {
   // not recommended to define inner component type,
   // as it can only be used inside this file
   component InnerComponentTypeDefinition {
    // …
   }
   instance InnerComponentTypeDefinition inner;
}
```

Figure 3.29.: Model contains hierarchy cycle, and is therefore invalid.

EMA

```
1 component SensorProcessing { // incomplete
2   ports in  C signal[6],
3         out (0m : 0.5m : 25m) distance;
4   instance Filter filter[6];
5 }
```

Figure 3.30.: Example model with port and component instantiation arrays.

component type definitions and component instantiations, confused students looking only at the textual models during labs. This extra `instance` keyword is neither needed for more advanced modelers nor for technical reasons as an algorithm can derive whether a component is defined or instantiated.

MontiArc models contain often inner component type definitions (cf. [Hab16, Rin14]). In *EmbeddedMontiArc* it is not recommended to define nested component types as shown in Figure 3.29, because no other component type definition can reuse the inner component type one. *EmbeddedMontiArc* supports inner component type definitions, but using them will cause a warning, as this limits the reuse of components. Therefore, this theses will not cover this case.

3.6.2. Arrays of Ports and Component Instantiations

One feature of *EmbeddedMontiArc*, missing in most other C&C languages (cf. Section 3.2), is the ability to create arrays of ports and component instantiations. The array concept avoids copying of ports and component instantiations, as well as it introduces more flexibility.

Figure 3.30 shows an incomplete *EmbeddedMontiArc* model levering the array concept; this subsection omits connections between arrays of ports or component instantiations (cf. Subsection 3.6.6 for simple and more advanced connection patterns).

Figure 3.31 represents the graphical C&C representation of Figure 3.30. The `SensorProcessing` component has 6 input ports (cf. l. 2), which receive raw signal data from a hardware as complex numbers, and it instantiates 6 subcomponents (cf. l. 4) to filter invalid input parallel. Most component and connector architecture description languages, e.g., *MontiArc* and *Simulink*, do not support arrays of component instantiations; and therefore, instantiations (in our example the 6 filter instantiations) are copied multiple times resulting in bad readable models.

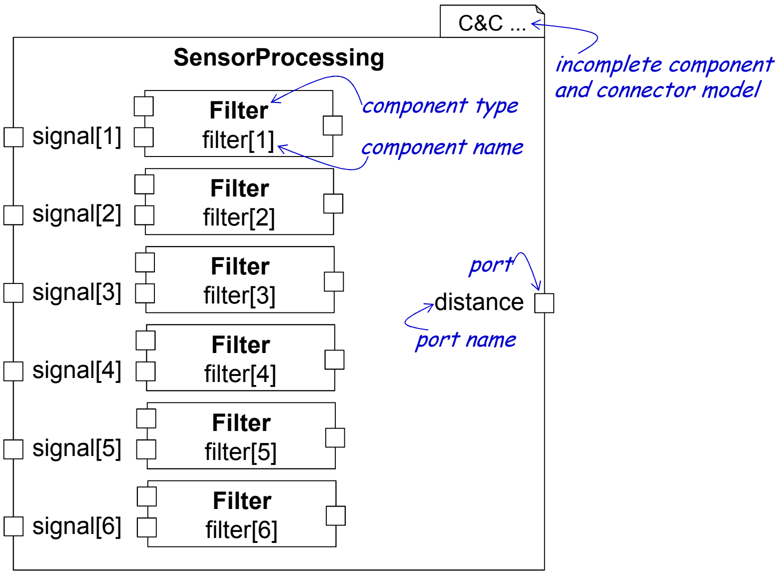


Figure 3.31.: Graphical representation of Figure 3.30 as incomplete component and connector example model. Connections are skipped; Subsection 3.6.6 presents different connection patterns.

EmbeddedMontiArc supports only one-dimensional arrays of ports and component instantiations right now. If a use case requires a multi-dimensional array, e.g., to model clusters, then the *EmbeddedMontiArc* language must be extended.

3.6.3. Component Interfaces

In *EmbeddedMontiArc* components communicate only via ports of component instances. Therefore, the internally decomposition or the atomic behavior of a component type is not important for data exchange with component instances of this component type. The component interface addresses this issue. The interfaces between C&C models and their simulators - e.g., car simulator *MontiSim* [GKR+17], SuperMario simulator [KRRvW18], or the PacMan simulator [KRRvW18] - use component interfaces on the *EmbeddedMontiArc* side and compatible C++ or Java interfaces on the simulator side. Therefore, data exchange between *EmbeddedMontiArc* and simulators are explicitly defined, and different model behaviors are easily possible. A component interface has no behavior, i.e., it does not contain any implementation block or neither it is decomposed of other subcomponents.

MontiArc supports besides component interfaces also component extensions whereby all input, output ports as well as subcomponent instantiations and their connections are inherited [Hab16, p. 42]. One drawback (in the opinion of the author of this thesis) of this component extension mechanism is the unclear semantics of connections inside a component which are not needed anymore. For example, the connection `port1 -> sub1.portIn` is replaced by the chain `port1 -> newSub.portIn` and `newSub.portOut -> sub1.portIn` during

EMA

```
1   // interface defines the contract with developed physic engine simulator
2   component interface Car {
3      ports in  GPS posCar,
4                (0 km/h : 0.1 km/h: 250 km/h) carSpeed,
5                (0 cm : 0.1 cm : 100 cm) distanceFront[20],
6                      …
7            out (0 lx : 2 lx : 60 lx) leftFrontLights[40],
8                (-10 m/s^2 : 0.01 m/s^2: 15 m/s^2) acceleration,
9                (-90°: 0.2°:90°) steering;
10  }
```

EMA

```
11  component PorscheCayenne implements Car {
12    ... // has all the ports of Car, but adds its own implementation
13  }
```

EMA

```
14  component Fiat500 implements Car {
15    ... // has all the ports of Car, but adds its own implementation
16  }
```

Figure 3.32.: Definition of Car component interface and their implementations.

component extension; but the extension mechanism (similar to Java's one) does not support to remove the old connections. Therefore, in *EmbeddedMontiArc* it is not possible to extend component types by any kind of ports.

Figure 3.32 shows an example of *EmbeddedMontiArc*'s interface mechanism. The `Car` interface (cf. l. 1) defines the ports (cf. ll. 3-5) that the simulator needs to update the physical car model. The `PorscheCayenne` (cf. l. 11) and the `Fiat500` (cf. l. 14) are two different C&C models implementing this `Car` interface. Therefore, both `PorscheCayenne` and `Fiat500` can interact with the *MontiSim* simulator, which results in two different driving behaviors of the car in the simulator.

The component interface can also contain generic or configuration parameters to facilitate more flexible data exchange between models and simulators. For example, the number of left front lights or the maximum value of the type of the car speed port could be a generic parameter in the `Car` interface.

3.6.4. Reference Architectures with Configuration Parameters

A component library reference architecture (cf. Figure 3.33) does not specify the implementation behavior of all atomic components. Thus, the reference design is reusable in different scenarios. For this case *EmbeddedMontiArc* supports component-interfaces as configuration parameters. For example, the `PumpActuator` (cf. gray subcomponent in Figure 3.33) might differ in various situations due to safety restrictions in countries, height it must pump water, or weather conditions.

Figure 3.37 presents the *EmbeddedMontiArc* model of the reference architecuture. It defines the `PumpActuator` component interface (cf. ll. 1-4) dealing as variation point. The reference architecture component type `PumpingSystem` (cf. ll. 6-9) has one configuration parameter

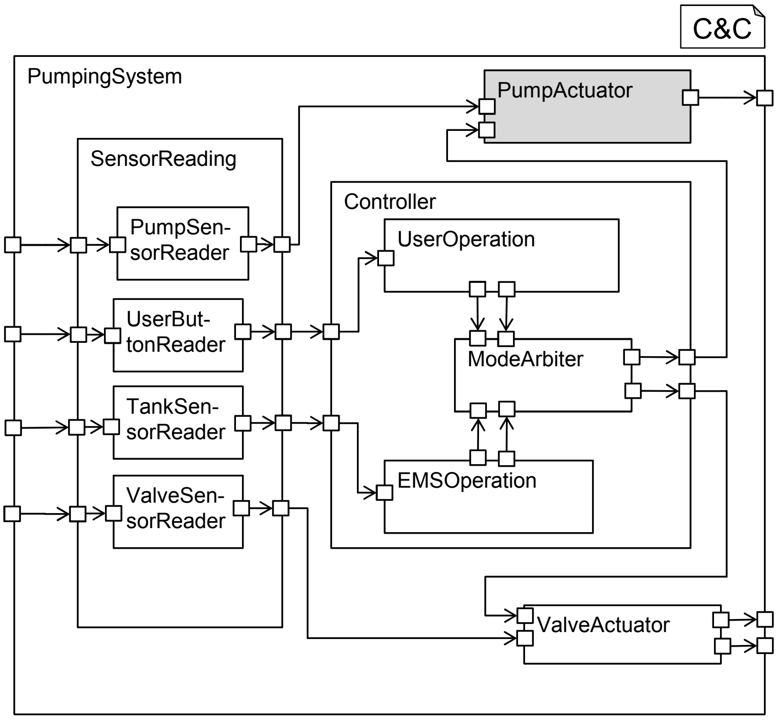


Figure 3.33.: PumpingSystem reference architecture (copied from [Rin14]).

EMA

```
// library component
component  Integrator<T as Numeric> {
  ports in   T value,
              B reset,
              (0s : oo s) time,
         out  T sumValue;
  // implementation skipped
}
```

EMA

```
// library component
component Differentiator<T as Numeric> {
  ports in   T value,
              B reset,
              (0s : oo s) time,
        out  T diffValue;
  // implementation skipped
}
```

Figure 3.34.: Library components from another company. These are packed and cannot be modified.

having the `PumpActuator` component interface as type (cf. l. 6). The `PumpingSystem` component type uses this configuration parameter to instantiate the subcomponent `pumpActua-`

EMA
```
component interface TimeDependentCalculator<T as Numeric> {
  ports in   T inValue,
             B reset,
             (0s : oo s) time,
         out T outValue;
}
```

EMA
```
component Controller(TimeDependentCalculator calc) { … }
```

EMA
```
// create wrapper to use library component
component IntegratorWrapper<T as Numeric>
                                     implements TimeDependentCalculator<T> {
   instance Integrator<T> integrator;
   connect inValue -> integrator.value;
   connect this.* -> integrator.*; // see 3.6.6
   connect integrator.sumValue -> outValue;
}
```

EMA
```
// create wrapper to use library component
component DifferentiatorWrapper<T as Numeric>
                                     implements TimeDependentCalculator<T> {
   instance Differentiator<T> differentiator;
   connect inValue -> differentiator.value;
   connect this.* -> differentiator.*; // see 3.6.6
   connect differentiator.diffValue -> outValue;
}
```

EMA
```
// reuse both library components
component ComplexController {
  …
  instance Controller(IntegratorWrapper<(0m/s^2 : oo m/s^2)>) controller1;
  instance Controller(DifferentiatorWrapper<(0m : oo m)>) controller2;
}
```

Figure 3.35.: Using library components in reference architecture with wrappers and without duck typing. Duck typing can be disabled in *EmbeddedMontiArc* via context condition flag.

`tor` in line 8. If the variation point is deeper in the hierarchy of the reference architecture, then the configuration parameter `PA` is passed to a subcomponent instantiation.

Figure 3.38 shows how to create and pass two specific component types. Both implement the `PumpActuator` interface (cf. ll. 1-4), and both types are passed to the `PumpingSystem` reference architecture (cf. ll.7, 12). The main component instantiation mechanism (cf. Subsection 3.6.7) enables reusing reference architectures as top-level element without creating any wrapper component (as it is done in this example with the `HydrolicPowerStationWes-tEur` and `ElectricPowerStationHawaii` components).

EMA

```
component interface TimeDependentCalculator<T as Numeric> {
  ports in  T inValue,
            B reset,
            (0s : oo s) time,
        out T outValue;
}
```

EMA

```
component Controller(TimeDependentCalculator calc) { … }
```

EMA

```
// reuse both library components directly via duck typing
component ComplexController {
  …
  instance Controller(Integrator<(0m/s^2 : oo m/s^2)> via duck typing)
           controller1;
  instance Controller(Differentiator<(0m : oo m)> via duck typing)
           controller2;
}
```

Figure 3.36.: Using library components in reference architecutre via duck typing.

EMA

```
// it is very specific to the climate
component interface PumpActuator {
   ports … ;
}
```

EMA

```
// general library model reference model
component PumpingSystem (PumpActuator PA) {
   … // large model with different hierarchy levels
   instance PA pumpActuator;
}
```

Figure 3.37.: Reference Architecture (incomplete model) as shown in Figure 3.33.

A general question is whether duck-typing [CRJ12] for component types should be supported or not. From a modeling-in-the-large point of view, duck-typing is really of advantage, as a project can define a component-interface and all library models (created before your project interface) can be imported and used (if they are compatible). In duck typing library components (which did not explicitly implement the new project's interface) automatically implement the interface, if the library component type is compatible to the interface. Without duck typing, the new project must wrap all library component types just to add the component-interface implementation.

Some persons [Beu05] see duck typing as a risk and it should not be used at all. *EmbeddedMontiArc* addresses both parties by supporting duck typing in general, and by providing the `no-duck-typing` flag. Using this flag, *EmbeddedMontiArc* activates a context condition to forbid duck typing for all components in this project.

EMA

```
component WestEuropePump implements PumpActuator
 { … } // no hurricans, but snow
```

EMA

```
component HawaiiPump implements PumpActuator
 { … } // hurricans, very hot, but no snow
```

EMA

```
component HydrolicPowerStationWestEur {
   // reuse reference architecture via general library model
   instance PumpingSystem(WestEuropePump) ps;
   …
}
```

EMA

```
component ElectricPowerStationHawaii {
  // reuse reference architecture via general library model
  instance PumpingSystem(HawaiiPump) ps;
  …
}
```

Figure 3.38.: Usage (incomplete model) of Reference Architecture (cf. Figure 3.37).

Figure 3.34 shows the two library components provided by a model repository. These components are not modifiable. Figure 3.35 shows how to reuse these two library components in the controller reference architecture by wrapping both of them. The wrapper variant enables renaming ports. Figure 3.36 shows the equivalent code of Figure 3.35 using the convenient duck typing concept. *EmbeddedMontiArc* forces the modeler to add the `via duck typing` (cf. ll. 11, 13) keywords to enable passing of components not implementing the interface. This way the duck typing is directly visible (e.g., to search later for such locations), and it avoids passing wrong component types via typos.

3.6.5. Product-Line Modeling with Configuration Parameters and Default Values

Delta Modeling supports powerful product-line modeling; cf. Section 3.7. In contrast to delta modeling stands the 150% modeling concept. *Simulink* uses `Enabled Subsystems` [The18k, p. 10-11ff.] for 150% modeling. `Enabled Subsystems` are especially useful to model optional features in an easy way.

This subsection shows how to create a product-line with optional features in *EmbeddedMontiArc*. Figure 3.39 shows a shortened product-line of an advanced driver assistance system. Figure 3.40 shows the 150% *Simulink* model for this product-line. Features are enabled (value 1[9] or `true`) or disabled (value 0 or `false`) via feature constants (cf. `FeatureTempomat`, `FeatureRepeater`, and `FeatureEmergencyBrake` in Figure 3.39). These constants can

[9] *Simulink* interprets any value different than 0 as `true` [The18d].

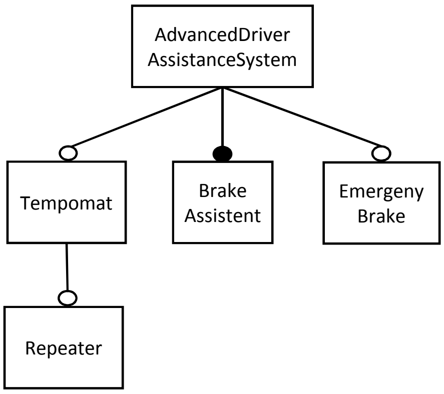


Figure 3.39.: Shortened Product-Line of advanced driver assistance system version 4 [BMR+17a].

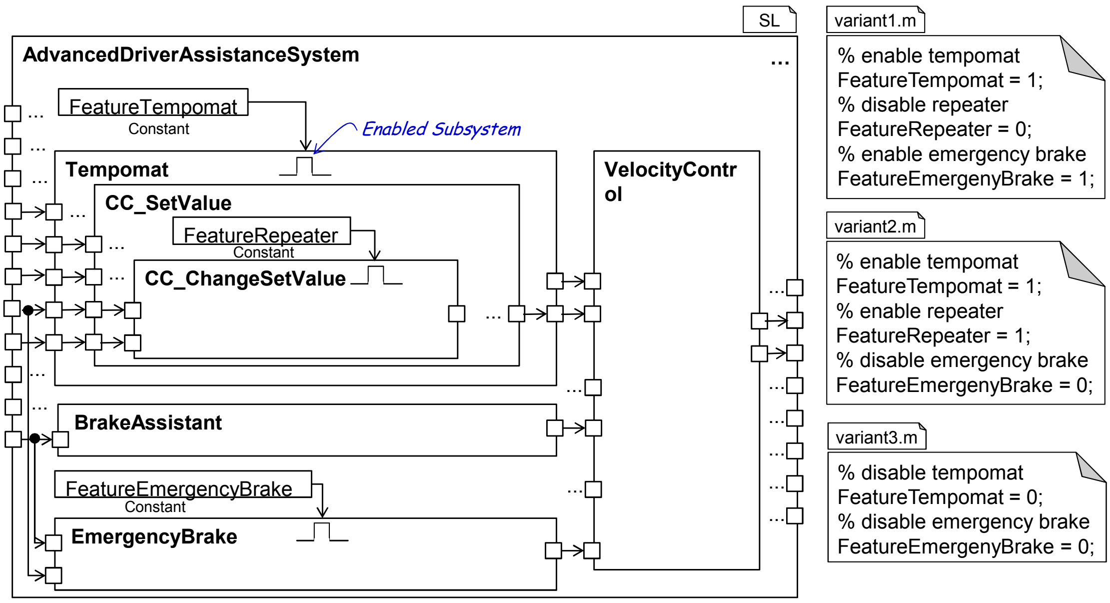


Figure 3.40.: Excerpt of *Simulink* model being compatible to product-line of Figure 3.39 (cf. [BMR+17b]).

be mapped to external tools (e.g., dSpace VariantManager). These tools produce *MATLAB* scripts (cf. right side of Figure 3.40) to enable or disable features. An `Enabled Subsystem` is a special *Simulink* subsystem which subcomponents are only executed when the current value of the input port `control signal` is `true`. Therefore, output ports of an `Enabled Subsystem` must declare what output value to produce when the enabled subsystem is disabled (cf. [The18k, p. 10-51f.]), because the output value cannot be calculated by its decomposed subcomponents.

Figure 3.41 shows the equivalent *EmbeddedMontiArc* model. For each optional feature a component interface and two components implementing this feature are written by the developer.

EMA

```
component interface TempomatFeature {
  ports in  (0km/h:0.1km/h:250km/h) vehicleSpeed, …,
        out (-10km/h:0.1km/h:10km/h) vehicleSpeedDelta, …,
}
```

EMA

```
component TempomatFeatureDisabled implements TempomatFeature {
  // terminate all input signals
  connect vehicleSpeed -> #;
  …
  // write down constant values for all output ports
  connect 0km/h -> vehicleSpeedDelta;
}
```

EMA

```
component TempomatFeatureEnabled
          (RepeaterFeature featureRepeater = RepeaterFeatureEnabled)
                          implements TempomatFeature { … }
```

EMA

```
component interface RepeaterFeature { … }
```

EMA

```
component RepeaterFeatureDisabled implements RepeaterFeature { … }
```

EMA

```
component RepeaterFeatureEnabled implements RepeaterFeature { … }
```

EMA

```
component interface EmergencyBrakeFeature { … }
```

EMA

```
component EmergencyBrakeFeatureDisabled implements EmergencyBrakeFeature
{ … }
```

EMA

```
component EmergencyBrakeFeatureEnabled implements EmergencyBrakeFeature
{ … }
```

EMA

```
component AdvancedDriverAssistanceSystem(
            TempomatFeature featureTempomat = Tempomat,
            EmergenyBrakeFeature featureEmergencyBrake = EmergencyBrake) {
  ports ...;
}
```

Figure 3.41.: *EmbeddedMontiArc* model for this product-line shown in Figure 3.39.

Main.txt

```
1 // variant 1
2 Main-Component-Instantiation: AdvancedDriverAssistanceSystem(
3     TempomatFeatureEnabled(RepeaterFeatureDisabled));
```

Main.txt

```
4 // variant 2
5 Main-Component-Instantiation: AdvancedDriverAssistanceSystem(
6     featureEmergencyBrake=EmergenyBrakeFeatureDisabled);
```

Main.txt

```
7 // variant 3
8 Main-Component-Instantiation:  AdvancedDriverAssistanceSystem(
9     TempomatFeatureDisabled, EmergenyBrakeFeatureDisabled);
```

Figure 3.42.: Three variants showing how to instantiate the *EmbeddedMontiArc* model in Figure 3.41 with different features.

One component disables this feature; this component just terminates all input signals (cf. l. 7) and produces constant output values (cf. l. 10). The other component enables this feature containing the actual logic. Since the `Tempomat` feature has the subfeature `Repeater`, the `TempomatFeatureEnabled` component type has one configuration parameter of the type `RepeaterFeature` (cf. l. 13). The `AdvancedDriverAssistanceSystem` component type has two configuration parameters (cf. l. 23), because the advanced driver assistance system has two direct subfeatures.

Figure 3.42 illustrates how to instantiate different variants of the product-line by enabling or disabling features via configuration parameters. Lines 2 and 3 show how to initialize the first version with the disabled `Repeater`. The default value of the second configuration parameter may pass the main component instantiation (is explained in detail in Subsection 3.6.7) as only configuration value. Lines 5 and 6 create the variant with enabled `Tempomat` and enabled `Repeater`, but disabled `EmergencyBrake`. Lines 8 and 9 shows the code for the variant disabling all optional features. A main component instantiation without passing any configuration parameter enables all features. The default values in this scenario are chosen in a way that most users want to activate these features.

In *EmbeddedMontiArc* all variation points of a product-line are visible in the component signature via the configuration parameters. In contrast, *Simulink* subsystems do not show variation points at all; the signature of the root subsystem `AdvancedDriverAssistanceSystem` contains only the signal input and output ports, but no feature constant value. Therefore, *EmbeddedMontiArc* with its strong type concept enables a “cleaner” modeling for optional features of a product-line.

EMA

```
component SensorProcessing {
  ports in    C signal[10],
              GPS posCar,
        out  (0m : 0.5m : 25m)
                              distance;
  instance Filter filter[10];
  instance SensorFusion<10> sf;
  connect signal[:] ->               ▲
          filter[:].signal;
  connect posCar -> filter[:].posCar;
  connect filter[:].distance ->
          sf.inValues[:];
  connect sf.outValue -> distance;
}
```

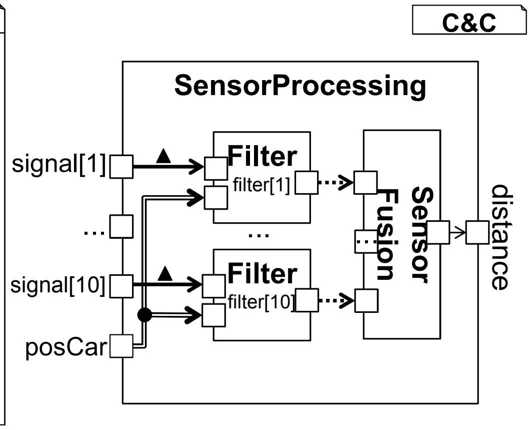


Figure 3.43.: Example showing component communications via connectors.

EMA

```
component InvalidConnection {
  ports in   (0m : 1m : 10m) source,
        out  (3m : 1m : 5m) target;
  connect source -> target;
```

domain(source) = domain(0m : 1m : 10m) = {0m, 1m, 2m, 3m, ..., 9m, 10m}
$\not\subseteq$ domain(target)=domain(3m : 1m : 5m) = {3m, 4m, 5m}

Figure 3.44.: Invalid connection, because domain of source port is not a subset of the domain of the target port.

EMA
```
component SteeringAct<N+ n> {
  ports in  (-45°:45°) steeringDeg,
        out (-5°/s:5°/s) steeringAc[n];
}
```

EMA
```
component SteeringActUsage<(2:4) m, (1:3) k> {
   ports in …,
         out (-5°/s:5°/s) steering[k*m];
   instance SteeringAct<m> sa[k];

   connect sa[:].steeringAc[:] -> steering[:];
   // is the same as:
   // forall i = 1..m, j = 1..k:
   //   connect sa[i].steeringAc[j] -> steering[i*(m-1)+j];
   …
}
```

Figure 3.45.: Example for two-dimensional matching via connector patterns.

3.6.6. Connections

Communication between component instances, also including communication between parent components to its subcomponents, is established via unidirectional asynchronous connectors. Figure 3.43 shows an example how to connect ports in *EmbeddedMontiArc*. The left part of the arrow symbol (->) is the source port of a connector, and the right part is the target port. The data exchange takes place from the source port to the target port. Since *EmbeddedMontiArc* is a logical modeling language, connectors do not loose data. If data loss in a connection is wanted, then a component actively loosing or modifying (noise) information must be added between the dataflow of two components.

Lines 8 and 9 in Figure 3.43 (`signal[:] -> filter[:].signal`) are a convenient abbreviation for `forall i in 1..10: connect signal[i] -> filter[i].signal`. Lines 8 and 9 connect the first signal port of `SensorProcessing` to the signal port of the first filter subcomponent and so on. The next line `posCar->filter[:].posCar` propagates the values of the `posCar` port to the corresponding port of all `filter` instances. An alternative syntax for line 10 is `forall i in 1..10: connect posCar -> filter[i].posCar`. Line 13 (`sf.outValue -> distance`) connects two output ports with each other; this syntax is identical to the one of the base grammar *MontiArc*.

The domain of the sender port must be a subset of the domain of the target port; Figure 3.44 shows an invalid example. In *EmbeddedMontiArc* a target port must not have different source ports. However, one source port may connect multiple target ports. In *EmbeddedMontiArc* constants can be directly connected to ports, e.g., `connect 7m/s^2 -> acceleration`. The route symbol in a target port (e.g., `connect unusedPort -> #`) terminates the data flow to suppress unused output port warnings.

Line 9 in Figure 3.45 shows an example of a two-dimensional matching via the connector pattern: The first dimension is a component instantiation array, and the second dimensions are port arrays for each component instantiation.

EmbeddedMontiArc resizes the matrix A_{source} automatically to the vector v_{source}. Now, *EmbeddedMontiArc* can connect elementwise this source port vector v_{source} with its target port vector v_{target}.

$$A_{source} = \begin{bmatrix} \text{sa[1].steeringAc[1]} & \cdots & \text{sa[1].steeringAc[k]} \\ \vdots & \ddots & \vdots \\ \text{sa[m].steeringAc[1]} & \cdots & \text{sa[m].steeringAc[k]} \end{bmatrix} \tag{3.1}$$

$$v_{source} = \begin{bmatrix} \text{sa[1].steeringAc[1]} \\ \text{sa[1].steeringAc[2]} \\ \cdots \\ \text{sa[1].steeringAc[k]} \\ \text{sa[2].steeringAc[1]} \\ \cdots \\ \text{sa[2].steeringAc[k]} \\ \cdots \\ \text{sa[m].steeringAc[k]} \end{bmatrix} \tag{3.2}$$

$$v_{target} = \begin{bmatrix} \text{steering[1]} \\ \cdots \\ \text{steering[k}\cdot\text{ m]} \end{bmatrix} \tag{3.3}$$

The lines `connect portX[:] -> sub[:].portY[:]` and `connect subA[:].portX[:] -> subB[:].portY[:]` enable resizing of port arrays. This automatic resizing facilitates very efficient ways to connect arrays of subcomponent instantiations with arrays of ports.

EmbeddedMontiArc uses the *MATLAB* array notations to create connection patterns (cf. colon operator [The18a, The18e] and reshape [The18f] documentation). Similar to *MATLAB*, indices in *EmbeddedMontiArc* start with 1, and not with 0 as in Java or C++. *AADL* also has connection patterns for one and two dimensions (cf. Figure 3.46). *AADL* uses words instead of indices to describes these patterns. It is a matter of taste, whether number-based or word-based indexing for connections is more beautiful. The number-based indexing of *EmbeddedMontiArc* is very powerful, and enables creating customized connection patterns in a few lines of code.

Name-based Connections

In practical applications it is often necessary to forward many ports from a component to one of its subcomponents or vice versa. Therefore, *MontiArc* (cf. [Hab16, Section 3.3.2]) and also *EmbeddedMontiArc* introduces the `autoconnect` keyword. Using this keyword in a component definition, all subcomponent instantiations' ports having the same port name are automatically connected.

However, in very rare cases (mostly due to wrong port namings) the `autoconnect` option is not available, because the connection is not unique. Figure 3.47 presents such a rare case where

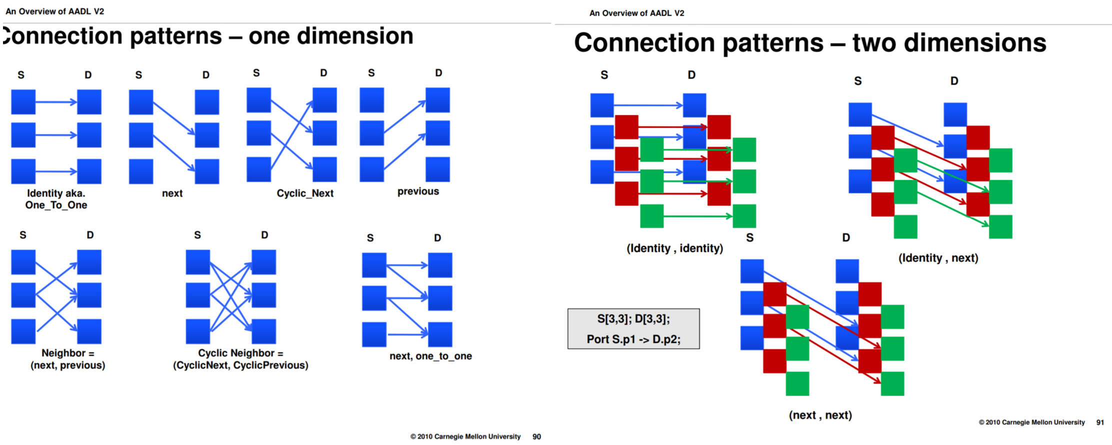


Figure 3.46.: Connection patterns in *AADL* (copied from [Fei10, slides 90-91]).
In the right picture `(identity, identity)` connects $\forall i, j \in \{0, 1, 2\} : S[i, j]- > D[i, j]$; `(identity, next)` connects $\forall i \in \{0, 1, 2\}, j \in \{0, 1\} : S[i, j]- > D[i, j + 1]$; `(next, next)` connects $\forall i, j \in \{0, 1\} : S[i, j]- > D[i + 1, j + 1]$.

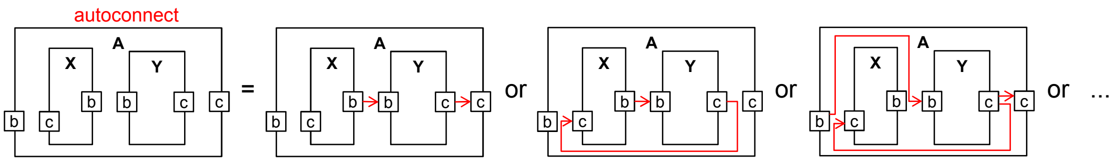


Figure 3.47.: *MontiArc*'s `autoconnect` option is not available (prohibited by context condition) as it could not be resolved uniquely. These cases are very rare.

EMA
```
component Inner {
  ports in  Z a, b, c,
        out Z x, y;
}
```

EMA
```
component Outer {
  ports in  Z a, b, c, d, e,
        out Z x, y, z;
  instance Inner inner;

  // connects: a -> inner.a; b -> inner.b; c -> inner.c;
  connect this.* -> inner.*;

  // connects: inner.x -> x; inner.y -> y;
  connect inner.* -> this.*;}
```

Figure 3.48.: Forwarding data using the wildcard operator in connectors.

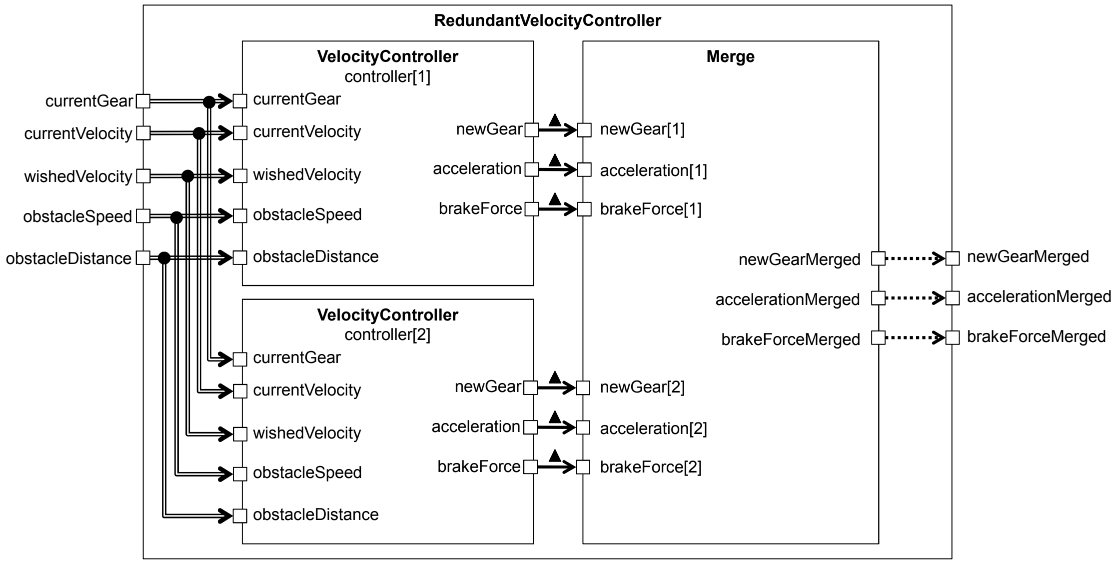


Figure 3.49.: Graphical C&C model of a redundant velocity controller.

`autoconnect` does not work. To still facilitate an efficient way of forwarding data for these use cases, *EmbeddedMontiArc* additionally supports the `.*` syntactic sugar (based on Java's `*` imports) to select all input or output ports.

Figure 3.48 shows an example. Of course, it is also possible to connect two subcomponents with the wildcard operator: `connect inner1.* -> inner2.*`. If the `inner1` subcomponent instantiation has the output ports `p1`, `p2`, and `p3`, as well as the `inner2` subcomponent instantiation has only the output ports `p1`, and `p2`; then `connect inner1.* -> inner2.*` connects only `inner1.p1 -> inner2.p1`, and `inner1.p2 -> inner2.p2`. However, if `connect inner1.* -> inner2.*` would result in no connections as port names do not match, then *EmbeddedMontiArc* throws an error.

Index- and Name-based Connections

Previously, this subsection explained how index-based connections and name-based connections work. In the following more complex connection patterns using both, index- and name-based, features in one connection statement are explained. A realistic example unveils the power of *EmbeddedMontiArc*'s connection patterns.

Figure 3.49 shows a redundant velocity controller containing of two controller instances to managed the velocity of a car. The two instances are needed to safety reasons to gain the wanted ASIL level. The input ports are the current gear of the car, the current vehicle velocity, the wished velocity (e.g., set by driver), as well as obstacle speed and distance of the car in front. The output ports are the new gear, acceleration, and brake force to get closer to the wished speed but avoiding a crash.

Figure 3.50 shows the combined index- and name-based connection pattern. The powerful pattern needs only 3 lines of code (cf. Figure 3.50) to create 19 connection instances (cf. Figure 3.49).

EMA

```
1  component RedundantVelocityController(N+ n=2) {
2    ports ... ;
3    instance VelocityController controller[n];
4    instance Merge<n> merge;

5    connect this.* -> controller[:].*;
6    connect controller[:].* -> merge.*[:]; ▲
7    connect merge.* -> this.*;
8  }
```

Figure 3.50.: Graphical C&C model of a redundant velocity controller.

The easiest way to understand this pattern is to unfold the connection statements step-wise (e.g., first all wildcards, and then all colons; or vice versa - the order does not matter). The connection `this.* -> controller[:].*` is equivalent to `this.currentGear -> controller[:].currentGear`, ..., `this.obstacleDistance -> controller[:].obstacleDistance`. The connection `this.currentGear -> controller[:].currentGear` is equivalent to `this.currentGear -> controller[1].currentGear` and `this.currentGear -> controller[2].currentGear`, because the left side is a single port and the right side is a port array and so the single port is connected to each port of the port array (cf. also Figure 3.43).

It is even possible to further shorten the listing in Figure 3.50. The double-lined (cf. l. 5) and the dashed (cf. l. 7) connections are subsumed by the `autoconnect` keyword; the code marked with with an triangle (cf. l. 6) is still necessary. This shows that the `autoconnect` and the name and index-based connection patterns complement each other very well.

In Figure 3.49 the `RedundantVelocityController` component type uses different output port names than the `VelocityController` component type. This is only for illustration purposes describing the name-based connection pattern. The real model uses the same output port names for both component types `VelocityController` and `RedundantVelocityController`, so that both component types can implement the same component interface. However, the name-based connection pattern does not work in this case anymore as the port names such as in `merge.newGearMerged` and `newGear` differ. In this case type-based connection patterns (skipped in this thesis) are available. In this example, `connect merge.** -> this.**` would connect `merge.newGearMerged -> newGear`, `merge.accelerationMerged -> acceleration`, and `merge.brakeForceMerged -> brakeForce` based on the output port types. However, type-based patterns are only available when every output or input port has a unique port type. The port type of `accelerationMerged` and `acceleration` is `m/s^2` and it differs from the port types of the other output ports having `N` and the enumeration type `Gear`.

All of these connection patterns enable to create more general component types, as shown in Figure 3.50, because the redundancy of the `VelocityController` component can be easily increased by adapting the configuration parameter `n` (cf. l. 1) when instantiating the `RedundantVelocityController`.

Main.txt

```
1  // mandatory part
2  Model-Paths:  [main/models/, lib/cars/, main/tags, test/streams];
3  Main-Component-Instantiation: controller.PID<°>
4                                (-45°, 45°, 0.2°*s, 2.3, 1.3, 0.2) steeringPid;

5  // mandatory, if needed
6  no-duck-typing: false;
7  1 EUR = 1,22838 USD;
8  1 GBP = 1,14402 EUR;

9  // optional, but recommended part
10 Authors: Michael von Wenckstern, Evgeny Kusmenko;
11 Date: 2017-12-15;
12 Version: 1.3;
13 EMA-Version: 2.5;
14 EMA-Compiler-Version: 4.1;
15 Simulator: 3dCarSimulator;
16 Simulator-Version: 2.3;

17 // optional part for trust
18 Public-Certificate: vonwenckstern.p12
19 File1:vonwenckstern.p12
20 File1-Digest: oA73o+KVb9kpTo8N4BtgEL4tbvR9eIFWC+lj0t+l/wYvRinu19m5ez09Ex5TimZt4M+NUlgpAfrw/k2zd1d6Ug==
21 File2: main/models/packageA/Inner.cmp
22 File2-Digest: fDcdHN6KYGuS32YO/GjCEApR06qQwVqlxBduCUYoZ5lz7ytDiDoUEwoHSycpmWeQ1/v2fHQZe5GFcN2RNHSXPQ==
23 File3: main/models/packageA/Outer.cmp
24 File3-Digest: Qxl+1fM+GvAzNNrYCswSZfdLc9yUV+ea4HUgS0Go76lr0Xatbf6M7ldrI7ydxGcFLraR8oKzZxzOJDv8H7wI9A==
25 …
```

Figure 3.51.: Example code snippet of `Main.txt` file.

3.6.7. Main Component Instantiation and Packaging

A complete model includes multiple component types and their interactions via connectors. Similar to Java [Ora17a], there is a need to specify the main (root) component instantiation of the modeled component types in a `Main.txt` file. Figure 3.51 presents such an example.

Similar to Java's JAR concept, all the models of the *EmbeddedMontiArc* family (cf. Section 3.4) plus the `Main.txt` file (which must be in the root folder) are zipped. This ZIP file presents one complete *EmbeddedMontiArc* component and connector model. Different tools can process this self-contained ZIP file directly, e.g., to generate a graphical component and connector instance structure, or to generate executable C++ code.

The C++ code generator [KRSvW18a] converts all units automatically. To convert currency units an exchange rate must be specified (cf. ll. 7-8).

The `Main.txt` file contains also an `EMA-Version` property (cf. l. 13) to which the model is compatible. This is needed for later compatibility, e.g., when updating the syntax or semantics. EMA is a short-form for *EmbeddedMontiArc*. This `EMA-Version` represents complete *EmbeddedMontiArc* modeling family. This means, if the syntax of any language of this family changes (e.g., *MontiMath*) or a new language is added, then this version is increased. Optionally, the `Main.txt` file contains a name of a registered (e.g., in an IDE) simulator and its current version. This enables tools to easily start the simulation with the given model. Besides the simulator and an *EmbeddedMontiArc* compiler, the `Main.txt` file can also contain other key (plugin-names) - value (plugin version) pairs. Based on the specified key-value pairs the IDE adds extra buttons in the toolbox, if these plugins with their versions are registered in the used IDE. Example plugins are the SVG generator, stream unit tester, extra-functional property verifier (cf. Chapter 6), and the C&C view design verifier (cf. Chapter 7).

The `Main.txt` file must contain the name of the root component, the `Main-Component-Instantiation` (cf. l. 3). The `Main-Component-Instantiation` has the same syntax as component instantiations (also with generic and configuration parameter bindings). The type of the instantiation must be a full-qualified type. The main component instantiation must not have a dimension.

The `Main.txt` file also contains model paths that are folders where the symbol table looks up when loading component types or other information. All the tag models must belong to one registered model path, otherwise these are ignored and the C&C model is not enriched with this extra information.

The `Main.txt` file enables IDEs to run different *EmbeddedMontiArc* models in different simulators (e.g., one model in a robotic arm simulator, and another one in a simulator for autonomous vehicles). To deliver *EmbeddedMontiArc* models to customers, generated C++ artifacts, C++ compiler, used mathematics libraries, as well as AI and optimization frameworks plus the needed simulator are packed into an extra ZIP file. The `EMA2WASM` compiler translates the *EmbeddedMontiArc* models to web assembly. This way the byte code (similar to JAR byte code) together with a web simulator (in JavaScript or web assembly) can be uploaded to a webserver. The webserver enables users to simulate scenarios in a web browser without installing any plugins or downloading the compiler and simulator frameworks. Examples of compiled models uploaded to the institute web server are:

- `http://www.se-rwth.de/materials/embeddedmontiarc/` shows the spectral clusterer for image processing, PacMan controller to eat food and avoid ghosts, and a SuperMario controller.
- `http://www.se-rwth.de/materials/ema_compiler/` presents an online car simulator with noise regulator.
- `http://www.se-rwth.de/materials/ema_tutorial/` contains two car tutorials: elk test and parking.

Similar to Oracle's JAR signing [Ora17c], a personal private key of a public client certificate can sign the ZIP file. It is recommended to use public client certificates authored by a trusted source, e.g., from the RWTH IT center [RWT18b]. The signed ZIP file includes the public key of the certificate [Ora17d]. If a model is signed, then a `Sha512.txt` file (skipped in this thesis) is additionally created. This file contains the base64 encoded Sha512 digest of every file except of its own `Sha512.txt` file (also the public certificate and the `Main.txt` file). The `Sha512.txt` file has the same task as the signature file in JARs. *EmbeddedMontiArc* uses SHA512 (SHA2) hashes instead of SHA1 hashes in signed JARs.

The signing mechanism with the certificate and the digests of all files creates trustful *EmbeddedMontiArc* models for library components. The hash values enable to verify that no content has been modified. More information about signature files and how to verify certificates are available from the Manifest Format page of the JDK documentation [Ora17e, Ora17b].

EMA

```
1  component interface VelocityController { … }
```

EMA

```
2  component RedundantVelocityController(VelocityController VC1,
3                                        VelocityController VC2) {
4    ports …;
5    instance [VC1, VC2] controller[2];
6    instance Merge<2> merge;

7    connect this.* -> controller[:].*;
8    connect controller[:].* -> merge.*[:];
9    connect merge.* -> this.*;
10 }
```

Figure 3.52.: *EmbeddedMontiArc* code with component type array (red text).

EMA

```
1  component RedundantVelocityController(VelocityController VC1,
2                                        VelocityController VC2) {
3    ports …;
4    instance [VC1, VC2] controller[6];
5    instance Merge<6> merge;
6    // connectors omitted, is the same code as in Figure 3.52
7  }
```

Figure 3.53.: *EmbeddedMontiArc* code with component type array being instantiated multiple times(red text).

3.6.8. Arrays of Component Types, Generic and Configuration Parameters

This subsection shortly explains how arrays of component types, generics, and configuration parameters help to further increase modularity of library components.

Figure 3.49 on page 86 and Figure 3.50 on page 87 show the redundant velocity controller. To become more resistant against logical design errors, the two controllers should not have the same component type. Therefore, the `VelocityController` component type is converted to a component interface which can be implemented by different component logics. To reuse the powerful connection patterns (cf. Subsection 3.6.6), an array of component types (cf. underlined text `[VC1, VC2]` in Figure 3.52) of the component interface `VelocityController` instantiates the two controller instances. This means `controller[1]` has the component type `VC1`, and `controller[2]` has the component type `VC2`.

To achieve an even higher ASIL level, the two logical different velocity controllers should be instantiated three times each; so that they are more resistant against hardware failures. Figure 3.53 presents the code snippet, where only lines 4 and 5 differ from Figure 3.52. The statement `instance [VC1, VC2] controller[6]` creates three component instances of type `VC1` and `VC2`. The first three controller instances (index 1 to 3) have type `VC1`, whereas the last three

EMA

```
1 // each type of the array VK is instantiated n times
2 component RedundantVelocityController <N+ k>
3                                          (VelocityController VC[k], N+ n=1) {
4   ports …;
5   instance VC[:] controller[n*k];
6   instance Merge<n*k> merge;
7   // connectors omitted, is the same code as in Figure 3.52
8 }
```

Figure 3.54.: *EmbeddedMontiArc* code with an an array of configuration parameters (red text).

EMA

```
// k=1 is derived from array size, n=1 the default parameter is used
RedundantVelocityController([VelocityControllerA]);

// k=2 is derived from array size
RedundantVelocityController([VelocityControllerA, VelocityControllerB], 4);

// k=2 is explicitely stated to ensure that the array size of VC is really 2
RedundantVelocityController<2> ( [ VelocityControllerA,
                                   VelocityControllerB ],4);

// it is wrong as k is set to 2 and the array size of VC is 1,
// this results in a compile time error
// RedundantVelocityController<2> ( [VelocityControllerA] );
```

Figure 3.55.: Examples how to instantiate the generic `RedundantVelocityController` component.

controller instances (index 4 to 6) are of type `VC2`. The array instance number (in our case 6) must be a multiple of the array size (in our case 2) of the component type array.

The downside of Figure 3.53 is that it is not generic anymore, because the number `6` is hard encoded and only two different types of velocity controllers can be used. Figure 3.54 addresses this issue by adding the configuration parameter `VC` (cf. `VC[k]` in l. 2) to the component type signature of the `RedundantVelocityController`. The parameter `VC` accepts a k-dimensional array of component types, where all elements implement the `VelocityController` interface. The generic parameter `k` can be skipped when instantiating the `RedundantVelocityController`, because `k` can be inferred from the array dimension of the parameter `VK`. The configuration parameter `n` states how often each component type of the array `VK` is instantiated. If the parameter is not bound during instantiation of the `RedundantVelocityController` component, its default value 1 is used. Figure 3.55 shows how to instantiate the `RedundantVelocityController` component.

Besides configuration parameters, generic parameters also support arrays of component types. An example is shown in Figure 3.57, which is equivalent to Figure 3.56. But Figure 3.57 can be easily generalized, by replacing the two in `in[2]` with another generic parameter. The zero in `(0 : limit[:])` is refilled to a vector so that it fits the array size of limit. This is the same

EMA

```
component X<N+ limit1, limit2> {
  ports in   (0 : limit1) val1,
             (0 : limit2) val2,
          out Z res;
}
```

EMA

```
instance X<10, 7> x1;
```

Figure 3.56.: Simple component type X with two generic parameters (cf. Subsection 3.5.4).

EMA

```
component X<N+ limit[2]> {
  ports in   (0 : limit[:]) vals[2],
         out Z res;
}
```

EMA

```
instance X<[10, 7]> x1;
```

Figure 3.57.: Component Type in generic parameter definition.

methodology as resizing single ports in sources to match the array size of port arrays in targets of connectors (cf. Subsection 3.6.6).

Let `A` be an one-dimensional array, then `A[:]` is equivalent to `A[1:end]` and both return the complete content of the array `A` by selecting a sub-array containing the first up to the last element of the array `A`. Therefore, instead of the long-form `A[:]` the equivalent short-form `A` could be used in *EmbeddedMontiArc*. This results in the following consequences:

- `connect port -> portArray` instead of `connect port -> portArray[:]`
- `instance TypeArray x[3]` instead of `instance TypeArray[:] x[3]`
- `port in (0 : limitArray) vals[2]` instead of `port in (0 : limitArray[:]) vals[2]`

If the names are all post-fixed with `Array` the reader knows that a port array is used instead of a single port. On the other hand, if intuitive names (e.g., all example code snippets in this section) are used, then the reader does not know whether it is an array or single element of ports, component types, or parameters. Therefore, *EmbeddedMontiArc* requires the array access `[:]` operator to use all array elements in `connect`, `instance`, or `port` statements.

3.7. Concepts of New Language Features

This section raises ideas and new features the *EmbeddedMontiArc* language may support in future. These features are not supported due to missing implementation man-power or because the concepts are not 100% clear, yet. The author still wants to summarize the new ideas shortly, since these concepts or variations of them may help in future.

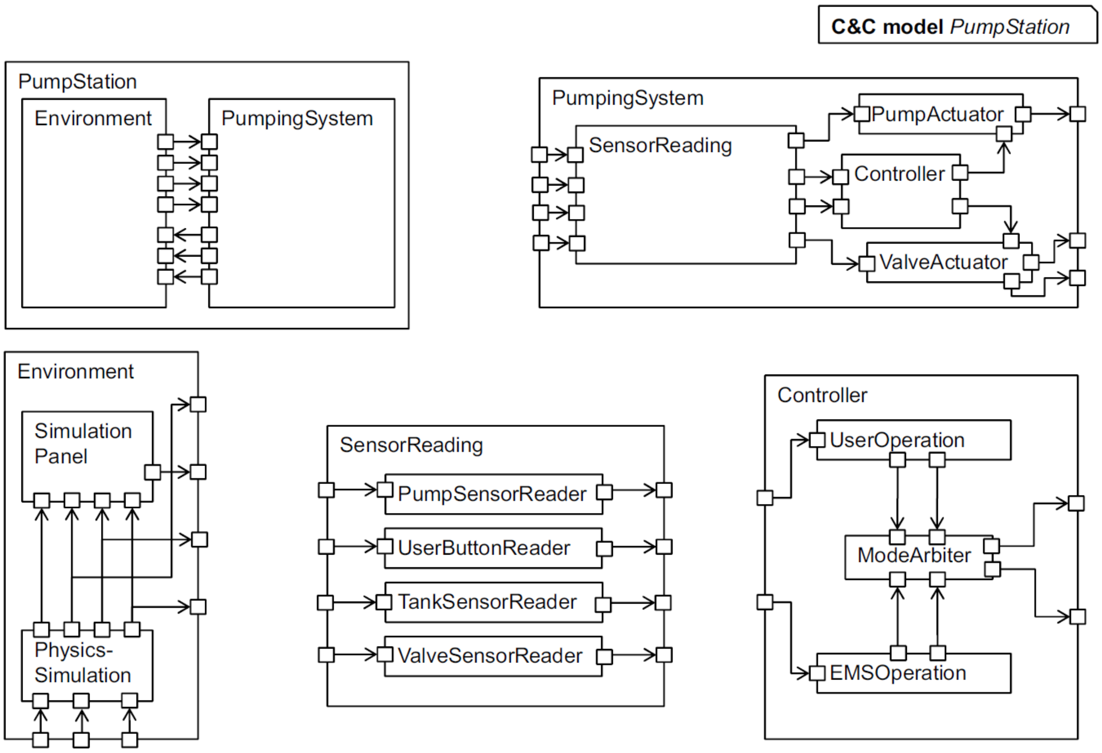


Figure 3.58.: PumpStation C&C model (copied from [Rin14]).

Delta Modeling and Enabled Component Patterns for Product-Lines. The delta concepts of Δ-*MontiArc* with `add`, `remove`, `modify`, and `replace` [MNR+13] operations can be mapped directly to *EmbeddedMontiArc* for product-line modeling.

Delta Modeling for Bug Fixing. Alt [Alt16] describes in his blog why programming language should not be closed by default as it is the case with Kotlin. Closed classes cannot be extended anymore to misuse these classes. But on the other side, this extension restriction makes it hard to fix bugs or enhance libraries at needed points where the library designer did not think of it at the beginning.

EmbeddedMontiArc does not support the extends mechanism of *MontiArc* anymore, so all component types are closed by default. In contrast to Java, where classes only contain type information and no instances; *EmbeddedMontiArc* and *MontiArc* component type is actually a mixture of instances (it instantiates a concrete number of subcomponents) and classes (components can be multiple time instantiated). This type mixture is the reason why a simple extension mechanism, as it is the case in *MontiArc*, is not strong enough for bug-fixing. The rest of this subsection gives an example why the delta language approach is a convenient way for bug-fixing and breaking the closed nature of *EmbeddedMontiArc*'s component types.

Figure 3.58 shows the pump station model from Jan Ringert's [Rin14] model library, which we import and reuse in our own *EmbeddedMontiArc* project. The library model includes 5 non-atomic

EMA

```
// old: buggy
component HydrolicPowerStation {
  instance PumpStation ps;
}
```

Δ-EMA

```
// new: bug fixed
component PumpStationFixed modifies PumpStation{
  replace component type EMSOperation
   with EMSOperationFixed
   for instance pumpingSystem.controller.emsOperation;
}
```

EMA

```
// use fixed version
component HydrolicPowerStation {
  instance PumpStationFixed ps;
}
```

Figure 3.59.: Example code snippet to replace the defect component with their bug fixed version using the Delta approach.

components and of 11 atomic components. **Imagine, the atomic `EMSOperation` component contains a bug**, hence this component must be exchanged with a corrected version. Since the bug is inside a model library, we cannot fix the bug directly by editing the `EMSOperation` file. Therefore, the only way to fix the complete pump station model is to copy the `Controller` text file to `ControllerFixed`, and replace `EMSOperation` component instantiation with the correct one. Since, the buggy `Controller` component was used by other components, we need to copy these too; otherwise, they do not use the new `ControllerFixed` component. This results in copying `PumpingSystem` to `PumpingSystemFixed` just to use `ControllerFixed` instead of `Controller`, and to copy `PumpSystem` to `PumpSystemFixed` to use `PumpingSystemFixed`. This is a lot of work! This task is also very errorprone: The bug fix is only successful, if every component in the complete hierarchy between the top-level component and the defect one is copied and modified to use the bug-fixed subcomponent. Otherwise, the wrong, still buggy, component type is instantiated and the bug is not fixed.

For this mentioned scenario, *EmbeddedMontiArc* should have a concept to exchange or modify the interior structure of large library components.

The Delta language approach as presented in *DeltaMontiArc* [HKR+11a] or *DeltaSimulink* [HKM+13, KRR15] looks like a suitable solution. Figure 3.59 shows how to replace the `EMSOperation` component with the corrected version. The `EMSOperationFixed` component type must have a compatible interface (input and output ports) to the `EMSOperation` component type to be replaced sucessfully.

The delta language approach also supports to add/delete components, ports and connections; thus new features can be easily integrated (e.g., adding a second pump actuator instance inside `PumpingSystem` for safety reasons).

C++

```
// returns the price in Eurocent
int fare(int  startPriceInCent,
         int  priceInCentPerKm,
         int  routeInKm,
         bool nightRide,
         bool baggagePresent) {

  int basePrice = priceInCentPerKm * routeInKm;

  int discount = 0;
  if (routeInKm > 50) {
    discount = std::round(0.1 * basePrice); // 10% discount, rounded
  } else if (routeInKm > 10) {
    discount = std::round(0.05 * basePrice); // 5% discount, rounded
  }

  int extraCharge = 0;
  if (nightRide) {
    extraCharge = std::round(0.2 * basePrice); // 20% extra charge, rounded
  }
  if (baggagePresent) {
    extraCharge += 300; // Cent; extra charge for baggage
  }

  return startPriceInCent +  basePrice + extraCharge - discount;
}
```

Figure 3.60.: Clean Code version of fare fees calculation in C++ (translated from [SV18, Listing 1b]).

Logic

```
basePriceInCent  > 0 AND basePriceInCent  <= MaxBasePrice
    AND
priceInCentPerKm > 0 AND priceInCentPerKm <= MaxKilometerPrice
```

Figure 3.61.: Contract for parameter in `fare` function (cf. Figure 3.60) for better testing (translated from [SV18, p. 11 left column bottom]).

3.8. Example Use Case for *EmbeddedMontiArc* in Business Domain

All sections in this chapter used *EmbeddedMontiArc* to model cyber-physical or embedded systems, because this is the main domain why this modeling family has been invented. This subsection shows how *EmbeddedMontiArc* can help to create clean code in a business domain. The clean code fare fees calculation function of Andreas Spillner [SV18, Listing 1b] serves an example.

Figure 3.60 shows the clean code of this fare function. Since C++ does not support units as first level language concept, all variable names in the function's signature are postfixed with their unit (e.g., `routeInKm`). But this postfix does not help to assign wrong units such as `routeInKm = priceInCentPerKm` as both have the type `int`. A generic unit framework would help, but this would lead to complexer type definitions.

To test only relevant intervals, Spillner et. al. suggests to create a contract for the interface (cf. Figure 3.61) to restrict the prices based on the price policy in the specification or of the customers opinion. The down-side of this approach is that the contract of Figure 3.61 is added either (i) informal as comment to a test suite, or the contract is added as (ii) assertion to the code in Figure 3.60 - resulting in runtime exceptions[10]. Both possible ways do not enforce the contract when using this function at compile time: if the base price is above the limit, then (i) it may crash as this case is not tested, or (ii) it will crash due to the assertion. Both cases result in unexpected behavior for the taxi driver.

Even Spillner et. al. do not follow the complete clean code guidelines, as in `extraCharge += 300; // Cent` the comment is used to explain what the expression means and not why he uses it. Clean code also postfixes variable names inside functions with units; thus, the line must be changed to `extraChargeInCent += 300`. This new version of the line is also readable without the old comment. Since this code is printed in a journal article and thus reviewed many time, it shows how much discipline is needed to create good readable code when units are involved and the used programming language does not support natively any unit concept. The rest of this section presents a much more type-safe version of this code in *EmbeddedMontiArc*.

Figure 3.62 contains the equivalent *EmbeddedMontiArc* code of Figure 3.60. The code is as easy to read as the C++ one, but it includes type safety checks of units. Additionally, the component supports other currencies, because *EmbeddedMontiArc* converts them automatically based on the exchange rate given in the `Main.txt` file (cf. Subsection 3.6.7). Thus, the user of the function must not care if the function works with Euro, American Dollar, or British Pounds.

Figure 3.63 shows how to instantiate the `Fare` component. This listing also shows that in contrast to the C++ version in Figure 3.60 and Figure 3.61, the *EmbeddedMontiArc* version supports different design contracts as these are bound via generics and not defined globally.

This subsection elucidated that *EmbeddedMontiArc* may also be a perfect choice for finance calculations. The strong unit concept, also supporting currencies, plus the universal generic concept enables to define contracts in what area the component operates (calculation is defined).

3.9. *EmbeddedMontiArcStudio*: Tooling for Users

The previous sections of this chapter presented the *EmbeddedMontiArc* family and its main language. The tooling around *EmbeddedMontiArc* breathes life into the theoretical concept. The tooling of these languages is the prerequisite to create many *EmbeddedMontiArc* artifacts. Only the nice user experience features for the *EmbeddedMontiArc* language family motivates students and professionals to create larger models of *EmbeddedMontiArc*. The 3D visualization presenting the simulation results are good for visual feedback and acceptance testing at the end to see whether the controller behaves in a correct way. The 3D visualization unveils “ugly” movements of cars or figures (e.g., PacMan or SuperMario), and then the components are refined by adding more intelligence resulting in smoother motions and resulting in larger *EmbeddedMontiArc* models. Only the creation of medium up to medium-large models enables the possibility to validate the language features presented above. Additionally, reuse of components as well as

[10]The author did not explicitly mention where he adds the contract. The contract should only help for equivalence testing.

EMAM

```
// generic design contract using constants, etc.
// are repliced by first level design contracts in the
// interface enabling static type checking
component Fare<
    (1 ct : oo ct) MaxBasePrice,
    (1 ct/km: oo ct/km) MaxKilometerPrice,
    (1 km : oo km) MaxRoute,
    (0 ct : oo ct) SmallestCoin = 1 ct, // in Germany it is 1ct,
                                      // in Netherland is rounded up to 5ct
    (1 ct : oo ct) MaxFarePrice =
                 100 EUR + 4 * (MaxBasePrice + MaxKilometerPrice * MaxRoute)
> {

  ports //skip here units in names as types contain units
    in  (0 ct : SmallestCoin : MaxBasePrice) startPrice,
        (0 ct/km : MaxKilometerPrice) priceInCentPerKm,
        (1 m: 1 cm : MaxStrecke) routeLength,
        Boolean nightRide,
        Boolean baggagePresent,
    out (0 ct : SmallestCoin : MaxFarePrice) farePrice;
  implementation Math {
    // data type const, means that this value is fixed and never changes,
    // thus the range of data type can be automatically inferred based
    // on the expression on the right hand side of the assignment
    const basePrice = round(routeLength * priceInCentPerKm, SmallestCoin);

    (0 ct : 1 ct : MaxFahrPreis) discount = 0 EUR;

    if routeLength > 50 km
      // rounds up or down so that the result is a multiple of SmallestCoin
      rabatt = round(basePrice * 10% , SmallestCoin);
    elseif routeLength > 10 km
      rabatt = round(basispreis * 5%, SmallestCoin);
    end

    (0 ct : 1 ct : MaxFahrPreis) extraCharge = 0 EUR;
    if nightRide
      extraCharge = round(basePrice * 20%, SmallestCoin);
    end
    if baggagePresent
      extraCharge += 3 EUR;
    end

   farePrice = max(startPrice + basePrice + extraCharge  - discount,
                                                   MaxFarePrice);
  }
}
```

Figure 3.62.: *EmbeddedMontiArc* code of Figure 3.60. This code contains the complete contract information via generics. This model has not been shorten to illustrate how a published business code can be completely modeled in *EmbeddedMontiArc*.

EMA

```
// use later
instance Fare<20 EUR, 10 EUR/km, 5'000 km> farePriceGermany;
instance Fare<25 EUR, 15 EUR/km, 15'000 km> farePriceItaly;

// NL has no 1 and 2 cent coins anymore
instance Fare<25 EUR, 15 EUR/km, 15'000 km, 5 ct> farePriceNetherlands;
instance Fare<10 USD, 20 USD/mi, 20'000 mi, 0.01 USD> farePriceUSA;

// 1 GBX = 1 Penny
instance Fare<15 GBP, 20 GBP/mi, 10'000 mi, 1 GBX> farePriceEngland;

// main file must contain now exchange rate between EUR and USD and GBP
// see http://jscience.org/api/org/jscience/economics/money/Currency.html#setExchangeRate(double)
// 1 EUR = 1.17 USD
// 1 GBP = 1.13 EUR
```

Figure 3.63.: Examples how the component function of Figure 3.62 can be used.

building component libraries, is only needed when a component model has more than just a couple of components. The *MontiSim* simulator of *EmbeddedMontiArc* is completely decoupled from the 3D visualization. Therefore, the simulator can be used for automatic black-box testing of closed-loop controllers interacting with the environment in CI tools such as Jenkins.

Table 3.64 shows that *MontiArc* and *MontiArcAutomaton* has been evaluated on 15 examples (two by Haber, seven by Ringert, and six by Wortmann).

Table 3.65 shows that the *EmbeddedMontiArc* language family has been evaluated on more than 15 examples, whereby eight of these models contain even more than 200 component instances. Ievgen created in his master thesis ten different racing lap models; Table 3.65 shows only the largest of these ten models. All these models are public available under the links presented in Subsection 3.6.7.

The first four models are translated Simulink models provided by Daimler AG. ADAS is the abbreviation for advanced driver assistance system. ADASv1 represents the first version. ADASv4 represents the latest evolution version provided to us. The ADAS models receive as input the logical sensor data, e.g., vehicle speed, recognized speed sign, set tempomat speed by driver, distance and speed to obstacle (also other car, bike, or pedestrian) in front of this vehicle. Based on these input signals, the *EmbeddedMontiArc* ADAS models calculate the optimal brake force or the car acceleration.

The fifth model is an adaptive light system provided as Simulink model by Daimler AG. The model's input signals are user controls such as turning on headlights, hazard flashing, or high beams. Based on these user controls the *EmbeddedMontiArc* ALS models calculates the brightness of many light bulbs.

The PacMan model controls the PacMan figure. It receives as input the current position of ghosts and of the food item as well as an integer matrix for the map having three different values to represent a wall, a way with coins, and a way without coins. The output of this controller is the movement position (left, right, forward, or backward) of PacMan. The most complex part of the PacMan controller is the optimal path finding algorithm using a cone-like search; it minimizes the

Author	Model	Nb. component instances
Haber	TCP/IP [Hab16, Section 8.2]	40
Haber	FlexRay [Hab16, Section 8.3]	25
Ringert	PumpStation [Rin14, Section 3.1]	16
Ringert	BumperBot [Rin14, Table 4.17]	12
Ringert	NavigationUnit [Rin14, Section 8.4]	12
Ringert	CoffeePreparingRobot [Rin14, Section 8.5]	11
Ringert	BumperBotEmergencyStop [Rin14, Section 8.1]	10
Ringert	RotationalJoint [Rin14, Section 8.4]	8
Ringert	AvionicsSystem [Rin14, Section 8.4]	6
Wortmann	NXT Java Coffee Delivery [Wor16, Subsection 9.1.1]	60
Wortmann	Robertino SmartSoft Java Transport Services [Wor16, Subsection 9.1.3]	58
Wortmann	iserveU Hospital Logistics Project [Wor16, Subsection 9.2.3]	57
Wortmann	Robotino *ROS* Python Transport Services [Wor16, Subsection 9.1.2]	31
Wortmann	Lego NXT Distributed Toast Service [Wor16, Subsection 9.2.1]	$\approx 15^{\dagger}$
Wortmann	Multi-Platform BumperBot [Wor16, Subsection 9.2.2]	$\approx 15^{\dagger}$

Table 3.64.: Models to evaluate *MontiArc* and *MontiArcAutomaton* languages.

† Guessed on the figures and the project description, no number is present in the thesis [Wor16].

number of movements to eat the food, but to avoid the ghosts. Documentation of the controller is available in an *EmbeddedMontiArc* case study paper [HH18].

In the racing lap model [KRSvW18b, Str18a], the controller moves a car which needs to pass a number of tests on the lap: (a) the elk test (driving around cones), (b) overtaking a car, (c) avoiding obstacles on the track, and (d) finish the lap by parking in a parking lot. The input values of the controller are distances from radar sensors, and the output values are the steering angle and the acceleration/deceleration value.

The SuperMarioBros model controls the SuperMario figure to pass one world. The figure must jump over obstacles, collect coins, and defeat enemies. The input of this model is very close to the one of PacMan, the output are the direction arrows and two Boolean values whether SuperMario should jump or fire.

The simple autopilot controller moves a car from the current position to a specified point in OpenStreetMap; it uses the *MontiSim* simulator [GKR+17]. The most interesting part of this controller is to calculate trajectory points, having small distances such as 10cm based on navigation points, having large distances containing only intersections. The navigation system component is not modeled, it is written in Java. The input and output ports are very close to the racing lap model.

The object detector model [KRSvW18a] using a clustering algorithm on a given image (the input port is a matrix array representing the channels red, green, and blue), and the output port is a Boolean image (matrix output port) where true represents the identified object.

Author	Model	Nb. component instances
Daimler AG	ADASv1 (cf. Chapter 8)	639
Daimler AG	ADASv2 (cf. Chapter 8)	1 396
Daimler AG	ADASv3 (cf. Chapter 8)	2 278
Daimler AG	ADASv4 (cf. Chapter 8)	2 309
Daimler AG	ALS (cf. Chapter 8)	1 086
Heithoff	atomic version of PacMan	143 296*
Heithoff	normal version of PacMan [HH18]	239+
Ievgen	racing lap model [Str18a]	220
Haller	SuperMarioBros [HH18]	55
Moktharin	simple autopilot controller	32
Schneiders	object detector [KRSvW18a]	21
Ringert	pump station (remodeled from [Rin14] in *EmbeddedMontiArc*)	16
von Wenck-stern	turbine controller [MRRvW16]	12
Kusmenko	traffic sign detection	≈ 10
Mehlan	weather balloon sensor [MMR+17]	5

Table 3.65.: Models to evaluate *EmbeddedMontiArc* language family.

* Behavior of atomic components, e.g., `And`, `Multiplication`, and `Smaller`, is mostly one simple expression.

+ The large difference between component instances and component instantiations results that component instances analyzing the world, i.e., ghosts, food, and obstacles, are created via arrays of component instantiations. The atomic version of PacMan was created after the normal version of PacMan to test the performance of *EmbeddedMontiArcStudio*; only the visualization had problems as generating four HTML and four SVG files for each component instance causes the PC to run out of hard disk space. Loading the large model and creating all 143 296 component instances was no problem for *EmbeddedMontiArc* and the *MontiCore* symbol table infrastructure.

The traffic sign detection model uses *EmbeddedMontiArcDL*. It receives an image with a speed sign, and it produces the recognized output value such as `30 km/h`. This model is a trained CNN model.

The pump station model controls the pump valve and pump actuator. The turbine controller controls the pitch angle to generate most electricity, but to avoid damages due to too large wind speeds. The weather balloon sensor collects GPS, temperature, and pressure information and decodes its values to send them via an antenna to the base station.

EmbeddedMontiArcStudio is the IDE for the *EmbeddedMontiArc* language family. Together with Evgeny Kusmenko in more than 30 bachelor and master theses and in 2 labs (together over 60 lab participants) many powerful features around *EmbeddedMontiArc* have been developed. The features of *EmbeddedMontiArc* can be used-standalone, e.g., via command-line interfaces on servers or continuous integration environments such as Jenkins, TravisCI or GitLabRunners.

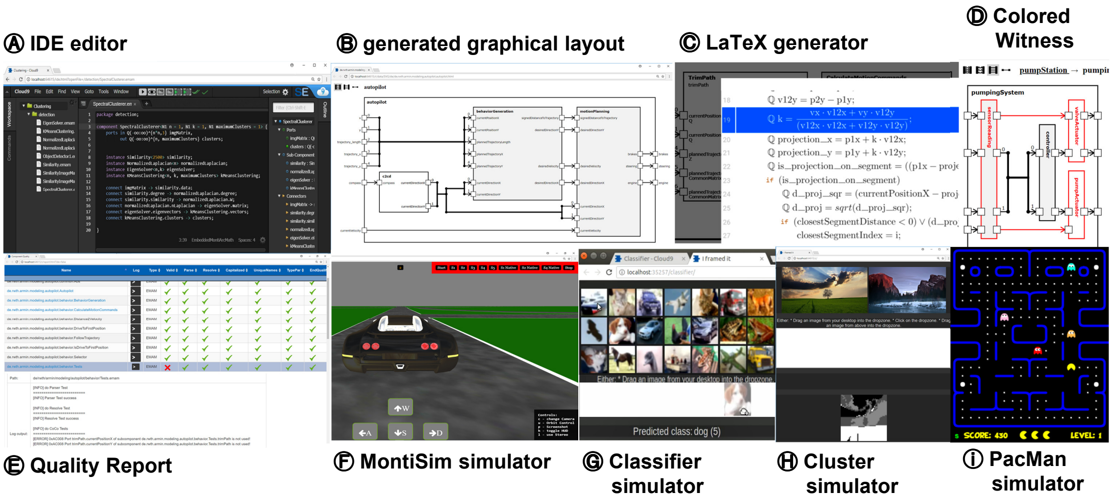


Figure 3.66.: Screenshots of *EmbeddedMontiArcStudio* 1.7.5 and the integrated tools.

EmbeddedMontiArcStudio integrates nearly all of these features in a development environment for modelling embedded and cyber-physical systems with the *EmbeddedMontiArc* modeling family (cf. Section 3.4). The premise for *EmbeddedMontiArcStudio* is that one button click of a developer is enough to execute a specific user experience feature.

EmbeddedMontiArcStudio is available from `http://www.se-rwth.de/materials/embeddedmontiarc/`. Version 1.7.5 contains among others the following features (cf. Figure 3.66):

- IDE with Outline, Syntax Highlighting and Parser Error Messages (cf. Ⓐ) [KRRvW18, Ron17];
- Optimized native C++ generator and compiler supporting SIMD and GPU [KRSvW18a, Sch17];
- Automatic generation of graphical C&C layouts for textual *EmbeddedMontiArc* models (cf. Ⓑ) [Sch18] - the generated graphical layout is available in four different abstraction levels;
- Automatic test environment for component black-box testing;
- Verification of extra-functional property consistency (cf. Chapter 6);
- Verification of design C&C views against C&C models (cf. Section 7.4) and creation of (colored) witnesses (cf. Section 7.5, Ⓓ);
- Component quality analysis inclusive report output (cf. Ⓔ);
- Many complete examples such as Autopilot model for self-driving cars (cf. Ⓕ), Image classifier (cf. Ⓖ), Cluster model to cluster images for object detection (cf. Ⓗ), and PacMan (cf. Ⓘ);
- 3D-Driver Simulator inclusive Physic Engine (provided by Evgeny Kusmenko) [Ilo18b, Ryn18]; and
- Model Explorer[11] with over 1 500 *EmbeddedMontiArc* component types to import from.

[11] `https://embeddedmontiarc.github.io/webspace/reporting/report/componentQuality.html`

Chapter 4.

Internal Representation of *EmbeddedMontiArc*

The previous chapter presented the language concepts and the concrete syntax of *EmbeddedMontiArc* based on examples. *EmbeddedMontiArc* is a functional component and connector (C&C) language to model the logical layer of embedded systems in an efficient, agile, and intuitive way. In *EmbeddedMontiArc* the instantiated main component represents the static architecture of a system. This architecture is decomposed of instantiations having different (generic) component types. Due to the modular and reusable nature of *EmbeddedMontiArc*, the decomposed component types are stored into multiple text files.

To enable an efficient navigation through the data structure of *EmbeddedMontiArc*'s component libraries and/or subcomponent instantiations of the main component, the first part of this chapter explains the abstract syntax (also named meta model in some papers) provided by the *EmbeddedMontiArc* language. As this thesis describes the abstract syntax of *EmbeddedMontiArc* via class diagrams to easier define *OCL* constraints on, the first section explains how *MontiCore* derives these class diagrams based on given grammar files.

The next section in the first part presents these class diagrams; and this section also contains important rules to express whether a C&C model is valid.

The second part of this chapter introduces the C&C instance structure. Models of the C&C instance structure language can be derived from valid C&C models of the *EmbeddedMontiArc* language by binding all generic, configuration parameters and component interfaces as well as by creating all component instances starting from the main component instantiation. The instance model describes the complete structure of one cyber-physical system. The instance structure of the architecture is better suited for further validations of structural and behavioral properties [RSvW$^+$15, RRS$^+$16, BRRvW16, BMP$^+$16, BRvW16, RSvW16, HRvW17].

The fourth section elucidates how to derive the C&C instance structure from a C&C model. This section explains this transformation on many examples.

The next section of this chapter compares the abstract syntax models of the second and third sections with the ones of other *MontiArc* derivatives, i.e., Ringert's formal C&C model definition, as well as the abstract syntax of Haber's *MontiArc* and Wortmann's *MontiArcAutomaton* languages.

The last section describes how both abstract syntax models are realized using *MontiCore*'s symbol management infrastructure. The last section also explains how the abstract syntax of other languages can be easily integrated into the presented abstract syntax models, so that these new languages can reuse all analyzes working on these two abstract syntax models.

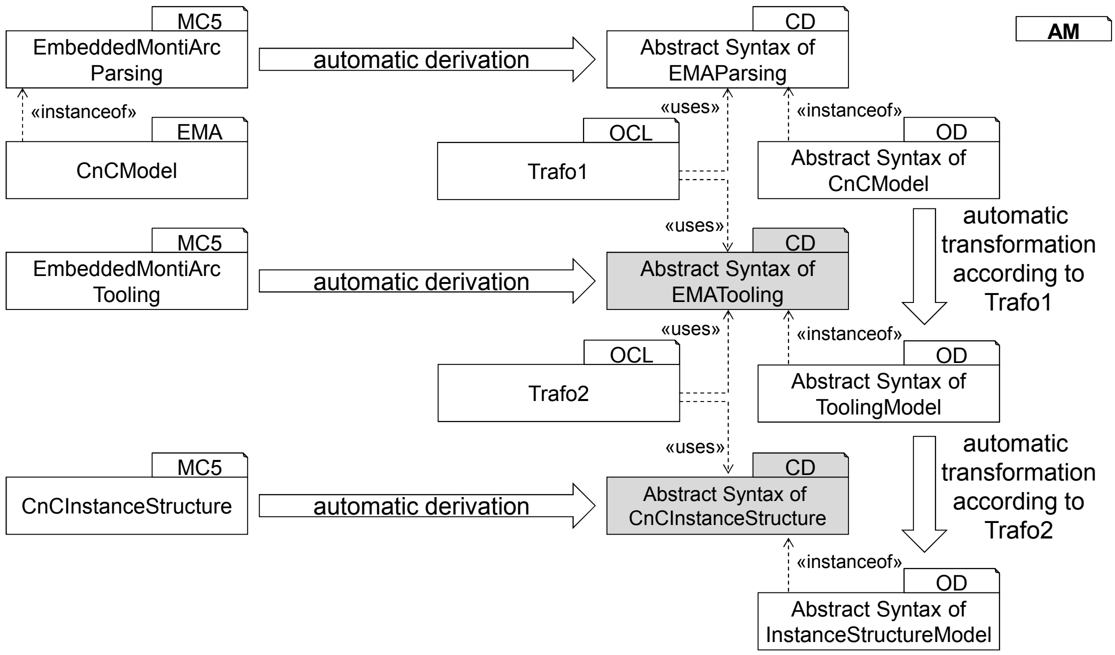


Figure 4.1.: Artifact model shows dependencies between the three different *EmbeddedMontiArc* grammars and the derived abstract syntax of these grammars. The gray parts highlight the internal representation of *EmbeddedMontiArc* presented in this chapter. Context conditions about *EmbeddedMontiArc* and their extra-functional properties are formulated via *OCL* on the gray marked abstract syntax structures (cf. Chapter 6).

4.1. Deriving Class Diagrams from *MontiCore* Grammars

The internal representation of *EmbeddedMontiArc* is directly derived from *MontiCore* grammars. The *MontiCore* grammar format defines the concrete and abstract syntax of a language. *MontiCore*'s production rules of grammar files generate class diagrams for the abstract syntax of this language (cf. [HR17, Chapter 5]). The transformation from *MontiCore*'s EBNF-based grammar format to class diagrams representing the abstract syntax of this language is full-automatic (cf. [BJRW18]).

Section 4.2 and Section 4.3 present the internal structure of *EmbeddedMontiArc* as class diagrams, because Chapter 6 and Chapter 7 formulate *OCL* constraints against these class diagrams. For the most readers it is more convenient to read *OCL* constraints against graphical class diagrams. However, the author of this thesis wants to emphasize that the internal structure of *EmbeddedMontiArc* does not use class diagrams as primary artifacts. The internal structure of *EmbeddedMontiArc* is defined by different grammar files.

Figure 4.1 shows the relations between the artifact types of *EmbeddedMontiArc*. *EmbeddedMontiArc* has three *MontiCore* main grammars[1] for different purposes: The first one exists to parse the textual *EmbeddedMontiArc* models with its nice syntactic sugar as presented in Chapter 3. The second main grammar is for tooling and context conditions; its main focus is to define a convenient abstract syntax to easily express *OCL* well-formedness constraints on, and to calculate auto-completions and outlines. The third main grammar represents the instance structure, the complete static architecture of the main *EmbeddedMontiArc* component (cf. Subsection 3.6.7); the C++ code and SVG visualisation generator uses the C&C instance structure.

A textual *EmbeddedMontiArc* model is an instance of the first main grammar. *MontiCore* generates based on the `EmbeddedMontiArcParsing` grammar a class diagram representing the abstract syntax of this grammar as well as a parser reading the *EmbeddedMontiArc* model and creating an object diagram (it is actually a Java object structure, but it can be reported as an object diagram) being an instance of this abstract syntax class diagram of the `EmbeddedMontiArcParsing` grammar.

MontiCore also generates the abstract syntax representations for the `EmbeddedMontiArcTooling` grammar and for the `CnCInstanceStructure` grammar. A special subset of *OCL* (*OCL* constraints following a specified pattern - cf. Section 6.4) specifies the relationship between the objects of the different abstract syntax structures. Based on these *OCL* transformations, the object structure of the textual *EmbeddedMontiArc* model, `Abstract Syntax of CnCModel`, is transformed to the object structures of the abstract syntax of the other two main grammars. The C&C model developer does not create textual models being instances of the `EmbeddedMontiArcTooling` and the `CnCInstanceStructure` grammars. However, the tool developer, e.g., to test these transformations, creates models of these two grammars to specify test results first as it is suggested by test-driven-development.

Section 6.4 explains how models of the first grammar (object diagrams of the abstract syntax of the first grammar) are transformed to models of the second grammar. The first transformation mainly extends convenient syntactic sugar (e.g., `ports in B val, out B notVal;`) to a long-form containing all information explicitly (e.g., the one-dimensional port array size in `ports in B val[1], out B notVal[1];`). As the first grammar is primarily used for parsing, and its abstract syntax is similar to the second grammar and the abstract syntax of the first grammar is not so important for this thesis; Section 4.2 and Section 4.3 only present the abstract syntax of the second and the third grammar of *EmbeddedMontiArc*. Section 4.4 explains how models of the second grammar are transformed to models of the third grammar.

The rest of this section shortly explains how *MontiCore* translates grammar definitions to class diagrams automatically. More details about this transformation are presented in the MontiCore language reference manual [HR17, Chapter 5] and in the paper *Translating Grammars to Accurate Metamodels* [BJRW18].

Figure 4.2 shows an excerpt of a simplified `EmbeddedMontiArcTooling` grammar. This simplified grammar copied the `Range` non-terminal in it instead of extending an existing grammar; however, this grammar is better suited to demonstrate the class diagram derivation.

[1] EmbeddedMontiArc has actually more grammars as it uses *MontiCore*'s modular language patterns to engineer these three main grammars. For example, `SIUnit`, matrix-based `Expression`, and `Type` grammars are *MontiCore* grammars being reused by all these three main grammars.

MC5...

```
1  grammar EmbeddedMontiArcTooling {
2    interface ComponentType;
3    symbol scope Component implements ComponentType =
4      "component" Name /*...*/ "{"
5        "ports" (Port || ",")* ";"
6       (subs:ComponentInstantiation | Connector)* "}";

7    enum Direction = IN:"in" | OUT:"out";
8    symbol Port =
9       Direction type:PortType Name "[" dimension:NaturalNumber "]";

10   symbol ComponentInstantiation /*...*/ =
11     "instance" type:Name@ComponentType /*...*/ Name
12     "[" dimension:NaturalNumber "]" ";" ;

13   PortInstantiation =
14     (sub:Name@ComponentInstantiation subIndices:Range | "this") "."
15     port:Name@Port portIndices:Range;

16   Connector =
17     "connect" sourcePort:PortInstantiation
18     "->" targetPort:PortInstantiation ";" ;

19   Range =
20    "[" start:NaturalNumber ":" step:NaturalNumber ":" end:NaturalNumber "]";
21 }
```

Figure 4.2.: Excerpt of `EmbeddedMontiArcTooling` grammar. This grammar file is modified for demonstration purposes, e.g., the `Range` nonterminal rule is actually part of another grammar file which `EmbeddedMontiArcTooling` one extends.

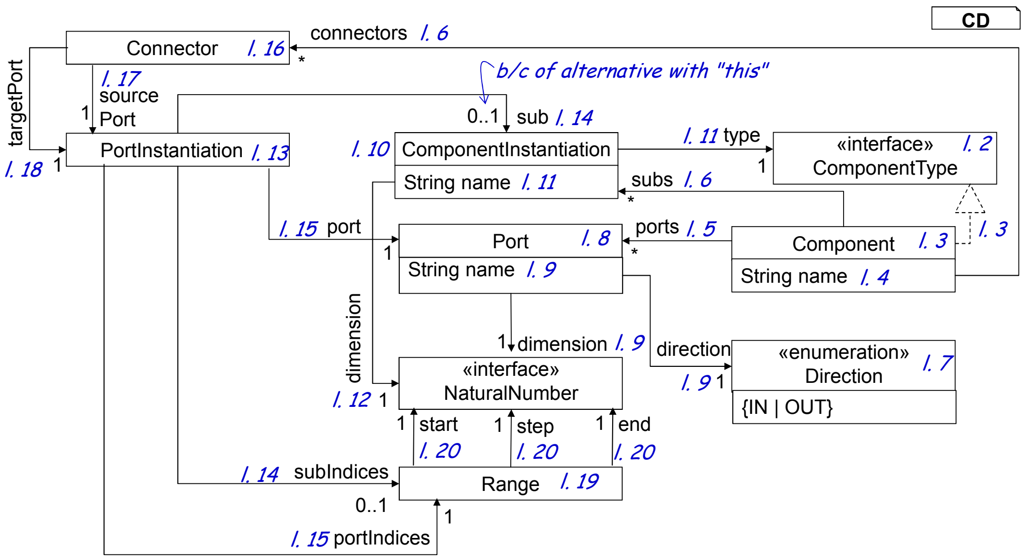


Figure 4.3.: Automatically derived class diagram from `EmbeddedMontiArcTooling` grammar in Figure 4.2.

Figure 4.3 illustrates the automatically derived class diagram based on the *MontiCore* grammar presented in Figure 4.2. The blue texts are comments; all classes and associations are linked to the line number of Figure 4.2.

The left-hand side (LHS) of a production rule is transformed to a class, to an interface (if it starts with the `interface` keyword), or to an enumeration (if it starts with the `enum` keyword) inside the class diagram. Thus, line 2 creates the `ComponentType` interface, line 3 the `Component` class, line 7 the `Direction` enumeration, line 8 the `Port` class, line 10 the `ComponentInstantiation` class, line 13 the `PortInstantiation` class, line 16 the `Connector` class, and line 19 the `Range` class.

The right-hand side (RHS) of a production rule is mapped to properties (if the type is `Name` or `String`) or to outgoing associations to the referenced non-terminal. Therefore, the `Component` class has the property `name` (cf. l. 4), as well as three outgoing associations to `Port`, `ComponentInstantiation`, and `Connector` (cf. ll. 5-6). The properties and outgoing associations of `Direction`, `Port`, `Connector`, and `Range` are derived in the same way.

A special case is a non-terminal reference with an `@` sign: The expression `sub:Name@ComponentInstantiation` in line 14 means that the concrete syntax expects a word matching the Java name token, and *MontiCore* maps this word to a `ComponentInstantiation` object having this word as name. For this reason, `PortInstantiation` has an outgoing association to `ComponentInstantation` with the role name `sub` instead of having a String property `sub`.

The cardinalities of the associations in the class diagram are derived from the possible occurrences of referenced non-terminals in the RHS of production rules. The cardinalities of the `sub` and the `subIndices` associations starting from the `PortInstantiation` class is optional (`0..1`), because the pipe symbol ("|") in the MontiCore grammar defines an alternative, and thus, parsing `this.portA[1:1:1]` leads to a not defined `sub` name and to a not defined `subIndices` range.

4.2. Component and Connector Model

This chapter uses class diagrams to formalize the abstract syntax of *EmbeddedMontiArc*. The graphical notation of class diagrams is based on Rumpe [Rum16, Chapter 2]. Since the complete class diagram of the abstract syntax of *EmbeddedMontiArc* is too complex to be shown at once, this chapter shows several graphical views (as suggested by Rumpe [Rum16, Section 2.4]) of the large class diagram focusing on different aspects of the abstract syntax of *EmbeddedMontiArc*. Classes, fields, and associations in different class diagram views with the same (role) name represent the same element of the complete class diagram.

The complete class diagram is created by merging (cf. [FALW14]) all graphical class diagram views. Appendix B contains the complete class diagram in textual CD4A syntax. The advantage of the textual CD4A syntax is that its concrete syntax is not ambiguous (e.g., in graphical diagrams it is not always clear to which association arrow the role name belongs to) and the CD4A syntax has a unique semantic by providing a mapping to Java source code (cf. [Rot17, Chapter 5]). The CD4A syntax is very good to express large class diagrams in an unique way; the graphical representation of the large class diagram has to many cross-cutting association lines. However, in the opinion of the author of this thesis, the graphical representation of smaller class diagram views is much easier to comprehend than the textual representation. Thus, the merged CD4A class diagram in the appendix together with the *OCL* constraints presented in Chapter 6 formally defines the abstract syntax of *EmbeddedMontiArc*; and this chapter explains this class diagram stepwise on graphical view representations of it.

This thesis assumes that the reader is familiar with class diagrams, and so this thesis does not introduce the graphical syntax and semantics of class diagrams; Rumpe [Rum16, Chapter 2] and Roth [Rot17] introduce class diagrams. If the source/target role name of an associations is equal to the source/target type of this association modulo the capitalization of the first character and modulo singular/plural differences for star cardinalities, the graphical representation may skip the role name due to clarity reasons[2].

This chapter adds *OCL* constraints for completeness and to explain the abstract syntax only once directly below the corresponding class diagram views. However, Chapter 6 - presenting the *OCL* framework - only introduces and explains *OCL* in detail. Therefore, the author suggests for readers being unfamiliar with *OCL* the following reading order: (i) this chapter with ignoring *OCL*, (ii) Chapter 6 to get familiar with *OCL*, and (iii) scan this chapter again with focusing on *OCL*.

Many of the following classes in the abstract syntax contain a `name` field. For the semantics this `name` field is (except for the `Port` class) not necessary and from a mathematical point of view it can be deleted in each class diagram. However, the classes also serve as abstract syntax structure for the underlying tools and, therefore, the `name` field has been added to generate user-friendly error messages containing the name of the model elements (e.g., when some conditions are violated).

All `name` fields are short names (no extended or full names such as `package.component .port`). The port definition needs, in contrast to the rest, a `name` field in its definition to force that port names of a component match the port names of its implemented component interface. The ports of the component must not be identical with the ports of the component interface, as their type can be different (the type must only be compatible).

The rest of this section introduces the abstract syntax for the C&C language *EmbeddedMontiArc* (it is the generated abstract syntax of the *EmbeddedMontiArcTooling* grammar) step-wise.

4.2.1. Port Type System

Figure 4.4 illustrates the relation between `Type` and `Value` interfaces. A `Value` has always one specific `Type`. The `Type` interface is very general. The `PortType` interface extends this general `Type` interface. All types used to communicate between components via ports implement this `PortType` interface, and the concrete values passed between the components implement the `PortValue` interface. A `Quantity` also implements the `Type` interface, because quantities may be types of parameters (cf. Subsection 4.2.2). Every `Quantity` has one base unit; e.g., `Length` has the base unit `Meter`. Every unit belongs to exactly one quantity; e.g., the quantity

[2] For example, an association going from `Component` (source type) to `Port` (target type) having the cardinality star at `Port` and one at `Component` (`association [1] Component -> Port [*]`) and omitting role names, automatically introduces the source role name `component` and the target role name `ports` (`association [1] Component (component) -> (ports) Port [*]`).

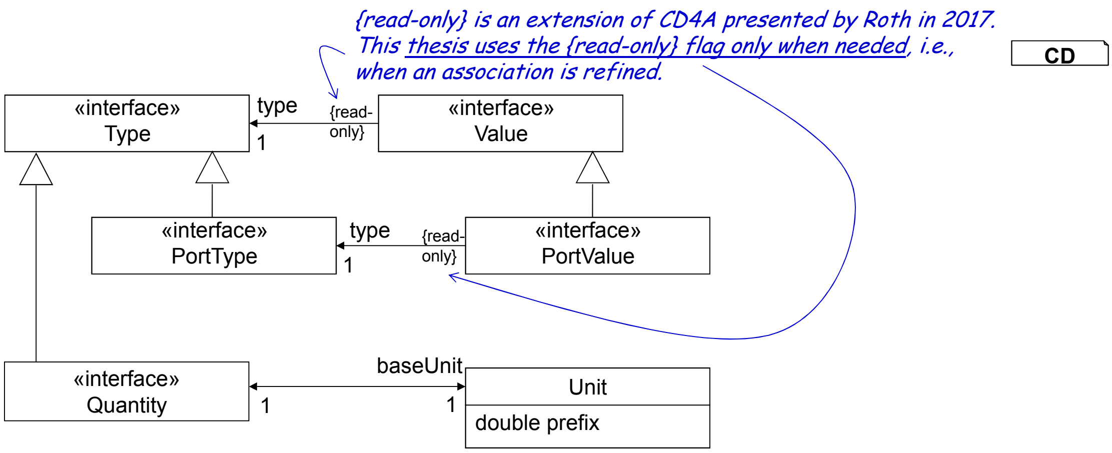


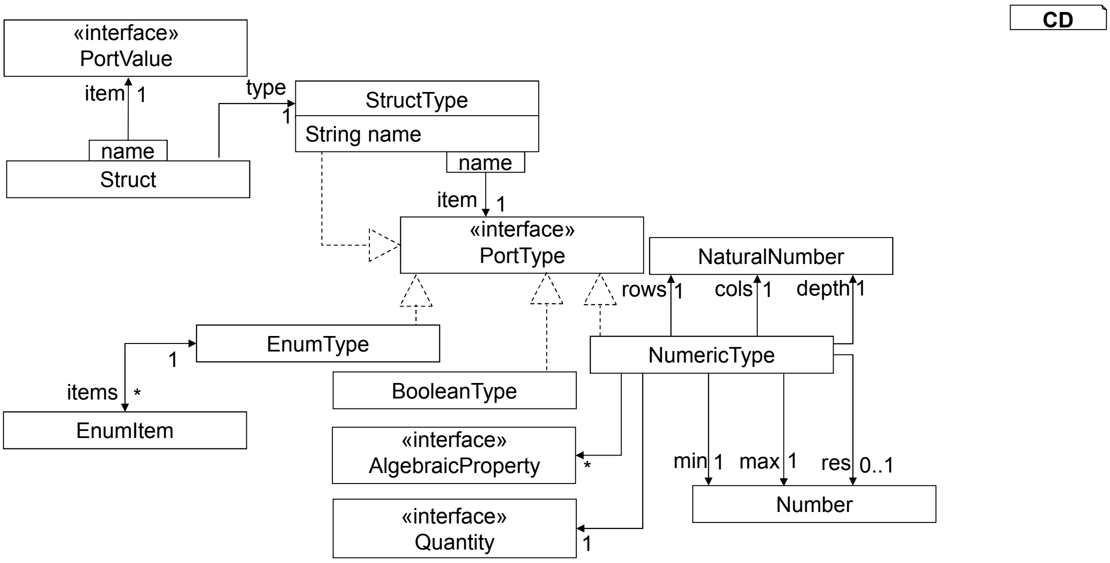
Figure 4.4.: Abstract syntax of `Type` and `Value` interfaces.
This thesis uses a derived version of CD4A presented in [Rot17]; cf. Appendix B for slightly modified textual syntax. The `{read-only}` tag presented in this class diagram has been added. The `{read-only}` flag allows it to have two associations with the same name (i.e., `type` in this figure) if both are marked as `{read-only}` and if the source and target class of both associations are in an inheritance relation (`implements` or `extends` relation in the class diagram). The advantage of this new keyword is that `{read-only}` associations can be refined. For example, every `Value` has (read-only access to) a unique `Type`; however, if the `Value` is a `PortValue` (the value has been refined), then we can now also express that the `Type` has also been refined to `PortType`.
One could argue that the top `type` association going from `Value` to `Type` is uninteresting for the concrete syntax of this language, however, this top association makes it much easier to express context conditions in *OCL*; and the main purpose of the abstract syntax of `EmbeddedMontiArcTooling` grammar is to have a convenient internal structure for tooling and to express context conditions.
This thesis uses the `{read-only}` flag only when an outgoing association (as in this example the `type` association) **is refined.** This means removing any `{read-only}` flag causes in an inconsistent textual class diagram when merging all these graphical representations. In all other cases, this thesis omits the `{read-only}` flag in associations to keep the graphical representations as simple as possible.

of `Mile` is `Length`. Figure 3.18 on page 63 lists available classes implementing the `Quantity` interface[3].

Figure 4.5 shows the abstract syntax of the `PortType` interface. *EmbeddedMontiArc* has four port type kinds:

- `BooleanType`: This type presents a Boolean value with `true` and `false`.
- `EnumType`: This type presents an enumeration. Each enumeration contains (multiple) enumeration items (`EnumItem`).
- `NumericType`: This is the most interesting type. This type represents numeric numbers or matrices (cf. Subsection 3.5.1).

[3] *EmbeddedMontiArc* uses JScience quantities. All available quantities are listed at: `http://jscience.org/api/javax/measure/quantity/Quantity.html`.



OCL
```
1 context Struct inv:
2   items.keySet.size == type.items.keySet.size &&
3   forall name in items.keySet:
4     items[name].type == type.items[name]
```

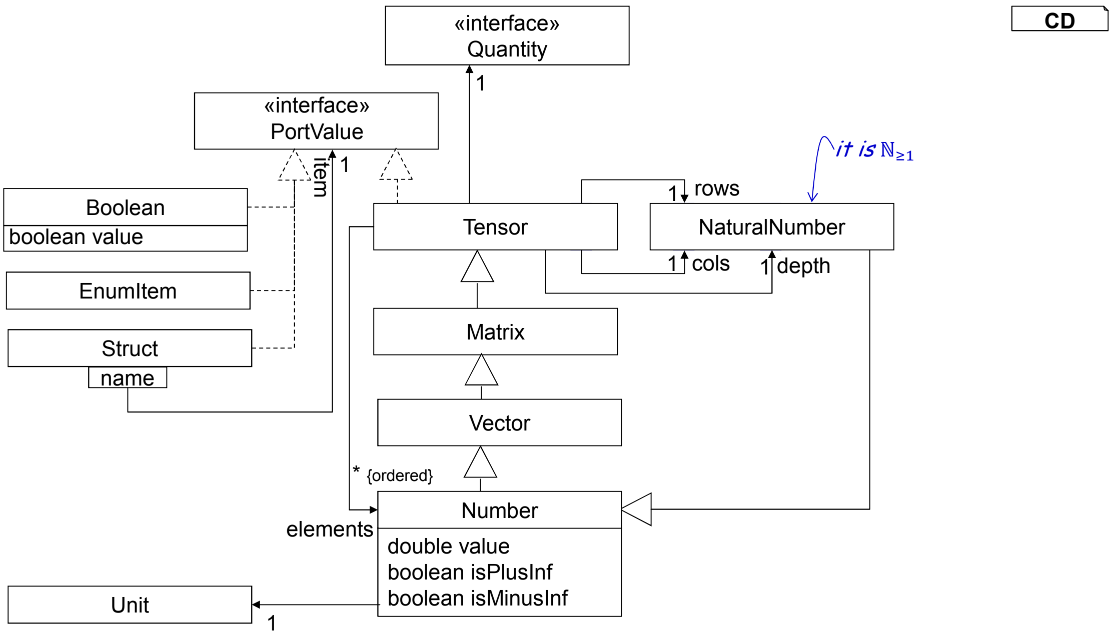
Figure 4.5.: Abstract syntax of `PortType` interface (extended abstract syntax of Figure 3.18). The cardinality of the `type` associations of the abstract syntax in this figure is `1`, because duck typing in *EmbeddedMontiArc* (cf. Subsection 3.6.4) works only on component types and not on port types. The duck typing inference algorithm of *EmbeddedMontiArc* automatically adds component interface implementations to existing component types.

- `StructType`: This type encapsulates data in a structure. Each `item` in this structure has a unique name. Since `StructType` implements the `PortType` interface and the items of structures (cf. qualified association) are elements of the `PortType` interface again, structures can be nested.

The *OCL* constraint in lines 1 to 4, say that `Struct` items have the same names as their `StructType` items and that the type of a `Value` of a `Struct` item with a given name is the same as the `StructType` item with the same name.

Figure 4.6 presents the abstract syntax of the `PortValue` interface. Analog to the port type kinds, *EmbeddedMontiArc* has four port value kinds. The classes `Tensor`, `Matrix`, `Vector`, `Number`, and `NaturalNumber` are numeric values; the type of these classes is the `NumericType` class.

The first *OCL* constraint says that a natural number has a value greater or equals to 1 and it is dimensionless and it cannot be plus or minus infinity (if one of these two boolean flags is `true`, the double field `value` is ignored). The second constraint says that a matrix is a tensor with `depth` equals to one. The next constraint classifies that a vector contains only of one row, this means every vector in *EmbeddedMontiArc* is a row vector. The last constraint formulates that a number is a vector with one column, meaning it is a 1×1 matrix.




```
context NaturalNumber inv:
  value >= 1 && unit.quantity instanceof Dimensionless &&
  !isPlusInf && !isMinusInf
```
OCL

```
context Matrix inv:
  depth == 1
```
OCL

```
context Vector inv:
  rows == 1
```
OCL

```
context Number inv:
  cols == 1
```
OCL

Figure 4.6.: Abstract syntax of `PortValue` interface (extended abstract syntax of Figure 3.18).

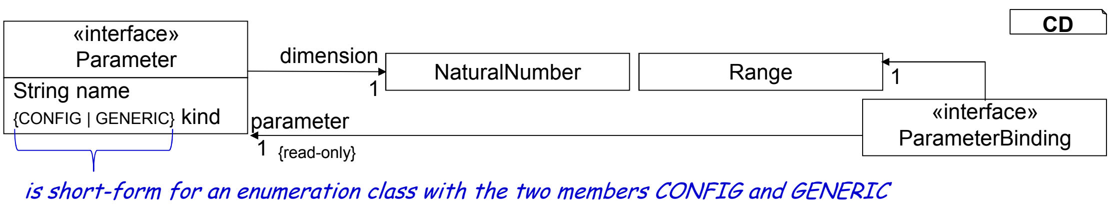


Figure 4.7.: Relationship between `Parameter` and `ParameterBinding` interfaces.

4.2.2. Parameter Definitions and Parameter Bindings

Figure 4.7 illustrates the abstract syntax of parameter definitions and parameter bindings. Every parameter has a `kind` attribute which is an enumeration with the two values `CONFIG` and `GENERIC` to model configuration and generic parameters. Additionally, parameters have a

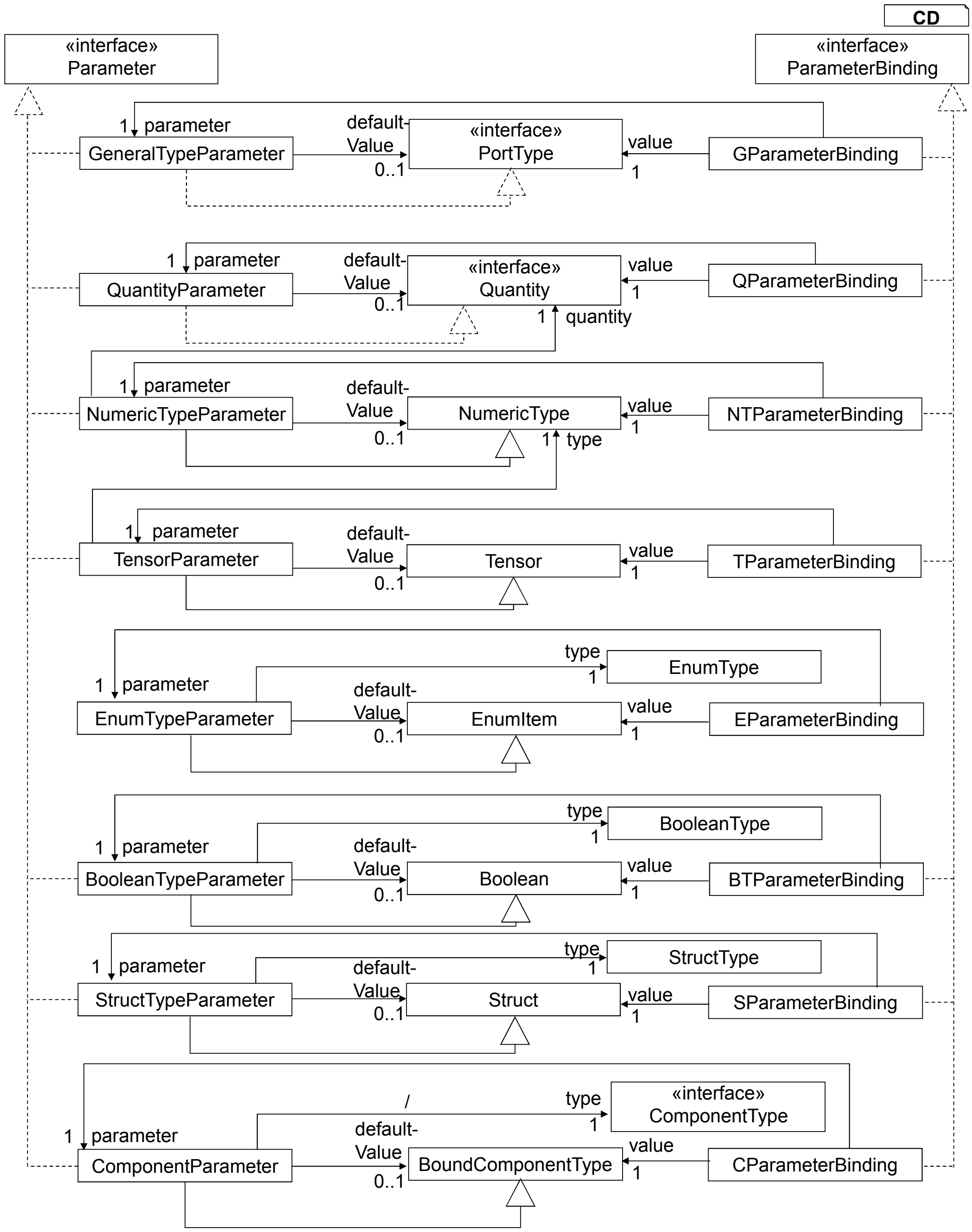


Figure 4.8.: Abstract syntax of concrete parameters and parameter bindings implementing the general `Parameter` and `ParameterBinding` interfaces.

```
context ComponentParameter inv:
  type instanceof ComponentInterface
```

OCL

Figure 4.9.: *OCL* constraint for `ComponentParameter`.

dimension and parameter bindings have the corresponding counterpart range to address the indices of a parameter definition (cf. example in Figure 3.54).

In *EmbeddedMontiArc* every parameter has an optional default value. However, every parameter kind accepts a different default value; therefore, the `defaultValue` association is not modeled in Figure 4.7.

Most parameters have a `Type` in *EmbeddedMontiArc*. Nonetheless, the `GeneralTypeParameter`, `QuantityParameter`, and the `NumericTypeParameter` have no `type` association; thus, in Figure 4.7 the `Parameter` interface contains no outgoing `type` association.

EmbeddedMontiArc has eight different parameter kinds as shown in Figure 4.8:

(i) **General type parameters.** These parameters accept as value every class implementing the general `PortType` interface, i.e., all four port type kinds mentioned in Subsection 4.2.1. Examples are `component G<T>`, `component G<T1 = Z, T2 = T1>`. Since the `GeneralTypeParameter` implements the `PortType` interface, the (default) value of a general parameter can be another general parameter.

(ii) **Quantity parameters.** These parameters have as value a class implementing the `Quantity` interface. Examples are `component X<Qt1 as Quantity = Length, Qt2 as Quantity = Velocity>`. The keyword `as` introduces the quantity parameter in *EmbeddedMontiArc*. The `QuantityParameter` implements the `Quantity` interface to use quantity parameters as (default) values for other quantity parameters; e.g., `component Y<Qt3 as Quantity, Qt4 as Quantity = Qt3>`. A quantity parameter binding (`QParamaterBinding`) binds a concrete value to a quantity parameter, e.g., when instantiating a component type. The expression `instance X<Acceleration, Dimensionless>` binds the parameter `Qt1` to the value `Acceleration`, and `Qt2` to the value `Dimensionless`; this example creates two objects of the class `QParameterBinding`.

(iii) **Numeric type parameters.** These parameters define the numeric type, supporting arithmetic operations, of in- and output ports. The keyword `is` introduces numeric type parameters. Examples of numeric type parameters are: `component B<T2 is Mass = (0 kg : 1t)>`, `component C<QtT3 as Quantity, T3 is QtT3>`, and `component D<T4 is Acceleration, T5 is Acceleration = T4>`. These parameters have as value an object of the `NumericType` class (cf. Figure 3.18). The `quantity` association of a numeric type parameter specifies the quantity property of the `NumericType` class. The `NumericTypeParameter` class extends the `NumericType` class so that numeric type parameters can be values of other numeric type parameters as shown for component `D`. The numeric type parameter binding (`NTParameterBinding`) binds one value to one numeric type parameter. The expression `instance A<T1 = Acceleration>` binds `T1` to `(-oo m/s^2 : oo m/s^2)`, because using quantity names in types in *EmbeddedMontiArc* is syntactic sugar for the type, having this quantity as generic parameter, with the largest possible range. This syntactic sugar is especially useful, when instantiating subcomponents with the same type having another parameter as quantity; e.g., component D<Qt5 as Quantity> { instance A<Qt5> a1; instance D<Qt5> d1; }.

(iv) **Tensor parameters.** Matrices are $n \times m \times 1$ tensors, and numbers are 1×1 matrices. The type attribute of a tensor parameter is the NumericType class. The quantity of a NumericType for the tensor parameter's type attribute is the same as the quantity of the unit in a Tensor for the value attributes of TensorParameter or TParameterBinding; thus, component M<Z^2 vector1 = [2 cm, 5 cm]> is wrong, as the type attribute for the parameter object vector1 is NumericType with quantity Dimensionless and the value attribute for vector1 is Tensor with quantity Length. Also the tensor dimensions (rows, cols, and depth attributes, cf. Figure 4.5 and Figure 4.6) of the type attribute and the value attribute must fit; therefore, component N<Z^3 vector2 = [2, 3]> is invalid, because the dimension of vector2.type is $3 \times 1 \times 1$ and the dimension of vector2.value is $2 \times 1 \times 1$. TensorParameter extends Tensor, so that one parameter can be another parameter's value; e.g., component P<N+ dim1, N+ dim2 = dim1>. Numbers as matrix parameters are often used to define the dimension of port or component instantiation arrays; e.g., component Or<N+ n> { ports in B values[n], out B result; }. For easier reading of this thesis, TensorParameters has the following extension hierarchy graph (analog to Tensor in Figure 4.6): MatrixParameter, VectorParameter, NumberParameter, and PositiveParameter having as values only NaturalNumbers; vector1 is a VectorParameter, and dim1 is a PositiveParameter.

(v) **Enum type parameters.** These parameters have an enumeration item as (default) value. The enumeration item bound to an enumeration type parameter must belong to the enumeration type of the type attribute of an enumeration type parameter. Thus, enum E1 { A | B}, enum E2 { C | D}, and component W<E1 en = D> is invalid, because D does not belong to en's type attribute which is E1.

(vi) **Boolean type parameters.** These parameters store as value either true or false.

(vii) **Structure type parameters.** These parameters have as value a structure, whereby the structure must be a valid instance of the structure type defined by the parameter's type attribute. This means the structure contains exactly the same names as the structure type does, and all values of the structure are compatible to the types of the defined structure type. An example is struct GPS { (-90° : 90°) latitude; (-180° : 180°) longitude; } and component S (GPS position = { latitude = 45°; longitude = -20°; }).

(viii) **Component parameters.** In contrast to the other seven parameter kinds, this parameter kind does not belong to the port type system. Reference architectures (cf. Subsection 3.6.4) use component parameters to enable different behavior of atomic components. The type of a component parameter is a component interface (cf. restriction in Figure 4.9)[4]. The value of component parameter is a bound component type whereby another component parameter is

[4]The type association in the class diagram is ComponentType and not ComponentInterface, as otherwise the class diagram merging algorithm does not work due to a conflict with the later introduced association BoundComponentType -> (type) ComponentType and ComponentParameter extends BoundComponentType. Therefore, the type of every ComponentParameter is restricted via *OCL*.

also a bound component type. An example is `component interface Interface1 {}`, `component Atomic(N+ n) implements Interface1 {}`, and `component Ref1 ( Interface1 I1 = Atomic(3) ) { instance I1 i1; }`.

A nice component definition with the three kinds (ii) - (iv) of parameters is `component General<Qt as Quantity, T is Qt> (Qt val)`. The first parameter is a quantity parameter, the second one is numeric type parameter with the quantity association bound to the value of the first parameter, and the third one is a matrix parameter with quantity `Qt`, and type of `NumericType` with minimum to minus infinity and maximum to plus infinity . A valid instantiation for this component is `instance General<Length, (0m : 10m)> ( 1 km)`. The value of `val` must not be of type `T`, thus `10 km` which is larger than `10 m` is a valid parameter. To force that the parameter `val` is inside the range created by the parameter `T`, the component must be defined as follow `component General2<Qt as Quantity, T is Qt> (T val)`, and now `instance General2<Length, (0m : 10m)> ( 1 km)` results in a compile error.

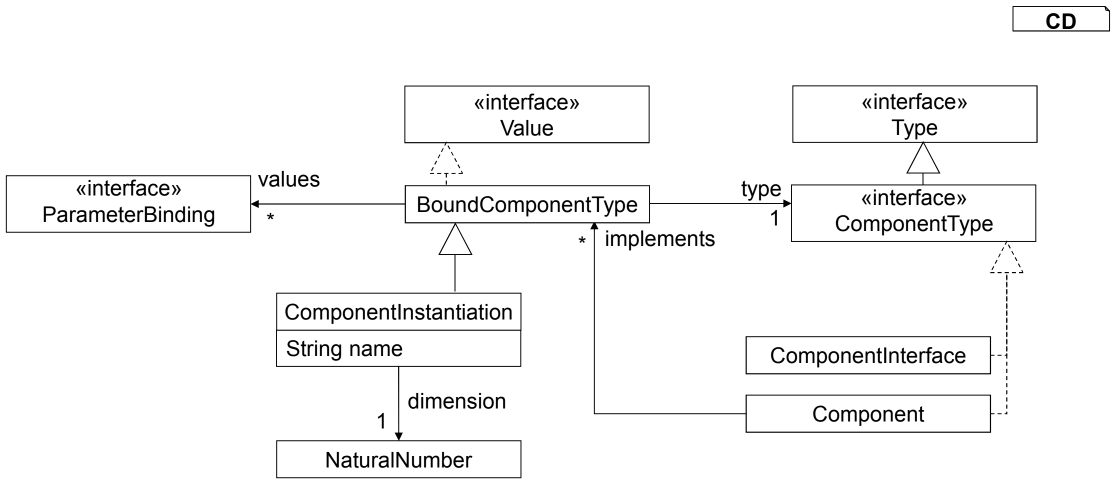
The general PID controller in Figure 3.23 is defined as follow `component PID<Qt1 as Quantity, Qt2 as Quantity, Qt1 lower, (lower : oo) upper> (...) { ports in ... time, Qt2 error, out (lower : upper) output; }`. Instead of passing `lower` and `upper` as two parameters and then building the type of the output port with these two values, a numeric type parameter can be used instead. This would look as follow: `component PID2<Qt1 as Quantity, Qt2 as Quantity, T is Qt1> (...) { ports in ... time, Qt2 error, out T output; }`. The difference in the component instantiation is `instance PID<Velocity, Acceleration, 0 m/s, 7m/s>(...) pid1` versus `instance PID2<Velocity, Acceleration, (0 m/s : 7 m/s) (...) pid2`. The first version (`PID`) is better suited when the *MontiMath* implementation also needs the `lower` or `upper` generic parameters; otherwise the second version (`PID2`) is to prefer.

Quantity and type parameters (i) - (iii), (v) - (vii) are mostly generic parameters, and the component parameter is mostly a configuration parameter. The matrix parameter, esp. numbers, are in general both: generic parameters when they address the dimension of ports as well as configuration ones when they address only the dimension of subcomponent instantiations or are factors (cf. `P`, `I`, and `D` of the generic `PID` controller) used in the implementation part.

4.2.3. Component Instantiation

Figure 4.10 shows the abstract syntax around the `ComponentInstantiation` class. A component instantiation has a `dimension`, and `name`; additionally it inherits from `BoundComponentType` all bound parameter `values`, as well as its component `type`. If the dimension is missing in the concrete syntax, then this `dimension` is set to 1. An example of *EmbeddedMontiArc* syntax without a dimension is: `instance A a1`. Examples of *EmbeddedMontiArc* syntax with dimension are: `instance A a2[3]`, and `instance A a3[n]` whereby `N+ n` is a configuration parameter. The dimension is always a natural number; either a positive whole number ($\mathbb{N}^{+}$) or a parameter which type is a subset of a positive whole number.

A component can implement multiple bounded component types (cf. `implements` association). A component implementing a component interface may bound some parameters of this




```
context Component inv:
  forall t in implements.type:
    t instanceof ComponentInterface
```

OCL

Figure 4.10.: Abstract syntax ComponentInstantiation class.

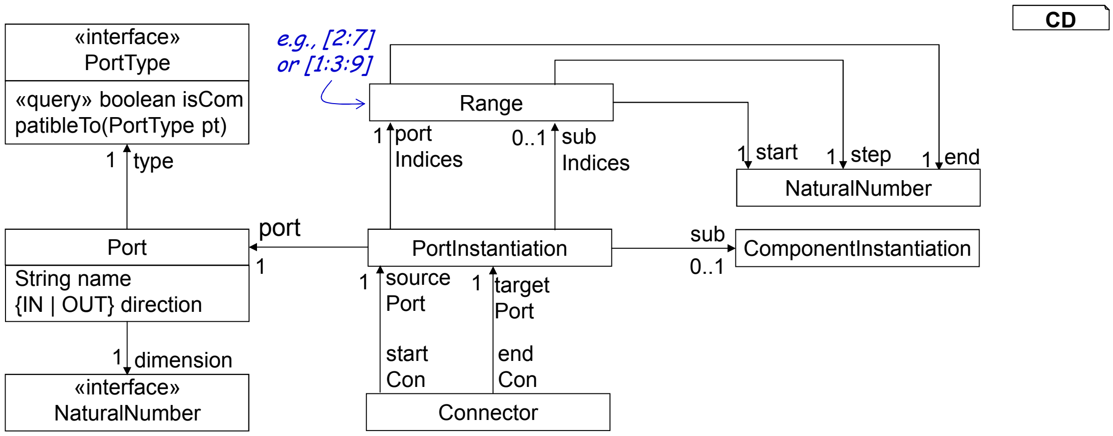
interface; therefore, the implements association goes from Component to BoundComponentType and *not* to ComponentInterface. An example is the following: component interface I5<N+ n> and component X implements I5<5>, whereby I5<5> is a bounded component type with component interface I5 as type and n = 5 as parameter binding.

The *OCL* constraint says that a component can only implement bounded component types whose types are component interfaces. Hence, component Y<T is Acceleration> and component Z implements Y<(-2m/s^2 : 2m/s^2)> is invalid (cf. discussion in Subsection 3.6.3 why *EmbeddedMontiArc* does not support extension of components).

4.2.4. Ports and Connectors

Figure 4.11 displays the abstract syntax of Port and Connector classes. A port has a name, a direction, a type, and a dimension. The direction of a port is either IN or OUT. A port instantiation is a port of a subcomponent instantiation or a port instantiation of the parent component (sub attribute is absent). Since ports and subcomponents have a dimension, a port instantiation contains index ranges of the subcomponent (subIndices) and of the port (portIndices). Index ranges are needed as port instantiations of library components cannot be "flattened", yet. A connector models dataflow from a source port instantiation to a target port instantiation.

An example for a connector of a library component is component LibA<N+ n> { instances LibB lib1[n], lib2[n]; connect lib1[:].result -> lib2[:].value; }. The connector connects the source port instantiation lib1[:].result with the target port instantiation lib2[:].value. The source port instantiation has the following values: sub = lib1, subIndices = 1:1:n (start = 1, step = 1, and end = n), port = LibB.result, and portIndices = 1:1:1. Since the value of subIndices.end is a



OCL
```
context Connector inv:
  sourcePort.port.type.isCompatibleTo(targetPort.port.type)
```

OCL
```
context PortInstantiation inv:
  sub.isPresent == subIndices.isPresent &&
  portIndices.end <= port.dimension &&
  subIndices.end <= sub.dimension
```

OCL
```
context Range inv:
  start <= end
```

Figure 4.11.: Abstract syntax of `Port` and `Connector` classes.

`PositiveParameter` which is not bound yet, the port instantiation, and thus the connector, cannot be flattened to `lib1[1].result -> lib2[1].result`, and so on.

The second *OCL* constraint in Figure 4.11 says that the `portIndices` and `subIndices` must be in range, thus, not larger than the dimension of the port definition and the subcomponent instantiation. `component A{ ports in In1[3], Out1[4]; connect In1[1:4] -> Out1[1:4]}` is invalid, because the port `A.In1[4]` does not exist.

The third *OCL* constraint says that the `start` value of a range is not larger than its `end` value. This constraint prevents empty (invalid) ranges. Line 3 in Figure 4.12 shows exactly how to define the values of a range.

The first *OCL* constraint in Figure 4.11 says that connectors connect only source ports with target ports when their types are compatible. Figure 4.12 defines exactly when two port types are compatible.

The first *OCL* constraint in lines 1 to 15 in Figure 4.12 says that `NumericType` t1 is compatible to `NumericType` t2 (source port type t1 can be connected to target port type t2), when the quantities and the tensor dimensions are equal (cf. ll. 9-12), as well as the algebraic properties are compatible (cf. l. 13) plus the range of t2 includes all values of the range of t1 (cf. l. 14-16). This thesis skips the concrete definitions when algebraic properties are compatible; their definitions are available in the matrix taxonomy paper [Bor06].

OCL

```
1  context NumericType t1, NumericType t2 inv:
2    let
3       maxSteps1 = {1 .. 2*Math.abs(t1.max / t1.res) + 1};
4       range1 = { v | v = t1.min + k * t1.res, k in maxSteps1, v <= t1.max };
5       maxSteps2 = {1 .. 2*Math.abs(t2.max / t2.res) + 1};
6       range2 = { v | v = t2.min + k * t2.res, k in maxSteps2, v <= t2.max };
7    in
8      t1.isCompatibleTo(t2) <=>
9      (t1.quantity == t2.quantity &&
10     t1.cols == t2.cols &&
11     t1.rows == t2.rows &&
12     t1.depth == t2.depth &&
13     t2.algebraicProperties.areCompatibleTo(t1.algebraicProperties)) &&
14     t1.res.isPresent == t2.res.isPresent &&
15     ( t1.res.isPresent implies range2.containsAll(range1) ) &&
16     ( !t1.res.isPresent implies t1.min >= t2.min && t1.max <= t2.max )
```

OCL

```
17 context StructType t1, StructType t2 inv:
18   t1.isCompatibleTo(t2) <=>
19   t1.items.size == t2.items.size &&
20   (forall item1 in t1.items:
21     exists item2 in t2.items:
22       (item1.name == item2.name &&
23        item1.type.isCompatibleTo(item2.type)))
```

OCL

```
24 context BooleanType t1, BooleanType t2 inv:
25   t1.isCompatibleTo(t2) <=> true // booleans are always compatible
```

OCL

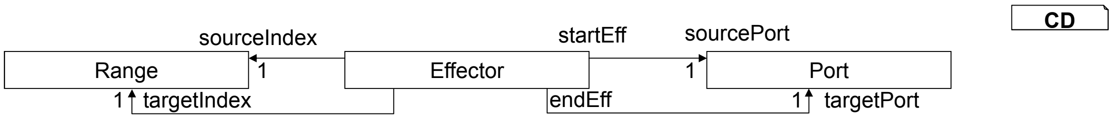
```
26 context EnumType t1, EnumType t2 inv:
27   t1.isCompatibleTo(t2) <=>
28   t1.items == t2.items
```

Figure 4.12.: Constraints about port type compatibility.

The source port type `(0:2:6)` is not type compatible to `(-1:2:7)`, since $4 \in (0:2:6) = \{0, 2, 4, 6\}$ and $4 \notin (-1:2:7) = \{-1, 1, 3, 5, 7\}$. The source port type `diag (-1:1)^{10,10}` is not type compatible to `diag positive-definite (-1:1)^{10, 10}`, because the source port type may have negative elements on the main diagonal and the target one must not have negative elements on its main diagonal.

The second *OCL* constraint in lines 17 to 23 states that two struct types are compatible if and only if both structs contain the same struct type element names and all of their struct type elements are compatible. This means `struct S1 {N+ x; Z y;}` is compatible to `struct S2 {Z x; Z y;}`. However, `S1` is not compatible to `struct S3 {N x; N y; N z;}`, because first `S3` contains an element with name `z`, and second the element type with `y` of `S1` is not compatible to `S3` as $\mathbb{Z} \not\subseteq \mathbb{N}$.

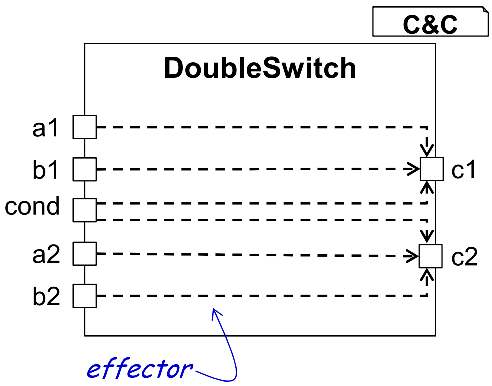
The third *OCL* constraint says that two boolean types are always compatible, and the fourth constraint forces that two enumerations are only compatible when they are equal.



OCL
```
context Effector inv:
  sourceIndex.end <= sourcePort.dimension &&
  targetIndex.end <= targetPort.dimension
```

OCL
```
context Effector inv:
  let
    sct = sourcePort.componentType;
    s = typeif sct instanceof Component then sct.subs else {};
  in
    sourcePort.direction == IN  &&
    targetPort.direction == OUT  &&
    sct == targetPort.componentType &&
    s == {}
```

Figure 4.13.: Abstract syntax of `Effector` class.



EMAM
```
component DoubleSwitch<T> {
  ports in T a1,
           T a2,
           B cond,
           T b1,
           T b2,
       out T c1,
           T c2;
  implementation Math {
    if cond
      c1 = a1;
      c2 = a2;
    else
      c1 = b1;
      c2 = b2;
    end
} }
```

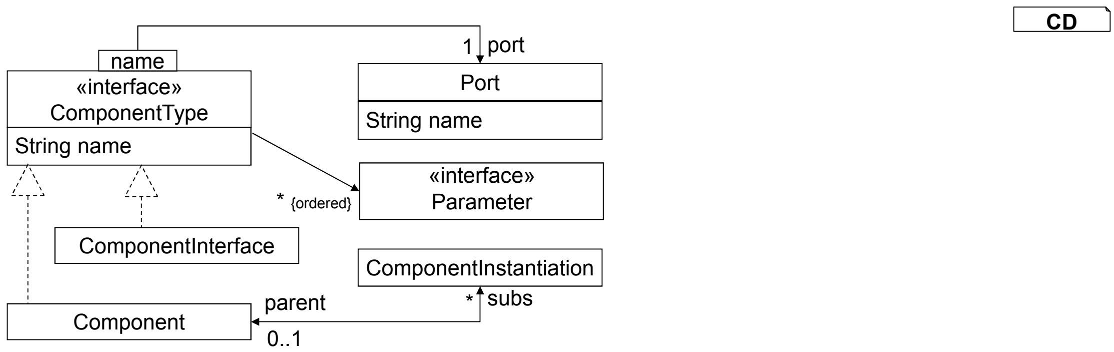
Figure 4.14.: `DoubleSwitch` component example to demonstrate effectors.

4.2.5. Effector

Figure 4.13 shows the abstract syntax of the `Effector` class. In contrast to a connector, delegating values one-to-one from one port to another port, an effector shows the effect of input ports to output ports of atomic components. Non atomic components do not have effectors.

The first *OCL* constraint in Figure 4.13 forces that the source and target index is in the range of the port dimensions. The second constraint says that an effector goes from an input to an output of an atomic component.

The embedded behavior language calculates the effectors; e.g., *EmbeddedMontiArcMath* calculates the effectors based on the control-flow graph of the mathematical expressions. In the default implementation of *EmbeddedMontiArc* (with no behavior language) every input port




```
1  context Component inv:
2    forall cmpType in implements.type:
3      cmpType.ports.name == this.ports.name
```

```
4  context Component inv:
5    let
6      typeDimParams = {p in ports.type.addAll(ports.dimension) |
7                       p instanceof Parameter};
8    in
9      this.subs != {} implies
10     this.parameters ==
11     subs.values.parameter
12       .addAll(typeDimParams)
13       .addAll( { param | param in this.parameters,
14                          exists param2 in this.parameters:
15                                 param == param2.type})
```

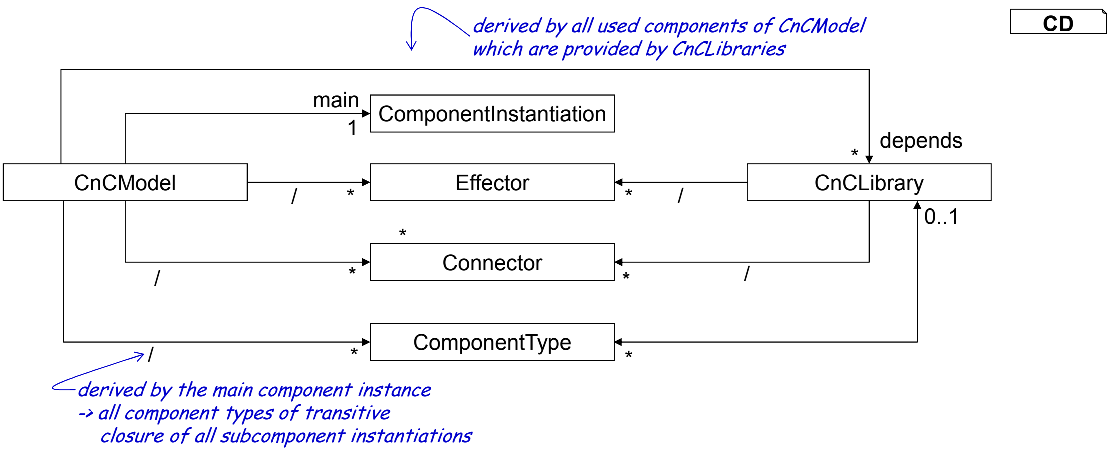
Figure 4.15.: Abstract syntax of `Component` and `ComponentInterface` class.

affects every output port. Assume we have a double switch as illustrated on the right side in Figure 4.14 with the input ports `a1`, `a2`, `b1`, `b2`, and `cond` as well as the output ports `c1` and `c2`. If `cond` is true, then `c1` returns the result of `a1`; otherwise `c1` returns `b1`. The output port `c2` works analog. The input port `cond` affects both output ports (cf. left side in Figure 4.14); but the input port `a1` and `b1` only affect `c1`, similar `a2` and `b2` affect only `c2`. This information what input port affects what output ports is useful to calculate structural effect chains crosscutting component hierarchies. The effectors for atomic components are later used to calculate effect chains (cf. Section 7.4) and to highlight abstract effectors in C&C views more accurately.

4.2.6. Component and Component Interface

Figure 4.15 displays the abstract syntax of the `Component` and `ComponentInterface` classes. Both, component interfaces and the components, have ports and parameters. Additionally, the component may contain subcomponent instantiations. If a component implements a bounded component interface, then the port names of the component must be identical to the port names of the implemented component interface instantiation (cf. first *OCL* constraint in ll. 1-3).

The second listing constraints that all parameters defined by non-atomic components must be used at least once. Atomic components may have additional configuration parameters that are



OCL

```
context CnCModel inv:
  let
    inPorts = {p | p in componentTypes.ports, p.direction == IN};
  in
    forall p in inPorts:
      forall pi in PortInstantiation:
        (pi.port == p && pi.sub != main) implies
        (exists con in connectors:
          pi == con.targetPort)
```

Figure 4.16.: Abstract syntax of `CnCModel` class.

used by their implementations in later language extensions. Concrete this means: All parameters (cf. l. 10) defined by a non-atomic (cf. l. 9) component must be used at least once in (1) a subcomponent instantiation (cf. l. 11), (2) as type or (3) dimension parameter in ports (cf. ll. 6, 7, 12), or as (4) parameter type of another parameter (cf. ll. 13-15). A correct example is `component X</* (4) */ Qt as Quantity, /* (2) */ T is Qt, N1 /* (3) */ n> (N1 /* (3) */ m) { port in T in1[n]; instance Y y[m]; }`.

4.2.7. Component and Connector Model

Figure 4.16 shows the abstract syntax of the `CnCModel` (component and connector model) class. A C&C model has one `main` component instantiation (cf. Subsection 3.6.7). Based on this `main` component instantiation, which has one unique component type, and on the transitive closure of all its subcomponent instantiations, all used component types of a C&C model can be derived. When all component types are derived and each component defines ports, then also all effectors of these ports can be derived; similar to all connectors.

A C&C library (`CnCLibrary`) is a collection of component type definitions, whereby each component type belongs at most to one C&C library. Effectors and connectors of a library can be derived. Since a C&C model knows which component types it uses, the imported C&C libraries can also be derived.

The *OCL* constraint in Figure 4.16 forces that for all input ports not belonging to the main component instantiation there must exist a connector providing data to this input port. The input and output ports of the main component instantiation must not be connected, because these ports serve as interface with the (simulator) environment. If an output port instantiation is not connected it produces a warning that the terminator block (`#` symbol) should be used. However, not connected input ports (except of the ports belonging to the main component instantiation) leads to the fact that components belonging to these ports cannot execute their calculations.

4.3. Component and Connector Instance Structure

Most tools, e.g., C++ code generator, graphical SVG code generator, or final context condition checks whether a C&C model is valid, work on the instance structure of a C&C model. The instance structure of a C&C model is derived from the main component instantiation. The instance structure does not contain any generic and configuration parameters. The instance structure only contains component instances, port instances as well as connector and effector instances, plus configuration parameter bindings for behavior implementations. Therefore, the instance structure is much easier to handle by tools than a complex C&C model, because all types have been completely resolved.

The C&C instance structure is the abstract syntax of the third main language of *EmbeddedMontiArc* (cf. Section 4.1). The abstract syntax of the second language `EmbeddedMontiArcTooling` of *EmbeddedMontiArc* does not exactly match the C&C instance structure, because `EmbeddedMontiArcTooling` supports reuse of component types and array concepts whereas these concepts are not present in `CnCInstanceStructure`.

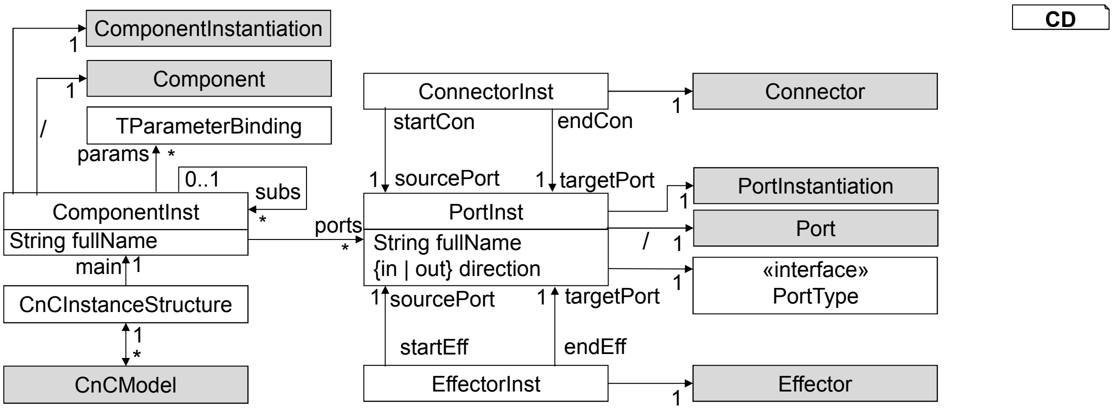
To have complete traceability between the instance structure and the text files, a link between the instance structure and the C&C model, it is derived from, is created. The C&C model contains links to the abstract syntax tree it is created from; and the parser adds source code positions of the matched text fragments to the abstract syntax tree.

Figure 4.17 shows the abstract syntax of the C&C instance structure. Every C&C model has exactly one C&C instance structure by binding all its parameters; because the textual *EmbeddedMontiArc* models describe a static architecture with no dynamic changes at runtime as it is the case in object oriented (modeling) languages.

However, different C&C models may instantiate the same C&C instance structure; e.g., instantiating both *EmbeddedMontiArc* models `And<n=2> {ports in B input[n], out B output; }` and `And {ports in B input[2], out B output; }` as main component results in the same C&C instance structure.

A C&C instance structure has one main component instance. A component instance can be decomposed of multiple subcomponent instances. Every component instance consists of several port instances to communicate with other component instances via connector instances. Atomic component instances, having no subcomponent instances, have effector instances to describe affects from input to output port instances. The type of a port instance is any type explained in Figure 4.5, but it is no parameter type.

The C&C instance structure does not contain a component interface, or any generic or configuration parameters. Also port instances and subcomponent instances do not have any dimension



OCL
```
context ComponentInst inv:
  (forall p in params.parameter:
    p.kind == CONFIG) &&
 (subs != {} implies params == {})
```

OCL
```
context PortInst inv:
  ! (type instanceof Parameter)
```

Figure 4.17.: Abstract syntax of C&C instance structure classes (gray are classes of C&C model; added for traceability).

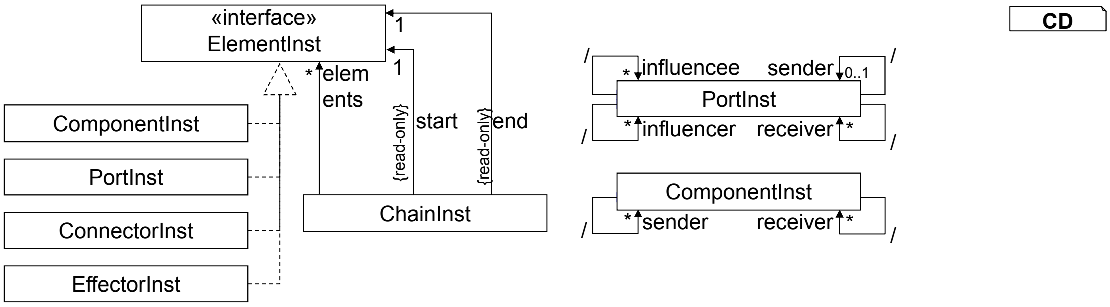
attribute. Atomic component instances may have bounded tensor configuration parameters so that later language extensions of *EmbeddedMontiArc* with a behavior implementation (e.g., *EmbeddedMontiArcMath*, or *EmbeddedMontiArcDL*) have access to the passed configuration parameters. Section 4.4 gives an example why tensors are needed for implementation parameters.

Figure 4.18 displays the abstract syntax of the chain instance class. A chain instance represents dataflow between two port instances. All element instances belonging to this dataflow via connector or effector (to describe dataflow inside atomic components) instances belong to this chain instance.

Additional, Figure 4.18 shows the derived associations for port instances and component instances. The two *OCL* constraints specify their semantics.

The `sender` association of a port instance `A` refers to another port instance sending data to `A`. The `receiver` association of a port instance `B` are all port instances that are connected to `B`. The `influencee` association extends the `sender` port instances with port instances linked via effectors; analog is the `influencer` association defined.

The `sender` association of a component instance `C` are all component instances that communicate via connectors with `C`. The `receiver` association is defined in an analog way. The self-associations of port and component instances enable an efficient navigation through the dataflow of C&C instance structures.



OCL

```
context ComponentInst inv:
  sender == { p.sender |
     p in ports}.componentInst &&
  receiver == { p.receiver |
     p in ports}.componentInst
```

OCL

```
context PortInst inv:
  sender.asSet == { PortInst p | exists con in ConnectorInst:
                    con.sourcePort == p && con.targetPort == this } &&
  receiver == {PortInst p | exists con in ConnectorInst:
               con.sourcePort == this && con.targetPort == p} &&
  influencer == sender.asSet.addAll(
    {p | exists eff in EffectorInst:
     eff.sourcePort == p && eff.targetPort == this} )    &&
  influencee == receiver.addAll( {p | exists eff in EffectorInst:
                           eff.sourcePort == this && eff.targetPort == p} )
```

Figure 4.18.: Abstract syntax of `ChainInst` class and derived associations of `PortInst` and `ComponentInst` classes.

4.4. Derivation of C&C Instance Structure from C&C Model

This section explains on two examples how to derive the C&C instance structure based on a given C&C model. To make the examples better readable, this section uses the more compact textual syntax representations instead of object diagrams to present concrete C&C instance structure examples. The concrete textual syntax of the C&C instance structure is very similar to the concrete syntax of *EmbeddedMontiArc* elucidated in Chapter 3. To better distinguish between *EmbeddedMontiArc*'s C&C model code and this C&C instance structure code; the C&C instance structure grammar (cf. Section C.1 on page 369) uses different keywords than the *EmbeddedMontiArc* grammar: `cmp-i` for component instance, `port-i` for port instance, `eff-i` for effector instance, and no keyword for connector instance. The C&C instance structure uses no keyword for the connector instance statement to make the code snippets for connections in this section shorter to fit in one line.

First, the transformation from the abstract syntax of `EmbeddedMontiArcTooling` to the abstract syntax of `CnCInstanceStructure` replaces the component instantiations with the complete contents of their component types; starting at the main component instantiation and it terminates when it reaches atomic components. Second, the transformation replaces port arrays by multiple single port definitions. Third, the transformation replaces connection patterns

EMA ...

```
1  component SensorProcessing<N1 n> {
2    ports in    C signal[n],
3                GPS posCar,
4          out  (0m : 25m) distance;
5    instance Filter filter[n];
6
7    connect signal[:] ->
8            filter[:].signal;
9    connect posCar ->
10           filter[:].posCar;
11 }
```

EMA ...

```
12 component Filter {
13   ports in    C signal,
14               GPS posCar,
15         out  (0m : 100m) distance;
16 }
```

Main.txt

```
17 Main-Component-Instantiation:
18   SensorProcessing<2>;
```

InstSt ...

```
19 cmp-i SensorProcessing {
20   port-i in C signal$1,
21          in C signal$2
22          in GPS posCar,
23         out (0m : 25m) distance;
24   cmp-i filter$1 {
25     port-i in C signal,
26            in GPS posCar,
27           out (0m : 100m) distance;
28     eff-i signal -> distance;
29     eff-i posCar -> distance;
30   }
31   cmp-i filter$2 {
32     port-i in C signal,
33            in GPS posCar,
34           out (0m : 100m) distance;
35     eff-i signal -> distance;
36     eff-i posCar -> distance;
37   }
38   signal$1 -> filter$1.signal;
39   signal$2 -> filter$2.signal;
40   posCar   -> filter$1.posCar;
41   posCar   -> filter$2.posCar;
42 }
```

Figure 4.19.: C&C instance structure derived from an *EmbeddedMontiArc* model with port and component instantiation arrays.

with single connection statements. The second and the third transformation also starts at the main component instantiation. The rest of this section explains these three transformations on examples.

Figure 4.19 shows how *EmbeddedMontiArc* models (cf. left side) containing component definitions with arrays of ports (cf. l. 2) and component instantiations (cf. l. 5) are transformed to a C&C instance structure (cf. right side). The main component instantiation (cf. ll. 17-18) passes the value `2` for the parameter `n` when instantiating the `SensorProcessing` (cf. l. 1) component definition. Therefore, the transformation creates for the `signal` port definition (cf. l. 2) with the dimension `n` two signal port instances (cf. ll. 20-21) for the `SensorProcessing` (cf. l. 19) component instance. For the same reason, the transformation maps the component instantiation array (cf. l. 5) to two component instances (cf. ll. 24-30 and ll. 31-37). Please note, that *EmbeddedMontiArc* models contain component instantiations to create subcomponents whereas the C&C instance structure contains directly the subcomponent instances. Thus, the C&C instance structure often contains the same information multiple times (cf. ll. 25-29 and ll. 32-36).

The transformation creates for the connection (cf. ll. 7-8) of the C&C model two connection instances (cf. ll. 38-39) in the C&C instance structure. It resolves the expression

EMA
```
1   component interface
2     PumpActuator <T is Length> {
3       ports in T pumpState,
4                T desiredPumpState,
5             out (0% : 100%)
6                         pumpActuator;
7   }
```

EMA ...
```
8   component PumpingSystem
9     (PumpActuator PA)     {

10    instance PA pumpActuator;
11  }
```

EMA ...
```
12  component WestEuropePump
13    <T is Length> implements
14    PumpActuator<T>        {

16    instance ShutOff protection;
17    instance PID<Qt2 = Length>
18              (1, 1, 1) pid;
19  }
```

Main.txt
```
20  Main-Component-Instantiation:
21    PumpingSystem (
22     WestEuropePump<(0m : 20m)> );
```

Port instance "in (0m:20m) pumpState" is linked to its port definition "in T pumpState". Thus, the binding information "(0m:20m)" -> "T" can be derived.

InstSt ...
```
23  cmp-i PumpingSystem {

24    cmp-i pumpActuator {
25      port-i in (0m:20m) pumpState,
26        in (0m:20m) desiredPumpState,
27        out (0%:100%) pumpActuator;

28      cmp-i protection { ... }

29      cmp-i pid {
30        port-i in (0ns : 1ns
31                   : oo ns) time,
32          in  (-oo m : oo m) error,
33          out (-oo m : oo m) output;
34        ...
35      }

36    }
37  }
```

Figure 4.20.: C&C instance structure derived from an *EmbeddedMontiArc* model with generic port type and component interface as configuration parameter.

`signal[:]` to `signal[1 : end]`, and this to `signal[1 : n]` which is for `n = 2` equals to `signal[1 : 2]`. The transformation maps the names `signal[1] to signal$1` and `signal[2] to signal$2`. The concrete syntax of the C&C instance structure uses a dollar sign for indices instead of squared brackets to avoid confusion with the port array dimension in *EmbeddedMontiArc*.

The effector instances in lines 28 and 35 are derived from the effectors in the C&C abstract syntax, which are added automatically (cf. Subsection 4.2.5) when building the C&C abstract syntax based on *EmbeddedMontiArc*'s abstract syntax. The effector instances together with the connector instances of the C&C instance structure enable to derive these chain instances; Section C.3 on page 371 presents the four longest chain instances.

These four chain instances support generators to optimize code, e.g., by parallelizing the calculations of the four chains at four different threads (CPU cores) [KRSvW18a]. The SVG generator uses these four chain instances to highlight dataflow (e.g., when clicking at an output port).

Figure 4.20 shows how the derivation algorithm creates the C&C instance structure of an *EmbeddedMontiArc* model with generic and configuration parameters. The main component instantiation (cf. ll. 20-22) binds the component interface parameter `PA` to `WestEuropePump` component type. Thus, the transformation maps the subcomponent instantiation in line 10 to the subcomponent instance shown in lines 24 to 35. Line 22 binds the generic port type parameter `T` to the type `(0m : 20m)`. Since the `WestEuropePump` component definition passes (cf. l. 14) this bounded parameter `T` to the implemented component interface `PumpActuator`, the type of the port instances `pumpState` (cf. l. 25) and `desiredPumpState` (cf. l. 26) is `(0m : 20m)`. The type of `error` (cf. l. 32) and `output` (cf. l. 33) port instances is `(-oo m : oo m)`, because the generic parameter `Qt2` is bound to `Length` (cf. l. 17) and no `upper` and `lower` parameter is set. Please remember that a quantity such as `Length` used as port type means the numeric type going from minus infinity to plus infinity of the corresponding quantity in *EmbeddedMontiArc*. This example showed why it is so convenient for further tools to work with the C&C instance structure, because they do not need to care about bounded parameters and component interfaces.

As shown in Figure 4.17, port instances (cf. `PortInst` class) are linked to its port definitions (cf. `Port` class). Helper classes use these links to recalculate the mapping of the data types, e.g., `(0m:20m)` to `T`.

Figure 4.21 illustrates how the two main component instantiations of the `Convolution` component type are transformed to two C&C instance structures. Similar to Figure 4.19 the port dimensions of `imageIn` and `imageOut` (cf. ll. 6-7) are unfolded (cf. ll. 30-36) in the first `Convolution` component instance. The first main component instantiation bounds the parameter `T` to a $1\,080 \times 720$ matrix, which elements are in the range between `0` and `255`. Therefore, the transformation replaces the port type `T` in lines 6 and 7 with the port type `(0 : 255)^{1080, 720}` in lines 31 to 36. The kernel array, which type is a $n \times n$ matrix, is transferred to a $n{\times}n{\times}dim$ tensor during the transformation process. This way the implementation languages need no knowledge about arrays of configuration parameters. Extending the kernel array to `kernel$1`, ..., `kernel$3` for the first component instance would not work as the implementation language resolves the configuration parameter according to its name, and then it expects one value with a specific type and not three values.

The powerful type inference algorithm[5] of *EmbeddedMontiArc* also infers the stricter type of the configuration parameter kernel from $\mathbb{Q}^{5\times5\times3}$ (`n` bounded to `5` and `dim` to `3`) to $(\frac{1}{256} : \frac{1}{256} : \frac{9}{64})^{5\times5\times3}$ based on the passed matrix.

The stricter type enables to generate C++ code leveraging much more hardware optimizations, as the entire $5 \times 5 \times 3$ tensor can be divided by 255 at the end and so *EmbeddedMontiArc* needs only to store the values $1, 2, \ldots, 36$. This way the tensor can be highly accelerated by using hardware accelerators, e.g., Google's TPUs (tensor processing units). TPUs are specific chips to execute 8-bit matrix multiplications for artificial intelligence applications. Due to the value limit of 8-bit (Integer values from 0 to 255), most TPUs offer throughput of 92 TeraOps/second [JYP$^+$17].

The importance to generate code for domain-specific hardware (e.g., using Intel's AVX-512 instructions, GPUs' single floating point calculations, or TPUs' 8-bit integer matrix multiplica-

[5]The algorithm encodes the matrix property rules [Bor06] in Prolog and a Java Prolog interpreter infers the data types; cf. [Gör17] for more information about the type inference algorithm.

EMA ...

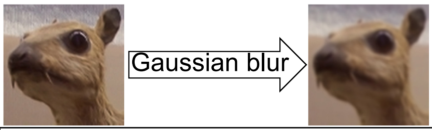
```
1   component Convolution
2      <T is dimensionless,
3        (1:2:3) dim, (3:oo) n>
4     (symmetric Q^{n, n}
5      kernel[dim]) {
6     ports in  T imageIn [dim],
7           out T imageOut[dim]; }
```

Main.txt

```
8   // HD-ready RGB-image, use 5x5
9   // Gaussian blur matrix for
10  // kernel[1], ..., kernel[3]
11  Main-Component-Instantiation:
12    Convolution
13     <(0 : 255)^{1080, 720}, 3>
14     ( kernel[:] = 1/256*
15           [1, 4,  6,  4,  1;
16            4, 16, 24, 16, 4;
17            6, 24, 36, 24, 6;
18            4, 16, 24, 16, 4;
19            1, 4,  6,  4,  1] );
```

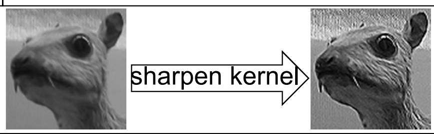
Main.txt

```
20  // Full-HD b/w image,
21  // use 3x3 sharpen kernel
22  Main-Component-Instantiation:
23   Convolution
24   <(0 : 2^24)^{1920, 1080}, 1>
25   ( [0, -1, 0; -1, 5, -1;
26      0, -1, 0] )        fullHD;
```



InstSt 1...

```
27  cmp-i Convolution (symmetric
28     (1/256 : 1/256 : 9/64)^{5,5,3}
29    kernel = [... /* tensor */ ])     {
30    port-i
31        in  (0:255)^{1080,720} imageIn$1,
32        in  (0:255)^{1080,720} imageIn$2,
33        in  (0:255)^{1080,720} imageIn$3,
34       out  (0:255)^{1080,720} imageOut$1,
35       out  (0:255)^{1080,720} imageOut$2,
36       out  (0:255)^{1080,720} imageOut$3;
37    eff-i imageIn$1 -> imageOut$1;
38    eff-i imageIn$2 -> imageOut$2;
39    eff-i imageIn$3 -> imageOut$3;}
```



InstSt 2...

```
40  cmp-i fullHD (symmetric
41     (-1 : 1 : 5)^{3,3} kernel =
42     [0, -1, 0; -1, 5, -1; 0, -1, 0]  ) {
43    port-i
44        in  (0:2^24)^{1920,1080} imageIn,
45       out  (0:2^24)^{1920,1080} imageOut;
46    eff-i imageIn -> imageOut;
47  }
```

Figure 4.21.: C&C instance structures derived from *EmbeddedMontiArc* models with generic matrix port type and atomic configuration parameters. Images for applying kernel convolution are copied from Wikipedia [Plo13].

tions) is the major improvement to handle the cost and energy consumption of new data-intensive algorithms [JYP+17].

Modern smartphones also contain specific TPU chips to handle virtual and augmented reality. The paper [ITC+18] benchmarks 10 000 Android mobile devices and more than 50 different mobile system-on-chips. *EmbeddedMontiArc*'s matrix type system with its type inference algorithms enables that the developer only focuses on the mathematical domain (what values need to be stored in the matrix) and to tag the C&C instance model with preferred hardware targets; the generator automatically produces high-performance hardware-specific C++ code.

The second main component instantiation in Figure 4.21 passes a 3×3 matrix to kernel and binds the variable `dim` to `1` instead of `3`. The parameter `dim` (cf. l. 3) accepts only the values `1` or `3`; `1` for black white images, and `3` for colored images whereby the three dimensions are the red, green, and blue channels.

The transformation of a C&C model to its C&C instance structure is unique. This means each C&C model is transformed to one unique C&C instance structure due to all parameter bindings passed through the main- or subcomponent instantiations. **The transformation is <u>not</u> injective**, because different C&C models, e.g., one with generic type parameters and one without, can be transformed to the same C&C instance structure.

4.5. Comparison of *EmbeddedMontiArc*'s Abstract Syntax Structures against the Ones of Other *MontiArc* Derivatives

This section compares both abstract syntax structures of Section 4.2 and Section 4.3, i.e., C&C model and C&C instance structure, with the abstract syntax structures defined by other *MontiArc* derivatives. The comparison starts with Ringert's formalized C&C model and C&C types definitions, over Haber's *MontiArc* abstract syntax, and finishes with Wortmann's *MontiArcAutomaton* abstract syntax.

Ringert's Abstract Syntax of *MontiArc* and *MontiArcAutomaton*

Ringert defines the abstract syntax of C&C models [Rin14, Definition 2.2 on p. 15] and C&C types [Rin14, Definition 6.8 on p. 164] via tuple structures. A unique translation of these tuple structures to a class diagrams is possible by mapping:

- Sets to classes, e.g., "$Cmps$ is a set of components $cmp \in Cmps$" is equal to the `Cmp` class, and
- Functions to 1-* associations, e.g., "each of which has a set of ports $ports(cmp) \subseteq Ports$" is equal to the association `[1] Cmp -> Port [*]`.

The C&C model definition in Ringert [Rin14, Definition 2.2] is very similar to the C&C instance structure of this thesis, because it only includes component (instances), port (instances), and connector (instances). The here presented C&C instance structure extends Ringert's definition with effector instances and configuration parameter bindings. Also the port type system of the here presented C&C instance structure is much more advanced.

The component and connector type definition [Rin14, Definition 6.8 on p. 164] of Ringert fits better to this thesis' definition of a C&C model. However, "[Rin14, Definition 6.8] abstracts *MontiArcAutomaton*'s component type name, the component parameters, and the type parameters to the single element $cType$. We omit the implementation details of these advanced concepts, which are not required for consecutive definitions and the techniques" [Rin14, p. 168]. In contrast, this thesis presented in this chapter the complete formalized abstract syntax using the class diagram and *OCL* semantics of a C&C modeling language with component types, component interfaces, configuration and generic parameters, as well as the bindings of these parameters. Ringert's restriction of connectors, "which connects two ports of the same type" [Rin14, p. 165], is very conservative. The here presented abstract syntax enables a more relaxed approach, inspired by *Simulink* and *SysML* models of industrial partners, to connect compatible types (cf. *OCL* expression in Figure 4.11 on page 117); e.g., connect the source type `(0:1:7)` with the target type `(-10 : 10)`. In the Java world, the relaxed restriction enables connecting (i) source type `int` with target type `double`; and

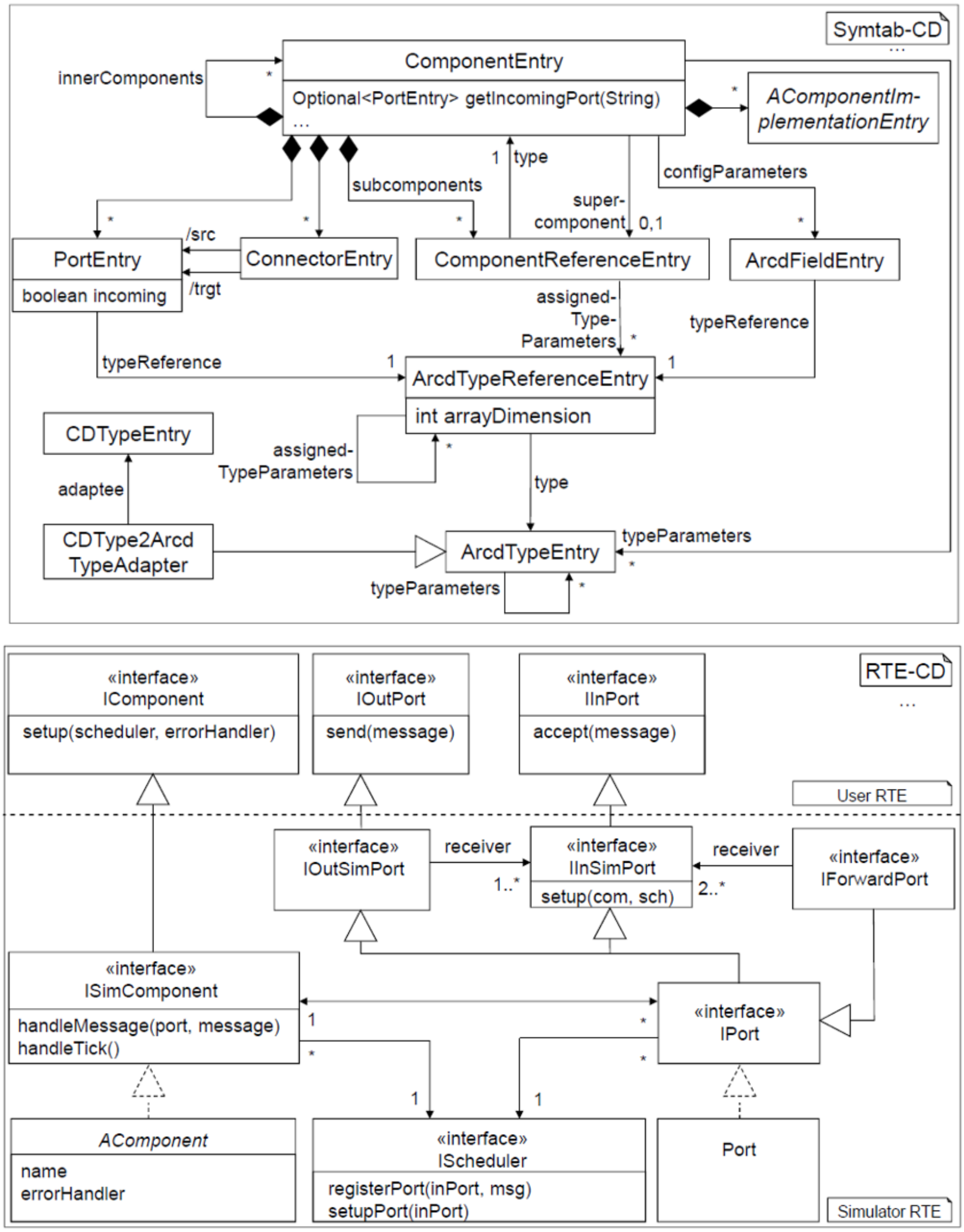


Figure 4.22.: Top part: Abstract syntax (symbol table) of *MontiArc* presented by Haber (copied from [Hab16, p. 135]).
Bottom part: Abstract syntax of simulator runtime environment (copied from [Hab16, p. 94]).

MontiArc

```
component And {
  port in  Boolean in1,
       in  Boolean in2,
       out Boolean out1;
}
```

MontiArc uses "component" keyword for component type definition and for component instantiation

MontiArc

```
component And3 {
  port in  Boolean in1,
       in  Boolean in2,
       in  Boolean in3,
       out Boolean out1;

  component And and1;
  component And and2;

  connect in1 -> and1.in1;
  connect in2 -> and1.in2;
  connect in3 -> and2.in1;
  connect and1.out1 -> and2.in2;
  connect and2.out1 -> out1;
}
```

Figure 4.23.: Example for missing `PortReferenceEntry` in *MontiArc*'s symbol table.

(ii) source type `ArrayList<String>` or `LinkedList<String>` with the target type `List<String>`, because `ArrayList<String>` and `LinkedList<String>` implement the `List<String>` interface.

We brake the name convention[6] of Ringert on purpose, so that our notation is compatible to the C&C model notation of Haber [Hab16], and Wortmann [Wor16] based on MontiCore's general approach that models are textual artifacts created by developers. In our case, developers cannot directly create C&C instance structure artifacts. Developers use the *EmbeddedMontiArc* language to define C&C models with component types to enable reuse. The purpose of the here presented C&C instance structure language was only to explain the transformation process from *EmbeddedMontiArc* models to the C&C instance structure.

Haber's Abstract Syntax of MontiArc

The top part of Figure 4.22 shows Haber's abstract syntax of *MontiArc*. Components can only have none or one supercomponent (cf. cardinality `0,1` above `ComponentReferenceEntry`). Thus one component type cannot implement multiple interfaces, even if all the component interfaces have the same or compatible ports. This way a component type cannot implement two component interfaces provided by two different simulators.

The `ConnectorEntry` belongs to the component type (`ComponentEntry`) and has direct connection to ports (`PortEntry`) and not, as in this thesis, to `PortReferenceEntry`

[6]Ringert's C&C model is our C&C instance structure, and our C&C model is Ringert's component type definition.

(which is missing in the symbol table of *MontiArc* and which would be the equivalent class to `PortInstantiation` of our abstract syntax)[7]. This results for Figure 4.23 to the following problem: The connector in line 13 has source port `And3.in1` (cf. l. 7) and as target port `And.in1` (cf. l. 2), and the connector in line 15 has source port `And3.in3` (cf. l. 9) and as target port again `And.in1` (cf. l. 2). This means that the object graph based on Haber's abstract syntax connects the port `And3.in1` and `And3.in3` with the one port `And.in1`; and this is wrong and leads even to an invalid model.

The bottom part of Figure 4.22 shows Haber's abstract syntax of the simulator runtime environment. Haber's simulator runtime instantiates the *MontiArc* models (cf. "Object Instantiation of a Simulation" [Hab16, p. 90]). The abstract syntax of the simulator runtime environment contains parts of our instance structure: The `ISimComponent` interface is similar to our `ComponentInst` class, and the `IPort` interface is similar to our `PortInst` class. The simulator runtime uses Java references of objects of the `IPort` interface for dataflow; thus it does not contain any equivalence to our `ConnectorInst` class.

Because *MontiArc* supports architectural changes at runtime and the generation/compilation process is highly modular, *MontiArc* generator does not optimize Java code according to control-flow graph analysis techniques, and therefore, *MontiArc* does not need the `ConnectorInst` class.

On the other side, *EmbeddedMontiArc*'s C++ generator uses the `ConnectorInst` and `EffectorInst` classes to analyze what computations can be executed parallel on different cores as well as what calculations maybe switched without modifying the result (cf. [KRSvW18a] for more details). Assume a component is decomposed of two matrix multiplication subcomponents: the first subcomponent multiplies the 100×20 matrix A with the 20×50 matrix B and the result is a 100×50 matrix C; the second subcomponent multiplies this matrix C with a 50×10 matrix D and the result is a 100×50 matrix E. Executing the decomposed component from left to right, i.e., $(A \cdot B) \cdot D$, needs about 300 000 operations[8], whereas reordering the calculations to $A \cdot (B \cdot D)$ based on the control-flow graph according to the connector and effector instances only 40 000 operations are needed; causing in a speed-up of at least 7.5. The actual speed-up is even higher as the second calculation does not create the very large temporary matrix C and thus, effects like loading and storing memory blocks are less present.

Wortmann's Abstract Syntax of *MontiArc* and *MontiArcAutomaton*

Figure 4.24 shows the abstract syntax of Wortmann's *MontiArc* and *MontiArcAutomaton* symbol table. Wortmann's abstract syntax contains no connector class and the `PortEntry` does not contain a self-reference to express the source or target port a specific port object is connected to. Thus, the abstract syntax does not hold any information about data flow between components. The absence of a reference class for `TypeEntry` results in missing links between type parameter

[7] The following statements in [Hab16, Tbl. 5.9] underlines this: "A ComponentEntry is created for each MontiArc component definition. A component entry consists of further entries that describe the component's interface and decomposition. The interface is given by a set of associated port entries." [Hab16, p. 135], "A component reference entry represents a reference to a component type. It is used to represent subcomponents as well as the reference to the type of a supercomponent." [Hab16, p. 136], and "Connector entries represent connectors in the model which connect a source port (src) with a target port (trgt)." [Hab16, p. 136].

[8] $A_{100\times 20} \cdot B_{20\times 50} \approx 2 \cdot 100 \cdot 20 \cdot 50 = 200\,000$ operations

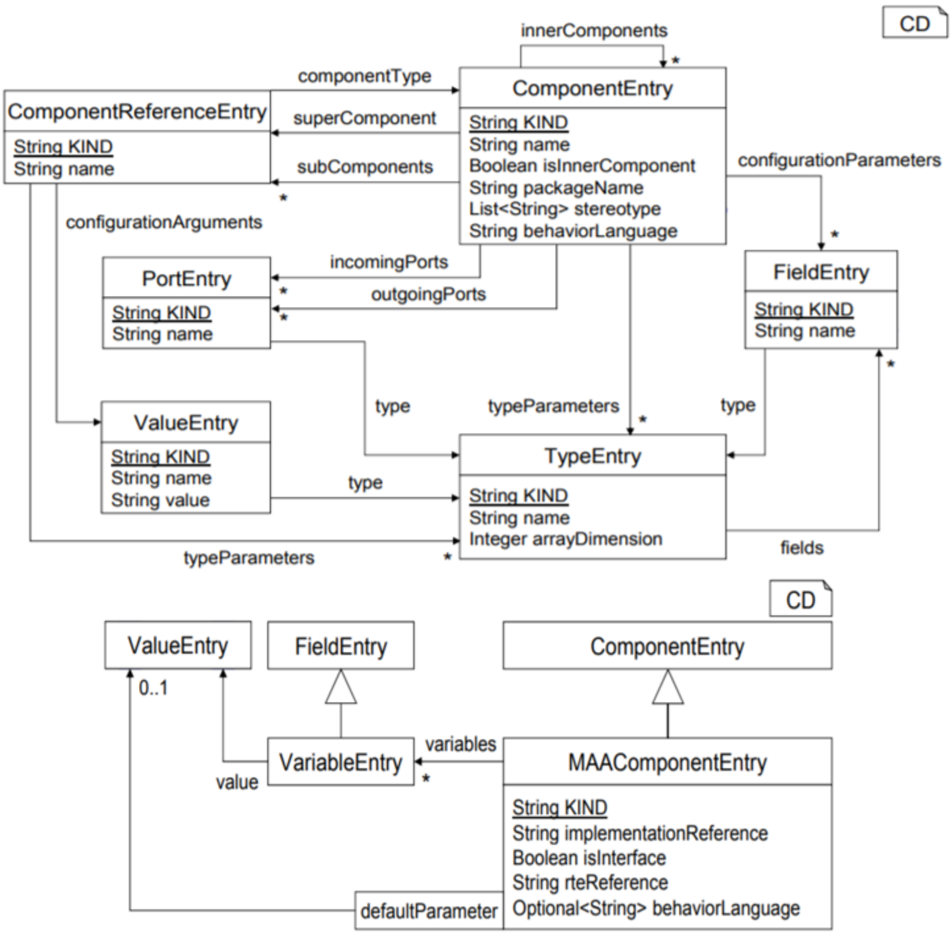
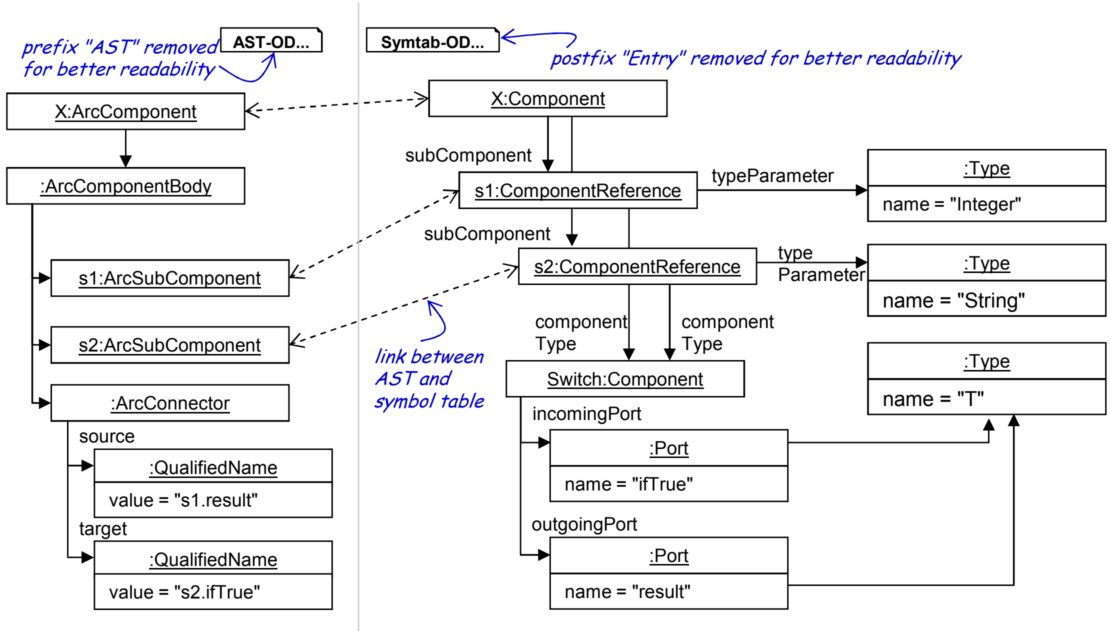


Figure 4.24.: Abstract syntax (symbol table) of *MontiArc* (top) and *MontiArcAutomaton* (bottom) presented by Wortmann (copied from [Wor16, p. 53 and p. 54]).

definitions and their binding with concrete values. The incomplete nature of the symbol table class diagram requires calculating the instance structures and their interactions for each tooling (e.g., context condition checks, or code generation).

Figure 4.25 shows the abstract syntax (AST + symbol table) of the component X. The dashed line between AST-OD and Symtab-OD shows the link between object diagrams of the abstract syntax tree and the symbol table. This link is automatically created when an AST node creates a symbol table entry. The structure of Wortmann's abstract syntax is modular and focuses only on the contents of a single file artifact. However, the additional C&C instance structure of *EmbeddedMontiArc* across component artifacts makes implementing context conditions more efficient. For example, the type incompatibility between source and target port of the connector, defined in lines 4 and 5, is hard to figure out using Wortmann's modular abstract syntax structure. This is the case, because neither AST nor the symbol table contains direct connections between



MontiArcAutomaton ...

```
1  component X {
2    component Switch<Integer> s1;
3    component Switch<String> s2;
4    connect s1.result ->
5            s2.ifTrue;
6  }
```

MontiArcAutomaton

```
7  component Switch<T> {
8    port in T ifTrue,
9         in Boolean cond,
10        in T ifFalse,
11        out T result;
12 }
```

Figure 4.25.: Object diagram for AST (abstract syntax tree) and Symtab (symbol table) created for left textual *MontiArcAutomaton* model. The AST-OD is created based on Wortmann's *MontiCore* grammar [Wor16, Listing A.1 on p. 250].

the two ports. Additionally, the abstract syntax also does not have a direct link between the bounded generics in the component reference (cf. ll. 2, 3) and the generic port type of the two `Switch`'s ports (cf. ll. 8, 11) due to the missing type reference class in the symbol table. In contrast, *EmbeddedMontiArc*'s C&C instance structure, shown in Figure 4.17, contains the `ConnectorInst` class having direct links to the source and target port objects of the `PortInst` class, which again has a direct link to the concrete bounded port type.

Both, *EmbeddedMontiArc* and *MontiArcAutomaton*, support component interfaces which are atomic and do not provide any behavior (cf. [Wor16, p. 51, and Listing 6.1 on p. 115]). *MontiArcAutomaton*'s *Application Configuration Language* (cf. [Wor16, Section 8.1 on pp. 176ff.]) also supports the definition of a main component and it supports to bind platform-independent component interfaces with platform-specific components [Wor16, pp. 115, and Listing 8.1 on p. 177].

However, the motivation of introducing component interfaces in *EmbeddedMontiArc* and *MontiArcAutomaton* is slightly different. *EmbeddedMontiArc* is a functional C&C modeling language (cf. Subsection 2.1.2); and thus, it models the problem-specific part (e.g., how a braking assistant works), and so it is by definition platform independent. Component interfaces in

EmbeddedMontiArc help to create mathematical reference architectures. In *EmbeddedMontiArc* component interfaces are logical variation points; e.g., filters used in image processing - all having other problem specific properties such as stability about different noise distributions. The platform specific part is added via tag models, e.g., ROS tags to define communication between components, or generate against different frameworks or simulators such as *Torcs*, or *OpenDavinci*. *EmbeddedMontiArc* toolchain also supports multiple targets such as native Windows and Linux platforms as well as client-side Browser platforms (cf. [KRSvW18a]).

In contrast, *MontiArcAutomaton* uses interfaces only to bind platform-independent components such as `Timer` class to different native API calls. For this reason, the binding is also only defined in the *Application Configuration Language* which also specifies the generator target. Thus, component interfaces cannot be used in *MontiArcAutomaton* to model variability in the logical platform-independent layer.

Summary

The two abstract syntax structures of *MontiArc* or *MontiArcAutomaton*- Ringert [Rin14], and Haber [Hab16] - do not introduce a main component instantiation mechanism. They provide a number of *MontiArc* artifacts (which could also be only a library), and based on the Java code, e.g., what artifacts are firstly loaded, different component and connector models are instantiated. This thesis here introduces a complete model-based approach where *EmbeddedMontiArc* models can be shipped as stand-alone (see ZIP file in Subsection 3.6.7) artifacts to be processed by different tools.

Table 4.26 summarizes the comparison results of the different abstract syntax structures (AST + symbol table, or formal definitions in mathematical tuples) of the different *MontiArc* derivatives. This table shows clearly that the two abstract syntax structures of *EmbeddedMontiArc* contain the most and best navigable information. These two structures facilitate to formalize context conditions in a few lines of *OCL* code (cf. Section 6.1), as well as they enable later to formalize the satisfaction relation between EmbeddedView (C&C view) language and *EmbeddedMontiArc* (C&C model) language precisely.

Table 4.26.: Overview of the elements of the abstract syntax of different *MontiArc* derivatives.

von Wenckstern [this thesis] (*EmbeddedMontiArc*)	Ringert [Rin14] (*MontiArcAutomaton*)	Haber [Hab16] (*MontiArc*)	Wortmann [Wor16] (*MontiArc-Automaton*)
CnCModel	- (no main component)	- (no main component)	Application Configuration Language
Component	structure cmp = (cType, CPorts, ...)	ComponentEntry	MAAComponent-Entry
ComponentInstantiation	CSubCmps	Component-ReferenceEntry	Component-ReferenceEntry
ComponentInterface	- (no extension or implements relation between cType)	ComponentEntry	isInterface flag in MAAComponent-Entry
Port	CPorts	PortEntry	PortEntry
PortInstantiation	tuple $(name, t) \in CSubCmps \cup \{(cType, cmp)\}$	- (is an error in the abstract syntax)	- (reference of Port-Entry is missing)
Connector	CCons	ConnectorEntry	-
Effector	-	-	-
Range	- (no arrays of port or component instantiations)	- (no arrays of port or component instantiations)	- (no arrays of port or component instantiations)
Type (type system with units, quantities, structs, enumerations, and matrices)	P (only a set, no relation between port types)	ArcdTypeEntry (Java type system), CDTypeEntry	TypeEntry (Java type system)
Parameter (parameters for port types)	-	ArcdFieldEntry, ArcdTypeEntry	FieldEntry, Type-Entry
ComponentParameter	-	-	-
ParameterBinding	-	ArcdType-ReferenceEntry	- (reference of Type-Entry is missing)
CnCInstanceStructure	structure m = (Cmps, Ports, ...) (main component must be computed)	-	-
ComponentInst	Cmps	ISimComponent	-
PortInst	Ports	IPort	-
ConnectorInst	Cons	-	-
EffectorInst	-	-	-
TParameterBinding	- (no behavior config. Parameter for structure m)	-	-

Java ...

```
1  public class ComponentSymbol extends CommonScopeSpanningSymbol {
2    public Collection<PortSymbol> getPorts () {
3      return this.getSpannedScope().<PortSymbol>
4                  resolveLocally(PortSymbol.KIND);
5    }
6    public Collection<ComponentInstantiationSymbol> getSubComponents() {
7      return this.getSpannedScope().<ComponentInstantiationSymbol>
8                  resolveLocally(ComponentInstantiationSymbol.KIND);
9    }
10 }
```

Figure 4.27.: Java code excerpt of `ComponentSymbol` class. This Java class represents the `Component` class of the abstract syntax shown in Figure 4.15.

4.6. Realization of the Abstract Syntax with Symbol Management Infrastructure

This section shortly explains how both abstract syntax structures are realized with symbol management infrastructure presented by Nazari [MSN17]. The first part of this sections describes some implementation details of the symbol table and its resolving mechanism. The second part of this section explains how the symbol table's resolving mechanism helps to easily integrate the different languages of the *EmbeddedMontiArc* language family.

Figure 4.27 shows a Java code excerpt of the `ComponentSymbol` class. This Java symbol class is the equivalent class to the `Component` one of the abstract syntax shown in Figure 4.15. The `getPorts` method (cf. ll. 2-5) in the Java class maps to the `ports` association going from `Component` to `Port` in the class diagram in Figure 4.15. The `getSubComponents` method (cf. ll. 6-9) in the Java class maps to the `subs` association going from `Component` to `ComponentInstantiation` in the class diagram in Figure 4.15.

The `ComponentSymbol` class extends the `CommonScopeSpanningSymbol` class, because the component symbol spans a new scope. The reader can interpret each scope as a repository which contains other symbols (cf. Section 1.1.3). In *EmbeddedMontiArc* only text files (`ArtifactScope`) and component type definitions (`ComponentSymbol`) open scopes. All other symbols (elements of the abstract syntax), e.g., `PortSymbol`, `ConnectorSymbol`, and `EffectorSymbol`, do not span a scope.

Links between symbols are not hard coded to support an easy adaption and extension of the abstract syntax. This is explained on an example later in this section. Therefore, the `ComponentSymbol` class does not contain any collection of `PortSymbols` in a field variable storing all the ports belonging to a component, nor does the `ComponentSymbol` contain any collection field for subcomponent instantiations. Instead, the `ComponentSymbol` asks his spanned scope repository to return all symbols of a special kind as it is shown in lines 3 and 4 to receive all ports of a component and as it is shown in lines 7 and 8 to receive all subcomponent instantiations.

The same mechanism holds for the `ConnectorSymbol`, it only stores the source and target names as String variables. The `ConnectorSymbol` asks its enclosing scope to resolve port

Java ...

```java
public class EmbeddedMontiArcSymbolTableCreator extends
                                          EmbeddedMontiArcSymbolTableCreatorTOP {
  @Override public void visit(ASTCompilationUnit node) {
    String cuPackage = Names.getQualifiedName(node.getPackageList());
    List<ImportStatement> imports = ...;
    ArtifactScope artifactScope = new EmbeddedMontiArcArtifactScope(
                Optional.empty(), cuPackage, imports);
    putOnStack(artifactScope);
  }
  @Override public void visit(ASTComponent node) {
   ComponentSymbol component = new ComponentSymbol(node.getName());
   // MontiCore opens new scope, as ComponentSymbol is scope spanning symbol
   addToScopeAndLinkWithNode(component, node);
  }
  @Override public void endVisit(ASTComponent node) {
    removeCurrentScope(); // MontiCore does not remove the scope yet
  }
  protected Boolean isInport = null;
  @Override public void visit(ASTPort node) {
    PortSymbol port = new PortSymbol(node.getName());
    port.setType(...);
    port.setDimension(node.getDimensionOpt().orElse(1));
    // handling syntactic sugar to allow "ports in B in1, Z in2,(0:3) in3;"
    isInput = node.getDirectionOpt().orElse(isInput);
    if (isInput == null) {
      Log.error("0xE1053 no direction at input port specified",
          node.get_SourcePositionStart()); }
    port.setDirection(isInput);
    addToScopeAndLinkWithNode(port, node);
} }
```

Figure 4.28.: Java code excerpt of `EmbeddedMontiArcSymbolTableCreator` which uses the abstract syntax tree to create the C&C model using the symbol table management infrastructure.

instantiation symbols with the target or source name. Since each connector is defined inside a component type definition in `EmbeddedMontiArc`, the enclosing scope of a `ConnectorSymbol` is the spanned scope of a `ComponentSymbol`.

Java developers using the *EmbeddedMontiArc* language do not notice that the symbols of the C&C abstract syntax are loosely coupled via the symbol management infrastructure. Java developers just call the get methods of each symbol to receive the wanted information. In this sense, the *EmbeddedMontiArc* symbol implementation encapsulates all the technical details of the symbol management infrastructure. To receive the first component symbol of the abstract syntax a developer needs only to create an `EmbeddedMontiArcModelingFamily` object having the path to the `Main.txt` file. In a next step, the developer calls `getMainComponentInstantiation` or `getMainComponentInstance` to receive component type instantiation or the component instance of the root component defined in the `Main.txt` file. The root component (instance) supports developers navigating through the Java classes as illustrated in the abstract syntax models in Section 4.2 and Section 4.3.

Figure 4.28 shows Java code excerpt of the `EmbeddedMontiArcSymbolTableCreator`. The symbol table creator builds the abstract syntax based on the abstract syntax tree. The `EmbeddedMontiArcSymbolTableCreator` extends the generated symbol table creator in line 2. *MontiCore* uses the grammar definition file to generate the basic symbol table infrastructure as well as visitor classes to traverse the abstract syntax tree in an efficient way. The generated symbol table creator extends the generated *EmbeddedMontiArc* language visitor. The `EmbeddedMontiArcSymbolTableCreator` overwrites the by default empty `visit` methods to extract all necessary information stored in abstract syntax tree nodes in order to create the elements of the abstract syntax. The first `visit` method in lines 3 to 9 is called when the root AST node of a text file is traversed. Line 4 extracts the full qualified package name, and line 5 collects all import statements in a list. Lines 6 and 7 initialize the artifact scope and add it to the global scope. All scopes added in the next visited `visit` methods automatically belong to the given package name. The symbol table management extends all names in later added scopes to their full-qualified names based on this package and import information.

The second `visit` method in lines 10 to 14 creates the component symbol based on component AST node. **Ports and subcomponents are not added to the component symbol.** Line 13 adds the component symbol with its introduced scope to the previously created artifact scope as well as it links the component symbol to the component AST and vice versa. Line 16 closes the current component scope in the artifact scope. This is needed for nested inner component definitions, so that derived full-qualified names (e.g., port names) of inner components differ from the ones of outer components.

Lines 18 to 29 create the port symbol based on the port AST node. This code excerpt (it is still very incomplete) is a little bit longer to illustrate how the symbol table creator handles syntactic sugar. For example, line 22 sets the port dimension to 1 if it is absent in the AST[9]. Lines 24 to 28 extract the port direction information from the AST node; whereby the previous port direction is used when the port direction is not specified in the AST. However, the first port AST node must specify a port direction; lines 25 to 27 throw an error if this is not the case. Line 29 adds the port symbol to the current scope, which is the scope created by the component symbol in line 13. Additionally, line 29 links the port symbol to the port AST node. An alternative way (and for the author of this thesis the preferred way) of handling syntactic sugar is to specify the relation between *EmbeddedMontiArcParsing* and *EmbeddedMontiArcTooling* via *OCL* constraints as it is done in Section 6.4 and to generate this Java code.

This code snippet illustrates that **associations between symbols of the two classes in the abstract syntax are only linked via the symbol management infrastructure**. The symbol table creator plus its helper classes contain about 1 000 lines of code to create the abstract syntax structure based on the AST. It is so complex, because it must handle many kinds of syntactic sugar (e.g., direction is not necessary, name based connections using the `.*` notation, and index based short-cuts using the `[:]` notation) as well as the symbol table creator must define all parameter definition symbols and their according parameter binding ones.

The next part of this section explains how Go functions can be adapted to component definitions and how the symbol management infrastructure integrates them directly in the C&C model.

[9]This line is a large abstraction by assuming that the dimension is a number. However, the dimension can also be a generic parameter and this case is much more complicated.

Go

```
1  func addsub(x, y int)
2            (sum, diff int) {
3    sum = x + y
4    diff = x - y
5    return
6  }
```

adapter

capitalizes name

EMA

```
20 component Addsub {
21   ports in Z x,
22            Z y,
23         out Z sum,
24            Z diff;
25 }
```

EMA

```
7  component DoubleAddSub {
8    ports in Z x,
9             Z y,
10        out Z sum,
11            Z diff;

12   instance Addsub as;
13   instance Multiplier m[2];

14   connect this.* -> as.*;
15   connect as.sum -> m[1].factor1;
16   connect as.diff ->m[2].factor1;
17   connect 2 -> m[:].factor2;
18   connect m[:].res -> [sum,diff];
19 }
```

Figure 4.29.: Example how Go function can be integrated in C&C model.

The direct integration enables reusing the complete abstract syntax defined in Section 4.2 and Section 4.3 by other tools without any adaption. The adaption of Go functions is only a simple and illustrative example; other languages can also be integrated into the flexible abstract syntax implementation.

Figure 4.29 illustrates an example where the Go function definition (cf. ll. 1-6) is used as component type (cf. l. 12). To support this case, all what a language engineer needs to do is to define an adapter translating Go function symbols to component symbols; lines 20 to 25 show how a translation of such an adapter may look like. The adapter translates Go's integer data type to the whole number ($\mathbb{Z}$) data type of *EmbeddedMontiArc*. It transforms in parameters to input ports, and return parameters to output ports. Additionally, the adapter capitalizes the Go function name during the translation process to satisfy *EmbeddedMontiArc* code conventions.

Figure 4.30 shows how the symbol table management infrastructure resolves the type association of the `ComponentInstantiation` class of the abstract syntax defined in Figure 4.10. First, the language engineer, aggregating the Go language with the *EmbeddedMontiArc* one, creates a new modeling family. For this new modeling family, the language engineer registers all Go adapters, e.g., the one which translates a `GoFunctionSymbol` to a `ComponentSymbol`.

A developer uses this new modeling family as symbol table. The developer calls the `getType()` method of the `ComponentInstantiation` symbol to receive further information about the `as` subcomponent instantiation in line 12 in Figure 4.29. Now, the `ComponentInstantiationSymbol`, shown in Figure 4.30, calls the `resolve` method of its enclosing scope which delegates this request to the global scope of the symbol table. The global scope resolves this symbol further until it looks up the information in a map (cf. [MSN17] for complete workflow). Since the map does not have any symbol with the name `Addsub` it returns `null`

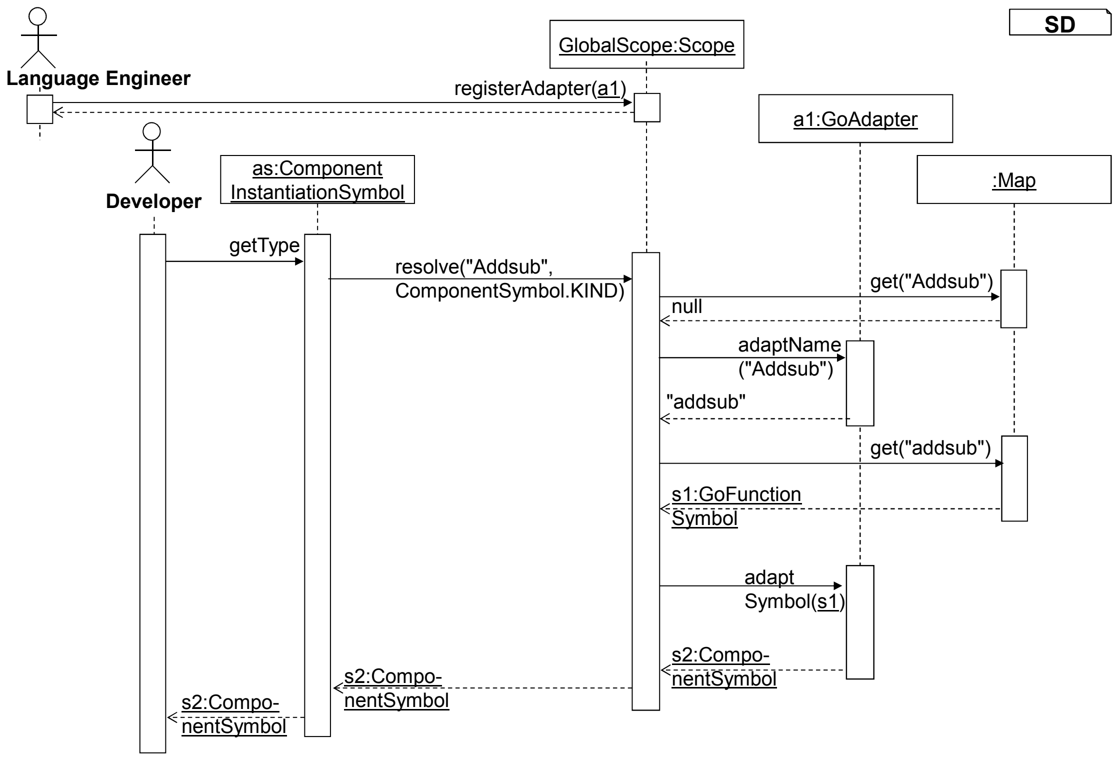


Figure 4.30.: Example workflow how the symbol table resolves the `Addsub` component type name and adapts the `addsub` go function to a `ComponentSymbol`.

for not found. Then, the global scope iterates overall registered adapters to adapt the name[10] to `addsub`; this is the inverse function of the name translation shown in Figure 4.29. In a next step in Figure 4.30, the global scope asks the map for a symbol with the new `addsub` name. Since this key exists in the map, the map returns a `GoFunctionSymbol` to the global scope. As the global scope was asked to resolve a component symbol kind, the global scope calls the adapter to translate the `GoFunctionSymbol` to a `ComponentSymbol`. Last, the global scope returns this `ComponentSymbol` to the `getType` method of the `ComponentInstantiationSymbol`. The `getType` method delegates this result to the developer.

The developer receives an adapted component symbol. If the developer calls the `getPorts` method on this adapted component symbol (not shown in Figure 4.30), this symbol calls `resolveLocally(PortSymbol.KIND)` (cf. ll. 3-4 in Figure 4.27) on the spanned scope of the component symbol to receive all ports. Now the scope iterates over all symbols it contains and checks if one of them has the symbol kind `PortSymbol.KIND`. This is not the case, because the `addsub` Go function does not define any ports. Therefore, the first iteration over the scope's symbols returns an empty set. Next, the scope calls all adapters to adapt the `PortSymbol.KIND`;

[10]In the *MontiCore* implementation this is done via filters, but for simplicity we abstract the filter and call the `adaptName` function of the adapter. Filters and adapters are pairs in the implementation; both are needed together.

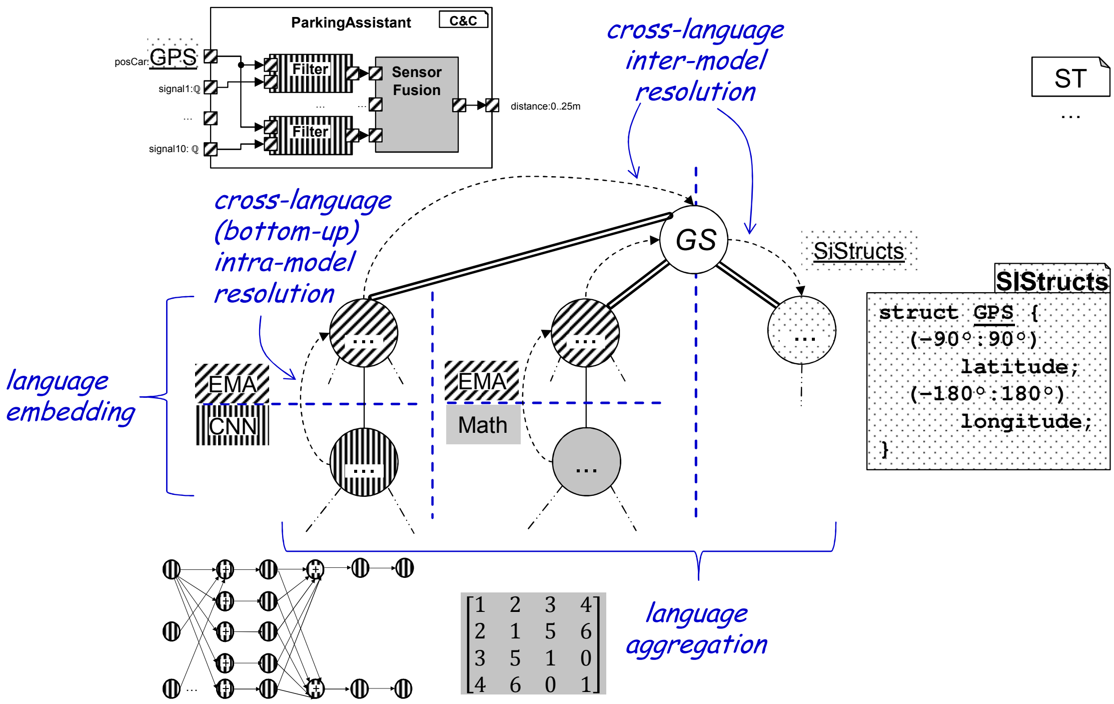


Figure 4.31.: Resolving of Symbols in a scope graph resulting from language composition in *MontiCore* (inspired by [MSN17, Fig. 8.1]).

the registered adapter `a2` translates this kind to `GoParameterSymbol.KIND`, and the adapter `a3` translates the port symbol kind to `GoReturnSymbol.KIND`. Iterating over the scope again and collecting all symbols with these new kinds, returns two `GoParameterSymbol` objects (`x` and `y`, cf. l. 1 in Figure 4.29) and two `GoReturnSymbol` objects (`sum` and `diff`, cf. l. 2 in Figure 4.29). Next, the adapters `a2` and `a3` translate the `GoParameterSymbol` and `GoReturnSymbol` objects to four `PortSymbol` objects. Finally, the scope of the adapted component symbol object `s2` returns these four adapted port symbol objects to the component symbol which delegates them to the developer.

The description of the two workflows (`ComponentInstantiationSymbol::getType` and `ComponentSymbol::getPorts`) elucidates why the implementations of both abstract syntax structures resolve associations between their classes via the symbol management infrastructure of Nazari [MSN17] as shown in Figure 4.27. The designed and implemented abstract syntax realizations are highly extensible for new language aggregations or language embeddings.

The next part of this sections explains on a parking assistant C&C model how the symbol management infrastructure supports to exchange information via symbols of four different languages.

The top left part in Figure 4.31 shows a C&C model belonging to a composed language containing of *EmbeddedMontiArc* (describing the C&C structure with components, ports, and connectors), *SIStructs* (which describes composed data types), *CNN* (to describe the functional behavior via neuronal nets), and *MontiMath* (describing the behavior in a declarative and functional style and supporting matrices) language.

The `ParkAssistant` component type is decomposed of three subcomponent instances having the component type `Filter` and `SensorFusion`. The atomic component type `Filter` describes its behavior via a neuronal net. The atomic component type `SensorFusion` models its behavior via matrix vector multiplications. Each of these three component types are described in its own textual artifact. The *EmbeddedMontiArc* language processes the artifact of the `ParkAssistant` component type; *EmbeddedMontiArcDL* language processes the artifact of the `Filter` component type; the *EmbeddedMontiArcMath* language processes the artifact of the `SensorFusion` component type; and the *SIStructs* language processes the artifact of the `GPS` port type. The main task of the symbol management infrastructure is to aggregate the abstract syntax of these four different artifacts.

The presented `ParkAssistant` component has the input port `posCar` that data type is `GPS`. To receive essential information about the `GPS` data type, e.g., type ranges, or unit kinds, the *EmbeddedMontiArc* language (as it defines the port) queries the symbol table for a `GPS` symbol. Now, the symbol table queries the scope, containing the port symbol, and its subscopes whether they have a `GPS` symbol and then a resolving workflow similar to Figure 4.30 is started; this process is called bottom-up or down resolving.

In our `ParkAssistant` example neither the scope nor its subscopes contain the `GPS` symbol; thus, the symbol table resolves up by asking the parent scopes until they receive the global scope (marked as `GS` in Figure 4.31). The global scope asks all artifact scopes whether they or their subscopes contain the `GPS` symbol. When the global scope resolves symbols, the symbol table also loads automatically text files, which may contain the symbol based on its kind and its name. Loading a text file means parsing the file, creating the AST and symbols, as well as registering symbols in the symbol table. In our example the symbol table would automatically load all component and struct files. While loading the GPS struct file, the `GPS` symbol is found and this symbol is returned to the global scope. Finally, the global scope returns the `GPS` symbol, found in the struct file, back to the *EmbeddedMontiArc* language as resolving result.

This explained process of resolving symbols asked in one language and found in another language is called cross-language inter-model resolution. Efficient language aggregation is only possibly due to this cross-language aggregation, as symbols defined by other languages can be used as they were defined in their own language. This means, it does not matter for tooling (e.g., context condition checks, or type inference) where the symbol is defined. And importantly, the checks for C&C models - such as ports only with the same data types can be connected - do not need to be updated when integrating the *EmbeddedMontiArc* language into the language family containing *SIStructs*.

The same concept holds for language embedding where the *CNN* language defines functional layers and the input data as well the output data is not defined in the *CNN* implementation. In contrast, the *CNN* input data are ports defined in the *EmbeddedMontiArc* language. Due to the intra-model resolution, the *CNN* language asks for a symbol name used in any *CNN* layer and the symbol table automatically resolves this information no matter where it is defined (e.g., in C&C models or in struct models). The resolving mechanism for the *Math* language is very similar to the *CNN* language: the atomic component implementation is a math formula reading input port values and writing its result to an output port.

MontiCore's ability to combine grammars and to exchange symbols between languages enables the development of modular language components and tools which can be completely reused to engineer large language families and powerful modeling tools.

Chapter 5.

Enriching *EmbeddedMontiArc* Models with Extra-Functional Properties

This chapter presents a model-driven approach to enrich component and connector (C&C) models with extra-functional properties. This tagging approach enables non-invasive extensions of the C&C modeling language *EmbeddedMontiArc* with new types of extra-functional properties.

The first section gives an overview of existing extra-functional properties in literature to show how flexible the tagging mechanism must be to support all of them. The second section presents existing approaches for annotating component and connector models with extra-functional properties. This section serves as basis to create the best fitting solution for our tagging approach by considering the best points of existing work. The third section lists the requirements, derived from the first two sections, of our tagging mechanism. The fourth section introduces a turbine controller model that is enriched with different extra-functional properties. The turbine controller is the running example for the rest of this chapter.

The last section presents details of the tagging mechanism for component and connector models. This larger section is divided into five subsections: Subsection 5.5.1 presents the general tagging approach; it introduces all involved artifacts and gives an overview of the relations between these artifacts. Subsection 5.5.2 elucidates the tag schema language to define concrete and abstract syntax of new extra-functional property types. Subsection 5.5.3 explains the tag model language to enrich C&C models with the extra-functional properties as defined in a tag schema model. Subsection 5.5.4 shows the derivation process of class diagrams based on the previously defined tag schema; it also illustrates how the generated class diagrams are merged with the ones representing the abstract syntax of *EmbeddedMontiArc* (cf. Chapter 4). Subsection 5.5.5 lists context condition rules for tag schemas, tag models, and between both.

The tagging approach of this chapter enables a complete model-driven workflow to enrich C&C models with extra-functional properties: (1) The tag schema defines the new extra-functional property type. (2) Tag Models, each conforming to one tag schema, annotate concrete extra-functional properties to existing C&C models. (3) The derivation of class diagrams for tag schemas and merging them with the class diagrams of *EmbeddedMontiArc*'s abstract syntax, integrates these new defined extra-functional properties directly in the well-known C&C model and C&C instance structure; this way *OCL* constraints (cf. Chapter 6) can define context conditions of enriched C&C models.

The here presented tagging approach has the following advantages, compared to most other solutions explained in Section 5.2 [MRRvW16]:

(i) *EmbeddedMontiArc* models are not polluted with extra-functional properties and, thus, these models stay easy to read.
(ii) Inherent separation of concerns enables different domain experts to decorate the C&C model with their own separated extra-functional tagging models.
(iii) Tag models reference C&C elements by their names present in the concrete syntax; hence, no knowledge about the implementation APIs of *EmbeddedMontiArc* are needed to tag these C&C models with extra-functional property values.

5.1. Overview of Existing Extra-Functional Properties

Before a tagging mechanism for extra-functional properties can be defined, we need to analyze what kinds of extra-functional properties exists. Examples of extra-functional properties/requirements are:

- Accuracy [KPMS01]: Mean magnitude of relative error.
- Accessibility [BP06]: Access control and audit for blind people, or older persons.
- Analyzability [BCvDV11]: When is a software optimal decomposed?
- Attractiveness [PSSK14]: Human activeness for features.
- Availability [LKD+03]: For example, service level agreements in cloud computing, and network connected components.
- Backup/Recovery [BI96, SPE11]: Cost, schedule, evolvability, performance, locality.
- Capacity [MA02]: Current and forecast.
- Certification [DEISS09]: ISO certificates, certificate ranking, communication certificates.
- Completeness [SRK+12]: For example, check how many requirements have been implemented, or how many variants covers one product-line via test coverage or model checking.
- Complexity [CSM+79]: Psychological Complexity of Software.
- Compliance of Software Systems [SSC96a, AK13]: Risk management.
- Configure-ability [CBCP02]: Internationalization (e.g., different countries, languages), or Personalization (personal user experience).
- Consistency [WYW+10]: Replica strategy, or consistency levels [Dat17].
- Deployment [HHW99, KH08]: Publishing, discovery, dependency resolution, downloading, installation, (re)configuration launching, activation process, deploying alternative combinations of components , Solution Deployment Descriptor [OAS08].
- Documentation [TH77]: JavaDoc, PSL / PSA.
- Efficiency [CMST03]: Resource consumption for given load, storage efficiency, or execution efficiency.
- Effectiveness: Resulting performance in relation to effort, e.g., via effectiveness metrics [Gac16].
- Emotional factors [vdWS10]: Fun or absorbing.
- Error and attack tolerance [AJB00]
- Expected market: Is the software or product for kids or for adults only, e.g., FSK 16 or FSK 18 [Sei12].
- Exploitability [WZX08]

- Extensibility: Ability to add features, carry-forward of customizations at next major version upgrade; it depends, e.g., on the number of free pins in communication buses [OPSS93].
- Failure management, cf. "Model-Based Failure-Management for Automotive Software" [EMOW07]
- Fault tolerance [DW02, dLdCGR06]: Coverage modes (not failed, failed covered, failed not covered).
- Legal and licensing [DPGA10]: Issues or patent-infringement avoidability instrumentation. Instrumentation of software refers to the process of enabling the software to be monitored at selected points to capture significant system state data at those points.
- Interoperability [KLH+02, CCW+05]
- Maintainability [SRK+08]: Coding guidelines, or cyclomatic complexity of components [CKK01].
- Maturity, cf. "The Capability Maturity Model: Guidelines for Improving the Software Process" [WCC95]
- Modularity: Design structure matrices [SGCH01]
- Performance/Response time: Jitter, response, latency, throughput, cf. palladio component model [BKR09].
- Platform compatibility
- Price
- Privacy, cf. "A Framework for Modeling Privacy Requirements in Role Engineering" [HA+03]
- Reporting: Severity level (warning, error, information), and output format.
- Resource constraints: Processor speed, memory usage, disk space, network bandwidth, energy efficiency, or response time.
- Reusability [SKS92]
- Robustness [Fir04]: Functions under abnormal conditions such as environmental tolerance, error tolerance (wrong user input), failure tolerance (defect in system execution).
- Safety/Factor of safety, e.g., ASIL [Int11]
- Scalability: Single/multi-thread, GPU support, or running on a cluster.
- Security, e.g., permissible information flows. [Den76]
- Stability
- Survivability: System must survive fire, natural catastrophes.
- Testability [VM93]
- Traceability
- Withdraw-ability: Degree to which a problematic version of a system or wrong data can be withdrawn and replaced with a previous versions.

This incomplete list (more properties are listed in [MMR14, Pou94, SCS11b, Rom85]) shows that extra-functional properties are varying very much, and thus the mechanism for defining these properties must be general and flexible. It may even be that further kinds of extra-functional properties will become of interest in the future.

```
property set AADL_Projects is
Time_Units: type units (
    ps,
    ns => ps * 1000,
    us => ns * 1000,
    ms => us * 1000,
    sec => ms * 1000,
    min => sec * 60,
    hr => min * 60);
-- ...
end AADL_Projects;
```

Listing 1

```
property set Timing_Properties is

  Time: type aadlinteger
     0 ps .. Max_Time units Time_Units;

  Time_Range: type range of Time;

  Compute_Execution_Time: Time_Range
   applies to thread, device, subprogram,
       event port, event data port);
end Timing_Properties;
```

Listing 2

```
thread foo
properties -- (1)
  Compute_Execution_Time => 3 .. 4 ms;
  Deadline => 150 ms ;
end foo;

thread implementation foo.impl
properties -- (2)
  Deadline => 160 ms;
  Compute_Execution_Time => 4 .. 10 ms;
end foo.impl;
```

Listing 3

```
process implementation bar.others
subcomponents
  foo0 : thread foo.impl;
  foo1 : thread foo.impl;
  foo2 : thread foor.impl
    {Compute_Execution_Time =>
      20 .. 40 ms;}; -- (3)
properties -- (4)
  Compute_Execution_Time =>
      30 .. 50 ms applies to foo1;
end bar.others;
```

Listing 4

Figure 5.1.: Example how to define and use properties in *AADL* (copied from [Ins15, Slides 16-17]).

5.2. Existing Approaches For Annotating Component and Connector Models with Extra-Functional Properties

In literature exist several approaches in different scenarios where models are enriched with different information. This section summarizes some language mechanisms to enrich C&C models with additional information.

AADL uses typed attributes to associate information to component types, implementations, subcomponent instances, or contained property associations [Ins15, Slides 16-17]. A typed attribute may have one or more properties, collected in a property set. Each property has a name, a type, and a list which component kinds are allowed to enrich. Figure 5.1 shows an example how properties are defined (two left listings), and how they are used (two right listings) in *AADL*. Listing 1 in Figure 5.1 defines time units in *AADL*. Listing 2 in Figure 5.1 defines the compute execution time property for threads, devices, subprograms, event ports and event data ports. The type of this compute execution time property is a time range, which is also defined in Listing 2.

EmbeddedMontiArc's units and type build-in mechanisms supports specifying the type simply by `(0 ps : oo s)`. This type contains all values being greater equals zero picoseconds (`0 ps`). The infinite seconds (`oo s`) means that the type `(0 ps : oo s)` has no upper limit; `-oo s` is the analogue syntax for types having no lower limit.

Listing 3 in Figure 5.1 defines values of this compute execution time properties inside (1) the component type thread, and inside (2) an implementation of a thread. Listing 4 in Figure 5.1 defines values of this compute execution time property inside (3) a subcomponent instance, and a (4) property association.

ACME uses property types to define properties. *ACME* supports multiple representations and views to add values to the defined property types in different artifacts. However, *ACME* does not support relations between property types defined in different artifacts[1]. The left side in Figure 5.2 shows an example how to define the property type `CallType` for a connector having the one property `returnsValue` and how to add this property types to connectors in the `LunarLander` system. The right side in Figure 5.2 shows different property views to enrich the client server architecture with `Visualisation`, `Source Code`, and `Performance Data`.

[1]"The ability to associate multiple representations with a design element (any of the *ACME* building blocks: components, connectors, and so on) enables *ACME* to encode multiple views of architectural entities (although there is nothing currently built into *ACME* that supports resolution of inter-view correspondences)." [GMW00a, pp. 53f.]

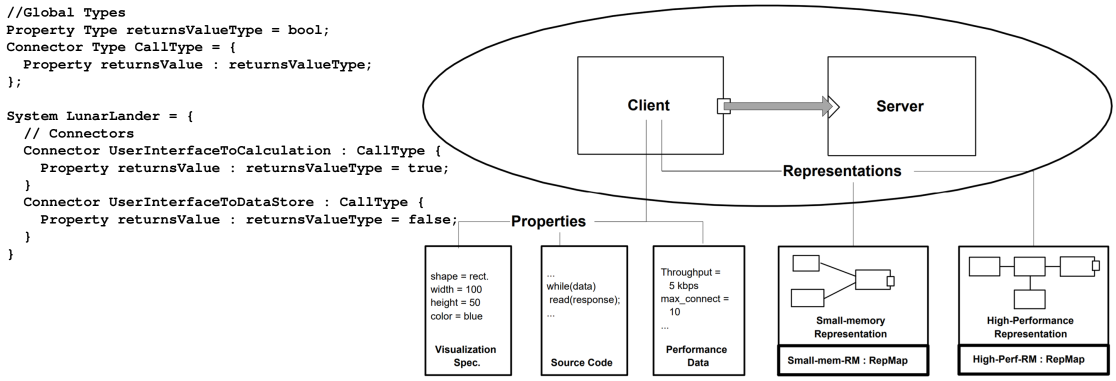


Figure 5.2.: Example how to define and add properties to *ACME* models (left part copied from [TMD10, slide 33]; and right part copied from [GMW00a, Figure 3.4]).

Haugen et. al. [HMPO+08] present an approach where not the base model, in this paper a simple arithmetic model, is directly enriched with annotations; instead, the authors create an additional variation model for specifying model feature combinations. Besides the one advantage of directly marking model elements with variability, the usage of annotations has the one large disadvantage: base models are cluttered with variability specifications, and thus, only one variability model exists for each base model [HMPO+08]. For this reason, Haugen et. al. suggest to separate DSL languages from variation ones. Besides addressing main disadvantage of annotations, the separation approach has the following advantages: (points 1. to 3. are copied from [HMPO+08])

1. Domain experts can concentrate on domain language concepts only.
2. The base DSL becomes compact and simple.
3. The separated approach supports division of labor and separation of concerns.

The ProMoBox [MDL+14] framework enriches domain specific models with temporal properties so that general constraints (e.g., elevator will not pass a passenger more than once) can be automatically verified with Spin [Hol97]. To easily define the temporal properties their approach generates five (design, runtime, input, output, properties) pattern languages, so that users do not need to specify error-prone LTL formulas. Similar to Haugen et al., this approach uses five different domain specific languages for defining different system properties; but furthermore, due to their generative aspect, ProMoBox can guarantee that the five pattern languages are consistent with the previously defined domain specific model.

Lara et al. [LGC14] presents an approach how to remove complexity, e.g., removing powertypes or stereotypes, from two-level modeling by introducing multi-level modeling. It also supports introducing dynamic features, e.g., new extra-functional properties, which are not given in the concrete meta-model.

Selic [Sel07] explains how to refine existing (widely-defined) *UML* diagrams with profiles, or stereotypes. The usage of this defined stereotypes or profiles is constrained via *OCL*. A profile can contain several stereotypes being in relations which each other. The advantages of this approach are separated abstract syntax models (the C&C abstract syntax model and the profile ones adding

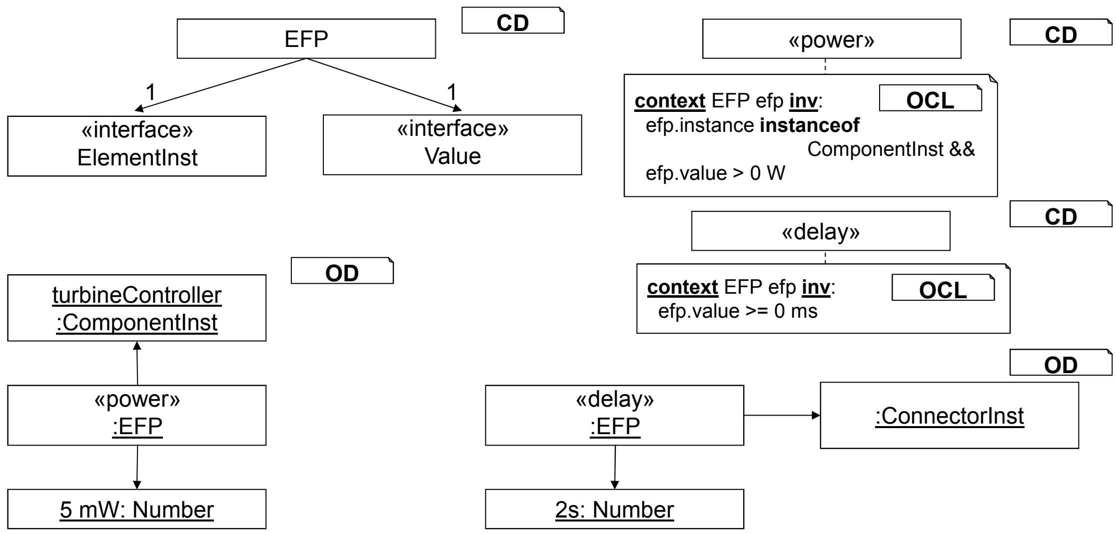


Figure 5.3.: Example how to add extra-functional properties via profiles.

new extra-functional properties), and separate object diagrams which can be merged (weaved) to one large diagram later. Figure 5.3 shows an example how to add extra-functional properties to models via profiles. The downside of this approach is that tagging many properties via object diagrams is time consuming; number objects `5mW` and `2s` are not completely modeled as all attributes (cf. Figure 4.5) are omitted.

Figure 5.4 shows how to add non-functional properties via tagged values to systems engineering diagrams using *UML/MARTE* NFP framework [EDG+05]. Special about this *MARTE* approach is that *MARTE* also adds the source property to extra-functional property; e.g., `calc` for calculated, or `req` for requirement in Figure 5.4. In *MARTE* complex extra-functional properties (`complexNFP`) may have multiple extra-functional property values. For example, the `Latency` property has the two values worst-case execution time (`WCET`) and `deadline`. *MARTE* NFP has full unit support: Numbers and units can be directly assigned as values, e.g., `WCET(5.0, ms, calc)`. Extra-functional property data types specify the allowed unit kinds, e.g., `DurationUnitKinds`; these unit kinds are very similar to our quantities defined in Figure 3.18. One drawback of the *MARTE* NFP approach is that the model (the activity diagram in Figure 5.4) is directly annotated with these extra-functional property values and not separated as suggested by Haugen et. al.

Figure 5.5 illustrates how extra-functional properties are defined and added to C&C models with the attribute framework [SSCC09] of ProCom. ProCom is a two layer (ProSys, and ProSave) component model for control-intensive distributed embedded systems [SVB+08, BCC+08].

The left side in Figure 5.5 presents an attribute type registry. It contains all defined extra-functional properties of an organization. The type identifier must be unique. It is also possible to group the registry into categories such as `resource usage`, `reliability`, or `timing` [SSCC09].

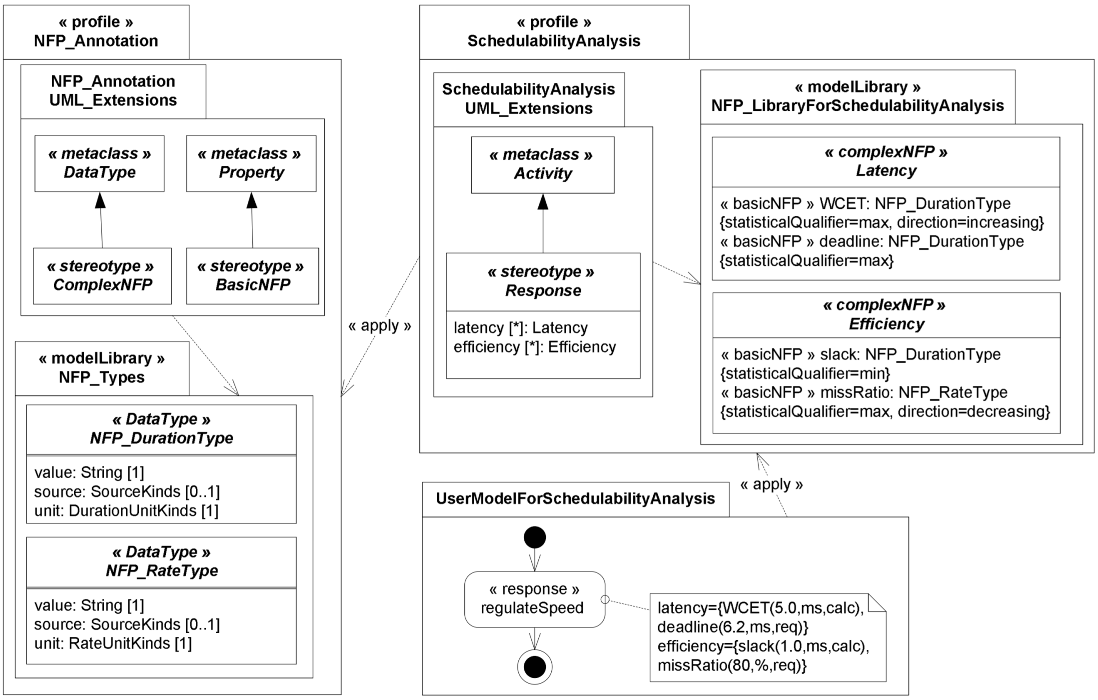


Figure 5.4.: Applying tagged values for annotating non-functional properties with the *MARTE* NFP framework (copied from [EDG+05, Figure 6]).

Sentilles [SSCC09] et al. can specify multiple values per attribute on their extra-functional properties including conditions when an attribute should be valid, e.g., testing or production, plus dependencies between attributes, and a version number. The data format of an extra-functional property are primitive types (e.g., `Integer`, or `Float`), structured types (e.g., arrays), or complex types (e.g., value distribution, external models, or images).

As shown in the right side of Figure 5.5, the framework also stores meta data for attribute values. Example meta data are the source of the value (e.g., requirement, estimation, measurement, simulation, formal analysis with tool X, or generated from implementation), `timestamp`, or accuracy [SSCC09]. Besides meta data, a value attribute may consist of multiple validity conditions; e.g., specific platform, usage profiles, or attribute dependencies [SSCC09].

Since one component may have different attribute values for an attribute, there must exist a selection strategy to filter the wanted extra-functional property values. In the right part of Figure 5.5, the attributes having a gray background color are deselected.

Look [GLRR15], [Loo17, Section 4.3] et. al. present an approach to derive tag languages and their tag schema languages systematically from existing domain specific languages (DSLs). The advantage of this approach are "clean, readable, and reusable" [GLRR15] DSLs as well as the tags follow a defined type schema. Look et. al. derive the tag schema and the tag model languages based on an existing DSL. This "systematic derivation considerably reduces the effort necessary to implement the tag language" [GLRR15].

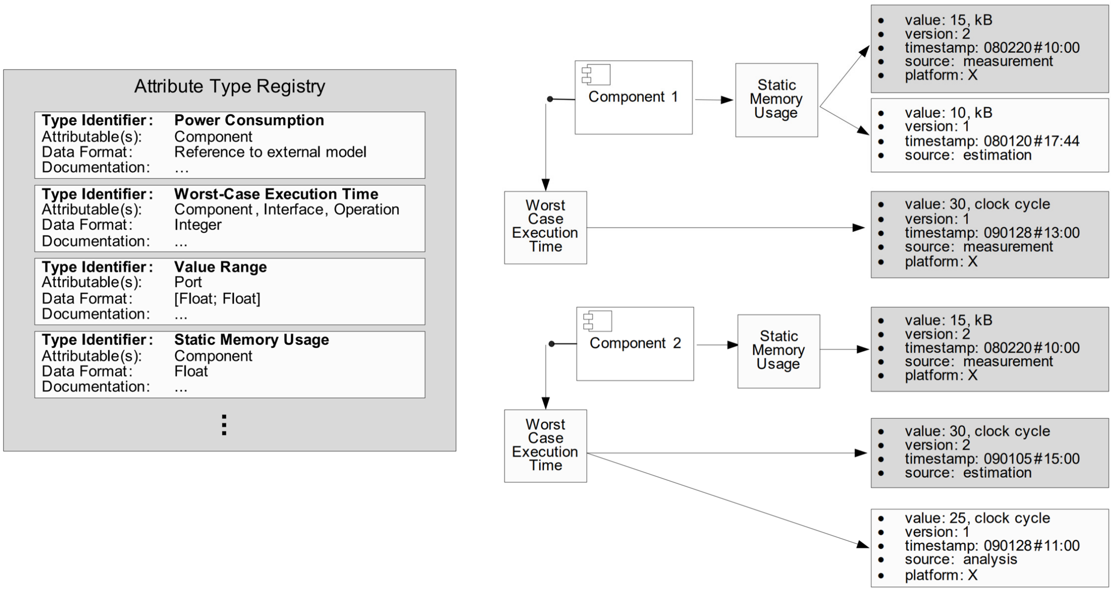


Figure 5.5.: Attribute Type Registry to define extra-functional properties (copied from [SSCC09, Fig. 2]). Right: Attribute configuration and selection (copied from [SSCC09, Fig. 8]).

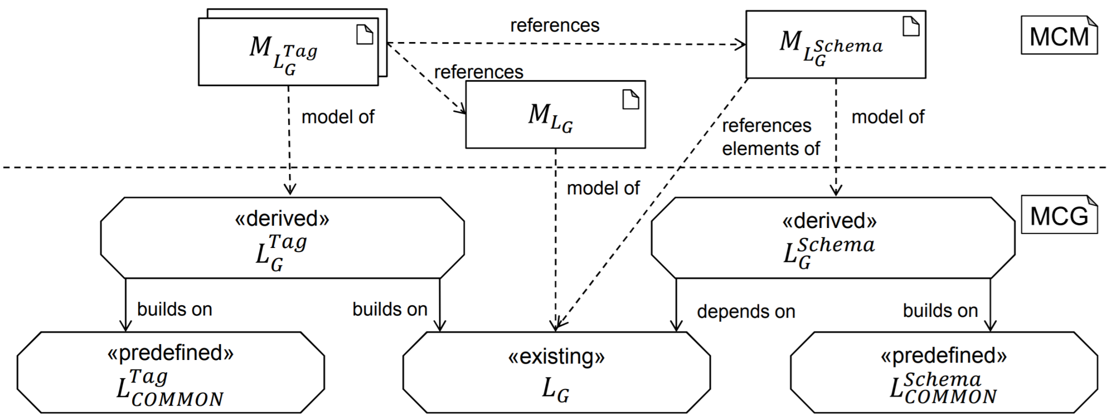


Figure 5.6.: Tagging approach of Look et. al. (copied from [GLRR15, Fig. 2]).

Figure 5.6 shows that based on an existing language grammar L_G and the predefined common tagging L^{Tag}_{Common} and schema L^{Schema}_{Common} languages, the grammar files of the tag model L^{Tag}_G and the tag schema L^{Schema}_G languages are derived. The schema model $M_{L^{Schema}_G}$ is a model of the L^{Schema}_G language and it defines new schemas, e.g., extra-functional property ones. The tag model $M_{L^{Tag}_G}$ is a model of the L^{Tag}_G language and it enriches models M_{L_G} with extra information conforming to an $M_{L^{Schema}_G}$ tag schema model.

Besides annotating models via tags, stereotypes, profiles or typed attributes; in literature also exist transformation languages and tools to transform models into a version enriched with appropriate information [Loo17]. Examples of transformation languages or tools are UMLAUT [HJLGP99], XSLT [MVG06], Atlas Transformation Language [JK06], concrete syntax-based graph transformation [GMPO09], MontiTrans [Wei12b, HRW15], T-Core [SVL15], and QVT Relations [Wes18]. This section does not take a closer look at this transformation languages and tools, as results of transformations are large models polluted with many extra-functional properties. And this thesis prefers an approach separating extra-functional properties as suggested by Haugen et. al. and Look et. al.

5.3. Requirement Analysis

Based on the literature survey of extra-functional properties (cf. Section 5.1), and already existing approaches (cf. Section 5.2) to enrich models (esp. C&C models) with information, our tagging mechanism should satisfy the following requirements:

- **(T1)** Modeling of extra-functional properties should be done in separated files due to separation of concerns.
- **(T2)** Tagging mechanism must support (a) to define new extra-functional property types and (b) to annotate models with consistent values to these types.
- **(T3)** Tagging mechanism must support units, because most extra-functional properties in embedded systems have units.
- **(T4)** Tagging mechanism should support tables, since prices (e.g., quantity discount), and mechanical properties (e.g., transmission ratios for gears) are specified in tables.
- **(T5)** Extra-functional properties may restrict its tagging capabilities, e.g., extra-functional property `delay` should tag connector elements in a C&C model.
- **(T6)** Elements may be tagged multiple times by extra-functional properties of the same type.
- **(T7)** Support to add meta-data (cf. Sentilles) to property types. Tags must be able to be tagged again.
- **(T8)** Tag values may specify multiple extra-functional properties, e.g., structures or sets.
- **(T9)** Mechanism to select extra-functional property attributes based on its meta data or based on values of these properties (cf. Sentilles).
- **(T10)** Conditions when an attribute of a type maybe reused (cf. Sentilles); e.g., to express dependencies on extra-functional property types.
- **(T11)** Definition of syntactical constraints about C&C models enriched with (different) tags.

(T1) is crucial, because “it is reasonable to assume that hundreds of attribute types or more will be introduced” [SSCC09, p. 5], and they should not all be defined in one file.

(T2) is needed when the extra-functional properties should be further processed; and due to the different structure (values, statistical distributions) of extra-functional properties there exists not one general abstract syntax structure.

(T3) is obvious, when looking at units of the extra-functional properties: jitter in ms, response in ms, throughput in Gbit/s, processor speed in GHz, memory in MB, energy usage in W, price in Euro, or temperature working area in °C.

(T4) is not needed, but it makes defining many key-value pairs easier as tagging one element several times with a structure representing a single row of table. A common example is rights management, where a table contains user names with trusted levels such as user or administrator for a component.

(T5) helps to have a consistent tagging of information.

(T6) is needed when modeling a product-line of hardware, because the same user functionality has different extra-functional properties such as down-time, accuracy, throughput, and price.

Most tags are enriched with meta data such as version number, date created, date modified, or source. Enriching tags with meta data (T7) enables to define the meta in one central place, and multiple tags can be tagged with the same meta-data, plus the extra-functional tags (latency, price) are not polluted with all the meta-data which is quite uninteresting for the extra-functional property expert.

(T8) it is often needed as the examples in *MARTE* NFP and ProCom's attribute framework (Figure 5.4 and Figure 5.5) showed.

(T9), (T10), and (T11) are beyond just enriching models with data. These requirements deal with more complex mathematical expressions between extra-functional property values. Some remarks to (T11): Enriching a C&C model with tags adds new properties and constraints to the model. However, tags may lead to inconsistent C&C models. Therefore, the tagging mechanism must support a way to define syntactical constraints to identify when C&C models enriched with (multiple) extra-functional properties are consistent. The syntactical constraints about extra-functional tags are derived by the semantics of the extra-functional tag types. For example, a constraint may restrict that the tagged price of a component is larger equals to the sum of the tagged prices of its subcomponents; this constraint is based on the meaning of the tag type price that bought items in the real world (i.e., our subcomponents) cost some money.

The next chapter explains how to express constraints to filter (return only elements satisfying this constraint) or to check properties with *OCL* expressions. This *OCL* framework also supports to define constraints or dependencies of different domain elements enriched with extra-functional properties.

5.4. Running Example

Figure 5.7 shows the turbine controller C&C model that is used as running example to explain the tagging mechanism in this chapter. The C&C architecture without extra-functional properties is a modified version of a *Simulink* wind turbine controller (cf. [SSCS16, Fig. 4]) of an industrial prototype. *Wind Turbine System: An Industrial Case Study in Formal Modeling and Verification* [SSS$^+$13] formalizes this turbine controller model as timed automata and it also shows simulation results executing this model. The paper *Wind Turbine Control Using PI Pitch Angle Controller* [HK12] presents recommended coefficients for a wind turbine controller to have the best performance of a 5 MW wind turbine without destroying the wind turbine - if it is too windy, the blades get in less optimal positions to limit the turbine performance. The `v` in the

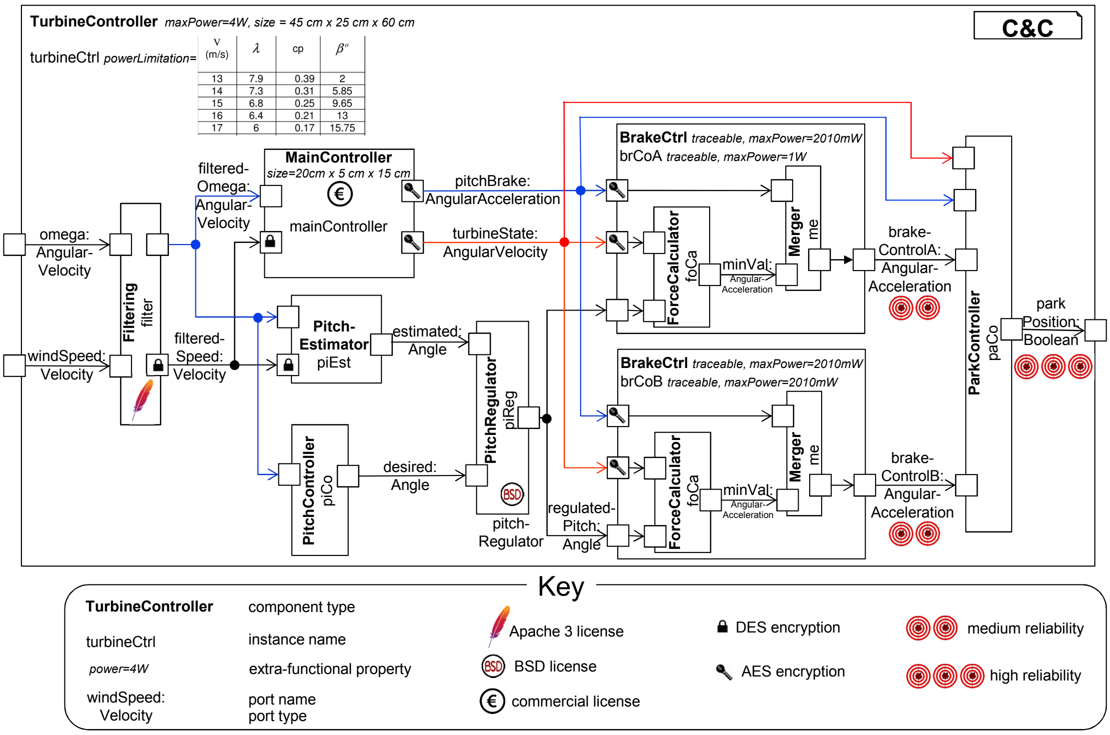


Figure 5.7.: Turbine controller model enriched with extra-functional properties (architecture slightly modified from [SSCS16, SSS+13]; power limitation table copied from [HK12]).

`powerLimitation` table presents the actual wind speed in meter per second, `lambda` is the tip speed ratio, `Cp` is the power coefficient of the wind turbine, and `beta` is the blade pitch angle.

The turbine controller consists of eight subcomponent instances. The `Filtering` subsystem transduces, filters and scales the wind and plant signals [SSS+13]. The main controller handles the performance and operations of the wind turbine to maximize the energy production and to prevent any damage [SSS+13]. Based on the environment conditions such as wind state, the controller selects the turbine's operational model, i.e., park, start-up, generating, or braking [SSS+13]. The pitch controller calculates the proper pitch angles to steer the rotor blades when starting up the turbine or when generating power [SSS+13]. The two brake controllers ensure the safety of the wind turbine, e.g., during wind turbulences [SSS+13]. The pitch estimator guesses the current pitch of the wind turbine by using interpolated history of sensor data.

Teams being responsible for different aspects (e.g., intellectual property, efficiency, and safety) of the wind turbine added important extra-functional properties to the turbine controller model. Since the main controller is bought-in as hardware solution, its size is completely specified. The overall size of the turbine controller chip is also specified, as the controller hardware must fit in the plant. The brake controller types specify the amount of energy they are allowed to use for braking; more energy does not work due to cooling issues. The two brake controller instances brake with different intensity, and, therefore, they have a different maximum energy consumption.

EMA ...

```
component TurbineController {
  port out B parkPosition;

  instance Filtering filter;
  instance MainController mainController;
  instance PitchEstimator piEst;
  instance BrakeCtrl(50%)  brCoA;
  instance BrakeCtrl(100%) brCoB;
  instance ParkController paCo;

  connect brCoA.brakeControl -> paCo.brakeControlA;
  connect brCoB.brakeControl -> paCo.brakeControlB;
  connect paCo.parkPosition  -> parkPosition;
}
```

EMA ...

```
component Filtering {
  port out (0 m/s : 20 m/s) filteredSpeed;
}
```

EMA ...

```
component MainController {
  ports in (0 m/s : 20 m/s) filteredSpeed,
        out (-5°/s^2 : 0°/s^2) pitchBrake,
            (0 °/s : 10°/s) turbineState;
}
```

EMA ...

```
component PitchEstimator {
  ports in (0 m/s : 20 m/s) filteredSpeed;
}
```

EMA ...

```
component PitchRegulator { }
```

EMA ...

```
component BrakeCtrl ( (0% : 100%) maxBrakeForce ) {
  ports in (-5°/s^2 : 0°/s^2) pitchBrake,
           (0 °/s : 10°/s) turbineState;
}
```

EMA ...

```
component ParkController {
  ports in (-5°/s^2 : 0°/s^2) brakeControlA,
           (-5°/s^2 : 0°/s^2) brakeControlB,
        out B parkPosition;
}
```

Main.txt

```
Main-Component-Instantiation: TurbineController turbineCtrl;
```

Figure 5.8.: *EmbeddedMontiArc* code of `TurbineController` C&C model of Figure 5.7. Only the elements, enriched with extra information, are shown.

The brake values, shared between the brake controllers and the park controller, are estimated values how hard the actuators actually brake. These brake values are not 100% reliable, its actual braking depends on the outside weather conditions. In contrast, the Boolean park condition connection from the park controller to the turbine controller is 100% reliable as the systems knows for sure whether the rotor blades are locked or not.

To protect the intellectual property from reverse engineering of the bought-in main controller chip, the communication of the speed input port and both output ports are encrypted. The `Filtering` component type uses an Apache 2 licensed library, and the pitch regulator component uses a library licensed under BSD 2. The main controller hardware is bought-in and thus has a commercial license. All other components are in-house developments and have no licenses, yet.

Figure 5.8 shows the textual *EmbeddedMontiArc* code of the graphical C&C model of the turbine controller in Figure 5.7. Figure 5.8 contains only the modeling elements, which are tagged with extra-functional properties later in this chapter. The data type `B` in lines 2 and 31 stands for Boolean ($\mathbb{B} := \{true, false\}$). The other data types are numerical data types representing a range, e.g., `filteredSpeed` in line 13 produces values between 0 meter per second and 20 meter per second. The `instance` keyword creates subcomponents of a given component type. Line 6 and line 7 create two `brCoA` and `brCoB` subcomponents of the `BrakeCtrl` type, whereby the first brake controller uses maximal 50% of the available brake force to save energy. The connect keyword connects the source port, left of the `->` arrow, with the target port, right of the arrow, to model data flow. Line 34 says that the `TurbineController` component type creates the root component `turbineCtrl` of this C&C model.

5.5. Tagging Mechanism for Component and Connector Models

This section presents the tagging mechanism for *EmbeddedMontiArc*. The tagging mechanism of *EmbeddedMontiArc* is based on the tagging engineering approach for domain specific languages by Look et. al. [GLRR15, Loo17]. The tagging mechanism contains two languages: the tag schema language, and the tag model language. The tag schema one defines the structure of an extra-functional property: what elements can be tagged and what format is used. The tag model language enriches existing C&C models with extra-functional properties from different domains without modifying the textual *EmbeddedMontiArc* files.

In contrast to Look et. al. [GLRR15, Loo17] where the tagging mechanism works on the abstract syntax tree, our tagging mechanism works on both abstract syntax structures, C&C model and C&C instance structure (cf. Chapter 4), based on the symbol management infrastructure [MSN17]. Thus, our tagging mechanism can address all (symbol) elements which have a concrete or derived name. The derived names of connectors and effectors are `sourcePortName -> targetPortName`.

The first part in this section presents the general approach of the tagging mechanism of *EmbeddedMontiArc*. The second part elucidates the tag schema definition language, the third part explains the tag model language, the fourth part shows the derivation process of class diagrams based on tag schemas, and the last part presents some general consistency rules between tag model and tag schema.

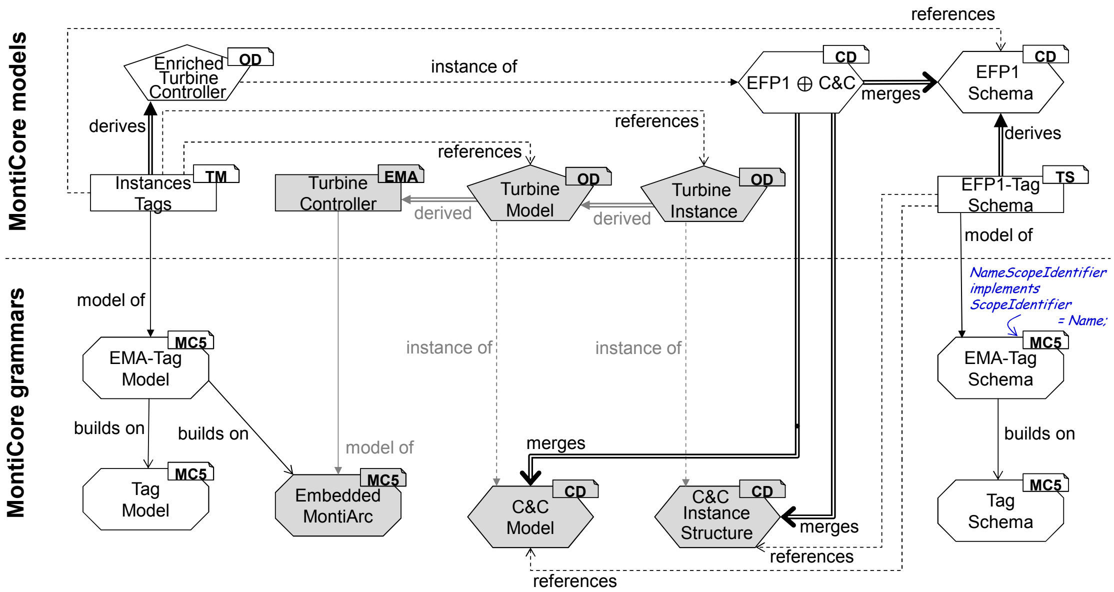


Figure 5.9.: Overview of artifact relations used by tagging mechanism (inspired by [GLRR15, Fig. 2], cf. Figure 5.6).

5.5.1. General Approach

Figure 5.9 illustrates the used artifacts and their relations of the tagging mechanism. The *EmbeddedMontiArc* tagging approach follows the general one presented by Look et. al. [GLRR15]. The gray parts of Figure 5.9 illustrate the artifacts and their relations explained in Chapter 4. The `Tag Model` and `Tag Schema` grammars are extended versions of L^{Tag}_{Common} and L^{Schema}_{Common}. `EmbeddedMontiArc` grammar and the two class diagrams, `C&C Model` and `C&C Instance Structure`, containing the abstract syntax of *EmbeddedMontiArc* language represent the existing L_G language in Look et. al.

The *EmbeddedMontiArc* tag model (`EMA-Tag Model`) maps to the derived L^{Tag}_G language in Look et. al. `EMA-Tag Model` builds on `Tag Model` to reuse the grammar structure of the five tag kinds (cf. Subsection 5.5.2), and it builds on `EmbeddedMontiArc` grammar to reuse the concrete syntax rules of connectors and effectors (but the `EMA-Tag Model` removes the `connect` and `effect` keywords) as well as the concrete syntax rules of arrays of ports and component instantiations. The `EMA-Tag Schema` grammar is the derived L^{Schema}_G grammar in Look et. al. The `EMA-Tag Schema` grammar extends the `Tag Schema` one to reuse all of its rules; the `EMA-Tag Schema` grammar adds only the `NameScopeIdentifier` rule (cf. comment in Figure 5.9) so that after the `for` keyword in a tag schema model every arbitrary name can be used. A context condition of the `EMA-Tag Schema` language checks whether the name after the `for` keyword can be resolved to any class names of imported class diagrams. If the two *EmbeddedMontiArc* class diagrams are imported, the `NameScopeIdentifier` restricts what C&C elements should be enriched with this tag (satisfying **(T5)**). If the class diagram of another tag schema is imported, the `NameScopeIdentifier` restricts what tag types of the other properties are allowed to enrich; this enables to enrich tags with meta-data (satisfying **(T7)**). The `EFP1-Tag Schema` is a model of the `EMA-Tag Schema`. When importing the two C&C class diagrams, the context condition checks are the two references from `EFP1-Tag Schema` to `C&C Model` and `C&C Instance Structure` in Figure 5.9. These context condition checks present the `depends on` arrow in Look et. al. (cf. Figure 5.6).

The `EFP1-Tag schema` is one tag schema model of the *EmbeddedMontiArc* `tag schema`. This tag schema defines some extra functional property types. This approach supports multiple tag schemas; this way for new extra-functional property types, a new tag schema can be defined (satisfying **(T2)**). Besides tag schemas for extra-functional properties, a tag schema can also be used for other properties, e.g., *ROS* tag schema [Hel18] or layout schema (cf. Subsection 5.5.3). The `EFP1 schema` is a class diagram representing the abstract syntax of the `EFP1-Tag schema`. The class diagrams are automatically derived from the defined tag schema models (cf. Subsection 5.5.4). Both, `EFP1-Tag schema` and `EFP1 schema`, map to $M_{L_G^{Schema}}$ in Look et. al. The `EFP1 Schema` class diagram and the two C&C class diagrams, i.e., `C&C Model` and `C&C Instance Structure`, are merged to the `EFP1` $\oplus$ `C&C` class diagram. The merged class diagram enables expressing *OCL* constraints (cf. Chapter 6) on *EmbeddedMontiArc* models enriched with tags conforming to the `EFP1-Tag Schema` in a convenient way. If multiple tag schemas exist, then the merged class diagram merges all these tag schemas with both C&C class diagrams.

`Instances Tags` maps to $M_{L_G^{Tag}}$ in Look et. al. `Instances Tags` is a model of `EMA-Tag Model` which tags the `TurbineController` *EmbeddedMontiArc* model. `Instances Tags` has references to, similar as in Look et. al., the `EFP Schema` as well as to `Turbine Model` and `Turbine Instance`. The `Enriched Turbine Controller` object diagram is derived from the `Latency Tags` and the `Turbine Controller` models, and it is an instance of the merged class diagram `EFP` $\oplus$ `C&C`.

The merged class and the enriched object diagrams provide developers a combined data structure containing all C&C architectural elements and all extra functional property values. The tagging approach combines the best of both worlds, i.e., tagging in separated artifacts and enriching models with profiles/stereotypes: (1) The textual artifacts of C&C architecture and extra-functional properties are separated (satisfying **(T1)**), so that independent domain experts can work on/version them separately; and (2) the combined data structure contains all the separated information in one object diagram, so that developers can easily access the marked elements such as they were all defined in one artifact.

5.5.2. Tag Schema

The tag schema defines the concrete and abstract syntax of the tags used to decorate C&C models. The tag schema language supports the five tag kinds:

- K1 Simple tags, when one only cares whether a C&C element is or is not tagged with this information, similar to a Boolean flag;
- K2 Single valued tags, decorating a C&C element with a tag containing a value, such as `Boolean`, `Number`, `String`, enumeration value or a JScience [Dau07] quantity (e.g., `Power` or `DataAmount`);
- K3 Complex tags to store several values, such as estimated worst-case-execution time [SSCC09]; e.g., `wcet = {time=800ms, confidence=50%};`

TagSchema

```
import embeddedmontiarc.*; // import all classes of EMA class diagrams
tagschema EFP1Schema {
  tagtype traceable for Component, ComponentInst;
  tagtype maxPower:Power for Component, ComponentInst;
  // enumeration type: license can one of the values GPL, ..., BSD2
  tagtype license: [GPL | Commercial | Apache3 | BSD2] for ComponentInst;
  // ports in component types are tagged with a set
  tagtype encryption: [AES |RSA | DES | DES3]* for Port;
  // ports of a component instance are tagged with at most one value
  tagtype encryption: [AES |RSA | DES | DES3] for PortInst;
  // use of SI type system with ranges
  tagtype reliability: (0% : 100%) for Connector;
  // regex type to define multiple values, e.g., "size = 45cm x 25cm x 7cm"
  tagtype size: { ${length: (0m : 100m)} x ${width: (0m : 100m)} x
                ${height: (0m : 100m)} } for Component;
  // table type to define multiple key value pairs efficiently
  tagtype powerLimitation: | v: (0 m/s : 100 m/s) | lambda: (0: 10) |
                             cp: (0 : 1)          | beta0: (0 : 40) |
                                                       for ComponentInst;
}
```

Figure 5.10.: Tag schema definition for extra-functional properties presented in turbine controller example (cf. Figure 5.7).

K4 Regex tags to store several values similar to complex tags in a more convenient way; e.g., `ConnectorLayout = { pos = (30, 50), end = (80, 90), mid = (70, 75) }`; and

K5 Table tags to assign a table as value to this property type; the type defines table header being a list of columns containing of names and types; e.g. `| keyCol: Type1 | col2: Type2 | col3: Type3 |`.

Our tagging paper [MRRvW16] presented the first four tag kinds. The first three tag kinds are similar to the ones of Look et. al. The second kind (K2) adds to the approach of Look et. al. unit support (satisfying **(T3)**) and support of ranges, e.g., `(0 : 20)`. The third (K3) and fourth (K4) kind enables multiple attributes for one extra-functional property (satisfying **(T8)**). The fifth kind (K5) enables to tag C&C elements with tables (satisfying **(T4)**).

Figure 5.10 shows the tag schema definitions for the extra-functional properties of the turbine controller example shown in Figure 5.7. All tags start with `tagtype`, have a name, and end with `for` plus the C&C model or instance structure element on which the tag type can be applied. Valued tags have after the name additionally a colon followed by a data type. Tag kinds (K2) to (K5) are valued tags. The `EFP1Schema` tag schema in Figure 5.10 contains one simple tag `traceable`, which can be applied to component type definitions (`Component`) and component instances (`ComponentInst`). Additionally, this tag schema defines four single valued tags `maxPower`, `license`, `encryption`, and `reliability`. The type of the `maxPower` tag is the quantity `Power`; thus, it accepts values such as `7 mW`, `-4 W`, or `100 kW`.

The type of the `license` tag is an enumeration with the values `GPL`, `Commercial`, `Apache3`, and `BSD2`. In contrast to the approach of Look et. al., the enumeration items do not need to be in quotation marks. This increases the readability a lot. The *MontiCore* grammar of our tag schema separates the enumeration items based on the pipe token; quotation marks are only needed if the enumeration item name contains spaces, the pipe token, or squared brackets.

The tag `encryption` is defined twice; once for ports of component types and once for ports of component instances. A tag can only be defined multiple times with the same name when elements it annotates are disjunctive. In this example, an element cannot be a port definition of a component type definition and a port instance of a component instance at the same time. The `encryption` tag for port definitions are a set of enumeration items (cf. `*` cardinality in l. 8). The `encryption` tag of a port instance is a single enumeration item, because it does not contain a `*` cardinality. The `encryption` tag for port definitions uses a list whereas the encryption tag for port instances uses no list, because a port definition may support multiple encryption modes, whereas a concrete port instance en-/decrypts its data using one concrete algorithm.

The tag reliability has a SI unit range type, which forces that all values are between `0%` and `100%`, whereby the value `0.25` is the same as `25%`. The reliability tag can only be used to enrich connectors in component type definitions; not in component instances for which the `ConnectorInst` class exists.

Figure 5.10 does not show a plain complex tag as they are supported by Look et. al. Instead, it shows in lines 14 and 15 the new regex tag type. The regex is defined between curly brackets. The regex itself can contain any regular expression, also escaped curly brackets. The regex expression has been extended with template variables similar to *FreeMarker*. A template variable is defined between `${` and `}`. Each template variable includes a name and a primitive type, e.g., `Boolean`, `String`, or even any SI unit range type. Based on the specified type, a regular expression is generated to match the variables. The generated regular expression based on the specified regex tag kind handles whitespaces in the same way as *MontiCore* does: One whitespace in the expression can match zero up to infinite whitespace, tabs, or new line characters. This means the regex tag kind defined in lines 14 and 15 matches `45cm x 25cm x 7cm`, `45 cm x 25 cm x 7 cm`, and even the bad readable one `45cmx25cmx7cm`. For all these three expressions, the generated Java code creates a size object having the following attribute values: `length = 45 cm`, `width = 25 cm`, and `height = 7cm`. The regex tag type facilitates creating nice syntactic syntax for complex tag types; developers defining many of these combined tags will be thankful.

The power limitation tag is a table tag; this means the value of one tag is a complete table. The table of the power limitation has four columns. The first column of a table tag is always the key column; thus, all elements in this column must be unique. The first column of the power limitation tag accepts wind speed values between `0 m/s` and `100 m/s`; `10 km/h` is also a valid value. The second, third and fourth column accept values between zero and ten, zero and one, as well as zero and fourty.

5.5.3. Tag Model

Figure 5.11 shows the `TypesTags` model to enrich component types, as well as ports and connectors of component types with extra-functional properties according to the TurbineController C&C model shown in Figure 5.7. The tag model is conforming to the previously defined

TagModel

```
1  conforms to EFP1Schema;
2  tags TypesTags {
3    tag TurbineController with maxPower = 4W, size = {45cm x 25cm x 60cm};
4    tag Filtering with license = Apache3;
5    tag Filtering.filteredSpeed with encryption = {AES, RSA, DES, DES3};
6    tag MainController with license = Commercial;
7    within MainController {
8      tag filteredSpeed with encryption = {DES, DES3};
9      tag pitchBrake, turbineState with encryption = {AES, RSA};
10   }
11   tag PitchEstimator.filteredSpeed with encryption = {DES, AES};
12   tag PitchRegulator with license = BSD;
13   tag BrakeCtrl with traceable, maxPower = 2010mW;
14   tag BrakeCtrl.pitchBrake, BrakeCtrl.turbineState with encryption = {AES};
15   within TurbineController {
16     tag paCo.parkPosition -> parkPosition
17       with reliability = 100%;
18     tag brCoA.brakeControl -> paCo.brakeControlA, brCoB.brakeControl ->
19         paCo.brakeControlB with realiability = 80%;
20   }
21 }
```

Figure 5.11.: Tag model `TypesTag` enriching component, port, and connector definitions with extra-functional properties.

`EFP1Schema` tag schema (cf. l. 1). Line 3 tags the `TurbineController` component definition (cf. l. 1 in Figure 5.8) with one single valued tag (K2) `maxPower`, and with one regex tag (K4) `size`. Line 3 is a short form for `tag TurbineController with maxPower = 4W` and `tag TurbineController with size = {45cm x 25cm x 60cm}`. Line 4 tags the `Filtering` component definition (cf. l. 13 in Figure 5.8) with one single valued tag (K2) of an enumeration. Similar to the tag schema defining enumeration items without quotation marks, a tag model can use these enumeration items also without quotation marks. Line 5 tags the `filteredSpeed` port (cf. l. 14 in Figure 5.8) of the `Filtering` component definition with `encryption`. Since the `encryption` tag for port definitions is a set of enumeration items (cf. `*` sign in l. 8 in Figure 5.10), the value of the encryption tag is a set with all four available encryption modes.

Lines 7 to 10 open the namespace of the `MainController` component definition (cf. l. 16 in Figure 5.8). The tagged names of lines 8 and 9 are names inside the `MainController` scope. Lines 8 and 9 enrich the `filteredSpeed`, `pitchBrake`, and `turbineState` ports with lists of encryption modes. Line 8 inside the `within` expression is equivalent to `tag MainController.filteredSpeed with encryption = {DES, DES3}` outside the `within` expression. Lines 11 to 14 work in the same way as lines 3 to 6. Lines 16 to 19 tag the connectors in the `TurbineController` component definition (cf. l. 1 in Figure 5.8) with reliabilities. The concrete syntax `paCo.parkPosition -> parkPosition` (cf. l. 16 in Figure 5.11) is the same one as in *EmbeddedMontiArc* model (cf. l. 11 in Figure 5.8); because the `EMA-Tag Model` builds on the `EmbeddedMontiArc` grammar (cf. Figure 5.9). Reusing the same concrete syntax makes defining tag models so intuitive.

TagModel

```
1   conforms to EFP1Schema;
2   tags InstancesTags {
3     tag turbineCtrl with powerLimitation =
4     // | v: (0 m/s : 100 m/s) | lambda: (0:10) | cp: (0:1) | beta0: (0:40) |
5        | 13 m/s               | 7.9            | 0.39      | 2            |
6        | 14 m/s               | 7.3            | 0.31      | 5.85         |
7        | 15 m/s               | 6.8            | 0.25      | 9.65         |
8        | 16 m/s               | 6.4            | 0.21      | 13           |
9        | 17 m/s               | 6              | 0.17      | 15.75        |;
10    within turbineCtrl {
11      tag filter.filteredSpeed with encryption = DES;
12      within mainController {
13        tag filteredSpeed with encryption = DES;
14        tag pitchBrake, turbineState with encryption = AES;
15      }
17      tag brCoA with traceable, maxPower = 1W;
18      tag brCoB with traceable, maxPower = 2010mW;
19      tag brCoA.pitchBrake, brCoA.turbineState, brCoB.pitchBrake,
20          brCoB.turbineState with encryption = AES;
21    }
22  }
```

Figure 5.12.: Tag model `InstancesTag` enriching component and port instances with extra-functional properties.

Figure 5.12 shows the `InstancesTags` model to tag the C&C instance structure of extra-functional properties. The tagging mechanism shown in Figure 5.9 enables tagging the C&C model and the derived C&C instance structure. The `InstancesTags` model is conforming to the `EFP1Schema` tag schema. Lines 3 to 9 in Figure 5.12 tag the `turbineCtrl` component instance (this is the main component, cf. l. 34 in Figure 5.8) with a table tag (K5). The concrete syntax of the table tag value is based on the syntax of Markdown, but without marking horizontal lines `|--|--|`. The concrete syntax is not new line sensitive; a single pipe represents a column break and two pipes in a row (cf. end of l. 5 and beginning of l. 6) stand for a line break. C&C instance elements are addressed via their full-qualified name regarding the main component instance. Therefore, all other tags are inside the `within` clause. Line 11 tags the `filteredSpeed` port instance of the `turbineCtrl.filter` component instance with an `encryption` tag. The `encryption` value of a port instance is - in contrast to the port definition - only a single item value (cf. missing star sign after enumeration type in l. 10 in Figure 5.10).

Line 17 tags the brake controller A (`brCoA`) component instance with a maximal power usage of 1 Watt, which is less than the maximal power usage of 2010 Milliwatt of its component type definition, because the component uses at most with `50%` of the available brake force (cf. l. 6 in Figure 5.8) to save energy. Line 18 tags the brake controller B component instance - using the complete available brake force (cf. l. 7 in Figure 5.8) - with the maximal power usage of 2010 Milliwatt.

TagModel

```
1  conforms to EFP1Schema;
2  tags InstancesTags2 {
3    tag turbineCtrl.brCoA with maxPower = 870mW;
4  }
```

TagSchema

```
5  import EFP1Schema.*;
6  tagschema MetaData {
7    tagtype source: [Calculated | Measured | Guessed] for maxPower;
8    // uses simple date format of JDK 8
9    tagtype timestamp: Date("dd.MM.yyyy 'at' HH:mm z") for maxPower, license;
10 }
```

TagModel

```
11 conforms to MetaData;
12 tags InstancesTags2 {
13   within turbineCtrl.brCoA {
14     tag maxPower = 1W with source = Calculated;
15     tag maxPower = 1W with timestamp = "04.05.2017 at 17:56 GMT+01:00";

16     tag maxPower = 870mW with source = Measured;
17     tag maxPower = 870mW with timestamp = "03.09.2018 at 12:23 GMT+02:00";
18   }
19   tag PitchRegulator.license = BSD with timestamp = "01.01.2018 00:00 GMT";
20 }
```

Figure 5.13.: Example how to enrich tags with meta data.

Tagging Meta Data

Figure 5.13 presents an example how tags are tagged again. Lines 1 to 4 define a new tag model conforming to the existing `EFP1Schema` (cf. Figure 5.10). Line 3 tags the `turbineCtrl.brCoA` component instance with `maxPower` again (satisfying **(T6)**). Lines 5 to 10 create the `MetaData` tag schema. This tag schema does not import the class diagrams of *EmbeddedMontiArc*, it imports the class diagram of the `EFP1Schema` to create tag types for the tags of the `EFP1Schema`. Lines 7 and 9 create the `source` and `timestamp` tag type for the `maxPower` tag. The type of the `timestamp` tag is `Date` and the configuration string is the simple date format of JDK 8 [Ora17f].

Lines 11 to 20 enrich the `maxPower` tags and one `license` tag with meta data. Line 13 opens the `turbineCtrl.brCoA` namespace which contains the port instances of the `brCoA` component instance and the tags added to this namespace. Lines 14 and 15 tag the `maxPower` tag, defined in Figure 5.12 (cf. l. 17), with `source` and `timestamp` meta-data (satisfying **(T7)**). The tag name plus its value (e.g., `maxPower = 1W`) identify the tag definition uniquely.

The advantage of tagging tags again, adding meta information to extra-functional properties, is that one meta information scheme (e.g, a company specific one containing `author`, `createdOn`, Boolean `approved`, `approvedBy`, and `approvedOn`) can be reused for different extra-functional properties types. Without this meta information mechanism the company specific fields must be always copied to all extra-functional property types.

Tag Models as Expected Test Result

Besides specifying extra-functional properties, the tag algorithm is very useful for tests. For example, in JUnit tests, tags describe the expected output results of algorithms of C&C models. Due to the nice regex tag kind and that the tagging mechanism works on the concrete syntax of *EmbeddedMontiArc*, the domain experts can specify the result of the algorithm without understanding any Java specific data structures. The algorithm, e.g., the layout algorithm, tags the C&C models or C&C instance structures with intermediate results (e.g., `LayoutSymbol` representing the layout tag) via the Java API of the tagging language. Finally, the JUnit test loads the *EmbeddedMontiArc* model A without tags, calls the algorithm to enrich model A with tags, loads the *EmbeddedMontiArc* model B with enriched expected result tags, and calls `assertEquals` on the calculated tags of model A and the expected result tags of model B. Of course, the models A and B are the same (they have also the same package name, but they are in different model paths) modulo tags.

Implementation projects of *EmbeddedMontiArc* use the tagging-based testing approach in:

- Checking the graphical layout position when generating a SVG graphic from its textual representation;
- Propagating the execution order of component instances (similar to *Simulink*'s `slist` [The18i]); and
- Substituting temporal variables in math expressions by the component's input port names to optimize the control-flow-graph [RSvW^{+}15] for speeding up the execution time.

5.5.4. Derivation of Class Diagrams based on Tag Schemas

Figure 5.14 shows the derived and merged class diagram of the `EFP1Schema` (cf. Figure 5.10) and `MetaData` (cf. Figure 5.13) tag schemas. Line 3 in Figure 5.10 defines the traceable tag type (`tagtype traceable for Component, ComponentInst`); therefore, the `Component` and `ComponentInst` class have an association to the `Traceable` class. The `Traceable` class extends the `Boolean` class; if the traceable marker tag is present for a component or component instance, then the value is true and otherwise it is false.

Line 4 in Figure 5.10 defines the maxPower tag type (`tagtype maxPower:Power for Component, ComponentInst`); thus, the `Component` and `ComponentInst` class have an association to the `MaxPower` class. Since the maxPower tag type is a valued one with type `Power`, the `MaxPower` class extends `NumberPower`, which always has `Power` as quantity, and `NumberPower` extends `Number`. All associations from classes of C&C model or C&C instance structure do not go directly to basic data types, i.e., enumerations, numbers, structures, or Boolean; because we do not want to extend the basic types when adding meta data to tag types. Line 9 in Figure 5.13 defines the timestamp meta tag type (`tagtype timestamp: Date for maxPower` - shortened); hence the `MaxPower` class has an association to the `Timestamp` class representing the timestamp meta tag type. Due to the other source meta tag type (`tagtype source: [...] for maxPower`), the `MaxPower` class has an association to the `Source` class.

Line 6 in Figure 5.10 defines the license tag type (`tagtype license: [GPL | Commercial | Apache3 | BSD2] for ComponentInst`); for this reason, the `Compo-`

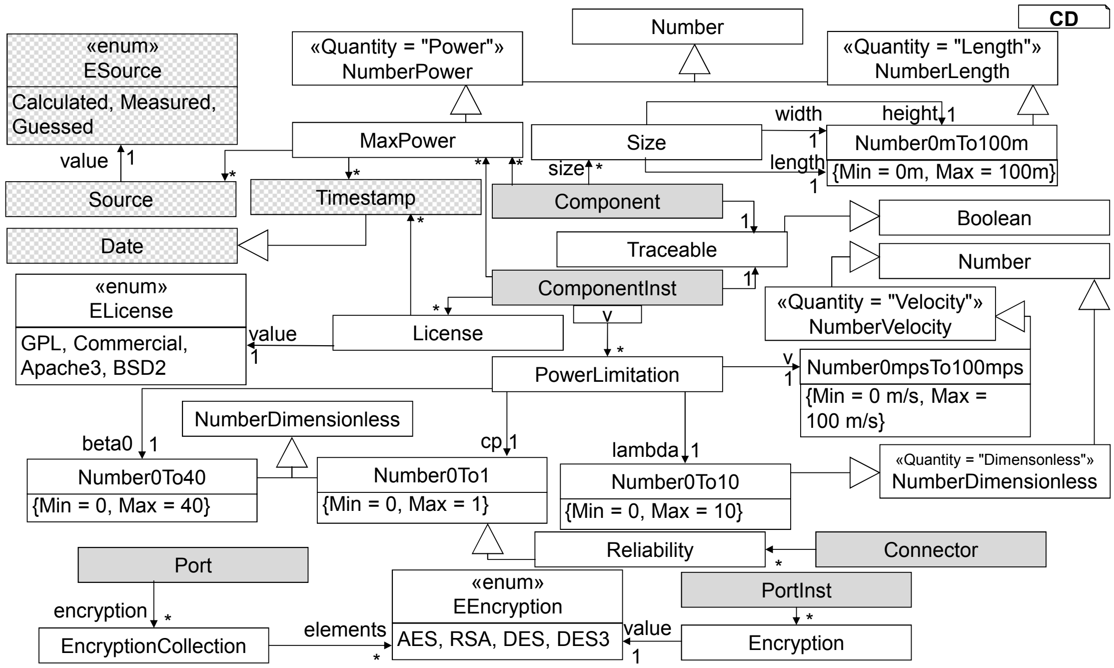


Figure 5.14.: Derived and merged class diagram of both tag schemas (cf. Figure 5.10 for `EFP1Schema` and Figure 5.13 for `MetaData` tag schemas). Classes with solid gray background (i.e., `Component`, `ComponentInst`, `Port`, `PortInst`, and `Connector`) are classes belonging to C&C model or C&C instance structure. Classes with chess background pattern (i.e., `ESource`, `Source`, `Timestamp`, and `Date`) are classes belonging to derived class diagram of `MetaData` tag schema.

`nentInst` class has an association to the `License` class. The `License` class has a `value` association with cardinality 1 to the `ELicense` enumeration class. In contrast to the `MaxPower` class extending the “normal” `Number` class, the “normal” `License` class cannot extend the enumeration class `ELicense`; therefore, the `License` class has the association and no inheritance arrow to `ELicense`. Line 9 in Figure 5.13 adds the timestamp meta type to the license tag type (`tagtype timestamp: Date for license` - shortened); hence, the `License` class has an association to the `Timestamp` class. This timestamp meta tag type also shows why the `ComponentInst` class does not have a direct association to the enumeration `ELicense` class and why the `License` class is not removed, because an enumeration is closed and so no outgoing association to `Timestamp` class can be added later.

Line 8 in Figure 5.10 defines the encryption tag type for the port definition (`tag encryption: [AES |RSA | DES |DES3]* for Port`). Due to the star cardinality of the tag type, the `Port` class has an `encryption` association to the `EncryptionCollection` class, which has zero, one, or many elements of the `EEncryption` class. The cardinality of the association going from `Port` to `EncryptionCollection` is a star one, because a port definition can be tagged multiple times (cf. requirement **(T7)**). The `EncryptionCol-`

`lection` can also define an empty set. Tagging an element with an empty set, e.g., `tag TurbineController.windSpeed with encryption = {}`, has a different semantics than not tagging the element at all. Line 10 in Figure 5.10 creates the encryption tag type for port instances (`tagtype encryption: [AES |RSA | DES | DES3] for PortInst`); therefore, the `PortInst` class has an association to the `Encryption` class which has an association with cardinality one to the `EEncryption` enumeration class - it is similar to the license tag type.

Lines 14 and 15 in Figure 5.10 define the regex size tag type for component types (`tagtype size: { ${length: (0m : 100m)} x ${width: (0m : 100m)} x ${height: (0m : 100m)} } for Component`). Since the size tag type introduces three variables, the `Component` class has an association to the `Size` class having three associations (i.e., `length`, `width`, and `height`) to the `Number0mTo100m` class, which extends the `NumberLength` class having quantity `Length`. `Number0mTo100m` has the two class diagram tags[2] `Min` and `Max` representing the valid range. Based on this tags *OCL* constraints and *FreeMarker* templates are derived to generate user friendly error messages when violating this range (cf. Subsection 6.1.2). For the variables `length`, `width`, and `height` no extra classes are created, because the internal structure of a tag cannot be tagged again. Meta data tags can enrich only the complete size tag resulting in a new outgoing association of the `Size` class.

Lines 17 to 19 in Figure 5.10 define the `powerLimitation` table tag type (`tagtype powerLimitation: | v : (0m/s : 100m/s) | lambda: (0:10) | cp: (0:1) | beta0: (0:40) | for ComponentInst`). The first column `v` is the key column; hence, the class diagram contains a qualified association with `v` as key going from `ComponentInst` class to the `PowerLimitation` class. For each column of the table header, also for the first one, the `PowerLimitation` class has outgoing associations to corresponding number classes, which are `Number0mpsTo100mps`, `Number0To10`, `Number0To1`, and `Number0To40`. All these number classes extend `NumberDimensionless`, as they have the unit `ONE`. The role names of the outgoing associations of `PowerLimitation` map the names of the table headers.

This class diagram in Figure 5.14 is merged with class diagrams presented in Chapter 4. The complete merged diagram (cf. EFP1 $\oplus$ C&C in Figure 5.9) contains a data structure to navigate through all C&C model and C&C instance structure elements as well as all extra-functional properties and their meta information (cf. Figure 5.13). The next chapter uses this merged diagram to formulate semantic based consistency constraints of extra-functional properties via *OCL*; e.g., the encryption mode of a port instance must be contained in the set of encryption modes of its corresponding port definition.

5.5.5. Consistency Rules between Tag Model and Tag Schema

To ensure consistency of tag models, tag schemas as well as between both of them, the following ten context condition rules apply:

Rule 1 Referenced data types used in a tag schema to define new tag types must exist.

[2]cf. `http://mbse.se-rwth.de/book1/index.php?c=chapter2-5`

Rule 2 The scope identifier in a tag schema is either a valid C&C element defined in class diagrams of Chapter 4 or another existing tag type.

Rule 3 The tag schema referenced by a tag model must exist.

Rule 4 Tag type names are unique per C&C model element kind.

Rule 5 Tagged C&C elements or tagged tags exist uniquely and are of the kind defined in the schema.

Rule 6 The tag value in a tag model is of the data type defined in the schema; it is also in the given range; and if it is an enumeration type, the tag value must contain one of the specified enumeration items.

Rule 7 The unit of the tag is compatible with the unit in the schema, e.g., `W` and `mW` but not `W` and `s`.

Rule 8 The tag value of a tag model fits to the specified cardinality: if the cardinality is missing only a single value can be specified; if the cardinality is a + or a * then a set of values must be specified, whereby the + cardinality excludes empty sets.

Rule 9 For complex and regex tags, the above (Rule 1 - Rule 8) applies to every value; for table tags the above applies to every value in a column.

Rule 10 The values in the key column, first column, of a table tag are different.

Rule 4, *Rule 5*, *Rule 6*, *Rule 7*, and *Rule 9* are already published in our paper *Consistent Extra Functional Properties Tagging for Component and Connector Models* [MRRvW16]. Most of these rules can also be mapped to the context conditions (*TD-1* to *TD-9*) defined by Look [Loo17, Subsection 4.3.3]: *Rule 3* maps to *TD-3*; *Rule 4* maps to *TD-4*; *Rule 5* maps to *TD-6*, *TD-7*, and *TD-8*; *Rule 8* maps to *TD-9*.

Chapter 6.

OCL Framework to Describe Structural and Extra-Functional Properties of Component and Connector Models

This chapter presents concrete formalizations of structural and extra-functional property rules. All formalization of this chapter can be processed automatically by tools to analyze these constraints. This section uses the popular and expressive Object Constraint Language (*OCL*) [WK99, WK03, Rum16] to specify these consistency rules for structural and extra-functional properties of component and connector (C&C) models.

The first section shows how to define C&C consistency constraints, also called context conditions, based on formal C&C definitions defined in Chapter 4. Implementation specific details, such as parsing workflows, abstract syntax trees, and symbol table information, are abstracted by providing C&C specific class diagrams (cf. Section 4.2, and Section 4.3), and powerful type-inference mechanisms when checking consistency. Six complete *OCL* examples illustrate how easily context conditions can be defined with this *OCL* framework. Additionally, this section elucidates how to define user-friendly error messages for violated *OCL* constraints via *FreeMarker* templates. Still it needs to be make clear that these *OCL* constraints generate on the sentence of the models, and thus, are used and evaluated during design time. These *OCL* conditions are designed by tool engineers, not the product developers.

The previous chapter introduced a tagging mechanism to enrich C&C models with extra-functional properties. The tagging mechanism is general and can be reused for nearly all extra-functional properties. The second section of this chapter explains how consistency constraints for measurable extra-functional properties are defined via *OCL*. The restriction to measureable properties is caused due to the fact that properties such as maintainability or user-friendliness are too imprecise to be formalized with *OCL*. The second section extends *OCL* with support for units, as many extra-functional properties describing physical properties of C&C models contain physical units. Thus, to enable domain experts define extra-functional consistency rules, a constraint language for them naturally should support automatic unit comparison and conversion. The second section illustrates the *OCL* framework for extra-functional properties on twelve examples, whereby three of these examples are consistency rules involving more than one extra-functional property.

The third section explains how witnesses of *OCL* constraints are generated. The positive and negative witness generation process uses the mathematical structure of *OCL* constraints for extra-functional properties. These witnesses intuitively demonstrate the reasons for consistency or inconsistency of a C&C model enriched with many different extra-functional properties.

The fourth section elucidates how to specify transformations between the abstract syntax of two different languages via *OCL*. This section explains this concept on transformations from the abstract syntax of *EmbeddedMontiArcParsing* to the one of *EmbeddedMontiArcTooling* (cf. Section 4.1): it presents *OCL* code snippets to transform name-based connections in the syntactic sugar version of *EmbeddedMontiArcParsing* to connections with specified port names in *EmbeddedMontiArcTooling*. The name-based connection `sub1.* -> sub2.*` is syntactic sugar for `sub1.portA -> sub2.portA`, `sub1.portB -> sub2.portB` and so on; it connects the port of `sub1` with the port of `sub2` if they have the same port name - Subsection 3.6.6 contains a more detailed example.

The fifth section gives some very short remarks about the implementation of the *OCL* language and the *OCL* to Java generator. The last section compares this *OCL* framework with related approaches existing in literature.

6.1. *OCL* Framework to Define Context Conditions of C&C Models

This section shows (the workflow) how to formulate structural constraints on C&C models. As an example, this section presents several *OCL* constraints for defining well-formedness rules (also called context conditions) of C&C models. Context conditions constraint the abstract syntax defined by *MontiCore*'s context-free EBNF-like grammar rules.

In this section, the context conditions and their identifiers are the same as the ones defined in Haber [Hab16] to enable easier tracings between these two theses. This section shows only a selection of Haber's context conditions, which are valid for both languages, i.e., *EmbeddedMontiArc* and Haber's *MontiArc*, and which do not address the resolvability of symbol names. In the current *MontiCore* version the symbol management infrastructure [MSN17] handles all the resolving constraints and throws suitable error messages.

6.1.1. Workflow to Define and Validate *OCL* Context Conditions

This subsection introduces the artifacts, generators and user roles being involved in the *OCL* verification process. Figure 6.1 shows the design and run time of the *EMA Validator*. The *EMA Validator* receives at run time a textual *EmbeddedMontiArc* model as input and it produces a Boolean flag whether the model is valid (i.e, the model satisfies all context conditions) and (possible empty) error messages as output (cf. right part).

The development (i.e, the design time) of the *EMA Validator* leverages a complete model-based approach. The internal representation of *EmbeddedMontiArc* (i.e, the two class diagrams presented in Section 4.2 and Section 4.3) is specified via three *MontiCore* grammars (cf. Section 4.1). The context conditions are defined as *OCL* constraints and the error messages (if the *OCL* constraint is violated) via *FreeMarker* text templates.

The *MontiCore* grammar generator produces Java classes for the abstract syntax based on the three *EmbeddedMontiArc* grammars. *EmbeddedMontiArc*'s internal structure describes the

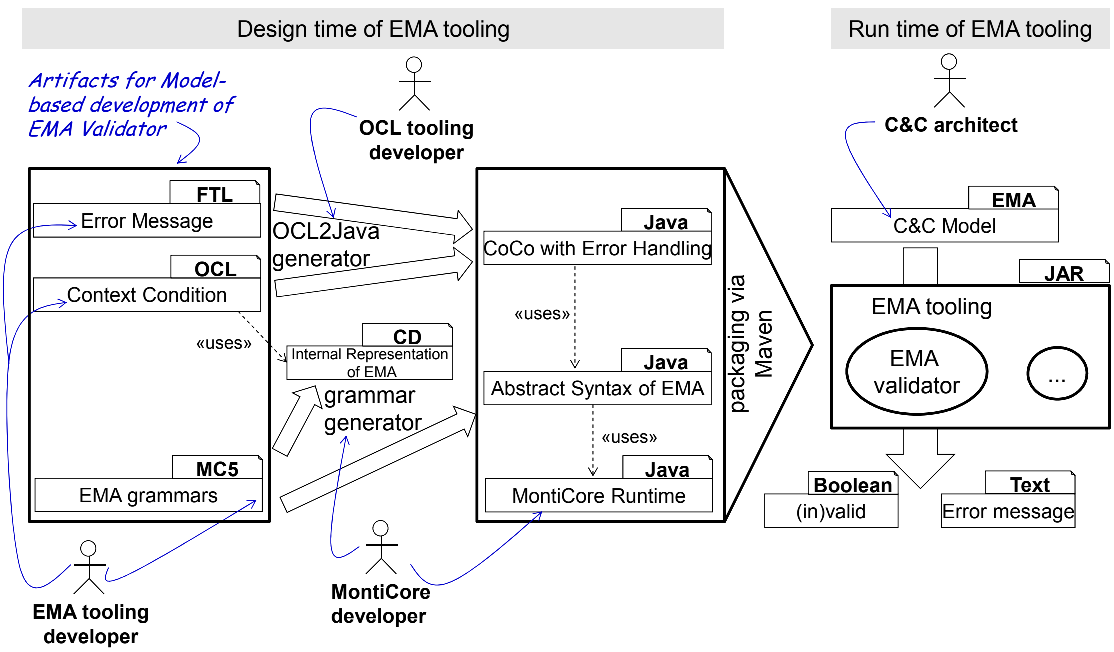


Figure 6.1.: Workflow, artifacts, and user roles of *OCL* Framework.

structural relationships between these generated Java artifacts as class diagrams; this way the *OCL* language can verify that *EmbeddedMontiArc*'s context condition are valid according to *EmbeddedMontiArc*'s abstract syntax Java files. The verification of *OCL* constraints against the class diagram representation of *EmbeddedMontiArc* is needed to avoid ugly Java compiler error messages. Because without checking the conformance of *OCL* against the abstract syntax, Java files generated by the *OCL2Java* generator may not be compatible to Java files generated by *MontiCore*'s grammar generator.

All generated Java files plus the *MontiCore* runtime environment are compiled and packaged to one *EMA Validator* JAR file. This JAR file has a command-line interface to specify input and output parameter options to validate C&C models programmatically.

6.1.2. CO1: Connectors May Not Pierce Through Component Interfaces

Haber defines this context condition as following: "Qualified sources and targets of a connector consist of two parts. The first part is a name of a subcomponent, the second part is a port name." [Hab16, p. 61]. But since this rule is specific to *MontiArc*'s concrete syntax, there exist several exceptions due to syntactic sugars in *MontiArc*. The next two rules are some of these exceptions: Haber supports writing "`connect msgIn -> af.msgs`" [Hab16, Listing 3.33 (line 8)] and "`[filteredMsgs -> bf.msgs]`" [Hab16, Listing 3.34 (line 6)].

Instead of defining several context conditions each possibly having exceptions on the *EmbeddedMontiArc* or *MontiArc* syntax, we suggest to define the context condition directly on its

EMA

```
1  component Outer {
2    ports in  (0:10) outerIn1,
3              (0:10) outerIn2,
4          out (0:10) outerOut1,
5              (0:10) outerOut2,
6              (0:10) outerOut3;
7    instances Filter inner1,
8                     inner2;

9    connect outerIn2 ->
10                    outerOut2;
11   connect outerIn1 ->
12                   inner1.in1;
13   connect inner1.out1 ->
14                   inner2.in1;
15   connect inner2.out1 ->
16                    outerOut1;

17   // invalid connection
18   connect inner2.in1 ->
19                    outerOut3;
20 }
```

EMA

```
21 component Filter {
22   ports in  (0:10) in1,
23         out (0:10) out1;
24 }
```

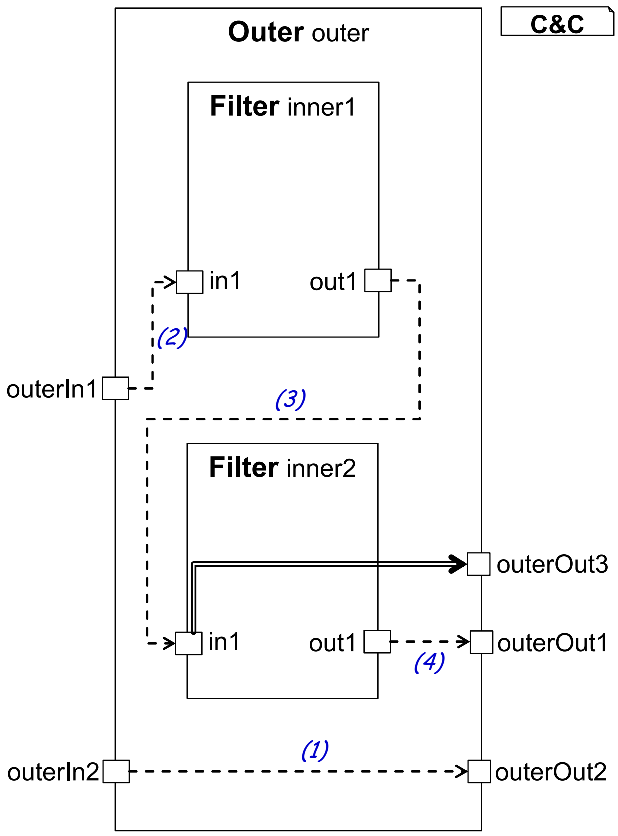


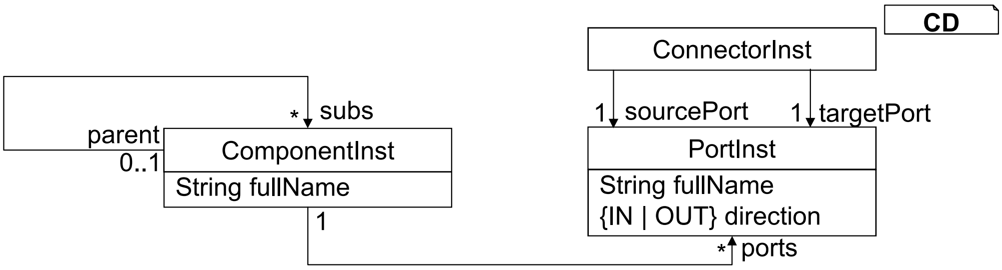
Figure 6.2.: Example *EmbeddedMontiArc* model for context condition *Connectors may not pierce through component interfaces*. The four green connections are valid, the red one is invalid.

underlying mathematical framework for component and connector models as defined in Chapter 4. The advantage is not to deal with many exceptions due to syntactical sugars.

Figure 6.2 shows an example containing all four cases of valid connections:

(1) Ports of the same component instance with different directions can be connected as shown in lines 9 and 10; `connect outerOut2 -> outerIn2` would also be possible.
(2) Source and target ports are input ports, and the component of the target port is a subcomponent of the source port's component as shown in lines 11 and 12.
(3) Source port is an output port, target port is an input one; and the components of both ports are different, but they have a common parent component - cf. ll. 13 and 14.
(4) Switched case of (2): Source and target ports are output ports, and the component of the source port is a subcomponent of the target port's component as shown in lines 15 and 16.

Figure 6.3 shows the *OCL* constraint for this context condition and the needed class diagram parts defined in Chapter 4. Line 1 says that for all connector instance objects the following invariant holds [Rum16, Fig. 3.1]. The prefix `context ConnectorInst` in line 1 is equivalent to `forall ConnectorInst: ...`, which means that the lines 2 to 11 must hold for every



OCL

```
1  context ConnectorInst inv CO1:
2    let srcCI = sourcePort.componentInst;
3        tgtCI = targetPort.componentInst;
4        srcD = sourcePort.direction;
5        tgtD = targetPort.direction;
6    in
7      srcCI == tgtCI && srcD != tgtD   || // in -> out, out -> in (loop)
8      tgtCI.parent == srcCI && srcD == IN && tgtD == OUT || // in -> sub.in
9      srcCI.parent == tgtCI.parent && tgtCI != srcCI &&
10                    srcD == OUT && tgtD == IN || // sub1.in -> sub2.in
11     srcCI.parent == tgtCI && srcD == OUT && tgtD == OUT // sub.out -> out
```

Figure 6.3.: *OCL* constraint for context condition CO1: *Connectors may not pierce through component interfaces.*

connector instance object at each observed point in time. The first part of the `let-in` construct in lines 2 to 5 defines auxiliary variables [Rum16, Subsection 3.1.2] which are used in the second part, the actual invariant constraint, in lines 7 to 11. Line 7 maps to case (1); line 8 maps to case (2); lines 9 and 10 map to case (3); and line 11 maps to case (4). Only eleven lines of *OCL* code define this context condition mathematically.

If the constraint fails, the *OCL* to Java generator returns a Java object structure with the values of the objects causing the constraint to fail. Figure 6.4 shows the object diagram representing the negative witness structure of the connector instance `connect inner2.in1 -> outerOut3`. The variable names in the object diagram are the ones used in the *OCL* constraint, i.e., the `conInst` name inside the `context` clause and two names inside the `let-in` clause.

The top part of Figure 6.5 illustrates an error template for the context condition of Figure 6.3. The bottom part of Figure 6.5 displays how the error message looks like for the example shown in Figure 6.2. This example shows that it is possible to specify context conditions of *EmbeddedMontiArc* via *OCL* constraints plus *FreeMarker* templates. The domain expert defining this context condition needs no knowledge about the underlying implementation of *EmbeddedMontiArc*. The expert only needs to understand the class diagrams introduced in Chapter 4, and have basic knowledge about *OCL* and *FreeMarker*. If the four conditions in Figure 6.3 would be separated into four *OCL* constraints, then even more accurate error messages are created - this thesis skips these four single constraints to avoid too much repeating content.

The *CoCo* language links to the *OCL* condition, and contains the *FreeMarker* template, the warning level, plus the *OCL* variables which corresponding text should be underlined in an IDE (cf. *xText* context conditions). In this example, the `conInst` variable should be underlined.

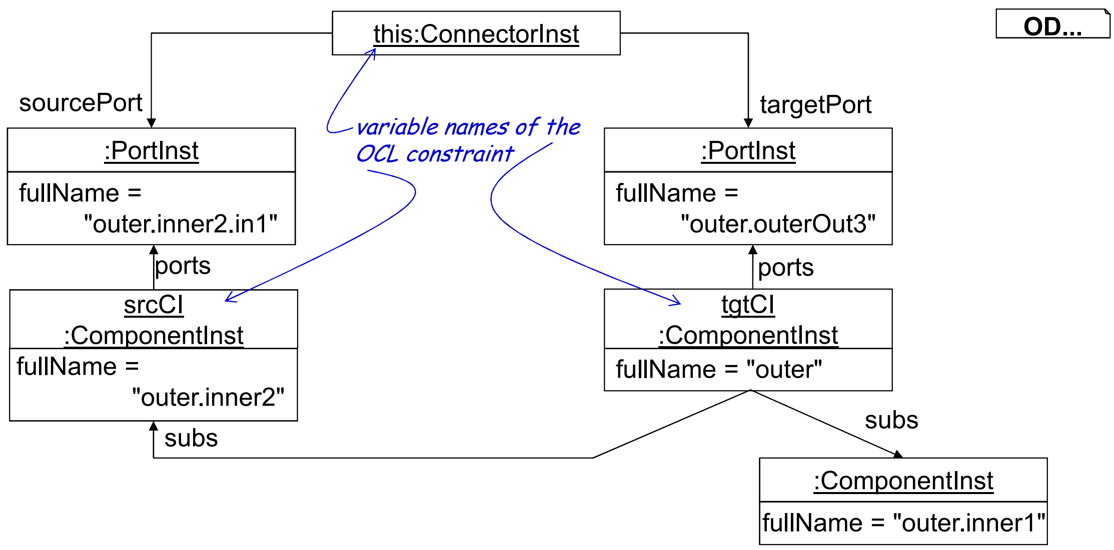


Figure 6.4.: Excerpt of generated witness structure of OCL2Java generator illustrated as object diagram. The implementation links to Java objects of the symbol management infrastructure; the structure of the Java objects is the same as the one presented in this object diagram.

Freemarker

```
1 The connector from port "${this.sourcePort.fullName}" to port
2 "${this.targetPort.fullName}" of the two components "${srcCI.fullName}" and
3 "${tgtCI.fullName}" pierces through a component interface.
```

Text

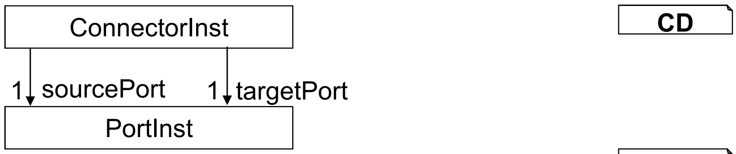
```
4 The connector from port "outer.inner2.in1" to port "outer.outerOut3" of the
5 two components "outer.inner2" and "outer" pierces through a component
6 interface.
```

Figure 6.5.: *FreeMarker* text (cf. ll. 1-4) to produce nice error message (cf. ll. 5-7) when the constraint is failing.

Backtracking from `conInst` to the its defining symbol of the type `Connector`, and then, form this symbol back to the abstract syntax tree containing the start (i.e., line 18, column 2 in `Outer.ema`) and the end (i.e., line 19, column 27 in `Outer.ema`) source position enables highlighting the text causing this error.

6.1.3. R1: Each Outgoing Port of a Component Type Definition Is Used At Most Once As Target Of a Connector / R2: Each Incoming Port Of a Subcomponent Is Used At Most Once As Target Of a Connector

Haber states “every receiving port only receives signals from a unique sender, while a sender can transmit its data to more than one receiver” [Hab16, p. 62], thus, Haber wants to check that



OCL

```
context ConnectorInst conInst1, conInst2 inv R1R2:
  conInst1.targetPort == conInst2.targetPort implies
    conInst1.sourcePort == conInst2.sourcePort
```

Figure 6.6.: *OCL* constraint for context condition R1: *Each outgoing port of a component type definition is used at most once as target of a connector.* and R2: *Each incoming port of a subcomponent is used at most once as target of a connector.*

different ports are not connected to the same target port. Due to the fact that Haber implemented the context conditions for *MontiArc* on the objects of abstract syntax tree which are directly derived from the concrete syntax, this context condition needs to differentiate between two different use cases.

Figure 6.6 shows the simple context condition when defining it on the abstract syntax graph of the C&C instance structure. This *OCL* definition could also be defined on connector definitions, because the target port and source port of a connector definition is a component instantiation (and no component type). An example why the target port must be a component instantiation is: `component X { ports in Z in1, Z in2; instances A a1, a2; connect in1 -> a1.in1; connect in2 -> a2.in1; }`. The target port is the component instantiation `a1.in1` and `a2.in1` which are different; but the port definition of both target ports is `A.in1`, the port definition of the component type `A`, cf. Subsection 4.2.3f. for further details.

This example illustrated why it is very important that the abstract syntax matches the essence of a language and it is not only the basic abstract syntax tree. Well-designed class diagrams, as the one in Chapter 4, present only the mathematical essence of the abstract syntax.

6.1.4. R13: Subcomponent Instantiation Cycles in Component Type Definitions Are Forbidden

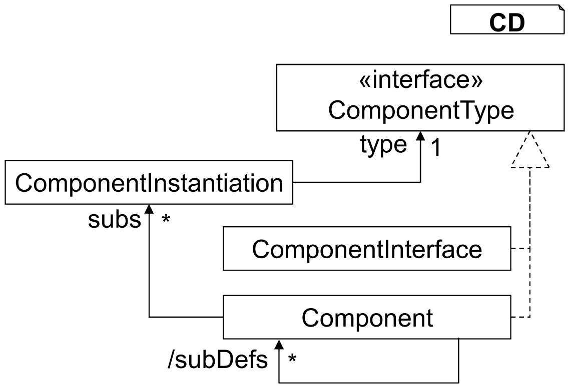
Lines 1 to 9 in Figure 6.7 show an example of a component type cycle via subcomponent instantiations. Lines 10 and 11 in Figure 6.7 define the derived self-association `subDefs` of `Component`. SubDefs contains the component types of the direct subcomponent instantiations. Line 12 and 13 in Figure 6.7 define the context condition, that no component type is part of the transitive closure of its own subcomponent types. The two stars represent the transitive closure operator [Rum16, Subsection 3.5.1].

A transitive closure on a binary relation $R \subseteq X \times X$ is the smallest relation on X containing R and being transitive: $(a,b) \in X \wedge (b,c) \in X \Rightarrow (a,c) \in X$. The self-association `subDefs` is a binary association of $Component \times Component$.

EMA
```
component A {
  instance B b1;
}
```
*A.subDefs** = {B, C, A}*

EMA
```
component B {
  instance C c1;
}
```

EMA
```
component C {
  instance A a1;
}
```



OCL
```
context Component inv:
  subDefs == {t | s in subs, t = s.type, t instanceof Component}
```

OCL
```
context Component inv R13:
  !(this isin subDefs**)
```
transitive closure

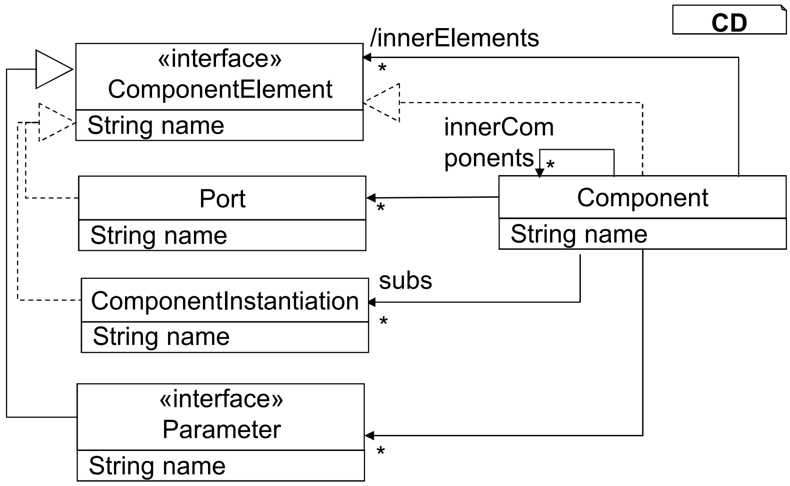
Figure 6.7.: *OCL* constraint for context condition R13: *Subcomponent instantiation cycles in component type definitions are forbidden*, and a simple example violating this constraint.

6.1.5. B1: All Names Of Model Elements Within a Component Namespace Have To Be Unique

Lines 1 to 5 in Figure 6.8 show a simple example violating the context condition B1, because the input port in line 2 has the same name as the subcomponent instantiation in line 4. Even though this thesis does not treat cases with inner component definitions, the *OCL* constraint handles it (cf. l. 8) in order to present the complete constraint. To have an elegant constraint of only three lines (cf. ll. 9-11), the `ComponentElement` interface has been added.

Figure 6.9 presents optimized Java code for this context condition. First, the Java code is not shorter than the *OCL* one. Second, to implement this context condition in Java you need to be familiar with the implementation details of the symbol management infrastructure of Nazari [MSN17]. The "stupid" *OCL* generator produces a nested `for` loop (cf. l. 10 in Figure 6.8) to iterate over the `innerElements` collections. Therefore, the Java code generated by our "stupid" *OCL* generator is much slower, with a run-time complexity of $\mathcal{O}(n^2)$ operations (n are the number of `innerElements`), than the optimized handwritten Java code, where the sorting algorithm with a complexity of $\mathcal{O}(n \cdot \log(n))$ is most the computing-intensive task. Since such element-wise comparisons often occur, the *OCL* generator could be extended to match this pattern and to produce an optimized Java code.

EMA
```
component X {
  port in C c;
  instance
    Controller c;
}
```



OCL
```
context Component inv:
  innerElements ==
    ports.addAll(parameters).addAll(subs).addAll(innerComponents)
```

OCL
```
context Component inv B1:
  forall e1, e2 in innerElements:
    e1.name == e2.name implies e1 == e2
```

Figure 6.8.: *OCL* constraint for context condition B1: *All names of model elements within a component namespace have to be unique*, and a simple example violating this constraint.

Java
```java
Collection<Symbol> symbols =
            componentSymbol.getSpannedScope().resolveDownMany(Symbol.KIND);
symbols = Collections.sort(symbols, Symbol::getName);
for (int j = 1; j < symbols.size(); j++) {
  if (symbols.get(j-1).getName().equals(symbols.get(j))
    Log.error(symbols.get(j).getName() + " is duplicated");
}
```

Figure 6.9.: Java code of context condition B1: *All names of model elements within a component namespace have to be unique*.

6.1.6. CV5: In Decomposed Components, All Ports Should Be Used In At Least One Connector / CV6: All Ports Of Subcomponents Should Be Used In At Least One Connector

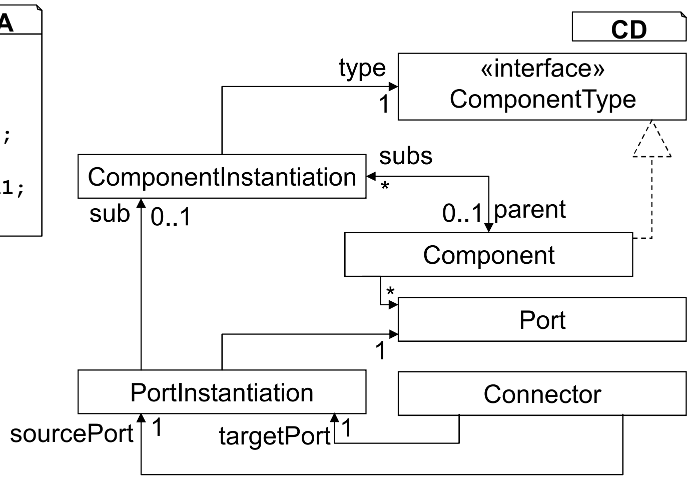
Both, CV5 and CV6, context conditions to connect all ports mean actually that all ports should be connected unless a component is atomic and has no parent one, i.e., if the C&C model has

EMA

```
1  component A {
2    ports in B b1,
3             B b2;
4    instance Delay<B> d1, d2;
5    connect b1 -> d1.in1;
6    connect d1.out1 -> d2.in1;
7  }
```

EMA

```
8  component Delay<T> {
9    ports in  T in1,
10         out T out1;
11 }
```



OCL

```
12 context Component cmp inv CV5:
13   cmp.subs != {} || (exists ComponentInstantiation ci: ci.parent == cmp)
14   implies
15     forall p in cmp.ports:
16       exists con in Connector:
17         (p == con.sourcePort.port || p == con.targetPort.port)
```

OCL

```
18 context Component cmp inv CV6:
19   forall ci in {ComponentInstantiation ci | ci.type == cmp}, p in cmp.ports:
20   exists con in Connector:
21     (p == con.sourcePort.port && ci == con.sourcePort.sub ||
22      p == con.targetPort.port && ci == con.targetPort.sub)
```

Figure 6.10.: *OCL* constraint for context condition CV5: *In decomposed components, all ports should be used in at least one connector* and CV6: *All ports of subcomponents should be used in at least one connector*, plus a simple example violating both conditions.

exactly one atomic component. Figure 6.10 presents the *OCL* constraint combining both context conditions as well as it illustrates an example violating CV5 and CV6.

It would be much easier to formulate this context condition on the C&C instance structure as shown in Subsection 6.1.2. The complex *OCL* constraints just illustrate how to deal with connectors in the C&C model.

Line 3 in Figure 6.10 shows a warning, because output port `b2` is not used. Line 10 shows a warning because the output port `out1` of the component instantiation `d2` is not used. However, the output port is used in the component instantiation `d1`. Therefore, it is not enough to check only if the port is used once, the *OCL* constraint must check whether all ports of all component instantiations of a given component type (cf. l. 18; in this example, the component type is `Delay`) are used.

EMA ...

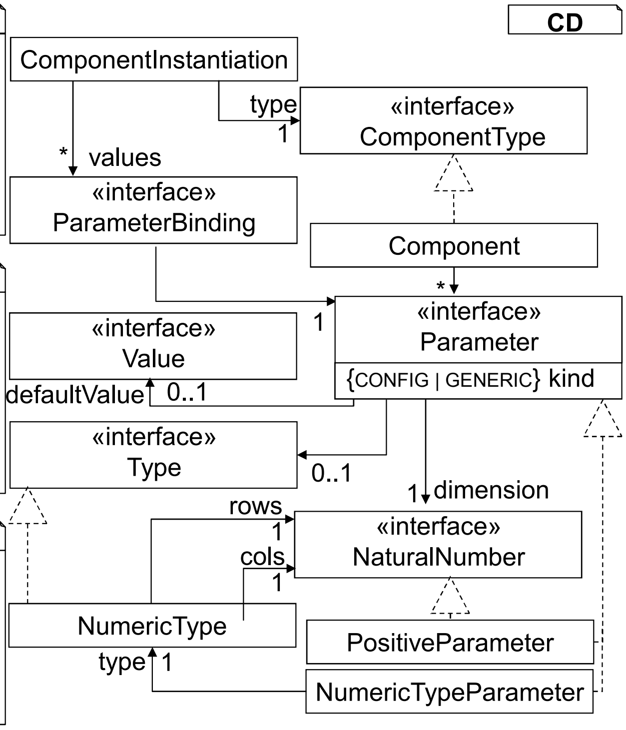
```
1  component Convolution
2     <T is Dimensionless,
3      (1:2:3) dim = 1, (3:oo) n>
4     (symmetric Q^{n, n}
5      kernel[dim]) {
6    ports in   T imageIn [dim],
7           out T imageOut[dim]; }
```

Main.txt

```
8  // valid, uses default value
9  // 1 for dim and n is derived
10 Main-Component-Instantiation:
11  Convolution
12  <(0 : 2^24)^{1920, 1080}>
13  ( [0, -1, 0; -1, 5, -1;
14     0, -1, 0] )       fullHD;
```

Main.txt

```
15 // invalid: configuration
16 //parameter Q is no specified
17 Main-Component-Instantiation:
18  Convolution
19  <(0 : 2^24)^{1920, 1080},
20    1,3> fullHD;
```



OCL

```
21 context ComponentInstantiation inv R9R10:
22   forall p in type.parameters:
23     !(exists p2 in type.parameters: // p can be inferred due to bounded
24         p2.kind == CONFIG &&        // configuration parameter p2
25         (p2.dimension == p || p2.type.rows == p || p2.type.cols == p)
26     ) &&
27     (p.defaultValue.isAbsent implies // no default value
28       exists pb in values:
29         pb.parameter == p)
```

Figure 6.11.: Positive and negative example plus *OCL* constraint for context conditions R9/R10: *If a component type is instantiated as a subcomponent, all generic and all configuration parameters have to be assigned.*

6.1.7. R9/R10: If a Component Type Is Instantiated As a Subcomponent, All Generic And All Configuration Parameters Have To Be Assigned

Figure 6.11 shows an example and the *OCL* code for the context condition R9/R10. The first listing in line 1 to 7 defines a `Convolution` component having three generic and one configuration parameters. The first parameter `T` defines a generic port type which quantity is dimensionless. The second parameter `dim` accepts the values 1 and 3, `(1:2:3)` represents the numeric data type starting at 1 and ending at 3 with a step-size of 2. The third parameter `n` accepts values greater equals 3; the `oo` symbol represents the plus infinity symbol meaning that `n` has no upper-bound. The last parameter `kernel` has as data type a symmetric $n \times n$ matrix of rational numbers (`Q^{n, n}` represents $\mathbb{Q}^{n \times n}$), the parameter `kernel` is a parameter array similar to a port or a component instantiation array (cf. Subsection 3.6.2).

The second listing instantiates the `Convolution` component definition. It binds the parameters `T` to $\left[0, 2^{24}\right]^{1\,920\times1\,080}$, `dim` to 1 (default value), `n` to 3 (automatically derived) and `kernel` to `[0, -1, 0; ... ]` being a 3×3 matrix.

The example in lines 17 to 20 instantiates the `Convolution` component definition with the following parameters: `T` to $\left[0, 2^{24}\right]^{1\,920\times1\,080}$, `dim` to 1, `n` to 3. However, it does not bind the configuration parameter `kernel` (cf. l. 5) of the component type definition; thus, this example is invalid.

Lines 21 to 29 present the constraint for this context condition. For *MontiArc* the *OCL* constraint would only consists of the lines 21, 22, 28, and 29. However, *EmbeddedMontiArc*'s parameter definitions may have default parameters, and these parameters may not be bounded (cf. l. 27). Additionally, *EmbeddedMontiArc* does not force to bind generic parameters whose values can be derived by mandatory configuration parameters (cf. ll. 23-25) as it is the case in our example: the component instantiation in lines 10 to 14 does not bound the parameter `n`, but `n` is automatically bound to 3 based on the passed configuration parameter `[0, -1, 0; ... ]` being a 3×3 matrix.

The expression in line 25 `p2.type.rows` automatically returns `false` when an error occurs (cf. [Rum16]), because every single *OCL* expression such as `p2.type` is executed in a Java `try-catch` block. However, the `type` association of a `Parameter` is the abstract interface `Type`, and this general interface does not have the associations `rows` and `cols`. To have shorter *OCL* expressions without many `typeif-instanceof` case distinctions, we extended the *OCL* generator, so that interfaces can navigate to an association when at least one class implementing this interface contains this association. In this case, the *OCL* generator produces automatic `typeif-instanceof` case distinctions to access the association of the classes implementing this interface and having this association. If the object of a class does not have this association the Boolean expression `p2.type.rows == p` is automatically evaluated to `false`. So `p2.type.rows == p` is a short-form of `typeif p2.type instanceof NumericType then p2.type.rows == p else false`.

In the case multiple subclasses of `Type` have the `rows` association, than a *OCL* context condition checks that target types of the associations of these subclasses have a common target type in the underlaying class diagram.

Figure 6.12 illustrates exemplary what Java code the *OCL* generator produces for the *OCL* expression `p2.type.rows == p`. This code is only pseudo code to present the general concept how *OCL* deals with associations of interfaces. As mentioned above, every sub expression which evaluates to Boolean, e.g., `p2.kind == config`, or `p2.dimension == p`, is surrounded by a `try-catch` block as shown in lines 6 to 18 and 21. The reason is, if any of these sub expressions throw an error, the complete logical expression with `!`, `||`, or `&&` can still be evaluated, and in some cases (e.g., concatenation of sub expressions) the failed sub expressions does not matter for the combined result.

Since the `Type` interface does not have the `rows` attribute, but one class implementing the `Type` interface has this attribute; the generator produces for this class the `if-instanceof` block as shown in lines 10 to 14. If the `p2_type` object belongs to a class not having this attribute, the code evaluates the Boolean sub expression to `false` as shown in line 17.

Java Pseudo Code

```
1  // p2.type.rows == p with p and p2 instance of Parameter interface
2  ParameterSymbol p;
3  ParameterSymbol p2;
4  boolean p2_type_rows__equals__p = false;
5  try {
6    TypeSymbol p2_type = p2.getType();
7    // Type interface does not have rows association, but the following
8    // classes, implementing Type interface, have rows association:
9    // NumericType
10   if (p2_type instanceof NumericTypeSymbol) {
11     NaturalNumberSymbol p2_type_rows =
12                                ((NumericTypeSymbol)p2_type).getRows();
13     p2_type_rows__equals__p = Objects.equals(p2_type_rows, p);
14   }
15   else {
16     // the object p does not belong to any class having the rows association
17     p2_type_rows__equals__p = false;
18   }
19 }
20 catch (Exception e) {
21  p2_type_rows__equals__p = false; // error evaluates to false
22 }
```

Figure 6.12.: Generated Java Pseudo code of the *OCL* expression `p2.type.rows == p` (cf. l. 25 in Figure 6.11).

The code in lines 11 to 13 is only added, because there exists at least one class (i.e., `PositiveParameter` class) implementing the `NaturalNumber` (left side of equals sign) as well as the `Parameter` (right side of equals sign) interface[1]. The Java code must cast `p2_type` to `NumericTypeSymbol`, otherwise it would result in a compilation error of the generated Java code. Line 13 uses `Objects.equals` as this method is more robust, because it also works for `null` references.

The here presented extension of the *OCL* generator is 100% compatible with the *OCL* semantics defined by Rumpe [Rum16], because the navigation of an association of an interface, which is only available by their subclasses and not by the interface itself, can be extended to longer *OCL* code using several of the safe `typeif - instance - then- else` rules (cf. Subsection *Conditional Expressions* [Rum16, SubSection 3.1.3]). This extension together with the automatic flattening (cf. [Rum16] for more details) operators, when navigating along two or more star or optional associations of *OCL*, are the main reason why context conditions formulated in *OCL* are much more compact than using any general purpose programming language such as Java, or C++.

[1] If such a class implementing both interfaces would not exist, the left and right hand side of the equals expression cannot be the same, and therefore, the generator would also evaluate the complete Boolean sub expression to `false`.

OCL

```
context ComponentInst inv Traceability:
  let
    selection = this;
    aggregation = component.traceable;
    compareTo   = traceable;
  in          // comparison
    aggregation implies compareTo
```

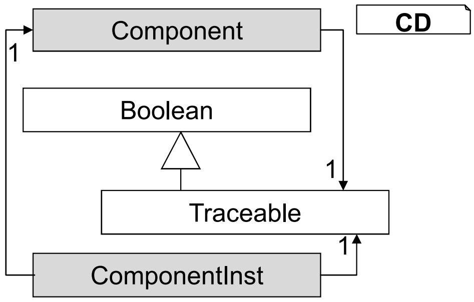


Figure 6.13.: *OCL* code for traceability consistency rule.

6.2. Defining Extra-Functional Properties in *OCL*

The previous section has shown how to define context conditions for *EmbeddedMontiArc* in *OCL*. This section presents how to define consistency rules of extra-functional properties in *OCL*. While these also apply during design time, it may be that product developers define and use these constraints themselves or even the tag designers.

All consistency rules presented in this section follow the selection, aggregation, and comparison structure introduced in our consistent extra-functional property tagging paper [MRRvW16]. This section uses twelve illustrative examples to elucidate the *OCL* framework. The *OCL* consistency rules are defined on the merged class diagram presented in Figure 5.14 on page 123. This merged class diagram combines the class diagrams of *EmbeddedMontiArc* and the generated class diagrams of the tag schema defining the concrete and abstract syntax of new extra-functional property types. For better readability, all *OCL* constraints skip the same import statement including all classes of the merged class diagram.

The first four rules are instantiation consistency examples. Instantiation consistency checks whether the extra-functional property values of C&C instances are conforming to the extra-functional property values of their definitions of the C&C model (cf. Figure 4.17 on page 123 for relations between C&C model and C&C instance structure in the abstract syntax) [MRRvW16].

The next eight rules are composition consistency rules. Composition consistency checks extra-functional properties across their composition [MRRvW16]. Most rules address consistency on type and/or instance level.

Most of the consistency rules are published in one of our two extra-functional properties papers [MRRvW16, MMR+17]. However, the *OCL* constraints in this thesis differ from the ones presented in the papers, because the steady improvement of the *OCL* generator (esp., its type inference algorithms) enables to shorten constraints.

6.2.1. Traceability for Component Instantiation

Consistency Rule: If the component type definition is traceable, all instances have to be traceable.

Figure 6.13 shows (except of the one import statement) the complete *OCL* code to formulate this consistency rule. All the consistency rules must define multiple `selection`, one `ag-`

OCL

```
1  context ComponentInst inv MaxPower:
2    let
3      selection = component;
4      aggregation =
5        max maxPower ?: 0W;
6      compareTo =
7        min selection.maxPower ?: oo W;
8    in           // comparison
9      aggregation <= compareTo
```

max/min: $\mathbb{T}^n \rightarrow \mathbb{T}$*; max/min:* $\emptyset \rightarrow \bot$

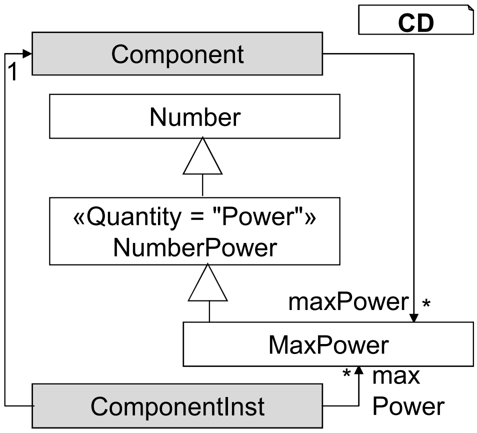


Figure 6.14.: *OCL* code for maximal power consumption consistency rule.

gregation, and one compareTo variables. These variables are used to generate positive consistency and negative inconsistency witnesses; Section 6.3 explains this in detail. Line 3 selects the current component instance. The aggregation value is true when the component definition is tagged with traceability (cf. BrakeCtrl component definition in Figure 5.7 on page 155 and l. 13 in Figure 5.8 on page 156). Line 5 is a short-form for compareTo = this.traceable. Line 7 is the comparison part: it says when the corresponding component type is tagged with traceable, then this component instance must also be tagged with traceable.

Please note that the other way around is not forced, meaning that the component instance can be marked as traceable and the component type definition of this instance is not marked as traceable.

The TurbineController C&C model in Figure 5.7 satisfies this constraint, because only the BrakeCtrl component definition is marked as traceable and both instances, brCoA and brCoB, are also marked with traceable.

6.2.2. Maximal Power Consumption for Component Instantiation

Consistency Rule: The maximal power consumption of an instance is at most the maximal power consumption of its type.

Figure 6.14 shows the complete *OCL* code (except of the import statement) for component instantiation consistency rule of maximal power consumption. Lines 3 to 7 define the mandatory variables needed for the witness generation.

The *OCL* language of Rumpe [Rum16] has been extended with many new set expressions. For example, the set expression max has as input variable any set of numbers and it returns the largest number. However, if the set is empty, max returns {} for not present. The empty set and the not present optional value use the same concrete syntax in the *OCL* language. This is wanted, because *OCL* treats everything as set: 0..1 cardinality is a set with zero or one element; 1 cardinality is a set with one element; and * is a set with zero, one, or more elements. The

OCL

```
1 context PortInst inv Encryption:
2   let
3     selection = port;
4     aggregation =
5       selection.encryption.elements;
6     compareTo = encryption.value;
7   in          // comparison
8     encryption.value.size <= 1    &&
9     aggregation.containsAll(compareTo)
```

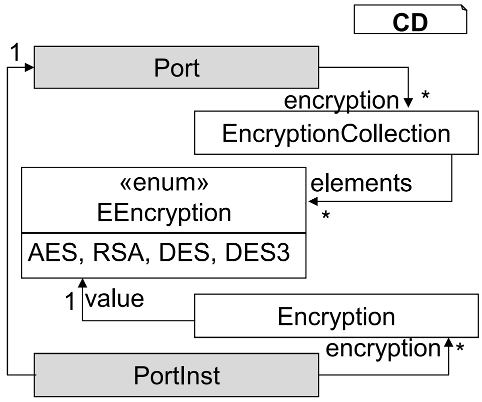


Figure 6.15.: *OCL* code for encryption consistency rule.

advantage of treating everything as a set is that the set flattening operators, defined by Rumpe [Rum16], can also be applied to optional and "normal" data types.

The Elvis operator `?:` is borrowed from Kotlin [Lei17] and `x ?: 0W` in line 5 is equal to `x.isPresent ? x : 0W`[2]; it is the same as `x.orElse(0 W)` in Java. Line 5 means: (a) if the component instance is not tagged with any `maxPower` value, then `0W` as default value is used; and (b) if the component instance is tagged with multiple `maxPower` values (cf. **(T6)** in Section 5.3), then the maximal value is used. If it is tagged with only one value, then of course the `max` operator returns this one value.

The new set operators and the Elvis operator enables to define this consistency constraint in less than 10 lines of *OCL* code.

The turbine controller example in Figure 5.7 satisfies this constraint, because:

(a) `TurbineController` component type definition is tagged with `maxPower = 4W`, but the only component instance of this component type is not tagged with `maxPower`. Therefore, `0W = aggregation <= compareTo = 4W`.
(b) `BrakeCtrl` component type definition is tagged with `maxPower = 2010mW`, and the two component instances are tagged with `maxPower = 1W` and `maxPower = 2010mW`.

6.2.3. Encryption for Port Instantiation

Consistency Rule: The encryption of a port instance must be in the encryption set of the port definition.

Figure 6.15 shows the complete *OCL* code, modulo one line of import statement, for port instantiation consistency of the extra-functional property encryption. Line 8 forbids tagging the port instance multiple times. Tagging an element several times with the same value results in a set with one value, because a set contains only different elements - in contrast to a list.

[2]*OCL* can access the content of optional values directly, so `x` is equals to `x.get()`. If the optional value is not present, then `x` returns an error resulting that the Boolean expression surrounding this error is evaluated to false.

TagSchema

```
1  // authentication
2  tagtype auth: [Pin, Voice, FaceID,
3     Finger] for Connector, ConnectorInst;
```

OCL

```
4  context Connector inv Authentication:
5    let
6      selection = connectorInst;
7      aggregation = selection.auth.value;
8      compareTo = auth.value;
9    in          // comparison
10     compareTo.containsAll(aggregation)
```

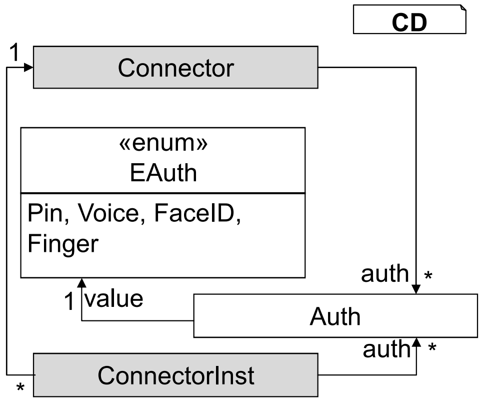


Figure 6.16.: *OCL* code for authentication consistency rule.

Lines 6 and 9 forces that the set `encryption.value` is a subset or equals to the `aggregation` set. The turbine controller example satisfies this rule, e.g.: `pitchBrake` and `turbineState` of `MainController` component definition are tagged with `{AES, RSA}` (cf. l. 9 in Figure 5.11), and the corresponding port instances are tagged with `AES` (cf. l. 14 in Figure 5.12).

Adding `tag turbineCtrl.windSpeed with encryption = DES3` causes this consistency constraint to fail, because the `windSpeed` port definition of the `TurbineController` component definition is not tagged at all; and line 5 flats this to an empty set which does not contain `DES3`.

6.2.4. Authentication for Connector Instantiation

Consistency Rule: The union of authentication methods of all connector instances must be a subset equal to the methods of the connector definition.

Figure 6.16 shows the tag type definition in lines 2 to 3 (as this extra-functional property have not been defined in Chapter 5), the class diagram derived of this tag type definition (cf. Subsection 5.5.2), and the *OCL* consistency constraint in lines 4 to 10. In contrast to the previous constraint starting with the instance, this constraint starts with the connector definition in line 4 and chooses all connector instances of this connector definition in line 6. *OCL* can navigate against the navigation direction of associations [Rum16]. The auxiliary variable `selection` is a set of connection instances (cf. star cardinality in class diagram). The `aggregation` variable is a set of `EAuth`; due to the automatic flattening of *OCL* (cf. Rumpe for further information [Rum16]) the type of `aggregation` is **not** a set of sets.

Assume we have the connector definition `in1 -> out1` with the two connector instances `a.in1 -> a.out1` and `b.in1 -> b.out1`. Therefore, the value of selection is the set of both connector instances. Furthermore, `in1 -> out1` is tagged twice: `auth = Pin` and `auth = Voice`. Due to automatic flattening `auth.value` is the set `{Pin, Voice}`.

TagSchema
```
1  // certificates
2  tagtype cert: String for ComponentInst,
3                           Port;
```

OCL
```
4  context ComponentInst inv Certificates:
5    let
6      selection = component.ports;
7      aggregation = intersection { s.cert |
8                         s in selection};
9      compareTo = cert;
10   in           // comparison
11     compareTo.containsAll(aggregation)
```

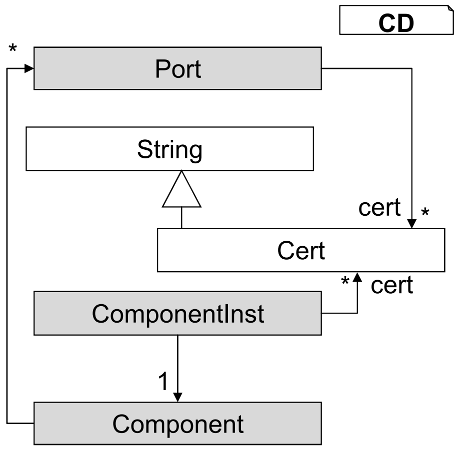


Figure 6.17.: *OCL* code for certificates consistency constraint.

Case A: If `a.in1 -> a.out1` is not tagged, and `b.in1 -> b.out1` is tagged with `auth = Voice`, then `aggregation` is the set `{Voice}`. This case satisfies the consistency constraint.

Case B: If `a.in1 -> a.out1` is tagged with `auth = Finger` and `auth = Voice` as well as `b.in1 -> b.out1` is tagged with `auth = Pin` and `auth = Voice`, then `aggregation` is the set `{Finger, Voice, Pin}`. This case does not satisfy the constraint, because `Finger` is in set `aggregration`, but it is not in the set `auth.value`.

6.2.5. Certificates for Component Instances/Port Definitions

Consistency Rule: The certificates of component instances must be at most the certificates common to all port definitions of the corresponding component type.

Figure 6.17 shows the tag type `cert`, the derived class diagram, and the *OCL* constraint to enforce certificate consistency. Line 6 selects all port definitions of the component definition which belongs to the given component instance. The expression `{ s.cert | s in selection}` creates a set of sets; it is not automatic flattened, because it is no navigation expression such as `selection.cert` is. The `intersection` operation receives as input a set of sets, and it returns a set whereby the set is the intersection of the input. For example, `interestion { {a, b, c}, {b, c, d} }` is equals to `{b, c}`. The intersection operator is the unary `retainAll` operator of *OCL*[3]. Therefore, `intersection { {a, b, c, d, e}, {b, c, d, e, f}, {c, d, e, f, g} }` is equals to `{a, b, c, d, e} .retainAll({b, c, d, e, f}).retainAll({c, d, e, f, g});` the unary `intersection` set operator is just more convenient. Line 11 is equivalent to `compareTo` $\supseteq$ `aggregation`.

[3] cf. `http://mbse.se-rwth.de/book1/index.php?c=chapterC-3`

OCL

```
1  context Component inv MaxPowerSubs:
2    let
3      selection1 = subs;
4      selection2 = this;
5      aggTags =
6        List { max s.type.maxPower ?: 0W |
7                   s in subs };
8      aggregation = sum aggTags;
9      compareTo = min maxPower ?: oo W;
10   in          // comparison
11     aggregation <= compareTo
```

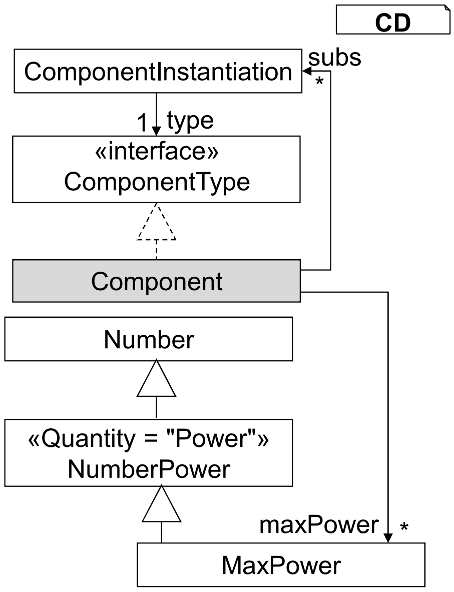


Figure 6.18.: *OCL* code for maximal power consumption of subcomponents.

6.2.6. Maximal Power Consumption of Subcomponent Instantiations

Consistency Rule: The combined maximal power consumption of all component definitions belonging to subcomponent instantiations of the decomposed component definition is at most the maximal power consumption of the composed component definition itself.

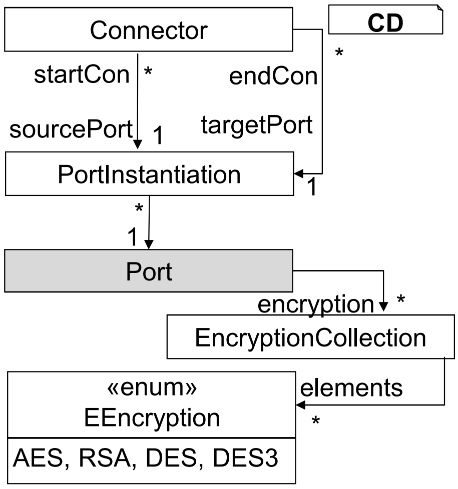
Figure 6.18 checks whether the aggregated value of the maximal power consumption of all subcomponent instantiations of a component definition is smaller or equal to the maximal power consumption of the component definition itself. Similar to Figure 6.14, `max X ?: 0W` for the set `X` returns `0W` if `X` is the empty set, and otherwise the element of `X` having the maximum value. The `aggTags` variable (cf. l. 5-6) is a list which elements store the maximal power consumption of each subcomponent instantiation inside `this`. Line 8 calculates the `sum` of all elements in the list; e.g., `sum List{2, 4, 9, 2}` is equals to `17`. Please note that lists can have duplicated entries, in contrast to sets. Line 9 in Figure 6.18 is the same as line 7 in Figure 6.14.

The `TurbineController` in component type in Figure 5.7 violates this rule, because it is tagged with `maxPower = 4W` and it contains the two component instantiations `brCoA` and `brCoB` having the type `BrakeCtrl`. Therefore, `aggTags` is a list with two elements `{2010mW, 2010mW}`. It uses the `maxPower` value of component definitions and not of the component instances. The variable `aggregation` is `4020 mW` which is the sum of both elements. Since `4020 mW` is not smaller or equals to `4W`, the constraint fails.

However, if this constraint would compare the subcomponent instance values, then the turbine controller would satisfy it. The `selection1 = subs` expression must be changed to `selection1 = componentInst.subs` and the expression `s.type.maxPower` must be modified to `s.maxPower` in Figure 6.18 when constraining the subcomponent instances instead of the component types of the subcomponent instantiations.

OCL

```
1  context Port inv Encryption:
2    let// selection is a set of sender ports
3      selection = portInstantiations
4                  .endCon.sourcePort.port;
5      aggregation = { p.encryption.elements
6                        | p in selection };
7      compareTo = encryption.elements;
8    in          // comparison
9      forall encSrc in aggregation:
10       encSrc.retainAll(compareTo) != {}
```



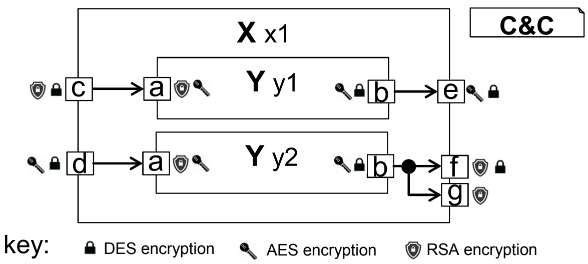


Figure 6.19.: *OCL* code for target ports and one simple example for encryption.

6.2.7. Encryption for Target Ports

Consistency Rule: A target port must support at least one encryption of its sender ports.

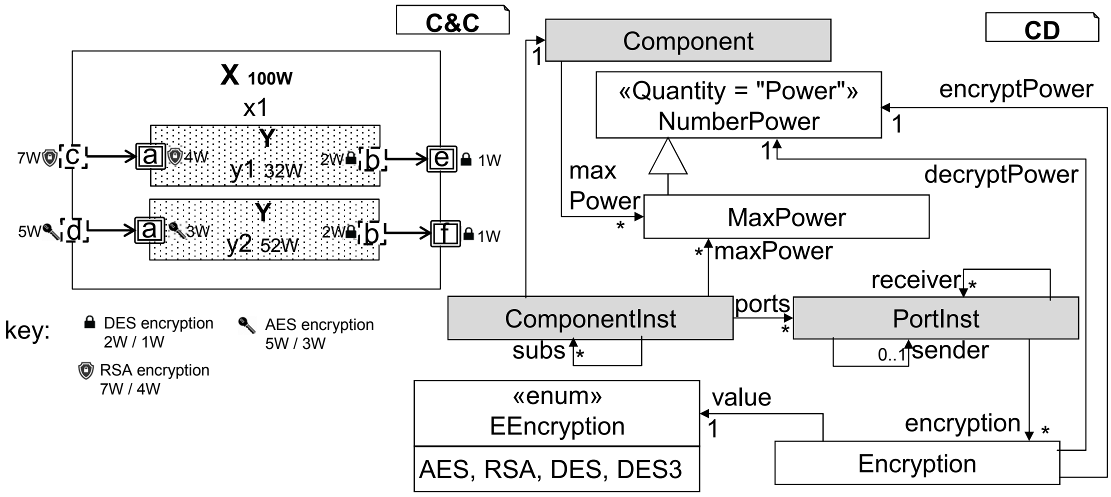
Figure 6.19 shows the *OCL* code for the target port consistency rule. Lines 3 and 4 are equivalent to `selection = { p | exists Connector con: con.sourcePort.port == this && con.targetPort.port == p}`.

For the port definition `b` (cf. C&C model in bottom left part) which is only a source port, `b.portInstantiations` evaluates to `{y1.b, y2.b}` and `y1.b.endCon` as well as `y2.b.endCon` evaluate to empty sets[4]. Therefore, the `selection` variable for the port `b` context is an empty set. If `selection` is an empty set, then the `aggregation` variable is also an empty set, and the constraint is satisfied (cf. l. 9).

For the port definition `a` the expression `a.portInstantiations` evaluates to `{y1.a, y2.a}`, hence, `a.portInstantiations.endCon` is equals to `{c -> y1.a, d -> y2.a}`. Therefore, `selection` is the set `{c, d}`. The variable `aggregation` evaluates to the following set of sets `{ {RSA, DES}, {AES, DES} }`. The encryption target (cf. `compareTo` in line 7) is the set of encryption elements of the port definition `a`; it is `{RSA, AES}`. Lines 9 and 10 are satisfied, because `{RSA, DES}`$\cap$`{RSA, AES}={RSA}`$\neq$`{}` and `{AES, DES}`$\cap$`{RSA, AES}={AES}`$\neq$`{}`.

The example in Figure 6.19 still fails the presented *OCL* consistency constraint, because the target port definition `g` supports only the encryption mode RSA (`aggregation = {RSA}`),

[4]Line 3 and 4 select only ports which are connected with `b` and `b` is target port. The set {e, f, g} does not count, because in these connections `b` is source port.



OCL

```
context ComponentInst inv PowerConsumptionWithEncryption:
  let// selections are subcomponent and port instances
    selection1 = subs;
    selection2 = ports.addAll(subs.ports);
    aggregation = sum List{ max s.maxPower ?: 0W | s in selection1 }
                + sum List{ max pi.encryption.encryptPower ?: 0W |
                        pi in selection2, pi.receiver != {} }
                + sum List{ max pi.encryption.decryptPower ?: 0W |
                        pi in selection2, pi.sender.isPresent };
    compareTo = min component.maxPower ?: oo W;
  in
    aggregation <= compareTo
```

Figure 6.20.: *OCL* consistency code using two different extra-functional properties.

and its sender port definition `b` only the encryption modes AES and DES. Therefore, `g` and `b` have no common encryption mode to communicate with (cf. l. 10: `{AES, DES}`∩`{RSA}={}`).

6.2.8. Power Consumption of Components considering Power needed for Encryption and Decryption

Consistency Rule: The combined energy consumption of subcomponent definitions plus the energy consumption needed for encryption and decryption data is at most the energy consumption of the composed component instance.

Figure 6.20 presents an *OCL* constraint which uses the two extra-functional properties `maxPower` consumption of component definitions and component instances as well as the encryption kind of port instances. The *OCL* code uses two selection auxiliary variables; this means the witness (cf. Section 7.3) contains the elements of `selection1` and `selection2`.

The small C&C example shown in Figure 6.20 does not satisfy the constraint, because its `aggregation` value is 109W (84W + 16W + 9W = (`32W` + `52W`) + (`7W` + `5W` + `2W` + `2W`) + (`4W` + `3W` + `1W` + `1W`)) and this is larger than the `100W` of the maximal power consumption specified by component definition `X`.

The power consumption needed for different encryption and decryption kinds are invented numbers to have a simple example. The paper *A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols* [PRRJ06] contains realistic numbers about the power consumption of different encryption kinds.

The expression `pi.receiver != {}` in line 7 evaluates to true if the port instance `pi` has a receiver port instance accepting the data; this means if the port instance itself is a sender port the expression is true; otherwise it is false. The expression `pi.sender.isPresent` in line 9 evaluates to true if the port is a receiver port. Please notice, that a port can be both; e.g., when the component instance `x1` is embedded into another component instance `z1` and the port `z1.g` is connected to `x1.c`, then `x1.c` has a sender port, i.e., `z1.g`, and one receiver port, i.e., `y1.a`.

The *OCL* expression can also be adapted to not add the power consumption of ports just delegating encrypted values, as a smart implementation does not decrypt and encrypt the by-passed values. In this case, line 7 needs to be modified to `pi in selection2 && pi.receiver != {} && pi.sender.isAbsent`, and analog line 9. This small discussion shows that consistency rules of extra-functional properties are not fix and must be adapted to the current context. However, the presented *OCL* framework enables defining these rules in few lines of code.

The `encryptPower` and `decryptPower` associations in Figure 6.20 are added manually: i.e., (1) add a new class diagram artifact with these two associations - all class diagrams are merged; and (2) extend the generated `EncryptionSymbol.java` file via the *MontiCore top mechanism* [HR17] by adding two Java methods returning for each encryption kind the correct power consumption value. The first step (1) is needed so that *OCL* language accepts the presented *OCL* code in Figure 6.20; the *OCL* language checks whether all associations exist. The second step (2) is needed to compile the generated Java code by the *OCL* to Java compiler; otherwise, an error arises that the Java methods `EncryptionSymbol.getEncryptPower` and `EncryptionSymbol.getDecryptPower` do not exist.

Figure 6.21 defines the mapping of needed power consumption to encrypt or decrypt data according to the used algorithm in a table tag (cf. ll. 11-14). First, lines 1 to 8 create a new tag schema defining the table structure. Lines 4 and 5 use the same enumeration items and the same table name as the encryption tag types (cf. ll. 8, 10 in Figure 5.10). Therefore, the tagging generator generates a class diagram where `EncryptPower` has an association to `Encryption` and the class diagram merger produces the class diagram shown in Figure 6.21. If this behavior is not wanted, then the table name in line 3 must be changed. If the table is the same but the enumeration items are different, then the class diagram merger throws an error that the diagrams are incompatible and cannot be merged.

The class diagram marks the new elements added due to the `encryptPower` tag schema bold. Lines 15 to 28 show the modified *OCL* code whereby the changed code parts are surrounded by dotted lines. Since the association `component.encryptPower` (cf. l. 18) is a qualified one, `ep` is a map and the key column `encryption` is the key element to access the map as it is shown in line 20 and 23.

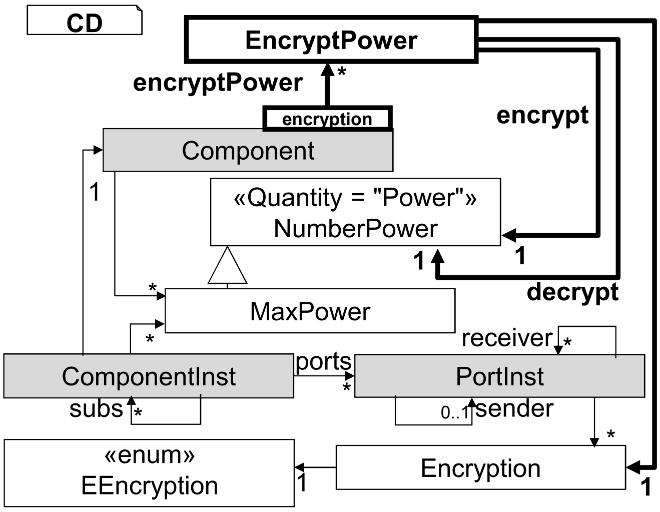
The expression `{ ep[encValue] | encValue in pi.encryption }.encrypt` is normal *OCL*/P as defined by Rumpe [Rum16]. Since `pi.encryption` has as inferred type **`Collection<Encryption>`** (cf. star cardinality in association), the access `ep[pi.en-`

TagSchema

```
tagschema EFPEncPower {
  tagtype encryptPower:
    | encryption:
        [ AES |RSA |
          DES | DES3 ]
    | encrypt:    Power
    | decrypt:    Power
    | for Component;        }
```

TagModel

```
conforms to EFPEncPower;
tags EPTable {
tag X with encryptPower =
   | DES | 2W | 1W |
   | AES | 5W | 3W |
   | RSA | 7W | 4W |;        }
```



OCL

```
context ComponentInst inv PowerConsumptionWithEncryptionTable:
  let selection1 = subs;
      selection2 = ports.addAll(subs.ports);
      ep = component.encryptPower;
      aggregation = sum List{ max s.maxPower ?: 0W | s in selection1 }
                  + sum List{ max { ep[encValue] | encValue in pi.encryption
                                  }.encrypt ?: 0W |
                             pi in selection2, pi.receiver != {} }
                  + sum List{ max { ep[decValue] | devalue in pi.encryption
                                  }.decrypt ?: 0W |
                             pi in selection2, pi.sender.isPresent };
      compareTo = min component.maxPower ?: oo W;
  in
    aggregation <= compareTo
```

Figure 6.21.: Complete model-driven workflow to define a consistency constraint involving two extra-functional properties.

`cryption]` is not defined in normal *OCL*/P, as `ep` is of type `Map<`**`Encryption`**`, Collection<EncryptPower>>`.

However, the author of this thesis thinks that the new element-wise access concepts similar to `indexOf` for array access (cf. Subsection 6.2.9) helps to improve the readability of *OCL* constraints. The element-wise access defines `map[{key1, key2}]` as `{map[key1], map[key2]}`. With this new introduced *OCL* concept the expression `{ ep[encValue] | encValue in pi.encryption }.encrypt` can be simply written as `ep[pi.encryption].encrypt`. And this supports the concept of *OCL* to focus only on the logic of constraints and not to bother the modeler with complex looking set constraints.

The table tagging approach enables defining for each component definition its own mapping from encryption kind to power consumption. If this mapping is the same for every component, then the best way is to tag the main component type of the model; because from any component instance a `main` association goes to the `CncInstanceStructure` class (cf. Figure 4.17 on

TagSchema

```
1  tagtype asil: [QM |ASIL_A | ASIL_B |
2                 ASIL_C | ASIL_D] for Component;
```

OCL

```
3  context ComponentInst inv ASIL:
4    let
5      asilNb = List{ QM, ASIL_A, ASIL_B,
6                     ASIL_C, ASIL_D };
7      selection = subs;
8      aggregation = min {
9        min asilNb.indexOf(s.asil.value) ?: 0 |
10       s in selection.component             };
11     compareTo = min asilNb.indexOf(
12                   component.asil.value) ?: 0;
13   in          // comparison
14     aggregation >= compareTo
```

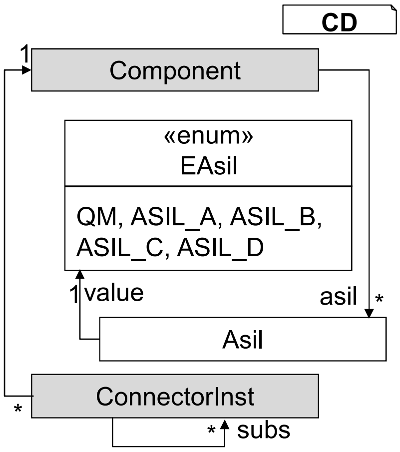


Figure 6.22.: *OCL* code to force that composed component cannot have higher ASIL than its subcomponents.

page 123) and from this class you can easily navigate to the component definition of the main component instance.

6.2.9. Automotive Safety Integrity Level for Component Definitions

Consistency Rule: The ASIL (Automotive Safety Integrity Level) of all subcomponents must be higher or equal than the ASIL of the composed component.

Figure 6.22 shows the *OCL* code checking that the ASIL of all subcomponents must be higher or equal than the ASIL of the composed component. Lines 5 and 6 define the `asilNb` helper list. This list maps each ASIL to a number so that a comparison of ASILs is possible. The expression `asilNb.indexOf(x)` returns `0` for `QM`, `1` for `ASIL_A`, `2` for `ASIL_B`, `3` for `ASIL_C` and `4` for `ASIL_D`.

The `indexOf`[5] function is extended to accept a set as parameter by applying the Java `indexOf` operator to each element of the set; e.g., `asilNb.indexOf( {QM, ASIL_C} )` returns `{0, 3}`. *OCL* extends all Java operators to set and list operators when applying it element-wise makes sense; this way *OCL* expressions - often dealing with sets due to its navigations of associations - come along with less `forall` or `exists` statements, which makes the code easier to read.

Line 9 takes the lowest number of an ASIL when a component is tagged with two different ASIL values, and it takes `0` for QM when the subcomponent has not been tagged at all. Line 14 does the comparison.

[5] cf. `http://mbse.se-rwth.de/book2/index.php?c=chapter3-2`

TagSchema

```
tagtype wcet: (0s : oo s) for Component;
```

OCL

```
context ComponentInst inv WCET_SingleCore:
  let
    selection = subs;
    aggregation = sum { max s.wcet ?: 0s |
                        s in selection.component };
    compareTo = min component.wcet ?: oo s;
  in          // comparison
    aggregation <= compareTo
```

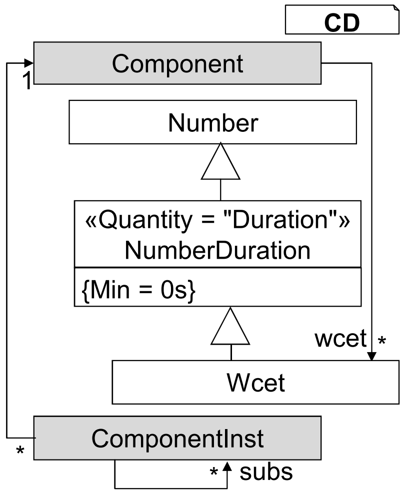


Figure 6.23.: *OCL* consistency constraint for worst-case execution time (WCET) of a single core processor.

6.2.10. Worst Case Execution Time for Single Core Processors

Consistency Rule: The worst-case execution time of a component instance is at most the worst-case execution time of its subcomponent instances.

Figure 6.23 shows one out of three semantic interpretations of worst-case execution time. This semantic does not parallelize anything, as it would be the case executing the model on a single core processor. Since this *OCL* constraint is very similar to the one for maximum power consumption in Figure 6.18, no further explanation is required.

6.2.11. Worst Case Execution Time for Processors with Infinite Cores

Consistency Rule: The worst-case execution time of a component instance is at most the maximum of the worst-case execution time of parallel executable direct subcomponent paths (we assume that the host PC has infinite many cores).

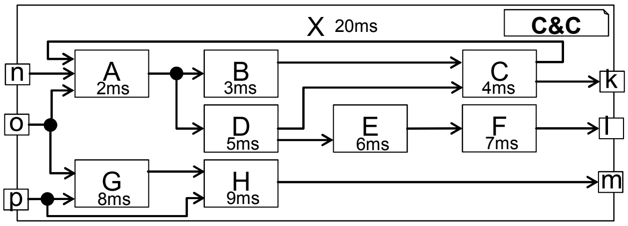
Figure 6.24 shows the second worst-case execution time semantics. The *OCL* code for composition consistency of worst-case execution time with the semantic interpretation to parallelize anything. This can be used if there is not sure how many CPU or GPU cores exist in the design phase, and if this constraint failed, the worst-case execution time constraint will fail no matter what the exact core number is.

Line 3 selects all subcomponents which sender is the current component instance. In the C&C example model, `startSubs` includes `A`, `G`, and `H` as these receive values directly from any input port of `X`. Line 4 stores all output ports in an auxiliary variable; in this example `outPorts` is the set `{k, l, m}`.

Lines 7 and 8 stores in `selection` all chains starting at any subcomponent instance of `startSubs` and ending at any port of `outPorts`. All chain instances contain all elements only once, thus, the cycle from `C` to `A` is contained at most once. Lines 9 and 10 filter out all

OCL

```
1  context ComponentInst inv WCET_InfiteCores:
2  let
3    startSubs = {s in subs | this in s.sender };
4    outPorts = {p in ports | p.direction == OUT};
5    // selection: chains from subcomponents to
6    // an output port
7    selection = { chain in startSubs.start |
8                  chain.end in outPorts };
9    subChains = { chain in selection |
10                 chain.elements.retainAll(subs)};
11   aggChains = { sum List{ max s.wcet ?: 0s |
12                s in chain.elements.component } |
13                 chain in subChains };
14   aggregation = max aggChains ?: 0s;
15   compareTo = min component.wcet ?: oo s;
16 in
17   aggregation <= compareTo
```



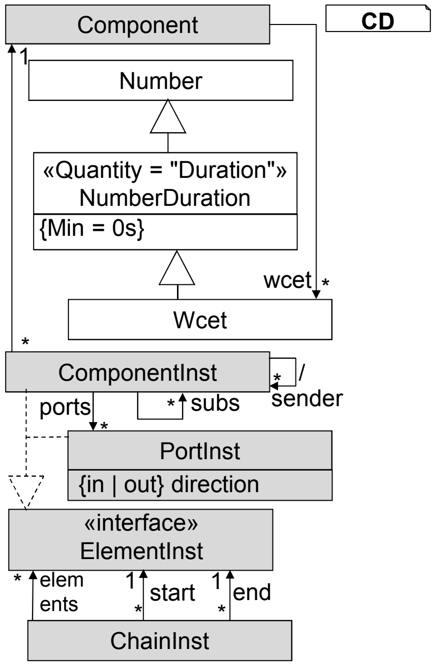


Figure 6.24.: *OCL* consistency constraint for worst-case execution time (WCET) using infinite cores.

subcomponent instances of the chains (`chain.retainAll(subs)` is the same as `chain` ∩ `subs`). In our example, `subChains` consists of five chains:

- chain 1 from A to k: A → B → C
- chain 2 from A to k: A → D → C
- chain 3 from A to l: A → D → E → F
- chain 4 from G to m: G → H
- chain 5 from H to m: H

Lines 11 to 13 adds all the worst-case execution time values of one `chain` in `subChains`. The expression `max s.wcet ?: 0s` is only there if one component instance has not been tagged or has been tagged multiple times. In the example, `aggChains` is the set `{9 ms, 11 ms, 20 ms, 17 ms}`; chain 1 and chain 5 have both a runtime of `9 ms`. In this example, the worst-case execution time is 20 ms when all five chains are parallelized on different cores. Since the decomposed component `X` is also tagged with `20ms`, the example satisfies this constraint.

The example in Figure 6.24 does not satisfy the constrain shown in Figure 6.23, because when all subcomponents are executed on one processor, the worst-case execution time of the subcomponents would be `44ms` which is not smaller or equal to `20ms`.

TagModel...

```
1  tag x with threads = 3;
```

OCL

```
2  context ComponentInst inv WCET_MultiCore:
3  let
4    startSubs = {s in subs | this in s.sender };
5    outPorts = {p in ports | p.direction == OUT};
6    // selection: chains from subcomponents to
7    // an output port
8    selection = { chain in startSubs.start |
9                  chain.end in outPorts };
10   subChains = {chain in selection |
11                 chain.retainAll(subs) };
12   threads = min cis.main.threads ?: 1;
13   // calculates all partition combinations
14   combChains=subChains.listPartitions(threads);
15   combSubs = { { {chain | chain in partition}
16            .asSet | partition in singleComb } |
17     singleComb in combChains };
18   partSums=  { { sum List{ max s.wcet ?: 0s |
19                            el in partition,
20                          s in el.component} |
21                 partition in singleCombs } |
22               singleCombs in combSubs }
23   maxTimeInComb = { max { p | p in singleComb }
24               ?: 0s | singleComb in partSums };
25   // aggregation takes best combination
26   aggregation = min maxTimeInComb ?: 0s;
27   compareTo = min component.wcet ?: oo s
28 in
29   aggregation <= compareTo
```

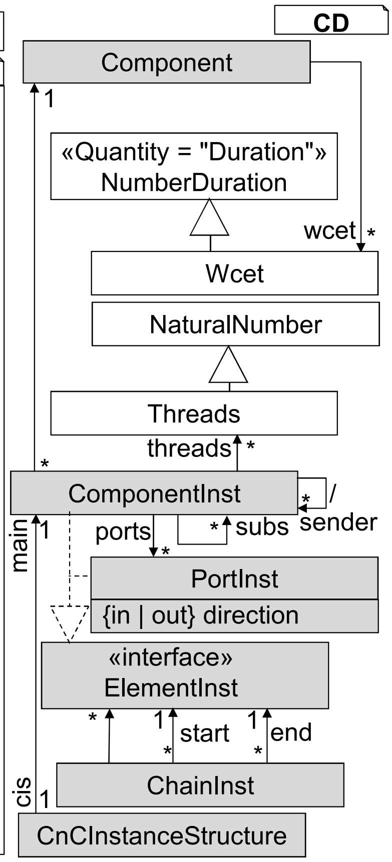


Figure 6.25.: *OCL* consistency constraint for worst-case execution time (WCET) using a multicore processor. Yellow high-lighted lines show the difference to *OCL* listing in Figure 6.24.

6.2.12. Worst Case Execution Time for Multi Thread Processors

Consistency Constraint: The worst-case execution time of a component instance is at most the maximum of the worst-case execution time of parallelizable executable direct subcomponent paths constrained by the number of available threads.

Similar to Subsection 6.2.8 combining maximal power consumption with used encryption kind, this constraint combines the two extra-functional properties: worst-case execution time and number of available threads.

Figure 6.25 shows the third semantics of worst-case execution time in combination with the `threads` extra-functional property (cf. l. 1). The lines enclosed by a dotted rectangle (i.e., ll. 12-26) in the *OCL* listing are different from the *OCL* listing in Figure 6.24. Line 12 receives the number of threads of the main component instance; the current component instance navigates to its `CnCInstanceStructure` via `cis` role name and the C&C instance structure navigates via `main` role name to the main component instance. For the example, we use the same C&C model as shown in Figure 6.24, the number of available threads are 3 (cf. l. 1).

Line 14 creates all combinations to partition the list with 5 elements into three sub-lists; each thread receives one sub-list executing the independent subcomponents chains.

For partitioning a list with 5 chains to three sub-lists, there exists 150 combinations. Three randomly selected combinations are:

- [[A → B → C], [A → D → C], [A → D → E → F, G → H, H]]
- [[A → D → C], [A → D → E → F, H], [A → B → C, G → H]]
- [[G → H],[A → B → C, A → D → C, A → D → E → F],[H]]

The variable `combChains` contains 150 combinations of three lists, each containing a list of chain instances. Lines 15 to 17 flatten the four dimensional structure `combChains` $\rightarrow$ `150 singleCombs` $\rightarrow$ 150×3 `partitions` $\rightarrow$ $150 \times 3 \times n$ `chains` $\rightarrow$ $150 \times 3 \times n \times m$ `element instances` to the three dimensional structure `combSubs` $\rightarrow$ `150 singleCombs` $\rightarrow$ 150×3 `partition` $\rightarrow$ $150 \times 3 \times k$ `element instances` whereby the n chain lists of m subcomponent instances are flattened to a set of k component instances. This flattening step is done to avoid executing one component twice in the same thread. Three combinations of `combSubs`, derived from the above presented `combChains` combinations, are:

- {{A, B, C}, {A, D, C}, {A, D, E, F, G, H}}
- {{A, D, C},{A, D, E, F, H},{A, B, C, G, H}}
- {{G, H}, {A, B, C, D, E, F}, {H}}

Lines 18 to 22 aggregate the worst-case execution time of all subcomponents inside one partition. This means the three dimensional structure `combSubs` shrinks to the two dimensional structure `partSums` $\rightarrow$ `150 singleCombs` $\rightarrow$ 150×3 `partition WCET values`. Three combinations of `partSums` are:

- {9 ms, 11 ms, 37 ms}
- {11 ms, 29 ms, 26 ms}
- {17 ms, 27 ms, 9 ms}

Lines 23 and 24 take for each combination the highest time value, because the execution is only finished when all threads are finished. Therefore, for the three combinations of `partSums` above, `maxTimeInComb` is a set containing - among others - the values `37 ms`, `29 ms`, and `27 ms`. Line 26 chooses the best single combination, i.e., the minimal value of `maxTimeInComb`, and stores it in `aggregation`. For the three combinations presented here, `aggregation` has the value `27 ms` by choosing the last combination. This `aggregation` value causes the *OCL* consistency constraint to fail in line 29.

This last *OCL* example shown in Figure 6.25 proves that our *OCL* framework is suited to describe real-world consistency constraints for extra-functional properties. The verification of the multi core worst-case execution time is time consuming when the number of subcomponents increases, because then the combinations how to deploy the chains of subcomponents onto different threads/cores increases dramatically. The generated Java code uses the Java library `combinatoricslib` of version 2.3[6] to calculate all partition combinations of line 14. This library also supports combinations with repetitions, permutations with and without repetitions, power sets, as well as Cartesian product calculations.

[6] cf. `https://github.com/dpaukov/combinatoricslib#7-list-partitions`

6.3. Extracting Consistency Witnesses from the *OCL* Constraints Describing Extra-Functional Properties

The normal *OCL* to Java generator produces witnesses based on the `let-in` construction as explained in Subsection 6.1.2 for the context condition on connectors. The verification tool of *EmbeddedMontiArc* for extra-functional properties extends the *OCL* to Java generator to produce more user-friendly positive consistency and negative inconsistency witnesses [MRR13, MRR14, Rin14]. The generated witnesses are parseable *EmbeddedMontiArc* models. However, the witnesses do not satisfy all context conditions, because they only contain the relevant C&C elements needed to explain why a consistency constraint is satisfied or not. For example, witnesses of `Traceability` (cf. Subsection 6.2.1), `MaxPower` (cf. Subsection 6.2.2), contain only components and component instances. In contrast, the witnesses of `Encryption` (cf. Subsection 6.2.3) would only contain port and port instances, whereas the witnesses of `Authentication` would only contain connector and connector instances.

To create better readable textual witnesses and nice graphical models, the verification generator extends the witnesses produced by the *OCL* to Java generator in the following way:

- The consistency witness contains all components and component instances for which a port or a port instance exist in the *OCL* witness.
- The consistency witness contains for each connector definition the source and target port instantiations inclusive the subcomponent instantiations and the port definitions.
- The consistency witness contains for each connector instance the source and target port instances as well as their component instances.
- If an extra-functional property is stored inside `aggregation` or `compareTo`, then the consistency witness also contains the corresponding C&C element.

Inconsistency witnesses contain the minimal C&C elements to violate an extra-functional property. For example, if the consistency constraint forces that the costs of the sum of all direct subcomponent instances are lower than their parent component instance; and one subcomponent instance is already more expense than the parent component instance, then the witness includes the parent component instance and only this one subcomponent instance. Filtering all not needed subcomponent instances facilitates to focus why a consistency constraint fails.

The rest of this section presents one positive consistency witness and one negative inconsistency witness. The witnesses shown in this thesis are relayouted, but no extra text or graphical elements have been added.

6.3.1. Positive Consistency Witnesses

Figure 6.26 shows the graphical representation of all generated consistency witnesses of the Traceability (cf. Figure 6.13) constraint for the the TurbineController example (cf. Figure 5.7 on page 155). Since the TurbineController model has 13 component instances, which are the context of the constraint (cf. l. 1 in Figure 6.26), the verification algorithm produces 13 positive consistency witnesses. Because the `selection` variable stores the current component instance, each witness consists of exactly this one component instance. Because `aggregation` and

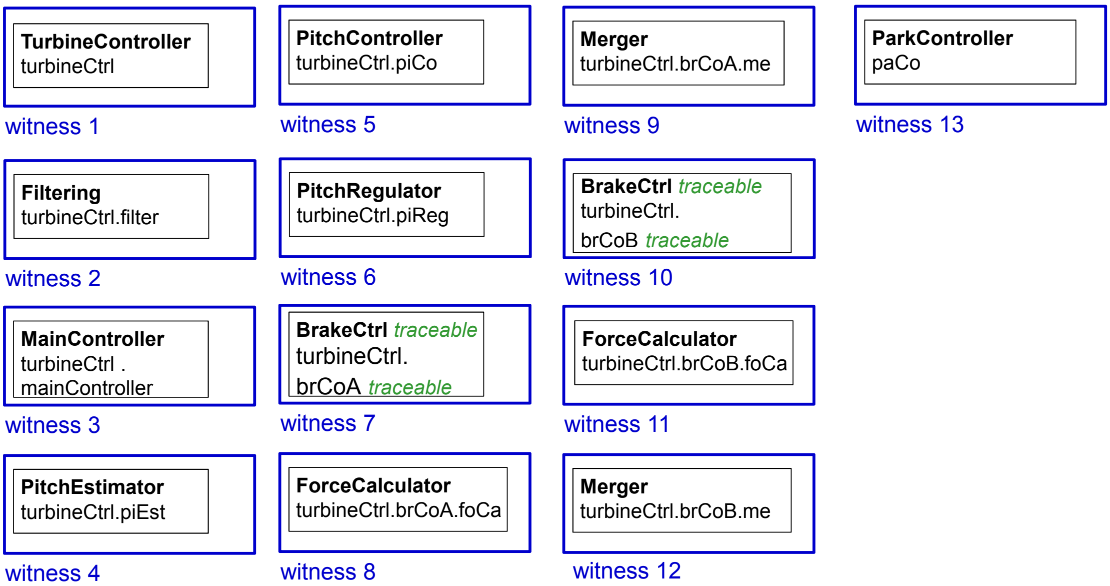


Figure 6.26.: Positive consistency witness of `Traceability` (cf. Subsection 6.2.1) constraint for `TurbineController` example (cf. Figure 5.7 on page 155). The values `aggregation` and `compareTo` are skipped in this witness.

OCL

```
context Component inv Traceability2:
  let
    selection = componentInsts;
    aggregation =
      and componentInsts.traceable;
    compareTo    = traceable;
  in           // comparison
    compareTo implies aggregation
```

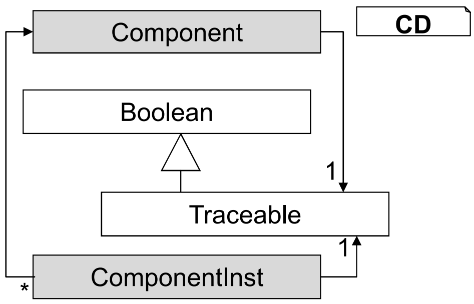


Figure 6.27.: Alternative to Figure 6.26 defining the traceable extra-functional consistency constraint.

`compareTo` address the `traceable` property this one is added to the witness. Please note that all other extra-functional properties of the `TurbineController` example are ignored.

Figure 6.27 shows an alternative *OCL* constraint to Figure 6.13. The context of the new `Traceability2` constraint is the component definition. Thus, the constrain generates ten witnesses as shown in Figure 6.28, as the turbine controller example has ten component types. *Witness 7* in Figure 6.28 contains two component instances because, the `BrakeCtrl` turbine controller has the two `brCoA` and `brCoB` component instances which are stored in the `selection` variable.

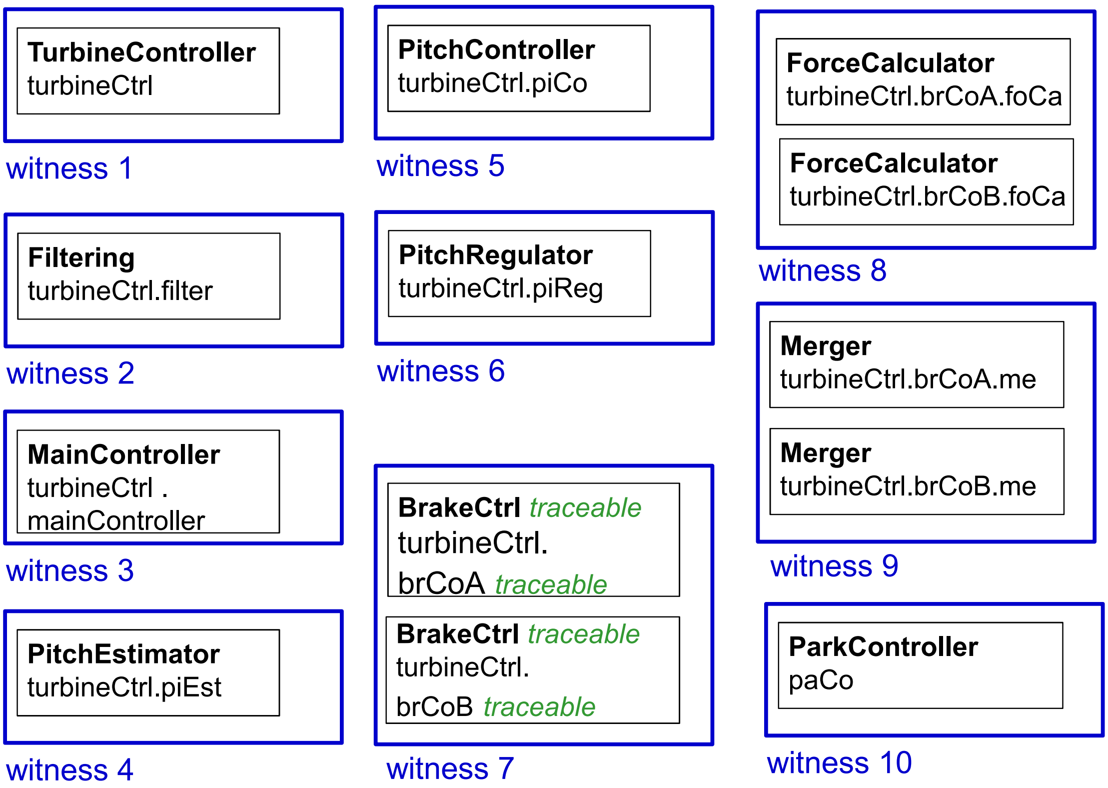


Figure 6.28.: Generated Witnesses of alternative consistency constraint. The values `aggregation` and `compareTo` are skipped in this witness.

6.3.2. Negative Inconsistency Witnesses

Figure 6.29 shows the C&C model of a simple weather balloon sensor [MMR+17]. This models serves to demonstrate how a negative witness looks like, because this model is inconsistent according to the `MaxPowerSubs` *OCL* rule shown in Figure 6.18.

Figure 6.30 presents the generated negative witness for the maximal power consumption rule of subcomponent instantiations (cf. Figure 6.18). The generated negative witness contains only of three subcomponent instantiations, because these three are enough to violate the constraint. The verification tool receives from the *OCL* to Java generator all subcomponent instantiations. Due to the <= relation between `aggregation` and `compareTo`, the verification generator sorts the subcomponent instantiations according to the used extra-functional property (e.g., `maxPower`) and it removes from the first selection statement as long as possible elements until this constraint is satisfied and then it adds the last removed one again. The result is a minimal inconsistency witness. The verification tool only optimizes according `selection` or `selection1` variable. It always shows the elements stored in `selection2` and so on, because this minimization analysis is too complex for the generator yet.

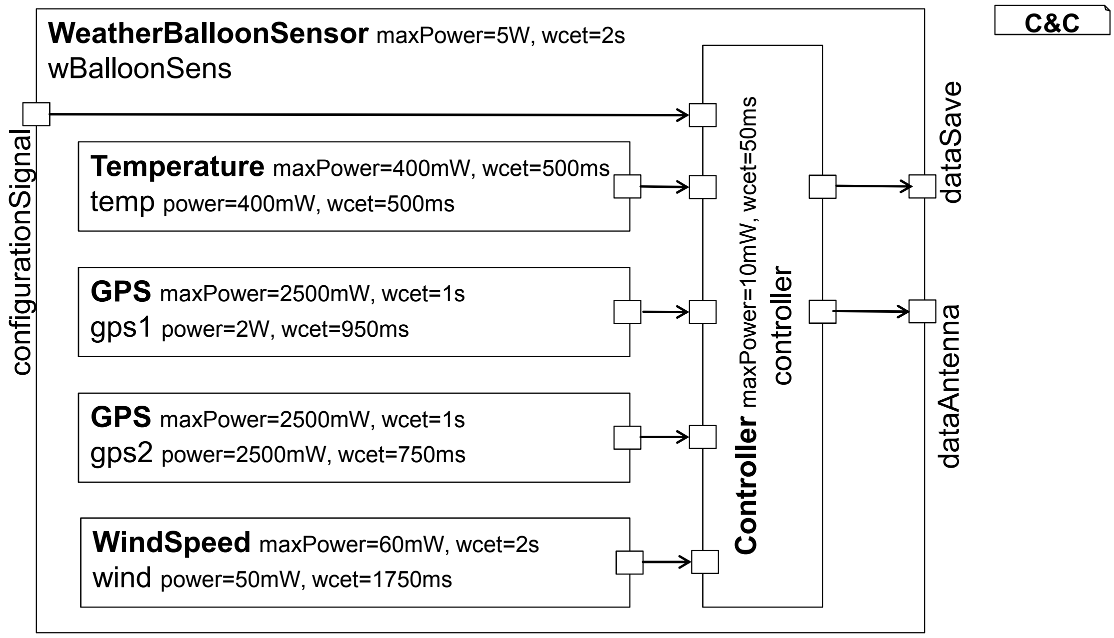


Figure 6.29.: WeatherBalloonSensor C&C model (adapted from [MMR+17]) to explain the negative inconsistency witness.

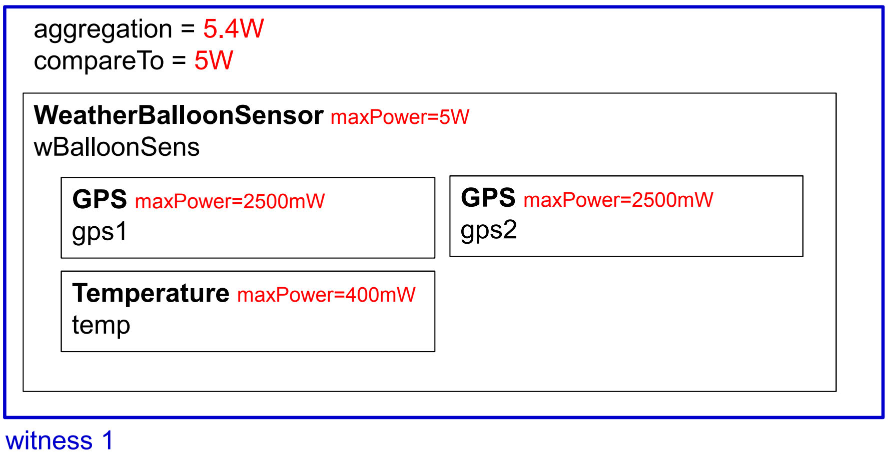


Figure 6.30.: Negative consistency witness showing that the sum of the maximal power consumption of the subcomponent instantiations exceeds the maximal power consumption of its parent component. Tooling marks the consistent extra-functional properties green and the inconsistent ones red.

MC5...

```
grammar EmbeddedMontiArcTooling {
  EmbeddedMontiArcArtifact = Package? ImportStatement* ComponentType;
  symbol scope Component implements ComponentType =
    "component" name:Name /*...*/ "{"
      "ports" (Port || ",")* ";"
     (subs:ComponentInstantiation Connector)* "}";

  enum Direction = in:"in" | out:"out";
  symbol Port =
     Direction type:PortType Name "[" dimension:NaturalNumber "]";

  symbol ComponentInstantiation /*...*/ =
    "instance" type:Name@ComponentType /*...*/ name:Name
    "[" dimension:NaturalNumber "]" ";" ;

  PortInstantiation =
    (sub:Name@ComponentInstantiation subIndices:Range | "this") "."
    port:Name@Port portIndices:Range;

  Connector =
    "connect" sourcePort:PortInstantiation
    "->" targetPort:PortInstantiation ";" ;

  Range =
   "[" start:NaturalNumber ":" step:NaturalNumber ":" end:NaturalNumber "]";
}
```

Figure 6.31.: Excerpt of normalized *EmbeddedMontiArc* grammar.

6.4. *OCL* to Specify Transformations between Abstract Syntax of Two Languages

The first sections of this chapter explained how to use *OCL* to describe structural or extra-functional property constraints. This section shortly explains how to transform the abstract syntax of *EmbeddedMontiArcParsing* to the abstract syntax of *EmbeddedMontiArcTooling*, and how *OCL* specifies these transformations. The transformations to the C&C instance structure defined in Section 4.3 work the same way; and thus, they are not explained in this section. In contrast to Hölldobler and Weisemöller et. al. [RW11, Wei12b, HRW15, HHRW15, AHRW17b, HRRW17, AHRW17a] describing transformations on the concrete syntax, *OCL* specifies the transformations on the abstract syntax.

The *EmbeddedMontiArc tooling developer* (cf. roles described in Figure 6.1 on page 171) specifies these *OCL* transformations between the abstract syntax of the three *MontiCore* languages *EmbeddedMontiArcParsing*, *EmbeddedMontiArcTooling*, and *CnCInstanceStructure* (cf. Figure 4.1 on page 104 for relationship between these grammars).

Figure 6.31 repeats an excerpt of the *EmbeddedMontiArcTooling* grammar (Figure 4.2 on page 106 presented a similar listing). *MontiCore* 5 generates the abstract syntax presented in Section 4.2 based on this grammar as explained in Section 4.1; cf. Figure 4.3 on page 106 or right part of Figure 6.34 for the derived class diagram of Figure 6.31.

MC5...

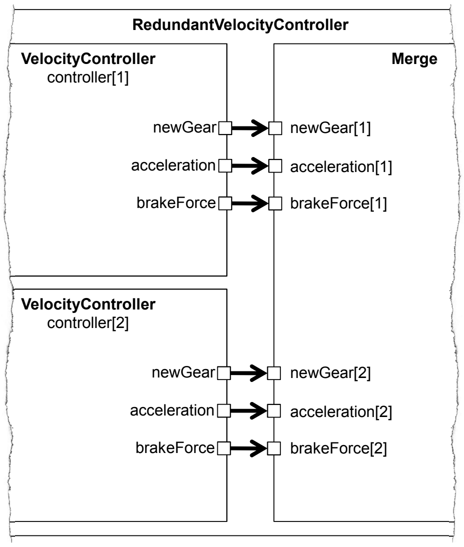
```
1  grammar EmbeddedMontiArcParsing extends EmbeddedMontiArcTooling {
2   start EmbeddedMontiArcArtifact;

3    symbol Port =
4       Direction? type:PortType Name ("[" dimension:NaturalNumber "]")?;

5    symbol ComponentInstantiation /*...*/ =
6      "instance" type:Name@ComponentType /*...*/ name:Name
7      ("[" dimension:NaturalNumber "]")? ";" ;

8    PortInstantiation =
9      (sub:Name@ComponentInstantiation subIndices:Range? | ["this"]?)
10     ("." port:Name@Port portIndices:Range? | nameBased:".*"
11      | indexBased:".**");

12   Range =
13    "[" (all:":" | start:NaturalNumber (":" step:NaturalNumber)?
14                                        ":" end:NaturalNumber) "]";
15 }
```

Figure 6.32.: Excerpt of *EmbeddedMontiArc* grammar; rules modified due to syntactic sugar are underlined.

Line 3 creates `Component` class extending the `ComponentType` interface as it is explained in Figure 4.10 on page 116. Line 5 and 6 create the `ports`, and `subs` association. The `parameters` association is skipped (cf. `/*...*/` in line 4). Line 6 also creates the association from `Component` to the `Connector` class, this one is skipped in Section 4.2 as it is not needed at all.

Line 8 creates the `Port` class; line 9 adds the `direction` enumeration attribute, and the `type` plus the `dimension` association (cf. Figure 4.11 on page 117). Line 13 creates the `PortInstantiation` class; line 14 adds the optional `sub` association to `ComponentInstantiation` as well as the optional `subIndices` association to `Range`. The associations are optional, because of the alternative (cf. pipe symbol). The expression `Name@ComponentInstantiation` means that the concrete syntax expects a word matching the Java name token, and *MontiCore* maps this word to a `ComponentInstantiation` object having this word as name. Line 15 adds to the `PortInstantiation` class the two associations `port` and `portIndices`. Lines 16 to 20 define the abstract syntax of `Connector` and `Range` as introduced in the bottom part of Figure 4.11.

However, the `Connector` definition in line 16 to 18 does not parse the following expression `connect controller[:].* -> merge.*[:]` illustrated in Figure 3.50 on page 87. One obvious solution is to extend the tooling grammar shown in Figure 6.31 with this nice syntactic sugar. However, this would destroy the abstract syntax of *EmbeddedMontiArcTooling* representing the essence of this language according to tool developers. Therefore, a better solution is to create the new *EmbeddedMontiArcParsing* grammar extending the *EmbeddedMontiArcTooling* one and to overwrite the inherited rules for adding syntactic sugar.



EMAParsing

```
component RedundantVelocityController(N+ n=2) {
  instance VelocityController controller[n];
  instance Merge<n> merge;

  connect controller[:].* -> merge.*[:];
}
```

trafo

EMATooling

```
component RedundantVelocityController(N+ n=2) {
  instance VelocityController controller[n];
  instance Merge<n> merge;

  connect controller[1:1:n].newGear[1:1:1]
    -> merge[1:1:1].newGear[1:1:n];
  connect controller[1:1:n].acceleration[1:1:1]
    -> merge[1:1:1].acceleration[1:1:n];
  connect controller[1:1:n].brakeForce[1:1:1]
    -> merge[1:1:1].brakeForce[1:1:n];
}
```

Figure 6.33.: Example code snippet how to transform *EmbeddedMontiArc* code to the normalized *EmbeddedMontiArc* version. The example is a snippet from Figure 6.31 and Figure 6.32.

Figure 6.32 shows the *EmbeddedMontiArcParsing* grammar extending the *EmbeddedMontiArcTooling* one to add support for nice syntactic sugar. Line 4 in Figure 6.32 adds support to write `ports in B in1, B in2` in *EmbeddedMontiArc*. Line 7 enables to omit the dimension during component instantiation when it is one. Line 9 enables omitting the `subIndices` when they are `[1:1:1]` as well as the `this` keyword when not using `.*` or `.**` (this is checked via a context condition). Lines 10 and 11 add the index- and name-based connection patterns to *EmbeddedMontiArc*.

Line 13 adds the `all` range to the *EmbeddedMontiArcParsing* grammar (e.g., `connect x[:] -> y[:]`) as well as `step` can be ignored when it is one.

The only thing which is left is to translate the abstract syntax of *EmbeddedMontiArcParsing*'s syntactic sugar version to the abstract syntax of the *EmbeddedMontiArcTooling* version. Figure 6.33 serves as demonstration example how to transform nice *EmbeddedMontiArcParsing* code to its version of the tooling grammar. Taking a first look at the difference between *EmbeddedMontiArcParsing* and *EmbeddedMontiArcTooling* in Figure 6.33 unveils that the syntactic sugar really makes daily life much more comfortable when writing *EmbeddedMontiArc* code.

The following part of this section presents two transformations: First, `[:]` to `[1:1:$end]`; and second, `.*` to the unfolded long version. Figure 6.34 shows the two class diagrams of the abstract syntax of both languages; the bold text on the left side marks the difference on the abstract syntax introduced by syntactic sugar.

Figure 6.35 shows the first *OCL* expressions to formalize the transformation of the *EmbeddedMontiArcParsing* abstract syntax to the one of *EmbeddedMontiArcTooling*. These three *OCL* constraints have a concrete structure so that the *OCL2Trafo* generator is able to generate Java code manipulating the data structure. All classes of *EmbeddedMontiArcParsing* are post-fixed

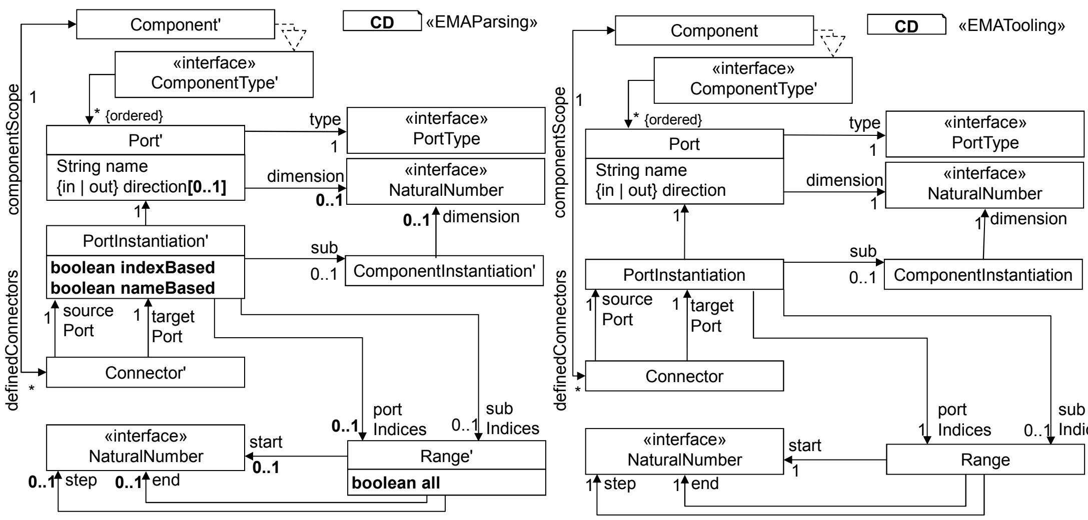


Figure 6.34.: Class diagrams of abstract syntax of *EmbeddedMontiArcParsing* with syntactic sugar (left) and *EmbeddedMontiArcTooling* (right). The classes in the left are marked with an apostrophe to differentiate between syntactic sugar and tooling classes of *EmbeddedMontiArc*'s abstract syntax. The generated classes of both class diagrams have the same short name, but different full-qualified names as they are in different packages.

with an apostrophe to differentiation between both class diagrams; the implementation differs by the package names. To improve readability the associations belonging to the syntactic sugar class diagram are also post-fixed with an apostrophe in *OCL*.

The context consists always of the pair which should be transformed. The expression `context Range rn, Range′ rs inv:` transforms the `rs` variable to `rn`. The generated Java code is a function `Ranges.setRange(Range rn, Range′ rs)` and a visitor with the method `void visit(Range′ rs)` creating a new `Range` object using the `Ranges.setRange` method. The `visit` method does this transformation step. After the context is always an equal (==) or similar[7] (`~~`) plus an iff (<=>) expression. The *OCL2Trafo* generator produces `if-else` Java expressions for `condition implies result` *OCL* ones (cf. ll. 7f). Line 7 handles the `[:]` case, lines 8 and 9 handle the "normal" case. The result is always on the left side: Lines 8 and 9 are translated to `Ranges.setStart(rn, rs.getStart()); Ranges.setEnd(rn, rs.getEnd()); Ranges.setStep(rn, rs.getStep() .orElse(NaturalNumbers.of(1)));`. `Ranges` and `NaturalNumbers` are helper classes for `Range` and `NaturalNumber`; the tool uses the Guava notation. These helper classes are generated to not overwrite the set methods of the abstract syntax generated by *MontiCore* or the adapted handwritten ones using the *TOP* mechanism.

[7]The similar expression has the same functionality as the equals expression. However, the equals expression checks via context conditions whether the left and right side type are compatible to detect typos in *OCL* which accidentally evaluate equals expressions always to `false`. Since `Range` and `Range′` are not in relation at all, these both types are not compatible. Therefore, the *OCL* expression uses `rn ~~ rs` instead of `rn == rs`.

transforms rs (source) to rn (target) -- or rn (target) is based on rs (source)

rn (target) should be equivalent to rs (source)

```
OCL
1  context Range rn, Range' rs inv:
2   let pi' = rs.portInstantiation';
3     dimension' = pi'.portIndices' == rs ? pi'.port'.dimension' ?: 1 :
4                                           pi'.sub'.dimension' ?: 1;
5   in
6     rn ~~ rs <=>
7     (rs.all' implies rn.start == 1 && rn.step == 1 && rn.end == dimension')&&
8     (!rs.all' implies rn.start == rs.start' && rn.end == rs.end'   &&
9                                           rn.step == rs.step' ?: 1)
```

case distinction

property of rn (target) equals number or property of rs (source)

```
OCL
10 context Port pn, Port' ps inv:
11   let ports' = ps.componentType'.ports';
12   in
13     pn ~~ ps <=>
14     pn.name == ps.name' && pn.dimension == ps.dimension' ?: 1 &&
15     pn.direction == ps.direction' ?:
16                     ports'[ports'.indexOf(ps) - 1].direction' ?: IN
```

```
OCL
17 context PortInstantiation pin, PortInstantiation' pis inv:
18  let
19    r1' = List{Range' r | r.start' == 1 && r.end' == 1 && r.step' == 1}[0];
20   in
21     pin ~~ pis <=>
22     pin.port == pis.port' && pin.sub == pis.sub' &&
23     pin.portIndices ~~ pis.portIndices' ?: r1'    &&
24     pis.sub.isPresent implies pin.subIndices ~~ pis.subIndices' ?: r1'
```

Figure 6.35.: *OCL* rules to express the transformation from *EmbeddedMontiArcParsing* to *EmbeddedMontiArcTooling* for `Range`, `Port`, and `PortInstantiation`.

Lines 10 to 16 show the *OCL* expression to transform the short to the long version of the port abstract syntax. Line 10 generates the method `Ports.setPort(Port pn, Port' ps)`. Line 16 shows the advantage of *OCL* to formulate the transformation: Similar to normal *OCL* constraints, the *OCL* code supports automatic flattening when navigating through associations and it handles all error cases automatically. Therefore, the inference of the missing direction with a default value can be easily described without carrying about an *out of bounds* exception as in Java. An error in *OCL* evaluates to `false` or to an empty set, and so the else part of the Elvis operator in line 16 is used.

Lines 17 to 24 transforms the short to the long version of the port instantiation. Line 19 is valid *OCL* code, and because the result `r1` is stored into `pin.subIndices`, the *OCL2Trafo* generator creates an object with this property if it does not exist. Whereas the normal *OCL* verifier code would evaluate line 24 to `false` if no Range `r` in line 19 exists, because `[0]` throws an error; *OCL2Trafo* handles this error situation by creating the suited object if needed. Lines 23 and 24 transforms `sub.port` to `sub[1:1:1].port[1:1:1]`.

Figure 6.36 shows one of the most complex transformations. The *OCL2Trafo* generator produces `if - else` Java expressions for `condition implies result` *OCL* ones (cf. l. 10 and 12). The expression `!or ps.indexBased` is equivalent to `!(sps.indexBased`

OCL

```
1  context Connector cn, Connector' cs inv:
2    let spn = cn.sourcePort;
3        sps' = cs.sourcePort';
4        tpn = cn.targetPort;
5        tps' = cs.targetPort';
7        ps' = {sps, tps}
8    in
9      cn ~~ cs <=>
10     ( (!or ps'.indexBased') && (!or ps'.nameBased') implies
11       spn ~~ sps' && tpn ~~ tps' ) &&
12     ( or ps'.nameBased' implies
13       let sCmp' = sps.sub'.parent';
14           tCmp' = tps.component';
15           names' = sCmp'.ports'.name'.retainAll(tCmp'.ports'.name');
16           sPorts = {Port sp | sp.name in names', sp.component ~~ sCmp'};
17           tPorts = {Port tp | tp.name in names', tp.component ~~tCmp'};
18           cons = {Connector c | c.sourcePort in sPorts,
19                                 c.targetPort in tPorts,
20                                 c.sourcePort.name == c.targetPort.name};
21           compN = List{Component cmp | cmp ~~ cs.componentScope}[0]
22        in
23          compN.containsAll(cons); // OCLTrafo generator changes it to addAll
24     ) /* && ... indexBased case similar to nameBased one: use position */
```

Figure 6.36.: *OCL* code to specify the normalization of the Connector.

`|| tps.indexBased)`; the `or` operator connects all elements in a set with the `||` operator. Using this set `or` operator leads to shorter and better readable code.

Line 11 says that the source and the target ports are similar when no special connector pattern is used. The expression `spn ~~ sps && tpn ~~ tps` use the previously generated `setPort` method twice: `Ports.setPort(spn, sps)` and `Ports.setPort(tpn, tps)`.

Lines 13 to 21 specify the transformation for the name-based connection pattern. The `names` variable contains port names which contain the source and target component definitions. In the example in Figure 6.33, names is equals to `{"newGear", "acceleration", "brakeForce"}`. In the example, `sPorts = {VelocityController.newGear, VelocityController.acceleration, VelocityController.brakeForce}` and `tPorts = {Merge.newGear, Merge.acceleration, Merge.brakeForce}`. Since these connections defined in lines 18 to 20 do not exist, *OCL2Trafo* creates them. Line 21 selects the component `cmpN` being identical to the one containing the name based connection. Line 23 adds all these connectors to the component `cmpN`, because `containsAll` is always satisfied when these elements are added before.

The normal *OCL* verifier can use these constraints to check whether the transformations are executed correctly, e.g., to test the *OCL2Trafo* generator.

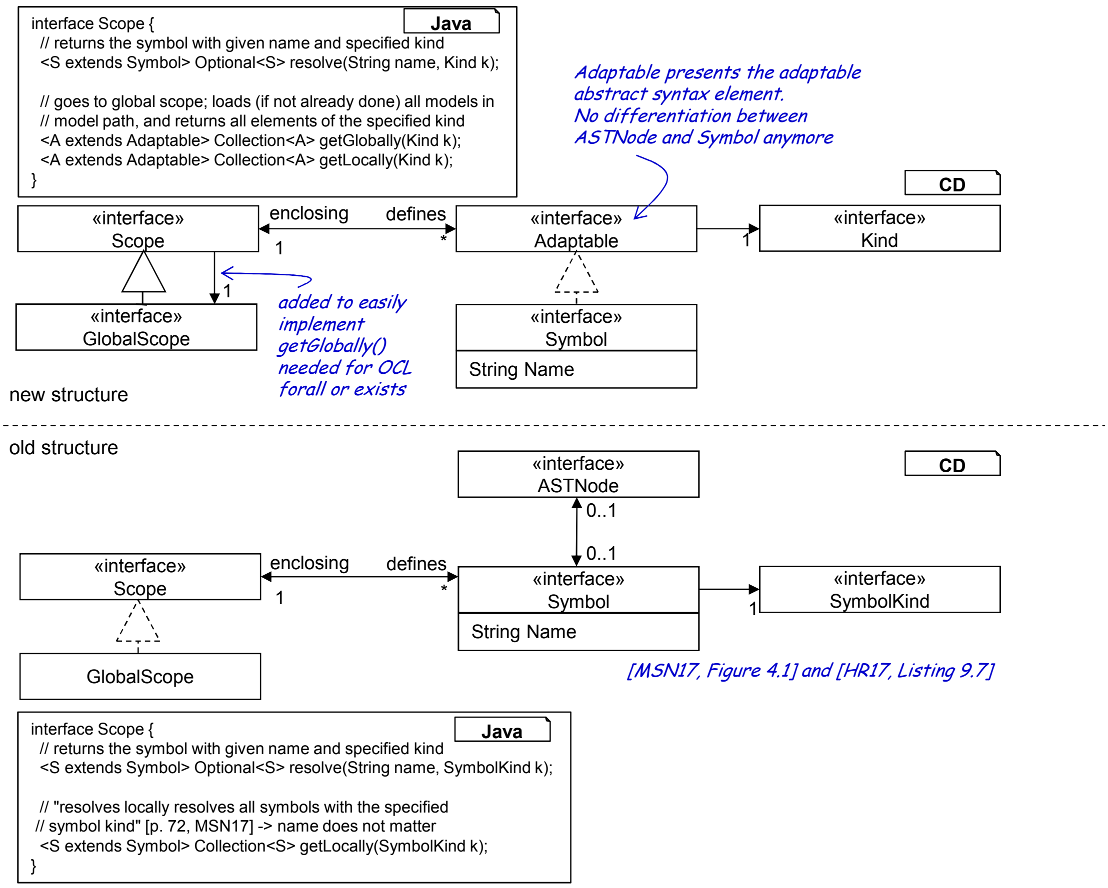


Figure 6.37.: Structure of new *MontiCore* version (top); and old *MontiCore* version, e.g., *MontiCore* 4.

6.5. Some Remarks about the Implementation

Previous *MontiCore* versions, e.g., *MontiCore* 4, differ between the abstract syntax tree and symbol elements of the abstract syntax. The implementation of Nazari [MSN17] only allows to adapt symbol elements; the new *MontiCore* version supports to adapt all abstract syntax elements. For this reason the `Adaptable` interface is decoupled from the `Symbol` interface in the top part of Figure 6.37.

Figure 6.37 shows the structure of the old *MontiCore* version and the structure of the new version. As the bottom part shows, only symbols are (locally and via `GlobalScope` globally) resolvable. Therefore, languages using this resolving mechanism define for each `ASTNode` a corresponding Symbol, where the name is the empty String when the `ASTNode` is not actually a symbol and has no unique name.

The *OCL* generator is able to handle both structures, the new and the old ones, of *MontiCore*. For the new structure, the developer must do nothing. For the old structure, the developer must

TagSchema

```
import de.monticore.umlcd4a.*;

tagschema OCL2JavaTags {
  tagtype SymbolName:String for CDType;
  tagtype ASTName:String for CDType;
  tagtype AssociationName:String for CDAssociation;
}
```

Figure 6.38.: Tag schema to define the mapping for classes of the abstract syntax to Java names.

specify how the classes of the abstract syntax are mapped to Symbol or ASTNode elements. This mapping enables the generator to resolve the corresponding associations by invoking the specified classes and get methods. Figure 6.38 defines the tag schema to enrich classes with this information. The new version of the *OCL* code generator does not use configuration files as bridge anymore as published in [MMR+17]; it uses tag models as explained above.

In contrast to Java, *OCL* when used as specification language has the ability to navigate against the navigation direction of associations. For this example association association [*] ComponentInst -> Component [1], the *OCL* generator rewrites the expression context Component inv: !**this.componentInst**.isEmpty to context Component inv: !**{ ComponentInst ci |ci.component == this}**.isEmpty. The *OCL* expression ComponentInst ci is mapped to the following Java code: for (ComponentInst ci : getEnclosingScope().getGlobally(ComponentInst .KIND)).

The most complex part of *OCL* is the type inference mechanism based on given class diagrams. The type inference mechanism must flatten Collection<Collection<X>>, Collection<Optional<X>> and Optional<Collection<X>> automatically to Collection<X> plus Optional<Optional<X>> to Optional<X>, and it must also infer types defined in nested sets. It is so complex, because most *OCL* constraints only define the types in the context clause; also most variables introduced with the let - in construct do not provide any further type information.

The *OCL* language has been extended to support units, e.g., context Person: size < 160 cm && weight > 120 kg implies fat. Thus, *OCL* must be able to infer the types of many constants, e.g., 160 cm has the Java type Number<Length> extending JScience Amount<Length> and 120 kg has the type Number<Mass>. The type inference algorithm must also evaluate all expressions, i.e., 160 cm + 120 kg throws an error; but 160 m / 2 s has the type Number<Velocity>.

Additionally, the *OCL* language supports calculation with plus and minus infinity as shown in various *OCL* expressions. This enables defining no limit in constraints, esp., needed for extra-functional properties. Similar to mathematics, the following rules hold for n as arbitrary number, but n is not plus or minus infinity: oo + n == oo, oo * n == n < 0 ? -oo : +oo, -oo + n == -oo, -oo * n == n > 0 ? -oo : +oo, n / oo == 0, n / -oo == 0, n < oo == true, and n > -oo == true. However, -oo + oo and oo / -oo result in an error.

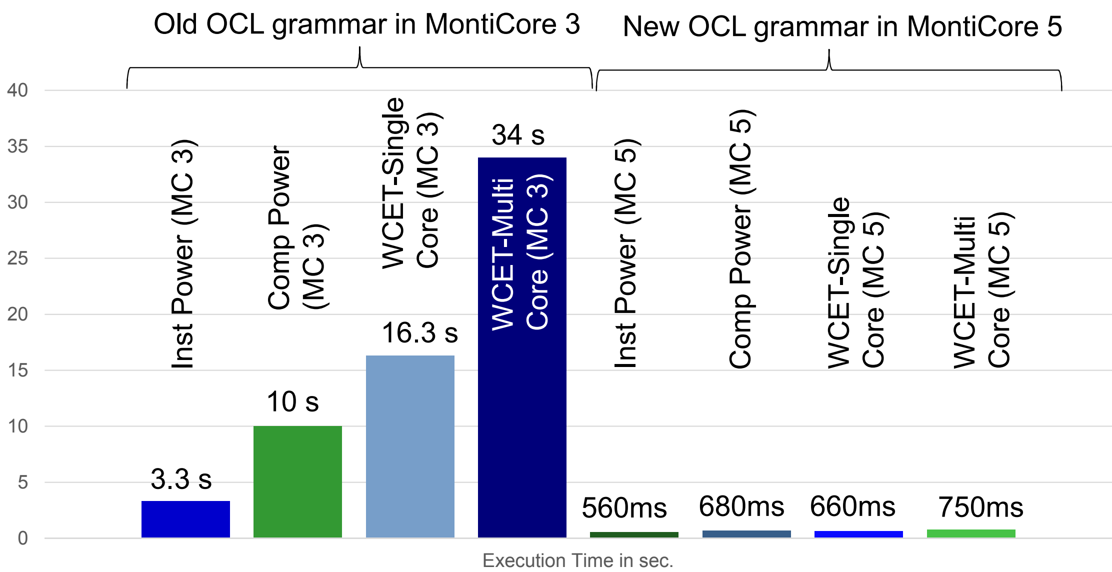


Figure 6.39.: Improved parsing speed of optimized *OCL* grammar. `InstPower` is the *OCL* rule defined in Subsection 6.2.2, `CompPower` is the *OCL* rule defined in Subsection 6.2.6, `WCET-Single Core` is the *OCL* rule defined in Subsection 6.2.10, and `WCET-Inf Core` is the *OCL* rule define in Subsection 6.2.11.

Besides adding unit, infinity, and type inference support to the *OCL* language, also the grammar file has been improved a lot. First, the `OCL X {` construct is now optional; hence, the concrete syntax of *OCL* is 100% compliant with the rules specified by Rumpe [Rum11, Rum16]. Now, the *OCL* language supports more operations (esp., new set operations such as `and`, `or`, `intersection`, `union`, `max`, `min`, `sum`, `prod`) to avoid the imperative `iterate` statement whenever it is possible; Section C.2 on page 370 summarizes all *OCL* operators according to their priority in a table. Figure 6.39 shows how the parsing speed of *OCL* text files improved a lot during the grammar refactoring. For the WCET-Inf Core constraint shown in Figure 6.39, the speed-up factor is 45! The new *OCL* grammar does not contain epsilon transitions, i.e., *MontiCore* rules matching empty input; e.g., `OCLContextDefinition = Type? varNames:(Name || ",")* ("in" Expression)?` has been refactored to `OCLContextDefinition = Type | Type? varNames:(Name || ",")+ ("in" Expression)?`.

Additionally, the new grammar does not contain optional of empty lists anymore if there is no concrete syntax between optional and an empty list, because `Rule1 = Rule2*` and `Rule3 = "word" Rule1?` causes a lot of back tracking for the *ANTLR* parser: For `Rule3` as start rule, maps `word` as input to `ASTRule1 = Optional.of(Collections.emptyList())` or to `ASTRule1 = Optional.empty()`.

6.6. Related Approaches to Constrain Structure of Architectures

ACME [GMW00b] specifies the structure of architectures via first-order predicates. Cichetti et. al. use the Epsilon Validation Language [KRGDP18] to define validity constraints. The Epsilon Validation language and the *ACME*'s first-order predicates provide similar features as our *OCL* framework. Some commercial *UML* tools (e.g., MagicDraw, Poseidon, XMF-Mosaic) also provide *OCL* support [GBR07].

Dresden *OCL* has an OCL2SQL and OCL2Java generator as well as a runtime interpreter to interpret all objects created during model execution [DW09]. Dresden *OCL* can also be used to animate stateful models; e.g., to describe which parts in a graphical editor should be highlighted. Dresden *OCL* also supports to query relational data bases [KPP06c]. Besides Dresden *OCL*, there exists UML2NoSQL to map *UML/OCL* code to graph database frameworks to check consistency of unstructured no SQL data bases [BCD$^+$16].

Dresden *OCL* has support for EOL (epsilon object language). EOL also supports defining variables, it mixes *OCL* constructs with constructs defined by other languages such as C++ or Java [KPP06b]. This is very similar to our used *OCL/P* language. EOL also only uses the dot operator to not differentiate between arrow or dot one as in *OCL* anymore. In contrast to *OCL/P*, EOL supports to manipulate data with the `:=` operator. EOL also enables reusability by importing constraints. Our paper *Encapsulation, Operator Overloading, and Error Class Mechanisms in OCL* [BRvW16] also presented a concept how to define *OCL* libraries and how to reuse operators in *OCL/P*. The current language implementation does not support to define operators in *OCL* directly; the *OCL MontiCore* grammar format must be extended to add new operators. Based on EOL there exists Epsilon Merging Language to define how models are merged.

Gogolla et. al. [GBR07] present USE (*UML*-based Specification Environment) to define *UML* diagrams. USE checks consistency between these diagrams via *OCL*, which is a very similar approach to our context condition checks defined in *OCL*. USE has an evaluation browser used as *OCL* expression debugger. USE additionally checks the consistency of models and constraints to identify contradicting *OCL* constraints [GBR07]. Hilken et. al. uses USE to specify derived properties [HSSG16]. The *OCL* verifier tool also supports inferring derive attributes.

In contrast to *OCL*/P, Dresden *OCL* and USE's *OCL* have, similar to OMG *OCL*, a four value logic. Therefore, evaluating a navigation chain fails when one of the objects is not present. This leads to many unwanted problems evaluating Boolean expressions. Some papers, e.g., *Safe Navigation in OCL* [Wil15] address this problem.

F-OML [BBD$^+$16] is a constraint and query language to define design patterns, reasoning about *UML* diagrams, and specification of DSLs. The PathLP part of F-OML supports smart querying, e.g., `?C.student[?S].name` bounds an object `c` to the variable `?C`, and binds `?S` to an object who is a student of `C` and it returns its `name` [BK11]. The `let-in` construct in *OCL/P* expresses the same content, but its code is much longer as all variables must be explicitly defined.

Herrera et. al. uses *OCL* as bridge from concrete to abstract syntax [HWP15]. They reformulate the "problematic" parts of the concrete syntax. This approach is similar to our OCL2Trafo one mapping the syntactic sugar *EmbeddedMontiArcParsing* grammar to the *EmbeddedMontiArcTooling* one. Jounault et. al. [JB16] also specifies transformation via *OCL* and generate incremental transformation code in Java.

Ahmed et. al. created textual constraint language for the *common data model* - an abstract data model for scientific data sets to be constrainted by *OCL* [AVKB14]. CdmCL translates the new language to *OCL*. A similar approach is done with our C&C architecture specification language in the next chapter. The syntax of it is also translated to *OCL*.

For *OCL* exist many tools to transform *OCL* to any solver, mostly any SMT solver such as Z3, CVC4 (cf. [ADEM14]) or mapping *OCL* to Haskell for its functional interpretation [CV16]. In a bachelor thesis, Nicolai Strodthoff also evaluated validation performance when translating *OCL* to Microsoft's Z3 solver and when generating unoptimized Java code (e.g., `forall` or `exists` expression are translated to `for` loops without any `break` statements). For plain structural analysis working on scopes (using locality constraints), the Java approach was much faster. This thesis even generated optimized SMT code (7 versions; first one is most readable one with own data types of Z3, and last one only uses integers and also bit patterns for quantifiers are generated). The evaluation of simple constraints, cf. constraint B1 in Section 7.1.4, for a simple model with only 59 components and 126 symbols (ports and components) needed between 15.17 seconds (version 7) and nearly 140 seconds (version 4). In contrast, the execution of the generated Java code to check this constraint needed less than 0.01 seconds, even for models with over 600 symbols. For further information, see bachelor thesis of Strodthoff [Str17].

Longuet et. al. [LTW14] model class diagrams and *OCL* in Isabelle jEdit and proof the constraints via Isabelle. However, they needed at least 9 GB of RAM to verify a constraint for 56 classes. If a model has 90 classes, Isabelle needs 28 GB RAM to verify constraints over it.

Grunske [Gru07] observed the need for a general language to formulate different extra-functional property types. Arjona et. al. define security constraints via *OCL* and translate them to CVC4 [ADEM14].

Cicchetti et. al. present for ProCom a value context condition language, a weaker version of our *OCL* framework, to define validity conditions and identify possible threads. The results are used to recalculate the satisfaction of extra-functional properties incremental. Cicchetti et al. [CCLS11] supports evolution of extra-functional values, and based on their change history, the algorithm suggests what components need to be updated.

Leveque and Sentilles [LS11] present refinement of extra-functional properties through instantiation and subtyping of components. Engineers can use *OCL* constraints to filter extra-functional attributes of components. Sentilles [Sen12, p. 88] uses only a simple selection and filter language supporting `and` conditions plus simple `if - else` statements. The *OCL/P* framework in this thesis, invented by Rumpe [Rum16], or OMG *OCL* [OMG05] support quantifiers and more complex set operations. The here presented mathematical framework with selection, aggregation, and comparison enables defining C&C-specific *OCL* constraints for consistency rules beyond simple refinement relations of extra-functional properties as presented in Sentilles et. al.

Defour et. al. describe Quality of Service extra-functional properties via a constraint logic programing language using *OCL* pre- and post-conditions [DJP04].

The combination of *OCL* with *FreeMarker* to define useful error messages for *OCL* constraints is new. The same holds for the mathematical structure of *OCL* constraints for extra-functional properties to generate useful consistency and inconsistency C&C witnesses. The author of this thesis is not aware of such a similar approach. Another highlight rarely present in existing constraint languages is the integrated unit support.

Chapter 7.

EmbeddedMontiView: A High-Level Design Language for Component and Connector Models of Embedded Systems

The previous chapter elucidated how to formulate structural and extra-functional properties of the component and connector (C&C) language *EmbeddedMontiArc*. Section 6.1 defined generic structural properties (also called context conditions or well-formedness rules) of *EmbeddedMontiArc* via *OCL*. This chapter introduces *EmbeddedMontiView*, a C&C design language, to specify architectural design decisions of embedded and cyber-physical systems.

EmbeddedMontiView is a C&C view language extending the abstraction concepts of Maoz, Ringert, and Rumpe [MRR13, MRR14, Rin14] and the functional net modeling approaches of Grönniger, Kriebel, and Rumpe [GHK+07, GHK+08a, GHK+08b]. In general, a C&C view addresses one specific concern of a large C&C model. C&C views serve as a layer between high-level textual requirements in *IBM Rational DOORS* [GHK+08a] and very large and complex logical architectures described as C&C models. The strength of C&C views is the ability to describe abstract relations between different hierarchy levels [MRR13]: For example, C&C views may skip C&C components and ports that are unimportant for a requirement. Furthermore, C&C views support connecting components directly with each other, even if they are no direct siblings in the corresponding C&C model.

A set of C&C views is called a structural specification of a C&C model; a C&C model satisfies a structural specification if and only if it satisfies all its C&C views. A C&C model satisfies a C&C view if and only if it concretizes all structural specifications defined in the C&C view: For example, if a C&C view introduces an input port with the name `p1` for the component `C1` and omits its port type, then the C&C model must contain a component with `C1` as type name and this component contains at least one input port with the name `p1`; the data type of the port in the C&C model does not matter as it is not specified in the C&C view. Section 7.4 presents this satisfaction relation, i.e., the satisfaction relation between *EmbeddedMontiArc* models and *EmbeddedMontiView* views. A component of a C&C model may occur in different C&C views, each focusing on a different concern of this component: For example, one view may specify important features/functionalities of this component by defining its (direct or indirect) subcomponents; and another C&C view specifies the interaction of this component with its environment (or other features) by focusing on its ports and its connections.

The *OCL* framework presented in the previous chapter also supports to define such structural architecture specifications on the abstract syntax of *EmbeddedMontiArc*. In contrast, *Embed-*

dedMontiView defines these constraints using (nearly the same) concrete syntax of *EmbeddedMontiArc*; this enables a much more intuitive and faster specification of design constraints. Furthermore, the concrete[1] structure (abstract syntax) of *EmbeddedMontiView* enables to improve the general witness creation algorithms of *OCL* to generate more helpful and user-friendly (non-) satisfaction models and natural text messages.

The structure of this chapter is the following: Section 7.1 lists all requirements and features of the C&C view language *EmbeddedMontiView*; it also discusses the new abstraction concepts (compared to previous publications [GHK+07, GHK+08a, GHK+08b, MRR13, MRR14, Rin14]) being added to *EmbeddedMontiView*. Section 7.2 presents related concepts for specifying and verifying architectural design decision. Section 7.3 introduces the concrete and abstract syntax of the *EmbeddedMontiView* language; it elucidates how *EmbeddedMontiView* extends *EmbeddedMontiArc* with new modeling elements to express underspecification (not known or unimportant information) of C&C models. Section 7.4 describes the binary satisfaction relation between *EmbeddedMontiArc* and *EmbeddedMontiView* in detail. Section 7.5 explains three kinds of witnesses: The satisfaction witness shows only the C&C model elements needed to reason why an *EmbeddedMontiArc* model satisfies an *EmbeddedMontiView* artifact; the tracing witness contains/highlights all C&C elements of an *EmbeddedMontiArc* model satisfying at least one abstract element in the *EmbeddedMontiView* artifact; and for each C&C view element - being not satisfied by the *EmbeddedMontiArc* model - its non-satisfaction witness contains C&C model elements violating this C&C view element's specification.

7.1. Requirements/Features on the C&C View Language

Component and Connector views as presented in several papers of Maoz, Ringert, and Rumpe [BMR+17a, MRR13, MRR14] introduce major abstraction mechanisms over hierarchy, connectivity, data flow, and interfaces. *EmbeddedMontiView* should support all features of the component and connector view profile of the C&C modeling language *MontiArc*. These features are (description is taken from [Rin14, Subsection 3.2.2 on p. 31ff.]):

- **Hierarchy abstraction**
  If one component is inside another one in a C&C view, then it does not necessary mean that the second one is a direct subcomponent of the first one. Rather, it means that the first component contains the second one, but not necessarily directly - i.e., the transitive closure of the subcomponent relation of the first component contains the second one. If two components are siblings in a C&C view, then it does not necessarily express that these both components are direct neighbors having a common parent; however, it specifies that neither of these two components contains (directly or indirectly) the other one. This abstraction enables to specify the hierarchical structure of an embedded system partially.
- **Connectivity abstraction**
  C&C views model abstract connections only. This means two elements connected via an abstract connector may not be directly connected with a single connector in the corresponding C&C model. An abstract connector expresses that these two components are

[1] compared to general and much broader expressiveness power of *OCL* constraints

connected via a chain of connectors (all transferring the same data). Whereas connectors in C&C models only connect ports, abstract connectors in C&C views may also connect components directly and abstract connectors may crosscut component hierarchies.

- **Incomplete interfaces**
  If not specified differently (cf. extension points), component interfaces in C&C views are incomplete. This means components in a C&C model may have more ports than specified by its C&C views. Additionally, C&C views may omit port data types or the port names.
- **Extension points**
  Engineers may add knowledge annotations to C&C views. For example, the stereotype `atomic` expresses that the component does not have subcomponents or internal connectors in any satisfying C&C model. Furthermore, the stereotype `interfaceComplete` specifies that the interface of an annotated component is complete, i.e., the component contains exactly the specified port names of the C&C view in any satisfying C&C model.

Additionally, *EmbeddedMontiView* should support the specification of abstract data flow[2] (this concept is already published in our C&C view case study paper [BMR+17a]):

- **Data Flow Abstraction**
  Effectors in a view describe data flow abstracting over chains of components (via their effectors) and connectors. In contrast to abstract connectors, the data passed from an abstract effector's source to its target may change. Effectors in component and connector models are only available to model data flow between input and output ports of atomic components. Effectors of atomic components in C&C models are not explicitly modeled, effectors are calculated based on the behavior implementations of atomic components. In contrast, abstract effectors in C&C views may connect any two arbitrary ports (even going from input to an output port of two different components).

All the previously presented abstraction concepts are only based on component and connector instance models described by the C&C view language profile *MontiArcView*. Since *EmbeddedMontiView* should provide abstraction concepts for all *EmbeddedMontiArc* language features (including unit types and matrix properties as port types, port and component arrays, as well as component types with interfaces), requirements representing major abstraction mechanisms for these new concepts are needed:

- **Support of Component Types**
  EmbeddedMontiArc supports component types, which can be instantiated several times. Thus, the C&C view language should not only support component instance names, but also component types. This requirement increases the complexity of the verification algorithm: The adapted algorithm must explore a much larger state space, because multiple components (having different names) may have the same component type. Since *EmbeddedMontiArc* also supports component interfaces, the component type may not even be unique: For example, the component interface type `Car` can be implemented by `A3`, `A4`, `X3`, and `SClass` component types.

[2] It is related to the German the phrases `Wirkkette` and `Wirkkettenanalyse`; e.g., cf. [AFBL14].

- **Unit Kind Abstraction**
  In an *EmbeddedMontiArc* instance model, the port data type or the matrix domain is completely specified by a minimum (which maybe minus infinity), a maximum (which maybe plus infinity), as well as a concrete unit (which maybe dimensionless) such as kilometer per hour. Therefore, port types in C&C views should support two abstraction kinds: First, omitting the port type at all as it is already possible in Ringert [Rin14]; and second, to specify only a unit kind instead of a concrete range with a concrete unit (e.g., `Velocity` as an abstraction of `(0 km/h :  250 km/h)`). In *MontiArc*'s Java type system this abstraction is identical by accepting a list of interfaces or super classes with or without generic bindings in *MontiArcView*; e.g., `ArrayList<String>` in *MontiArc* satisfies `Iterable & Cloneable` port type in *MontiArcView*.
- **Matrix Property and Dimension Abstraction**
  The C&C view language should abstract from the dimensions of tensors (also matrices and vectors) as well as it should describe matrix properties in an abstract manner. For example, a C&C view should specify an *underspecification parameter* `n` to specify with `port in (0 ms :  10 ms)^{n, n} inport` a quadratic matrix port type being an abstraction of `ports in (0 ms :  10 ms)^{10, 10} inport`.
- **Port Array Abstraction**
  C&C view interfaces cannot only abstract from port types and port names, but also from port array dimensions. A missing port array dimension in a C&C view is always an underspecification, whereas a missing port dimension in a C&C model always represents the default array dimension one as C&C models do not support underspecification.
- **Component Instance Array Abstraction**
  Component instances can be instantiated multiple times via arrays. Similar to port array abstractions, C&C views may describe component instance array dimensions in an abstract manner.
- **Support Anonymous References**
  A graphical model may contain an unnamed port object having two outgoing connections to two different components. Maoz, Ringert, and Rumpe [MRR13, MRR14, Rin14] do not support this use case, because ports used in abstract connectors automatically force a satisfying C&C model to contain this port name. *EmbeddedMontiView* should support schema variables (starting with a dollar sign) for referencing a concrete port object in the textual language without introducing a concrete port name. This way *EmbeddedMontiArc* allows to model all graphical C&C views. Transformation languages (cf. [HRW15]) use a similar concept.

7.2. Related Concepts for Verifying Component and Connector Models

As this thesis extends the C&C view concept and the verification process of Maoz, Ringert, and Rumpe; this concept builds on and refines their work. This chapter discusses the differences between our C&C view language *EmbeddedMontiView* and the *MontiArcView* language profile in detail. One of the biggest differences between *MontiArcView* and *EmbeddedMontiView* is that the

MontiArcView

```
1 <<view>> component UserButton {
2   <<interfaceComplete>> component UserButtonReader {
3     port <<untyped>> in button,
4          <<unnamed>> out UserInput;                    } }
```

EMV

```
5 view UserButton {
6   component UserButtonReader {
7     ports (c) in ? button,
8               out UserInput ?;                         } }
```

Figure 7.1.: Comparison of *MontiArcView* language profile and *EmbeddedMontiView* language (inspired by [Rin14, Listing 3.13, and Listing 3.14]).

second one is a completely separate language, whereas the first one only enriches the *MontiArc* language [Hab16, RRW15, RRW13a, HRR12] with stereotypes. Therefore, *MontiArcView* is bounded to the concrete syntax of *MontiArc*; *MontiArcView* just disables some of *MontiArc* context conditions and adds some new ones related to C&C views. As a result, *EmbeddedMontiView* supports much more underspecification: For example, due to parsing restrictions, *MontiArc* and thus also *MontiArcView*, do not support to define ports without port names and port data types. Additionally, stereotypes make the concrete syntax longer, and their restricted positions defined by the *MontiArc* grammar less intuitive.

Figure 7.1 highlights the differences between Ringert's C&C view specification and the one used in this thesis in a small example. The top part of Figure 7.1 (cf. ll. 1-4) shows the `UserButton` view defined in *MontiArc* enriched with the *MontiArcView* profile (cf. [Rin14, Section 3.6 on p. 40ff.], esp. [Rin14, Listing 3.13 and Listing 3.14] for more details on the concrete syntax). Line 1 needs to add the `component` keyword even though a C&C view is no component. Line 2 adds the «`interfaceComplete`» stereotype in front of the component keyword to mark that that this component defines all port names. Line 3 adds the «`untyped`» stereotype in front of the port direction to say that the name followed after the port direction is the port name and that the port type is skipped. The position of the stereotype is unintuitive, because in *MontiArc* the port type is defined between the port direction and the port name: so `in` «`untyped`» `button` would be a better choice. The same holds for line 4 defining an unnamed output port.

The bottom part of Figure 7.1 (cf. ll. 5-8) defines the same architecture specification in *EmbeddedMontiView* (cf. Section 7.3 for more details on concrete syntax). *EmbeddedMontiView* does not use stereotypes as it is a stand-alone language. Therefore, line 5 does not need the `component` keyword, as well as lines 7 and 8 can just replace the port type or the port name by question marks. The complete sign `(c)` (cf. [Rum16, Section 2.4] in class diagrams) specifies that the ports are completely specified. Due to *MontiCore*'s language extension features, *EmbeddedMontiView* can be as easy extended with new keywords as *MontiArcView* with new stereotypes. An advantage of extending *EmbeddedMontiView* with keywords instead of stereotypes is that newly added keywords directly appear in the generated abstract syntax of *EmbeddedMontiView* (or the new language extending it), whereas newly added stereotypes in *MontiArcView* are not directly visible - the Java code accessing the stereotype information must be scanned.

All work being related to C&C views of Maoz, Ringert, and Rumpe is also related to this chapter. A short list of "inherited" related work extended with new papers is (more detailed information is available in [Rin14, Subsection 3.7.5 on p. 47f]):

- Kruchten's 4+1 concurrent views [Kru95]. The 4 stands for the four views: logical, process, physical, and development; the 1 stands for the scenario model combining these different view kinds. Verdier et. al. [VST18] extends each of the four views in Kruchten's 4+1 views with platform-specific variability points to model product-lines efficiently. The views in this paper do not contain variability points; however, a separate feature model could select which views should be valid - this way views support product-line modeling, and additionally, the model artifacts are separated (cf. discussion in Section 5.2 for the advantages in separating product-line modeling and domain modeling).
- Runeson [RM14] presents an adaption of the 4+1 model for industry-academia collaborations; his four views are time (when), space (where), activity (how), and domain (what) - the plus 1 stands for the scenario view.
- *View-based Model-driven Software Development with ModelJoin* [BHK$^+$16] uses a DSL to define views declarative on existing meta-models. The views help to focus on parts of the meta-model. This approach differs from our one, as our approach does not use meta-models; our C&C view language defines views on concrete component and connector models. Another difference is that our C&C views are independent from the model; thus the views can be created before the C&C model. The declarative approach for *ModelJoin* needs references to an existing model. A commonality of this paper with this thesis is that both approaches use a human-readable DSL to specify views (cf. [BHK$^+$16, Listing 1]).
- The viewpoints in Taylor et. al. [MT10] specify different perspectives of design decisions related to a common concern.
- IEEE 1471 standard [Hil00] uses views to define a representation of a whole system according to a specific perspective related to a set of concerns.
- For Giese and Vilbig [GV06], architectural views represent a partial software of a C&C model to a particular concern.
- Clements et. al. [CGL$^+$03] add the relationship aspects of different aspects to view definitions.
- For Sabetzadeh and Esterbrook [SE06] views are a typed graph representing parts of an architecture.
- The *AADL* language [FG12] supports refinement of architectural elements.
- Chechik et al. [SFC12] present a mechanism for incomplete models. Chechik et. al. [SCFG15] use partial MAVO (may, abs, var, and OW partiality) models to express incomplete information, which can be refined to reduce uncertainty. The formal approach of Chechik et. al. [SCFG15] supports defining formal correctness conditions for partial model refinement transformations. Our C&C view approach is also a model refinement, because every C&C model is also a C&C view (cf. Subsection 7.3.11).

The C&C view language profile *MontiArcView* is inspired by functional net modeling of Grönninger, Kriebel, and Rumpe. In contrast to the approach of Maoz, Ringert, and Rumpe, C&C views of Grönninger et. al. also model environment elements in C&C views. The environment elements provides better understanding of closed loop controllers interacting with actuators than just considering the control part [GHK+07]. C&C views satisfaction relation can simply ignore environment components.

Besides environmental blocks, their views also support external blocks. Both blocks may not have a counter-part in the corresponding C&C model. The external block may be bought-in or is developed by a different department. "Non-signal communication is modeled by connectors that are connected to special ports in which 'M' represents mechanical influence, 'H' hydraulics, and 'E' electrical interactions." [GHK+08a, p. 3]. This separation of the influences of environment is not needed, because in the high-level design phase only the interaction should be modelled and the underlying physical law. If the brake works electrical, hydraulic or mechanical is uninteresting; if this decision has influence on the physical output of the brake, e.g., deceleration range, then this range should be modeled. Also the separation of external and environment is not needed, because an external component is logical the same as an environment component: it is modeled to understand the closed-loop, but it may not have a counterpart in the C&C model. Furthermore, a simulator must simulate both environment and external components; maybe external components are easier to simulate when a supplier delivers a DLL to the OEM.

Grönninger, Kriebel, and Rumpe introduce additional communication diagrams to views modeling the behavior of data-flow between connections in one concrete scenario. *EmbeddedMontiView* does not support behavior modeling; it focuses on the specification of structural design decisions.

Similar approaches to functional modeling are UML-RT [FOW01], *SysML* [OMG15], service oriented modeling of automotive systems [RFH+05, WFH+06], complex interface description including extra-functional properties [DVM+05], ATESST project based on *EAST-ADL* [GHK+08b].

Pittou and Tripakis [PMRT18] use multi-view modeling to describe the system under development by distinct models capturing different perspectives of the system. Reinecke and Tripakis [RT14, Subsection 3.2] interpret a view as an incomplete picture of a system. However, Reinecke and Tripakis only consider views for behavior and not for structural properties.

The Society of Automotive Engineers (SAE) Architecture Analysis & Design Language AS5506 [FLV06], provides a model-based development lifecycle including system specification (similar to views), their analysis as well as evolution of views via lifecycles.

Behere, Törngren, and Chen [BTC13] use views to describe conceptional and logical layers of reference architectures.

O'Reilly, Bustard, and Morrow [OBM05] use structures similar to views for team coordination. They present four different views concerning different tasks: (i) conveying effort, (ii) create a shared understanding of the context of different software pieces, (iii) track the implementation progress, and (iv) highlight conflicts during development activities.

A systematic literature review of Williams and Carver [WC10] unveiled five different logical views (dependency relationships, layers, inheritance structures, module decomposition, source structure) which are often suggested in literature to understand large and complex software projects.

Tools checking the consistence of Java/C/C++ software architecture projects create views to provide a better overview and understanding of large projects. Examples of such tools are the following:

- *ArchAngel* [OMB03] is a light weight architecture model describing the components of a system and their inter-relationship (containment and communication). They use a graph structure. "The main requirements of *ArchAngel* that have emerged so far are that it should: (i) support the building and maintenance of simple architectural descriptions; (ii) support the linking of an architectural description to an implementation; (iii) be proactive in determining whether or not an evolving implementation conflicts with the defined dependencies; and (iv) notify stakeholders (software engineers and architects) of inconsistencies that are detected" [OMB03]. *ArchAngel* also provides verification tools similar to our ones; but they do not create witnesses for satisfaction or non-satisfaction.
- *JDepend* [Mik17] is a free developer tool that can perform the same type of Java package constraint checking as the *ArchAngel* system.
- *Adele* [BEM93] provides a system model that is bound to the kernel of a software configuration management system.
- *Mae* is integrated into source code management environments [vdHMRRM01] to analyze the evolution of software architectures.
- The tools of [MMM02] and [SSC96b] check coding rules and compliance ones according to high-level design models.
- The *Architecture Alignment Checker* [MSN11] checks consistencies between Java implementations and their architectural descriptions specified in *MontiArc* (cf. [Hab16] for further information). Since this tool is based on an older *MontiCore* version, it needs a mapping language from Java to *MontiArc*. *EmbeddedMontiArc* supports this mapping via the adaption mechanism of the symbol management infrastructure [MSN17]. *EmbeddedMontiView* supports more abstraction mechanisms than the *Architecture Alignment Checker*.
- *RefJava* [Flo02] works similar as the *Architecture Alignment Checker*. However, it also detects, besides architecture inconsistencies, bad smells [MSN11].
- Passos et. al. [PTV$^+$10] present the dependencies of components in a quadratic dependency structure matrix instead of modeling it via associations as it is done in C&C views.
- Greifenberg, Müller, and Rumpe [GMR15] use a dependency constraint language for features of architectures. This language also enables to forbid architecture styles, e.g., bad design decisions. C&C views work in a similar way: positive views define dependencies via abstract connectors or abstract effectors, and negative views (skipped in this thesis, but they work the same as presented by Ringert [Rin14, p. 30]) forbid relationships between components. The dependency checker of Greifenberg et. al. is similar to our satisfaction verifier (cf. Section 7.4).
- *EVA: A tool for visualizing software architecture evolution* [NLM18] uses abstractions similar to our C&C views to present relations between modules, e.g., classes of a software component are inside one large circle, and different colors present the different groups of dependencies (effectors or connectors in our case). For different concerns, EVA uses different views, i.e., single-release architecture view, 3-D architecture-evolution view, and pairwise architecture-comparison view. C&C views (cf. Daimler Case Study on C&C views in Chapter 8) can also be used for software evolution.

EMV

```
1  view RedundandVelcoityControllerPorts {
2    enum Gear { D1 | D2 | D3 | D4 | D5 | D6 | D7 | R | N | P }
3    component RedundantVelocityController {
4      ports in  (0km/h : 250km/h) currentVelocity, // full specified port
5                ?                 wishedVelocity, // untyped port
6                Gear              ?, // unnmamed port
7            out ?                 ?, // untyped and unnamed port
8                (-oo m/s^2 : oo m/s^2) $acceleration, // anonymous port
9                ?                 $brakeForce; // untyped anonymous
10   }
11 }
```

Figure 7.2.: *EmbeddedMontiView* model showing underspecification of ports.

- The paper *An extensible benchmark and tooling for comparing reverse engineering approaches* [CN15] presents *UML* tools (generating *UML* class diagrams instead of *SysML* block diagrams) to analyze existing code bases. The best tools, all having 100% score in the class detection benchmark, are [CN15, TABLE IV]: ArgoUML [RVR+10], Astah Professional [Cha18], BOUML [Pag18], *Enterprise Architect* [Spa17], *Rational Rhapsody* [IBM18], and MagicDraw [No 18].

All these mentioned tools enable to create smaller viewpoints (e.g., only focusing on user interactions or class structures) based on a large software architecture. The witness extraction presented in Section 7.5, esp. the complete tracing witness in Subsection 7.5.2, creates viewpoints on an existing large architectural C&C model focusing on the important details of the viewpoint, i.e., in our case the specified C&C view.

7.3. Concrete and Abstract Syntax of *EmbeddedMontiView* Language

This section explains the concrete syntax of the *EmbeddedMontiView* language by examples. The *EmbeddedMontiView* syntax is similar to the *EmbeddedMontiArc* syntax. *EmbeddedMontiView* extends *EmbeddedMontiArc* by adding concrete syntax for underspecification. Additionally, this section presents the abstract syntax of *EmbeddedMontiView* and highlights the differences of the abstract syntax between *EmbeddedMontiView* and *EmbeddedMontiArc*.

7.3.1. Abstract Component Type Definition

Figure 7.2 shows how to define an abstract component type in *EmbeddedMontiView*. This figure is an abstraction of Figure 3.49 and of Figure 3.50 on page 87. All *EmbeddedMontiView* models start with the keyword `view` and a name (as shown in l. 1). In general, C&C views are small and focus only on a very specific part of a C&C model, and multiple small views specify one large C&C model. To reference a view later (e.g., for positive or negative witnesses) each view must have a unique name.

EMV

```
1  view RedundandVelcoityControllerPortsComplete {
2    enum Gear { D1 | D2 | D3 | D4 | D5 | D6 | D7 | R | N | P }
3    component RedundantVelocityController {
4      ports (c) // (c) stands for complete
5            in  (0km/h : 250km/h)       currentVelocity,
6                ?                       wishedVelocity,
7                Gear                    currentGear,
8                ?                       obstacleSpeed, // added
9                ?                       obstacleDistance, // added
10           out ?                       newGearMerged,
11               (-oo m/s^2 : oo m/s^2) accelerationMerged,
12               ?                       brakeForceMerged;
13   }
14 }
```

Figure 7.3.: Specify complete port interface in *EmbeddedMontiView*.

Analog to *EmbeddedMontiArc*, components in *EmbeddedMontiView* communicate with each other only via their component interface containing input and output ports. There are different abstraction levels to define abstract ports in *EmbeddedMontiView* (port array dimensions are handled later):

1. Line 4 specifies the port completely as it is done in *EmbeddedMontiArc*.
2. Line 5 specifies a port incompletely by omitting its data type.
3. Line 6 specifies the port incompletely by omitting its name; it means the component has at least one ingoing port with a `Gear` data type.
4. Line 7 specifies the port very abstractly by only presenting its direction; this means the component has at least one output port plus the output ports defined in lines 8 and 9, the name and the datatype does not matter.
5. Line 8 is similar to the third case. Both specify only the datatype. In contrast to line 6, line 8 defines the anonymous port name via a schema variable (starting with the $ sign), which is an anonymous placeholder to create connections to or from anonymous ports.
6. This case is similar to the fourth one; the data type is not specified and the name is an anonymous placeholder (cf. l. 9).

Since Figure 7.2 defines three anonymous output ports, a model satisfying this view has at least three outgoing ports. The view does not specify the names of the outgoing ports, so a port array with three elements matches these three ports defined in lines 7 to 9.

A `(c)` after the `ports` keyword, as shown in line 4 in Figure 7.3, specifies that the view defines completely all port names. Figure 7.3 specifies that `RedundandVelcocityController` has exactly five ingoing and three outgoing ports with these names. A specified port can also be a port array, thus the controller may have more ports, but the controller must not have another port with a name different from the specified one. *EmbeddedMontiView* does not support to specify ports with the compete symbol and to omit the port names (e.g., via schema variables or question mark signs for port names). Rumpe [Rum16, Section 2.4], [PFR01] already introduced the syntactical symbol `(c)` for complete information.

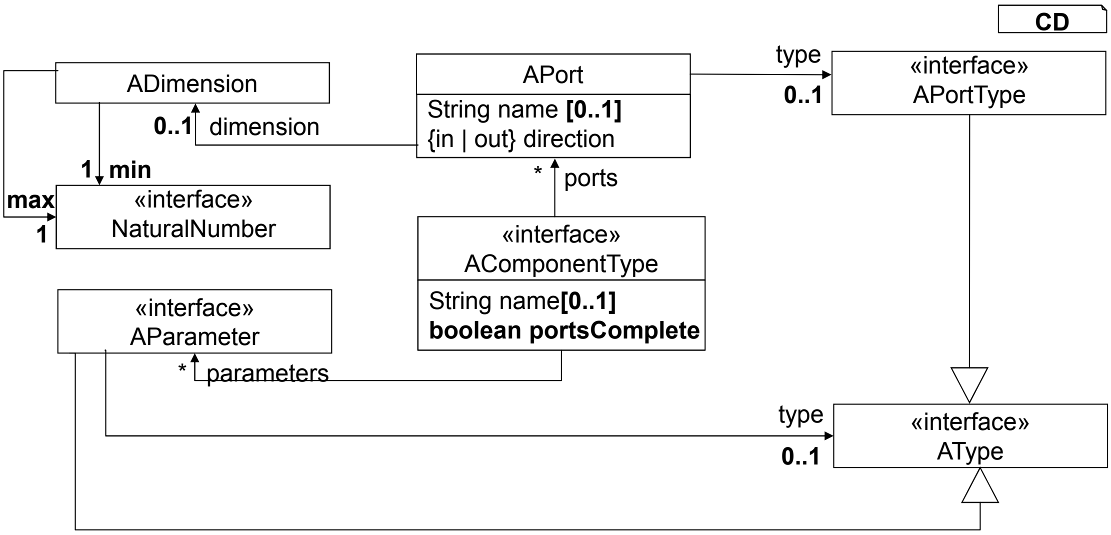


Figure 7.4.: Abstract syntax of abstract port class (`APort`) and abstract component type interface (`AComponentType`).

EMV

```
view WheelSensor {
   component Car {
     instance ? wheelSensor;
     instance ? controller;
     connect wheelSensor.airPressure -> controller;
  }
}
```

Figure 7.5.: Abstract component instantiations with unknown component type in *EmbeddedMontiView*. The red text highlights the differences between the internal structures of *EmbeddedMontiView* and *EmbeddedMontiArc*.

Figure 7.4 shows the abstract syntax of the abstract port class (`APort`) and the abstract component type interface (`AComponentType`). The bold text highlights the differences between *EmbeddedMontiArc* and *EmbeddedMontiView*. As shown in the examples in Figure 7.2 and Figure 7.3, a port may omit the port type, the dimension or the name (cf. question mark signs in *EmbeddedMontiView* examples). An abstract component type also may not have a name, see examples in Subsection 7.3.7. The optional dimension of an abstract port is an abstract dimension defining the minimum and maximum number of the port dimension, see examples in Subsection 7.3.3. An abstract component type may be marked as interface complete (cf. `(c)` sign after `ports` in Figure 7.3).

EMV

```
view RedundandVelocityControllerInstances {
  instance VelocityController vc1;
  instance VelocityController vc2;
}
```

Figure 7.6.: Abstract component instantiations in *EmbeddedMontiView*.

7.3.2. Abstract Component Instantiations

Similar to *EmbeddedMontiArc*, C&C views can also define an (abstract) hierarchical decomposition of component types as it is the case with the `Car` component type in line 2 in Figure 7.5. *EmbeddedMontiView* also supports to define component instantiations without knowing the component type of this instantiation as it is shown in lines 3 and 4 in Figure 7.5. The instantiation names are only local names of the view and may not match the instantiation names of a C&C model. The `WheelSensor` view specifies that there exists a `Car` component type which instances have (directly or indirectly) at least two different subcomponent instances and one of these instances has an output port `airPressure` being connected (directly or indirectly) to any input port of the other instance. The semantics of *EmbeddedMontiView* supports that there exist other component instances in a corresponding[3] C&C model between the `Car` instance and the `wheelSensor` instance. *EmbeddedMontiView* supports defining ports via connectors: Line 5 states that the `wheelSensor` instance has an `airPressure` output port with an unknown type. *EmbeddedMontiView* forces to write `this.$portName` when referring to an port of the enclosing component type. For example, `controller -> this.controlOutput` introduces the `controlOutput` port for the component type `Car`.

In contrast to *EmbeddedMontiArc* having exactly one top level component instance (cf. main component instantiation in Subsection 3.6.7), a C&C view may have multiple abstract component instantiations in its top level. Figure 7.6 shows such an example. The abstract component type `VelocityController` may not be defined in this C&C view. This view only specifies that there exist two C&C instances having the component type `VelocityController`, and that neither of these two instances is contained in the other one. The component type `VelocityController` can be defined in (multiple) other views. Since a model must satisfy all views, the semantics does not change whether `VelocityController` is completely defined in one or in multiple views.

7.3.3. Array of Abstract Component Instances and Abstract Ports

Analog to *EmbeddedMontiArc*, *EmbeddedMontiView* supports specifying dimensions for abstract port definitions or abstract component instantiations as shown in Figure 7.7. However, in contrast to *EmbeddedMontiArc*, where omitting the dimension as shown in lines 7 and 15 automatically sets the port or component instantiation dimension to 1, omitting the dimension in *EmbeddedMontiView* is interpreted as underspecification stating that the dimension is not

[3] This thesis uses the phrase corresponding C&C model in context of a C&C view, if and only if the C&C model satisfies this C&C view.

EMV

```
1  view Arrays {
2    component SensorProcessing {
3      ports in
4              // specification of port array dimension with at least 10
5              C signal[10],
6              // is the same as always, specification of port array size >= 1
7              C signal2,
8              // concrete specification of port array dimension (here it's 1)
9              C signal3[!1],
10             // define an allowed range
11         out (0m : 0.5m : 25m) distance[2-7];
12
13      instance Filter1 filter[3]; // minimum three Filter1 instances
14      // underspecification of instance array dimension (same as in line 17)
15      instance Filter2 filter2;
16      // minimum of one instance array dimension (here it is 1)
17      instance Filter3 filter3[1];
18      // maximum of one Filter4 component types (directly or indirectly)
19      instance Filter4 filter4[!1];
20   }
21 }
```

Figure 7.7.: *EmbeddedMontiView* example with arrays of abstract ports and abstract component instances.

known or not important. In *EmbeddedMontiView*, the dimension one (cf. `[!1]`) must be explicitly modeled as shown in lines 9 and 19. The exclamation in line 19 means that the component type `Filter4` exists exactly once in the transitive closure of all subcomponent instances of `SensorProcessing`. *EmbeddedMontiView* uses the exclamation mark before the array dimension number in the concrete syntax and not after the number, because `3!` can be confused with faculty of three (being six).

The differences of interpreting the omitted array dimension in the concrete syntax of *EmbeddedMontiArc* and *EmbeddedMontiView* result on the fact that *EmbeddedMontiView* is used in a design phase and *EmbeddedMontiArc* is used in the implementation phase of the logical architecture: In the design phase, all decisions should be modeled explicitly and the language takes as less as possible default interpretations; in contrast, the logical architecture contains no underspecification anymore and therefore, the default interpretations are syntactic sugar for frequent use cases. Every *EmbeddedMontiArc* model saved as *EmbeddedMontiView* artifact (i.e., by simple surrounding an *EmbeddedMontiArc* text with `view $Name {` and `}`) satisfies its own *EmbeddedMontiArc* model. The array dimensions are satisfied, because a missing dimension in *EmbeddedMontiView* is an underspecification, the missing dimension in *EmbeddedMontiArc* is an array of size one and an array dimension underspecification satisfies every *EmbeddedMontiArc* dimension.

EMV

```
1 view ViewWithCompleteInstances {
2   component FlipFlop {
3     ports (c) in B r, B s,
4               out B q, B notQ;
5     instances (c) Switch[2]; Not; Memory;
6   }
7 }
```

Figure 7.8.: *EmbeddedMontiView* example with complete component instances.

EMV

```
1 view ViewWithDirectInstances {
2   component FlipFlop {
3     ports (c) in  B r, B s,
4                out B q, B notQ;
5     instances direct Switch[2]; direct Not;
6   }
7 }
```

Figure 7.9.: *EmbeddedMontiView* example using the `direct` keyword.

7.3.4. Completeness of Abstract Component Instances

Similar to Figure 7.5 on page 223, marking an abstract component type as interface complete, the complete marker `(c)` also exists for instances as shown in Figure 7.8. The `(c)` in line 5 means that transitive closure of all subcomponent instances of `FlipFlop` includes two instances of the `Switch` component type, one instance of the `Not` component type and one instance of the `Memory` component type. For example, a C&C model having an instance of the `Not` component type inside an instance of the `Memory` component type also satisfies this model. If the keyword `instances (c)` is used, then no other `instance` or `instances` rules are allowed inside this component body. All array dimensions of instances defined after the complete sign are exact array dimensions. Therefore, line 5 is equivalent to `instances (c) Switch[!2], Not[!1], Memory[!1]`.

Figure 7.9 specifies that the `FlipFlop` component type directly contains the instances of types `Switch` and `Not`. The memory block (cf. Figure 7.8) can be represented by any component instance. The `direct` keyword forces that the C&C model `FlipFlop` directly contains (and not via other intermediate component instances) two instances of the `Switch` and one of the `Not` component type.

EmbeddedMontiView also enables defining a complete component hierarchy as shown in Figure 7.10. Line 5 says that the component instantiations of two `Switch`, one `Not` and one `Memory` component type are complete and also direct; therefore, these three component types must be atomic, too. To specify atomic components, not having any subcomponents themselves, explicitly, the `atomic` keyword exists; e.g., `atomic component Not`. The `atomic` keyword actually defines a negative view, specifying what is forbidden. Negative views are fine for C&C views verification, but they may cause runtime efficiency problems to C&C

EMV

```
view ViewWithNoUnderspecification {
  component FlipFlop {
    ports (c) in B r, B s,
              out B q, B notQ;
    instances (c) direct Switch[2]; direct Not; direct Memory;
} }
```

Figure 7.10.: *EmbeddedMontiView* example with no underspecification in component hierarchy.

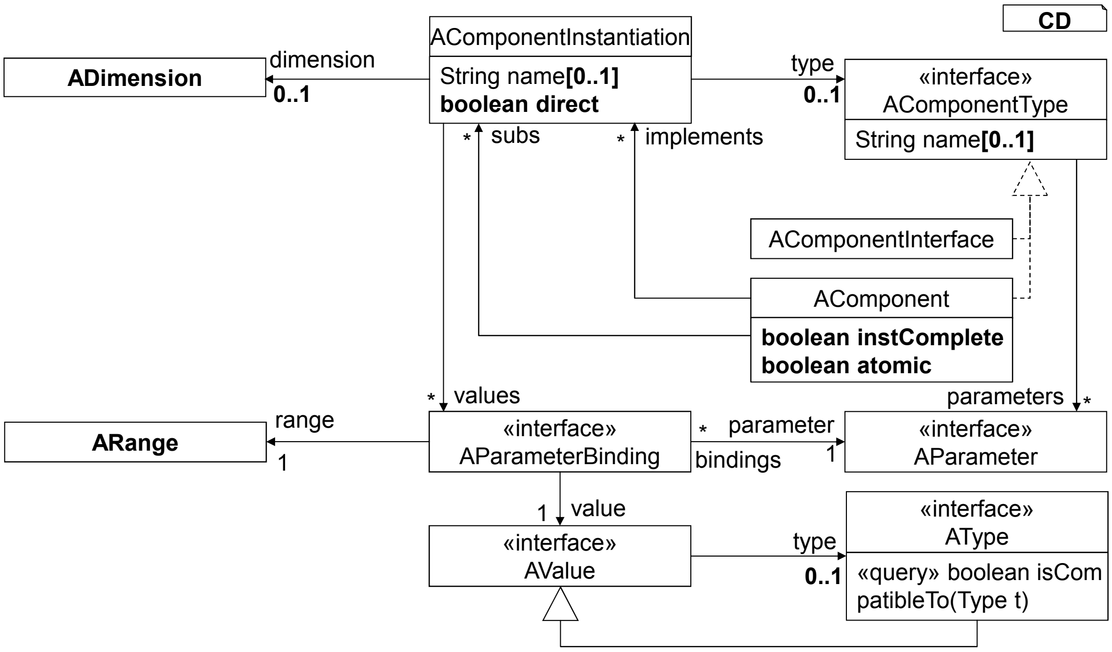


Figure 7.11.: Abstract syntax of abstract component instantiation class (`AComponentInstantiation`).

views synthesis. The Boolean flag `atomic` in `AComponent` is present to support all language features of C&C views of Ringert [Rin14].

Figure 7.10 only contains underspecification of the data-flow, as no connectors or effectors are modeled.

Figure 7.11 shows the abstract syntax of the component instantiation class (`AComponent-Instantiation`). The bold text highlights the differences of the abstract syntax between *EmbeddedMontiView* and *EmbeddedMontiArc*. As shown in Figure 7.5, an abstract component instantiation may not have an abstract component type. As shown in Figure 7.7 an abstract component instantiation may have an optional abstract dimension similar to abstract ports. As shown in Figure 7.8 an abstract component may not have a name and an abstract component may mark its abstract component instantiations as complete. As shown in Figure 7.9 a component

EMV
```
1  view TypeParameters {
2    component Max<T> {
3      ports in  T values,
4            out T maxValue;
5    }
6  }
```

EMA
```
7  component Max<T, N+ n=2> { /* copied from Figure 3.28 */
8    ports in  T values[n],
9          out T maxValue;
10 }
```

Figure 7.12.: *EmbeddedMontiView* model with abstract type parameters.

EMV
```
1  view TypeParameterInst1 {
2    instance Max<T=N+> naturalNumberMax;
3    instance Max<T=Q+> posNumberMax;
4  }
```

EMV
```
5  view TypeParameterInst2 {
6    struct Vec3 {R x, y, z; }
7    instance Max<T=Vec3> vecMax;
8  }
```

Figure 7.13.: Abstract type parameter bindings in *EmbeddedMontiView*.

instantiation may be marked as direct. As the next section shows abstract component instantiations also contain values binding abstract parameters.

7.3.5. Abstract Type Parameters

Component type definitions in *EmbeddedMontiView* may also contain type parameters as shown in Figure 7.12. The C&C view in lines 1 to 6 specifies what kind of type parameters are present, but the concrete C&C model as shown in lines 7 to 10 may have more type parameters. Type parameters in C&C views support to define library components in the design phase. The type parameter `T` in this example prevents that a non-generic (e.g., just for `N+`) maximum component is developed. Due to the underspecification of port arrays, the `max` component can be even more general by introducing another generic parameter `n` (cf. l. 7).

The usage of instances with type parameters introduces automatically component types having type parameters as it is shown in lines 1 to 4 in Figure 7.13. Thereby, no `is`-type is derived, because multiple views (cf. both views in Figure 7.13) use the defined component type. *EmbeddedMontiView* binds type parameters only via names (cf. ll. 2, 3, 7); binding via position as it is the case in *EmbeddedMontiArc* is not possible, since the parameter list in *EmbeddedMontiView* is incomplete.

EMV

```
1  view TypeParameterInst3 {
2    instance Max<T=N0> naturalNumberMax;
3    instance Max<T=Q+> posNumberMax;
4  }
```

EMA

```
5  /* n and T are switched on purpose for this example */
6  component Max<N+ n=2, T> {
7    ports in  T values[n],
8          out T maxValue;
9  }
```

EMA

```
10 component Controller {
11   instance Max<n=3, T=N0> naturalNumberMax3;
12   instance Max<4, Q+> posNumberMax4; // n=4 and T=Q+ see positions in l. 6
13 }
```

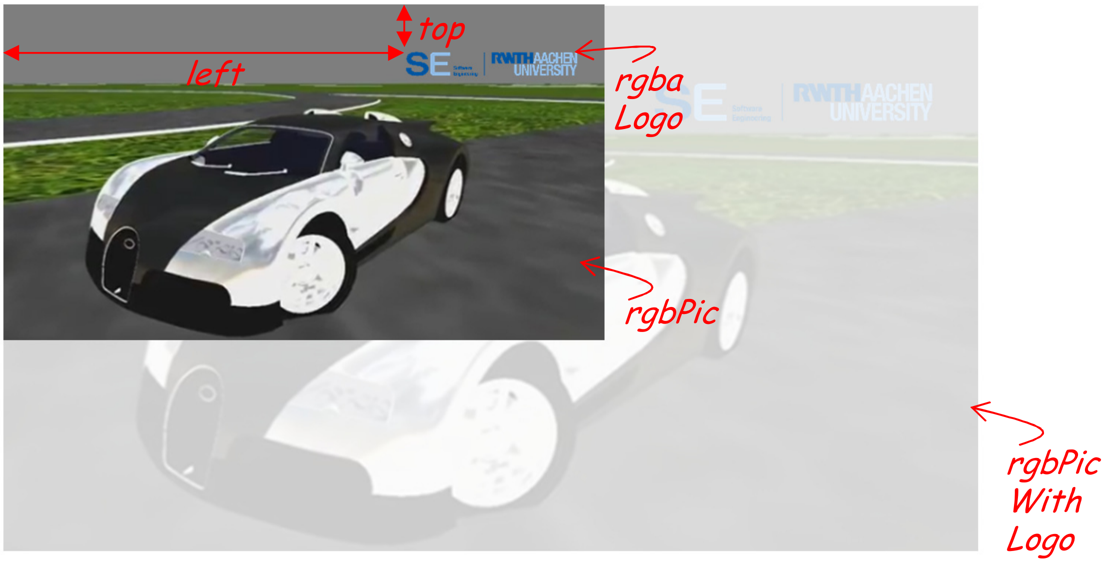
Figure 7.14.: *EmbeddedMontiArc* model satisfying parameter bindings.

The `Controller` component model in lines 10 to 13 in Figure 7.14 satisfies the `TypeParameterInst3` view in lines 1 to 4 in Figure 7.14, because the `Controller` contains two instantiations of a `Max` component type with the correct generic parameter bindings (l. 11 → l. 2 and l. 12 → l. 3). Please notice: The instance name is not important for subcomponents; the instance names are only needed to identify the correct connections of (sub)instances; and the concrete model may have more parameters (cf. parameter `n` and `T` in l. 6) and in a different order than the abstract view (only parameter `T` introduced indirectly in ll. 2, 3).

7.3.6. Matrices as Abstract Port Types

The view language supports defining underspecification parameters as shown in line 4 in Figure 7.15. These parameters express matrix or tensor dimension ratios such as 16:9 or 4:3 for TV formats. The ranges of these parameters may depend on previously defined parameters, cf. parameters `k` and `l`. Sure it would be a better strategy to model the `LogoAdder` with a generic parameter for reusability reasons. This simple and intuitive example should only demonstrate how *EmbeddedMontiView* abstracts from concrete matrix dimensions. The view does not specify concrete picture sizes, the C&C view only specifies that the image ratio is 16:9, and that the logo is smaller than the picture.

If the specification contains no constraints about matrix dimensions, then a question mark as shown in Figure 7.16 instead of underspecification parameters can be used. Please notice, that `Q` as data type in *EmbeddedMontiView* is no underspecification as it is the case with port dimensions. If the dimension is unknown, then the expression `Q^?` or `Q^{?, ?}` is used as data type. The reason for this decision is the fact that most port types of *EmbeddedMontiArc* and *EmbeddedMontiView* are single value ones.



EMV

```
view Matrix {
  component LogoAdder {
    // these parameters are no generics (only to express underspecification)
    underspecification params N+ n, (1:16*n) k, (1:9*n) l, N+ m;

    ports in (0:255)^{16*n, 9*n, 3} rgbPic, // 3 is for red, green, blue
             (0:255)^{k, l, 4} rgbaLogo, // 4 is for red, green, blue, alpha
             (1:16*n-k+1) left,      // left position where the logo starts
             (1:9*n-l+1)   top,      // top position where the logo starts
         // picture is rescaled automatically
         out (0:255)^{16*m, 9*m, 3} rgbPicWithLogo;
  }
}
```

EMA

```
component LogoAdder {
  ports in  (0:255)^{320, 180, 3} rgbPic,
            (0:255)^{100, 20, 4} rgbaLogo,
            (1:220) left,
            (1:160) top,
        out (0:255)^{1920, 1080, 3} rgbPicWithLogo;
}
```

Figure 7.15.: *EmbeddedMontiView* model with underspecification parameters for matrix dimension in ports.

EMV

```
view Matrix2 {
   component ProcessMatrices {
       ports in Q^? vector, // rational vector with unknown length
                Q^{?,?} matrix, // rational matrix with unknown dimensions
                tridiagonal invertible Q^{?,?} matrix,
           // any arbitrary tensor with unknown dimensions and data type
           out ?^{?, ?, ?} arbTensor,
   }
}
```

Figure 7.16.: *EmbeddedMontiView* model with unknown dimensions of vectors, matrices, and tensors.

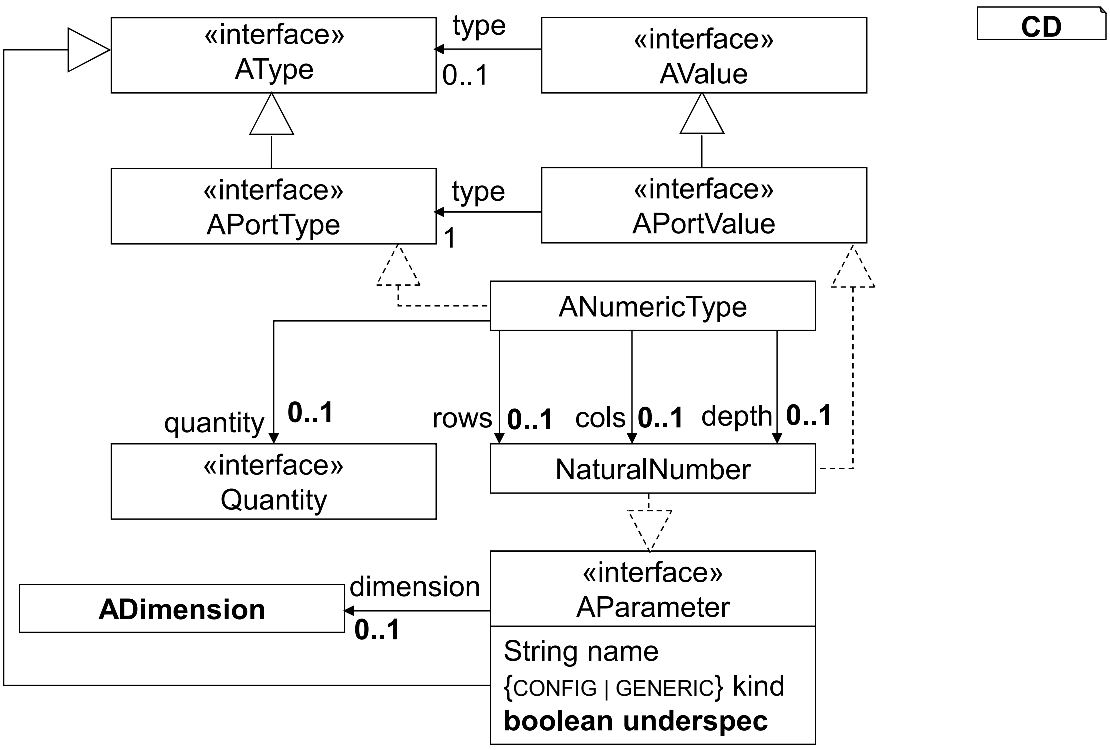


Figure 7.17.: Abstract Syntax of abstract numeric type.

Figure 7.17 shows the abstract syntax of the abstract numeric type. Its optional rows, columns, and depth are abstract natural numbers extending abstract parameters. Abstract parameters have an additional field `underspec` to define that these parameters are underspecified ones and that these abstract parameters may not exist as generic parameters in a component type of the C&C model. The `rows`, `cols`, and `depth` associations are empty, when the question mark operator is used. If the port type is `?^{?, ?, 2}`, then abstract numeric type has no quantity; whereas `Q^{?, ?, 2}` has the quantity dimensionless.

EmbeddedMontiView supports specifying matrix properties as shown in line 5 in Figure 7.16. Line 5 forces a quadratic matrix; the algebraic property `pseudo-invertible` forces that the domain of the matrix are real or complex numbers so that the *Moore-Penrose-Inverse* [Moo20, Pen55] matrix expression `A^+` exists ($A \cdot A^{+} \cdot A = A$ and $A^{+} \cdot A \cdot A^{+} = A^{+}$); the pseudo-inverse matrix A^{+} solves linear compensation problems used in logistics [CCF55].

7.3.7. Abstract Connections

MontiArcView, presented by Ringert [Rin14], always connects ports at the most outside-level. In *EmbeddedMontiView* connections can be inside every scope. The expression `component ParkingAssistant { connect this.signal -> filter; }` is mapped to `connect ParkingAssistant.signal -> ParkingAssistant.filter`.

EmbeddedMontiView also supports convenient syntactic sugar to focus only on the parts you want to specify. Lines 1 to 15 in Figure 7.18 shows the syntax of *EmbeddedMontiView* using syntactic sugar whereas the lines 16 to 37 show the same C&C view without syntactic sugar; the underlined text shows the added information. For example, syntactic sugar enables to access ports in connectors, even though these ports are not explicitly specified in the C&C view. Paths in views are relative ones to access elements, but these paths may not exist in the model satisfying this view. Reasons for this are: (1) hierarchies in views are abstract so that the path `distronic::tempomat.distanceFront` in *EmbeddedMontiView* may be mapped to a path containing other elements in between such as `distronic.distronic_enabled.speedControl.tempomat.distanceFront` in *EmbeddedMontiArc*; (2) additionally, instance names in a view are only for internal representation and may not match instance names in the component model satisfying the view, so the path `distronic::tempomat.distanceFront` in *EmbeddedMontiView* may satisfy `dist.temp.distanceFront` in *EmbeddedMontiArc*.

EmbeddedMontiView may pierce through component borders as shown in line 10 and it enables to directly connect subcomponent instantiations. Therefore, *EmbeddedMontiView* introduces the double colon `::` operator to navigate from a component instantiation to its subcomponent instantiation according to this view. Without this new double colon operator, `distronic.tempomat.distanceFront` would not be unique: does it means (a) the `distanceFront` port or (b) the `distanceFront` subcomponent instantiation. With this new operator `distronic::tempomat.distanceFront` means the port and `distronic::tempomat::distanceFront` means the subcomponent instantiation. Due to the four well-formedness rules of connectors (cf. CO1 rule in Subsection 6.1.2), the port direction of a port introduced by a connector may be derived as done in lines 22, 25, and 33. If the direction of a port cannot be inferred, e.g., when connecting new ports within the same component (`connect this.portA -> this.portB`), then the direction must be added to the target port (e.g., `connect this.portA -> this.portB {in}`), which indicates that `portA` is an outgoing port and `portB` is an incoming one.

Figure 7.19 shows how to connect arrays of ports in *EmbeddedMontiView*. The syntax similar to line 12 `signal -> filter[:].signal` in *EmbeddedMontiArc* means that the one `signal` port (having dimension one) is connected to the signal ports of all filter instances. The same line in *EmbeddedMontiView* means that there exists for each `filter` instance one `signal` port of `ParkingAssistant` which is connected to the `signal` port of the `filter`

EMV

```
view AbstractConnectorsSyntacticSugar {
  component ParkingAssistant {

    // automatically introduces instance `filter` and both `signal` ports
    connect this.signal -> filter.signal;
  }

  component ADAS { // Advanced Driver Assistant System
    instance ParkingAssistant parkingAssistent;
    instance Distronic distronic;

    // can pierce through component borders, :: to go through components
    connect this.distanceFront -> distronic::tempomat.distanceFront;
  }

  component Distronic {
   instance Tempomat tempomat;
  }
}
```

EMV

```
view AbstractConnectorsLongForm {
  component ParkingAssistant {
    port in ? signal;
    instance ? filter;
  }
  component ? {
     port in ? signal;
  }
  component ADAS {
    port in ? distanceFront;
    instance ParkingAssistant parkingAssistent;
    instance Distronic distronic;
  }
  component Distronic {
   instance Tempomat tempomat;
  }
  component ? {
     port in ? distanceFront;
  }
  connect ParkingAssistant.signal -> ParkingAssistant::filter.signal;
  connect ADAS.distanceFront -> ADAS::distronic::tempomat.distanceFront;
}
```

Figure 7.18.: How syntactic sugar of abstract connectors in *EmbeddedMontiView* is mapped to its long-form which is similar to Ringert [Rin14].

instance; this is less restrictive, because multiple connections in *EmbeddedMontiArc* can satisfy this abstract connection; e.g., `signal[:] -> filter[:].signal[1]` or `signal[1] -> filter[:].signal[1]`, where by the dimension of `ParkingAssistant`'s `signal` in the first case is ten and in the second case it is one. If *EmbeddedMontiView* does not restrict the dimension of `ParkingAssistant`'s `signal` both models are valid. If the modeler does

EMV

```
1  view AbstractConnectors {
2    component ADAS {
3      ports in ? signal[20];
4      instance ParkingAssistant pa[2]; //for longitudinal&transverse parking

5      // one signal port must be connected to any port of one filter instance
6      connect this.signal -> pa::filter;

7      // only one signal port must be connected with the signal port of
8      // any filter port
9      connect this.signal -> pa::filter.signal;

10     // each filter instance exists a signal port so that it connect to this
11     // signal port
12     connect this.signal -> pa[:]::filter[:].signal;

13     // long-form of line 12
14     forall i in 1..2, j in 1..10:
15       connect this.signal -> pa[j]::filter[i].signal;

16     // forces that each signal port is connected to a different filter
17     // instance
18     connect this.signal[:] -> pa[:]::filter[:];

19     // concrete connection (no abstraction anymore)
20     connect this.signal[:] -> pa[:]::filter[:].signal;

21     // long-form of line 20
22     forall i in 1..2, j in 1..10:
23       connect this.signal[(j-1)*10+i] -> pa[j]::filter[i].signal;        }
24   component ParkingAssistant { ports in ? signal[10]; instance Filter filter[10]; }}
```

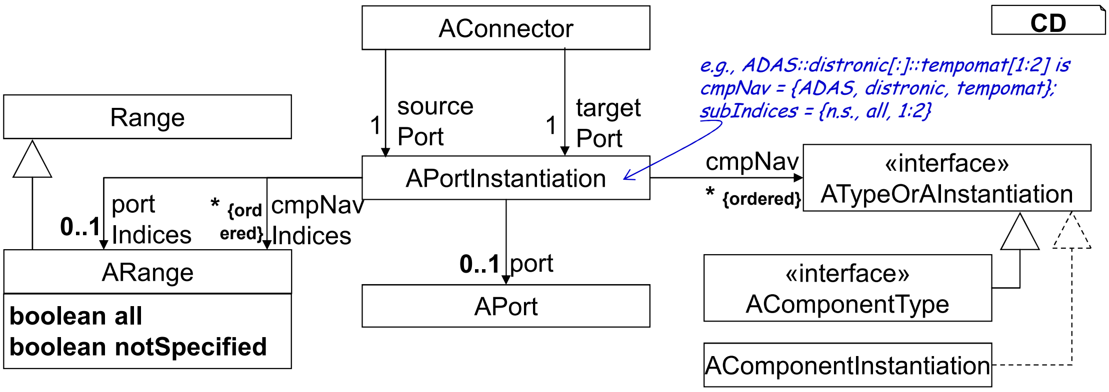
Figure 7.19.: Abstract Connection in *EmbeddedMontiView*.

not want this underspecification, then the second case must be defined explicitly: `signal[1] -> filter[:].signal` in *EmbeddedMontiView*.

Figure 7.20 shows the abstract syntax of the abstract connector. The `cmpNav` / `cmpNavIndices` list associations maps the optional `sub` / `subIndices` associations of `PortInstantiation` in *EmbeddedMontiArc*. The abstract range in *EmbeddedMontiView* extends the *normal* range with the two Boolean attributes `all` when explicitly defining `[:]` and `notSpecified` when no range is specified in the concrete syntax. In contrast to *EmbeddedMontiArc*, the `[:]` cannot be resolved to a range with minimum and maximum, as the maximum (which is the dimension of a port or a component instantiation) may not be specified in *EmbeddedMontiView*.

7.3.8. Abstract Effectors

Abstract effectors model the data-flow between components, abstract effectors may cross-cut component hierarchies. For example, an abstract effector can specify that the emergency brake component has (structural) impact on the brake output port of an advance driver assistant system; an *EmbeddedMontiArc* model satisfies this specification only if there exists a data-flow from an output port of this emergency brake component to the brake output port of the advance driver




```
context APortInstantiation inv:
  port.isPresent == portIndices.isPresent &&
  cmpNav.size    == cmpNavIndices.size
```

Figure 7.20.: Abstract syntax of abstract connector class (`AConnector`).

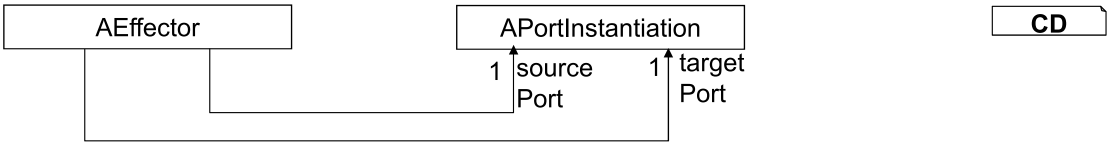


Figure 7.21.: Abstract syntax of abstract effector (`AEffector`).

assistant system. The abstract effector only forces a structural data-flow from its source to its target port; i.e., the behavior impact of the source to the target may be very less or even zero. However, the structural data-flow in C&C models is a necessary condition for behavioral data-flow.

Abstract effectors have the same abstract syntax (cf. Figure 7.21) and nearly the same concrete syntax - but with an `effect` instead of a `connect` keyword at the beginning - as abstract connectors. Abstract effectors must specify the direction of ports they introduce, for both source and target ports, because it exists no rules similar to CO1 (cf. Subsection 6.1.2). The syntactic sugar of introducing instances and ports in a connector or an effector statement saves much code, especially if the effector goes from the top level component to a very deeply nested inner one. The next section *Satisfaction-Relation between C&C Views and C&C Models* explains the semantical difference between abstract connectors and abstract effectors.

7.3.9. Imports and Full-Qualified Names

The C&C view language supports full-qualified component type names. If an *EmbeddedMontiView* artifact defines a package or an import statement, then an *EmbeddedMontiArc* model satisfies this view only if it matches all full-qualified names of the component types. If the component type of an abstract component instantiation is not fully qualified in an *EmbeddedMontiView*,

EMV
```
1 import p1.CmpA;
2 view Valid {
3   instance CmpA;
4 }
```

EMV
```
5 import p2.CmpA;
6 view Invalid {
7   instance CmpA;
8 }
```

EMA
```
9  package p1;
10 component Cmp {
11   ports …;
12   instance CmpA cmpA;
13   connect …;
14 }
```

Main.txt
```
15 Main-Component-Instantiation: p1.Cmp main;
```

Figure 7.22.: *EmbeddedMontiView* artifacts with import statements.

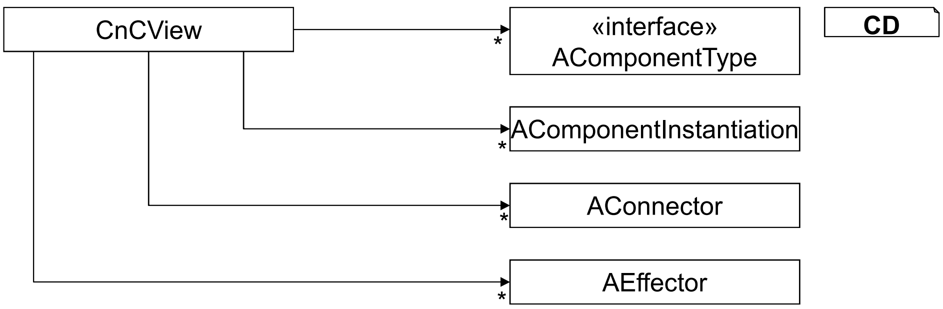


Figure 7.23.: Abstract Syntax of a C&C View.

then an *EmbeddedMontiArc* model satisfies this view if the short names of abstract and concrete component type names are equal.

The C&C model on the right (cf. ll. 9 - 15) in Figure 7.22 satisfies the top-left view `Valid` (cf. ll. 1-4), because the model has an instance with the full-qualified component type `p1.CmpA`. The C&C model does not satisfy the bottom-left view `Invalid` (cf. ll. 5-8) as the model does not has an instance with the full-qualified component type `p2.CmpA`.

7.3.10. Component and Connector View

The `CnCView` class of C&C views is analog to the `CnCModel` class (cf. Figure 4.16) of C&C models. Figure 7.23 shows the abstract syntax of it. A C&C View (`CnCView`) may consist of multiple abstract component type definitions (`AComponentType`), multiple abstract component instantiations (`AComponentInstantiation`), multiple abstract connectors (`AConnector`) and multiple abstract effectors (`AEffector`). In contrast to a C&C model having exactly one main component instantiation, a C&C view may have multiple, one, or no top-level abstract component instantiations.

7.3.11. Some Remarks

Similar to Grönniger, Kriebel, and Rumpe, *EmbeddedMontiView* can be easily extended with an environment keyword, e.g., `environment BrakeActuator { ports in (0V:12V) brakeSignal; }`. The C&C view verification ignores environment blocks. Thus, nearly no adaption of the verification algorithm is needed. However, modeling the environment enables simulating closed-loop controllers later, e.g., by adding physical constraints. Furthermore, the environment may be visualized in the graphical representation of *EmbeddedMontiView*.

Every Component and Connector Model is also a Component and Connector View when surrounding the content of every *EmbeddedMontiArc* model with `view $fileName {` and `}` and rename the file ending from `ema` to `emv`. It is the case as *EmbeddedMontiView* only extends concrete and abstract syntax of *EmbeddedMontiArc* with underspecification.

7.4. Satisfaction Relation between *EmbeddedMontiView* and *EmbeddedMontiArc*

The satisfaction relation between *EmbeddedMontiArc* and *EmbeddedMontiView* is straight forward: An *EmbeddedMontiArc* model satisfies an *EmbeddedMontiView* artifact if and only if, the *EmbeddedMontiArc* model refines all specified elements in an *EmbeddedMontiView* artifact.

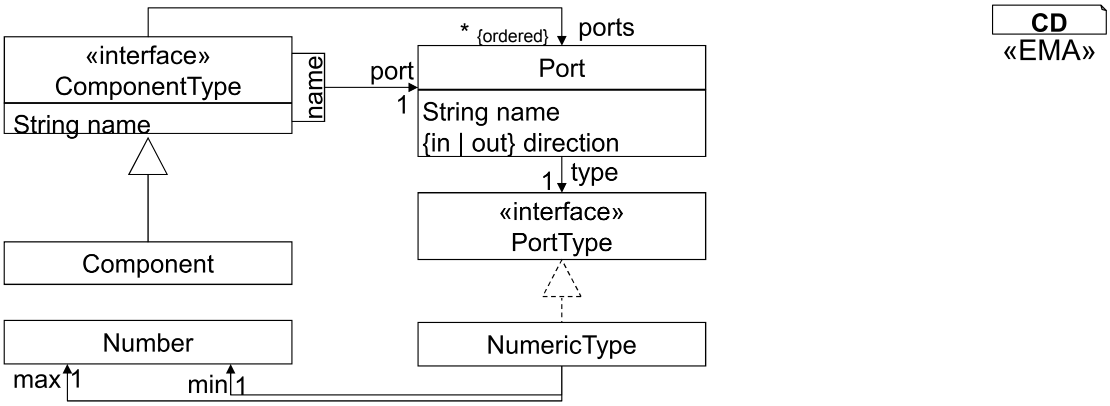
This section calls the C&C view models of *EmbeddedMontiView*, *EmbeddedMontiView* artifacts and not *EmbeddedMontiView* models to avoid confusion with the C&C models of *EmbeddedMontiArc* which are called *EmbeddedMontiArc* models.

For example, the *EmbeddedMontiView* artifact shown in Figure 7.3 on page 222 is semantically equivalent to the *OCL* constraint displayed in Figure 7.24. This means, based on the abstract syntax of *EmbeddedMontiView* a generator could produce the *OCL* code shown in Figure 7.24. An even easier solution is to formulate *OCL* constraints between the abstract syntax of *EmbeddedMontiView* artifacts and *EmbeddedMontiArc* models (where ever it is possible). This solution avoids to write an *OCL* generator, and we can only focus on the domain knowledge of these two languages specified in the *MontiCore* format. The next subsections define some of the satisfaction relations via *OCL*. However, some of the satisfaction relations are only described as text to avoid repeating *OCL* constraints having very similar patterns.

7.4.1. Abstract Ports

Figure 7.25 shows the satisfaction between abstract ports and ports. The satisfaction relation is not part of the concrete or abstract syntax of *EmbeddedMontiView*. The satisfaction relation describes the semantics of *EmbeddedMontiView*, i.e., the set of *EmbeddedMontiArc* models satisfying the specified C&C view.

The top part of Figure 7.25 shows the abstract syntax of *EmbeddedMontiView*. The abstract syntax of *EmbeddedMontiArc* has the following changes: The port `name` is not optional; the `dimension` association from `Port` to `NaturalNumber` has cardinality one; the `type` association from `Port` to `PortType` has cardinality one; the component type `name` is not optional; and the `ComponentType` does not have the Boolean property `portsComplete`.



OCL

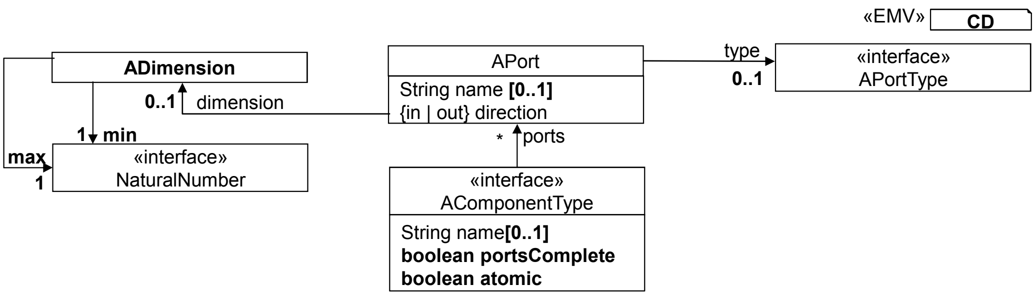
```
1  inv RedundandVelocityControllerPorts:
2   exists c in Component:
3     let cV = c.port["currentVelocity"]; wV = c.port["wishedVelocity"]
4     in
5       // component RedundantVelocityController {
6       c.name == "RedundantVelocityController" &&
7       // 3x ports in and 3x ports out
8       {p in c.ports | p.direction == IN}.size >= 3 &&
9       {p in c.ports | p.direction == OUT}.size >= 3 &&
10      // ports in (0km/h : 250km/h) currentVelocity
11      cV.direction == IN && cV.type instanceof NumericType
12         cV.type.min == 0km/h && cV.type.max == 250km/h &&
13      // ports in ?                   wishedVelocity
14      wV.direction == IN &&
15      // ports in Gear                ?
16      (exists p in c.ports: p != cV && p != wV && p.direction == IN
17         && p.type instanceof EnumType && p.type.name == "Gear") &&
18      // ports out (-oo m/s^2 : oo m/s^2) $acceleration
19      (exists p in c.ports: p.direction == OUT && p.type instanceof
20       NumericType && p.min == -oo m/s^2 && p.max == oo m/s^2)
```

Figure 7.24.: *OCL* constraint being semantically equivalent to C&C view model of Figure 7.3 on page 222.

The expressions `X ?== Y`, `X ?> Y`, `X ?>= Y`, `X ?< Y`, `X ?<= Y` mean if `X` is not present (i.e., `Optional.empty()` in Java) they evaluate to true, and if `X` is present they evaluate to `X == Y`, ..., `X <= Y` whereby `X` is the present value (i.e., is `X.get()` in Java). These operators enable efficient specifications of constraints including underspecification, because the comparison between `X` and `Y` must only be satisfied if `X` is specified.

The first constraint in lines 1 to 3 says when the abstract component type is marked as `portsComplete` then there must exist a component type which port names match the abstract port names of the abstract component type; see Figure 7.3.

The second constraint in lines 4 to 9 forces that the *EmbeddedMontiArc* model defines more ports (respecting the port dimensions) than the *EmbeddedMontiView* model. The expression `ports in B in1[4], out B out1[2]` defines four input ports (cf. ll. 6, 7) and two



`name ?== c.name` *is equal to*
`c.name == name ?: c.name` *or*
`name.isPresent implies name == c.name`

```
OCL
1  context AComponentType inv PortsComplete:
2    portsComplete implies exists ComponentType c: name ?== c.name &&
3      ports.name == c.ports.name
```

```
OCL
4  context AComponentType inv NbOfPorts:
5    exists ComponentType c: name ?== c.name &&
6      sum {ap.dimension.min ?: 1 | ap in ports, ap.direction == IN} <=
7                   sum {p.dimension | p in c.ports, p.direction == IN} &&
8      sum {ap.dimension.min ?: 1 | ap in ports, ap.direction == OUT} <=
9                   sum {p.dimension | p in c.ports, p.direction == OUT}
```

```
OCL
10 context AComponentType inv PortNames:
11   exists ComponentType c: name ?== c.name &&
12     forall ap in {ap in ports | ap.name.isPresent}: // ap: abstract port
13       exists p in c.ports: p.name == ap.name && p.direction == ap.direction
14              && ap.type ?~~ p.type && ap.dimension.min ?<= p.dimension &&
15              ap.dimension.max ?>= p.dimension
```

```
OCL
16 context AComponentType inv PortTypes:
17   exists ComponentType c: name ?== c.name &&
18     forall ap in {ap in ports | !ap.name.isPresent && ap.type.isPresent}:
19       exists p in c.ports: !ports.name.contains(p.name) &&
20         p.direction == ap.direction && ap.type ~~ p.type &&
21         ap.dimension.min ?<= p.dimension && ap.dimension.max ?>= p.dimension
```

Figure 7.25.: Satisfaction relation between port in *EmbeddedMontiArc* and abstract port in *EmbeddedMontiView*.

output ports (cf. ll. 8, 9) according to Figure 7.25. The second constraint handles the abstract input/output ports with unknown name and unknown datatype. The second constraint matches to lines 8 and 9 in Figure 7.24.

The third constraint in lines 10 to 15 in Figure 7.25 forces that the *EmbeddedMontiArc* model has the ports with the same name (cf. l. 13) as the named abstract ports (cf. l. 12) in *EmbeddedMontiView*. Furthermore, ports in *EmbeddedMontiArc* have the same direction (cf. l. 13), and same type (if present, cf. l. 14) as the abstract ports. Additional, the port dimension of the *EmbeddedMontiArc* port must be in the specified range, minimum (cf. l. 14) to maximum (cf. l. 15), of the dimension of the abstract port. This is equal to lines 11, 12, and 14 in Figure 7.24.

The fourth constraint in lines 16 to 21 forces that there exist ports in *EmbeddedMontiArc* matching the type and dimension of the abstract ports whereby ports already matching abstract ports in the second constraint (cf. `!ports.name.contains(p.name)` in l. 19 in Figure 7.25) cannot be used twice. This is equal to lines 16, 17, 19, and 20 in Figure 7.24.

Figure 7.25 omits the case `ports in ? ?[4]` which forces that a port in *EmbeddedMontiArc* exists having an arbitrary type but matching the dimension; `ports in B in1[2], B in2[2]` satisfies this constraint, because these are four input ports with the same type; whereas `ports in Z in1[3], B in2[2]` does not satisfy this constraint. Of course, this constraint also does not enable matching *EmbeddedMontiArc* ports several times, i.e., matched ports of the last two constraints in lines 10 to 21. This means `ports in B ?[2], ? ?[3]` is satisfied by `ports in B in1[2], Z in2[4]`. However, it is not satisfied by `ports in B in1[3], Z in2`, because `B ?[2]` is already matched by `B in1[3]` due to the Boolean data type and `? ? [3]` can again only be matched by `B in1[3]` due to the dimension. The avoidance of matching ports in *EmbeddedMontiArc* several times blows up the *OCL* constraint. This *OCL* constraint is too long to present it in this thesis and printing this constraint will not further help in understanding the satisfaction relation.

7.4.2. Abstract Subcomponent Instantiations

The *OCL* constraints to describe the satisfaction relation between *EmbeddedMontiArc*'s component instantiations and *EmbeddedMontiView*'s abstract component instantiations are very similar to the one of the port to abstract port relation: similar with the complete sign, similar to the minimum and maximum dimension, component types work analog to port types.

One difference is that for the not `direct` case, the abstract component instantiations must match elements in the transitive closure of the component instantiations of an *EmbeddedMontiArc* component type. The transitive closure of an *EmbeddedMontiArc* component instantiations of a component type `C` is the set `S` of all direct component instantiations of `C` plus all component instantiations of the component types of the elements in `S` (i.e., for all elements in `S` the component instantiation function calls itself recursively on the component type of the corresponding element).

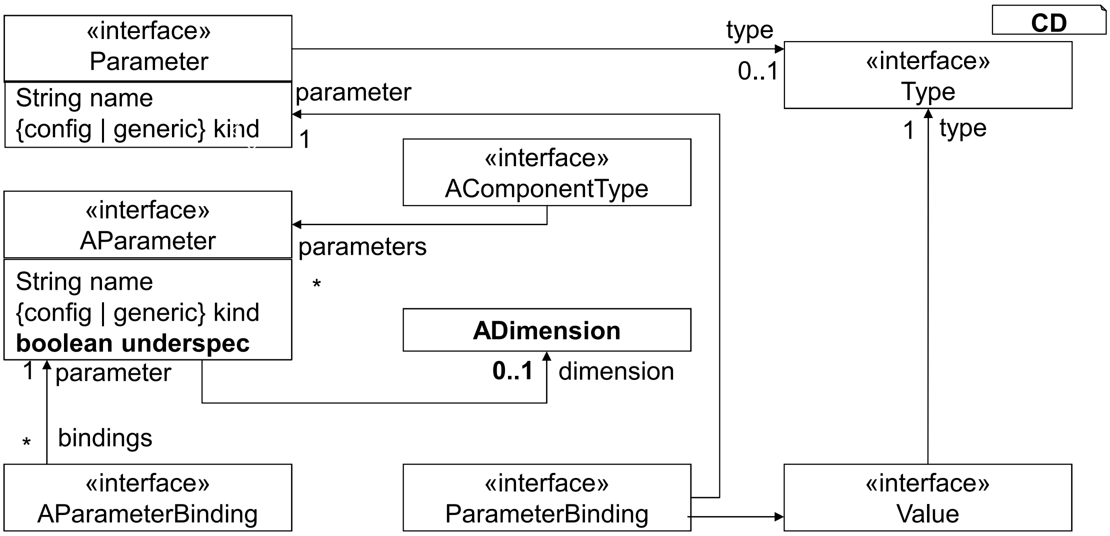
The other small difference is that the *OCL* constraints to describe this satisfaction relation do not compare the names of the component instantiations, because the (abstract) component instantiation names are only internal names and must not match.

Additionally, the following constraints hold: If an abstract component is marked as atomic, the matched *EmbeddedMontiArc* component does not have any subcomponent instantiations.

7.4.3. Abstract Type Parameters

Figure 7.26 shows the satisfaction relation between *EmbeddedMontiArc* parameters and *EmbeddedMontiView* abstract ones. For all abstract parameters (which are not underspecification parameters as in line 3) defined by an abstract component type, the *EmbeddedMontiArc* component type must contain a corresponding parameter (cf. l. 4). Corresponding parameter means in this case:

(i) the names and kinds of abstract and concrete parameter are identical (cf. l. 5);




```
1 context AComponentType inv Parameters:                                        OCL
2   exists ComponentType c: name ?== c.name &&
3     forall ap in {ap in parameters | !ap.underspec}:
4       exists p in c.parameters:
5         ap.name == p.name && ap.kind == p.kind && ap.type ?~~ p.type &&
6         (forall t in ap.bindings.value.type: t.isCompatibleTo(p.type)) &&
7         ap.dimension.min ?<= p.dimension && ap.dimension.max ?>= p.dimension
```

Figure 7.26.: Satisfaction relation between parameter definition in *EmbeddedMontiArc* and abstract parameter definition in *EmbeddedMontiView*.

(ii) if the abstract parameter defines a type; then the *EmbeddedMontiArc* parameter must also match this type (cf. l. 5); plus,

(iii) the values of parameter bindings of abstract parameter are assignable to the type of the concrete parameter (cf. l. 6); and the

(iv) the concrete parameter satisfies the dimension specifications of the abstract parameter if they are defined.

The function `AType::isCompatibleTo(Type t)` in case (iii) is pretty much the same as the compatibility one as defined in the *OCL* constraint in Figure 4.11 on page 117; non-numeric types are only compatible when they are identical. Case (iii) is needed when introducing parameters indirectly via abstract component instantiations. For example, `view V1 { instantiation And<1>; }` is not satisfied by `component And<(2:oo) n> { ports in B in1[n], ... }`, because the value `1` does not belong to the type `2:1:oo`.

7.4.4. Abstract Tensors as Port Types

A tensor in an *EmbeddedMontiView* artifact satisfies an abstract tensor, if and only if:

- If the abstract tensor has a type for the matrix elements, both tensor types must be equal.

- Missing dimension elements in the concrete syntax of the abstract tensor are interpreted as one. For example, `Q` is interpreted as `Q^{1, 1, 1}`, `Q^4` is interpreted as `Q^{4, 1, 1}`, and `Q^{4, 3}` is interpreted as `Q^{4, 3, 1}`. Underspecification in dimensions must be explicitly modeled; e.g., `Q^{4, 3, ?}`.
- Every dimension of the concrete matrix must match the specified dimension of the abstract matrix unless it is not specified (i.e., only the `?` sign) or it is expressed via underspecification parameters.
- If the underspecification parameters add constraint to the ratios of the dimensions of abstract matrices, the dimensions of the concrete matrix must respect these ratios.
- The concrete tensor/matrix has all the algebraic properties which are introduced by the abstract tensor.

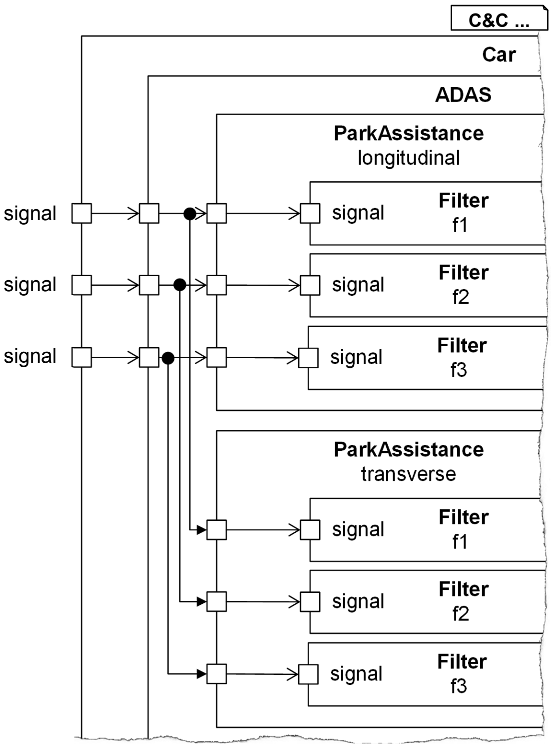
7.4.5. Abstract Connections

Abstract connections and abstract effects are very complex to formulate in *OCL*, because there must exist a connection chain satisfying a specified pattern in the transitive closure of all connection chains. Therefore, this subsection starts with the translation of a C&C view abstract connector to an example *OCL* constraint.

Figure 7.27 shows in the left part a C&C model satisfying the C&C view in the right part. Line 9 forces that the component `Car` exists and line 10 specifies that this component has at least one input port with the name `signal`. Line 11 says that the `Car` component must have directly or indirectly two instances of the same type: The C&C model `Car` has indirectly (`ADAS` is in between) the two instances `longitudinal` and `transverse` with the same component type, i.e., `ParkAssistance`. Lines 14 and 17 force that the component type of the two instances in line 11 have again directly or indirectly one instance whose type has the input port signal: In the C&C model the `ParkAssistance` component type has three (i.e., `f1`, `f2`, `f3`) subcomponent instantiations which type has one `signal` input port.

Lines 19 and 20 defines that *all* port instances of the `Car`'s `signal` port are connected via a connection chain to at least one signal port instance of `f1`, `f2`, and `f3` each being inside `longitudinal`, and `transverse` subcomponent instantiation. The expression `parkAss[1:2]` in line 20 specifies that only the two of the `ParkAssistance` (or any component type being in `Car` and having two instances) component types must be connected with `Car`'s `signal` port; if `Car` or `ADAS` would contain another `ParkAssistance` instance not being connected with `Car`'s `signal` port, then the constraint would still be satisfied.

Lines 1, 3, 4, and 5 in Figure 7.28 are similar to the condition 5 (c) of Definition 3.8 in Ringert's PhD thesis [Rin14, p. 37]. Line 2 only states that the elements contain *all* connectors. The elements attribute of a connector chain instance contains all elements (e.g., ports and components) involved in the connector chain. The second *OCL* constraint in lines 6 to 7 map to condition 5 (b) of Defintion 3.8 in Ringert's PhD thesis. The third *OCL* constraint in lines 8 and 9 introduces the `subs` association for component instantiations, which is the `subs` association of the component type of the component instantiation; this subs association is a self-association as it goes from `ComponentInstantiation` to `ComponentInstantiation`, and therefore, the transitive closure `subs**` exists. The fourth *OCL* constraint in lines 9 and 10 introduces the association `allSubs` of the component type which first navigates via `subs` to *all* subcomponent



EMV

```
1  view SignalConnections {
2    component Car {
3      port in ? signal;
4      connect this.signal[:] ->
5        parkAss[1:2]::filter[:].signal;
6    }
7  }
```

EMVNormalized

```
8  view SignalConnections {
9    component Car {
10     port in ? signal;
11     instance $PA parkAss[2];
12   }
13   component $PA {
14     instance $F filter;
15   }
16   component $F {
17     port in ? signal;
18   }
19   connect  Car.signal[:] ->
20      Car::parkAss[1:2]::filter[:].signal;
21 }
```

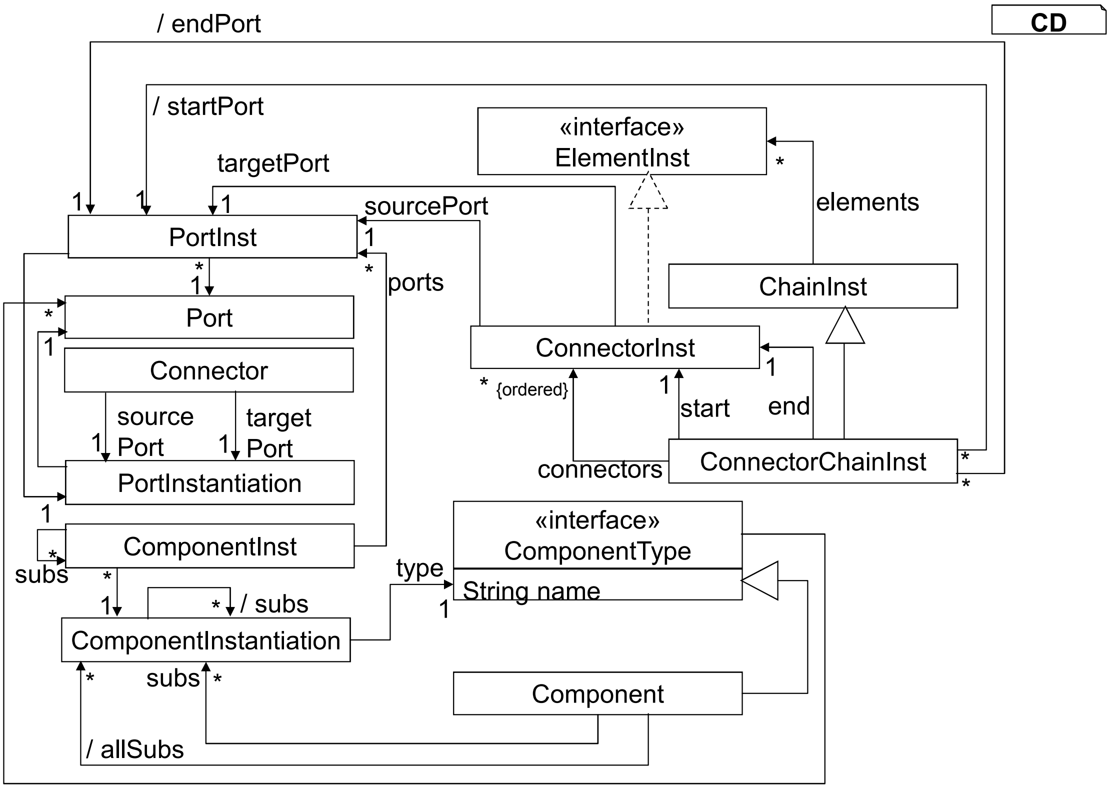
Figure 7.27.: Right side describes one C&C view: Right top part shows the short syntax with syntactic sugar, and the right bottom part shows the normalized *EmbeddedMontiView* syntax. The left side shows an excerpt of a C&C model satisfying this view.

instantiations and then calls there the transitive closure `subs**` one. The derived `allSubs` association goes from `Component` to `Set<ComponentIntantiation>`, and not from `Component` to `Set<Component>` as it would be the natural `subs**` one.

Additionally, the `ConnectorChainInst` class contains the two derived associations `startComponent` (which is `startPort.componentInst`) and `endComponent` (which is `endPort.componentInst`). These two associations are only skipped in Figure 7.28 due to clarity reasons to avoid crosscutting association lines.

Based on the class diagram with the new introduced derived associations in Figure 7.28, the C&C view in Figure 7.27 can be expressed as *OCL* constraint shown in Figure 7.28. Line b[4] in Figure 7.29 maps to `Car` in line 20 in Figure 7.27. The syntax `Car[:]` would result in a `forall` expression instead of an `exists` on in line b. Lines c and d map to `Car.signal[:]` in line 19; the `:` operator is mapped to the `forall` operator, which is independent from the number of port array size of the `signal` port in line 10; the port array number of `signal` is underspecified.

[4]Lines have letters as identifiers to reference in one sentence lines of two different figures later: where the one figure uses numbers and the other one letters as identifiers.




```
context ConnectorChainInst inv:                                    OCL
  elements.containsAll(connectors) &&
  connectors[0] == start && connectors[connectors.size - 1] == end &&
  forall i in {0 .. connectors.size - 1}:
    connectors[i].targetPort == connectors[i+1].sourcePort
```

```
context ConnectorChainInst inv:                                    OCL
  startPort == start.sourcePort && endPort == end.targetPort
```

```
context ComponentInstantiation inv:                                OCL
  subs == typeif type instanceof Component then type.subs else {}
```

```
context Component inv:                                             OCL
  allSubs == subs.subs**
```

Figure 7.28.: Abstract syntax of `ConnectorChainInst` class plus *OCL* constraints for derived associations.

OCL lines e to h map to `::parkAss[1:2]` in line 20. The `containsAll` function in line g is equals to the mathematical subset equals operator. The `[1:2]` specifies that there must exist at least two different `parkAss` component instances being involved in the connection chains; cf. l. h. The satisfaction relation may not match the component indices, as these differ between C&C model and C&C view anyway. These indices differ, because the C&C view may omit intermediate component (cf. `ADAS` component in C&C model in Figure 7.27).

```
OCL
a  inv SignalConnections:
b    (exists car in {c in ComponentInst | c.component.name == "Car"}:
c      (forall PortInst car_signal in {p in car.ports |
d                              p.name == "signal" && p.direction == IN}:
e        (exists ComponentType tParkAss:
f          (exists Set<ComponentInst> parkAss:
g           {s in car.subs**| s.component == tParkAss}.containsAll(parkAss) &&
h            parkAss.size == 2 &&
j            (exists ComponentType tFilter:
k              (forall ComponentInst filter in {s in parkAss.subs**)|
l                                                  s.type == tFilter}:
m                (exists PortInst filter_signal in {p in filter.ports |
n                              p.name == "signal" && p.direction == IN}:
o                  (exists ConnectorChainInst cci in car_signal.startPorts:
p                      cci.endPort == filter_signal
q    ) ) ) ) ) ) ) )
```

Figure 7.29.: *OCL* constraint derived from C&C view `SignalConnections` in Figure 7.53.

The indices in *EmbeddedMontiView* are used to indicate whether there must exist one element connecting multiple other ports, e.g., `parkAss[1:2]` and `parkAss[2:4]` have one common port instance `parkAss[2:2]` which must satisfy both conditions. Lines k and l map to `::filter[:]` in line 20.

Line o forces that the connector chain instance starts at a port instance of the `Car.signal` port. Line p states that this connector chain instance finishes at a port instance of `filter.signal`.

This example also unveils the expressive nature of *EmbeddedMontiView*: 16 lines of *OCL* code (cf. Figure 7.29) can be expressed by only seven lines of *EmbeddedMontiView* code (cf. ll. 1-7 in Figure 7.27). Additionally, the *EmbeddedMontiView* code is easier to read as it constraints the architecture on the concrete syntax whereas the *OCL* code constraints the architecture on the abstract syntax. The mapping of the *EmbeddedMontiView* syntax to Boolean *OCL* constraints about *EmbeddedMontiArc* models defines the semantics of *EmbeddedMontiView* uniquely.

For the complex abstract connector definition exists no *OCL/P* formula as it is the case in Ringert [Rin14], because the `[:]` and `[1:2]` operators may introduce mixed `exists`-`forall`-`exists` quantifiers as shown in Figure 7.29. These mixed quantifiers require to define lambda functions when iterating over the `cmpNav` parts of the abstract syntax of EmbeddedView's abstract connector (cf. Figure 7.20). Lambda functions map `forall x in X: boolean_expression(x)` to `boolean_expression` $\rightarrow$ `and { boolean_expression(x) | x in X}`. Lambda functions are not supported by the current *OCL* version.

To present the semantics of the abstract connector, we use a *FreeMarker* template which generates, based on the *EmbeddedMontiView*'s abstract syntax, the *OCL* expression for the *EmbeddedMontiArc* abstract syntax.

Figure 7.30 shows an excerpt of the *FreeMarker* template to generate the *OCL* code. This *FreeMarker* template can be interpreted as higher-order function having the signature $FTL : AConnector \rightarrow OCL$ and $OCL : EmbeddedMontiArc \rightarrow \mathbb{B}$. During the runtime of the C&C

FTL... :EMV→OCL

```
${tc.signature("ac")}
<#-- ac is the abstract syntax of the abstract connector -->
inv SignalConnections:
 <#assign sNav = ac.sourcePort.cmpNav>
 <#assign tNav = ac.targetPort.cmpNav>

 <#-- (1) handle sourcePort.cmpNav -->
 <#list sNav as cmp>
   <#if cmp?is_first><#assign prevName=cmp.name?lower_case>
      <#else><#assign prevName=sNav[cmp?index-1].name?lower_case></#if>
   <#assign cmpName = cmp.name?lower_case>
   <#assign ns = ac.cmpNavIndices[cmp?index].notSpecified>
   <#assign all = ac.cmpNavIndices[cmp?index].all>
   <#assign isCT = cmp.class.simpleName == "AComponentType">
   <#assign isCINoType = cmp.class.simpleName == "AComponentInstantiation"
                                                 && !cmp.type.isPresent()>
   <#if isCINoType>exists ComponentType t${cmp.name?cap_first}: </#if>
   <#if ns || all>
     <#if ns>(exists<#elseif all>(forall</#if>
     <#if isCT>${cmpName} in {c in ComponentInst |
                           c.component.name == "${cmp.name}"}:
     <#else>${cmpName} in {s in ${prevName}.subs** |
       <#if isCINoType>s.type == t${cmp.name?cap_first}
       <#else>s.type.name == "${cmp.type.name.get()}"
     </#if> }:
   <#else>(exists Set<ComponentInst> <= <#-- ... -->
   </#if>

   <#-- handle ...
    (2) sourcePort.port,
    (3) targetPort.cmpNav (skips elements handled by (1))
    (4) targetPort.port
   -->

   (exists ConnectorChainInst cci in
     <#assign sNavLast = sNav[sNav?size-1].name?lower_case>
     <#if ac.sourcePort.port.isPresent() && ac.sourcePort.port.get().name.isPresent()>
       ${sNavLast}_${ac.sourcePort.port.get().name.get()}.startPort:
    <#else> ${sNavLast}.startComponent: </#if>

    <#assign tNavLast = tNav[tNav?size-1].name?lower_case>
     <#if ac.targetPort.port.isPresent() && ac.targetPort.port.get().name.isPresent()>
      cci.endPort == ${tNavLast}_${ac.targetPort.port.get().name.get()}.endPort
    <#else> cci.endComponent == ${tNavLast}.endComponent: </#if>

   <#-- close brackets: counters are skipped here -->
```

Figure 7.30.: Excerpt of *FreeMarker* template generating *OCL* code from abstract syntax of *EmbeddedMontiView*'s abstract connector.

views verification tool, the *FreeMarker* template is executed to produce *OCL* code and this *OCL* code is directly afterwards evaluated to create the Boolean satisfaction answer.

The *FreeMarker* template is more complex than the *OCL* listings shown in Figure 7.25, and Figure 7.26. This is also the reason that the satisfaction relations in the previous subsections are defined via *OCL* constraints representing a function with the signature: *OCL* :

$EmbeddedMontiView$ × *EmbeddedMontiArc* → $\mathbb{B}$. This *FreeMarker* template approach is only used because *OCL/P* does not support higher order logic functions [Rum11, Section 3.5]; Rumpe states that the higher order logic functions can be emulated by using query functions of classes. This section uses *FreeMarker* instead of many query functions implemented in Java.

Using the *FreeMarker* DSL to describe the transformation process from an abstract connector to an *OCL* constraint has the following advantages:

1. Expressions in *FreeMarker* are navigated as in *OCL*, e.g., `ac.sourcePort.cmpNav` is interpreted as the Java code `ac.getSourcePort().getCmpNav()` [HR17, p. 151].
2. *FreeMarker* has many build-in functions for collections, e.g., `collection[index]` is mapped to `collection.get(index)` or `1..4` defines the same list as in *OCL*, and for strings.

However, *FreeMarker* is not typed; thus, the *FreeMarker* template may cause runtime exceptions when executing it. Line 1 in Figure 7.30 only says that the template is invoked with one parameter (cf. `TemplateController` class in [HR17, p. 166]), but this line does not state the type of `ac`. To overcome this problem in future, the `TemplateController` may be extended with a method `signatureTypes` and line 1 will be then replaced by `${tc.signature ("ac")} ${tc.signatureTypes("embeddedmontiview.AConnector")}`. Based on the additional type information of the template parameter, *MontiCore* would be able to resolve all types (e.g., against Java classes or CD4A class diagrams).

Our example C&C view in lines 19 and 20 in Figure 7.27, binds `sNav={Car}`, `tNav={Car, parkas, filter, signal}`. Lines 7 to 27 (esp., ll. 17-19) in Figure 7.30 create line b in Figure 7.29. The handling of `targetPort.cmpNav` works very similar to the handling of the `sourcePort.cmpNav` except that for elements which are also in `sNav`, i.e. `Car` in our example, no new *OCL* `forall` or `exists` text is produced. Therefore, the lines 7 to 27 for `tNav` create the lines e to l in Figure 7.29. Lines 33 to 42 produce lines o and p.

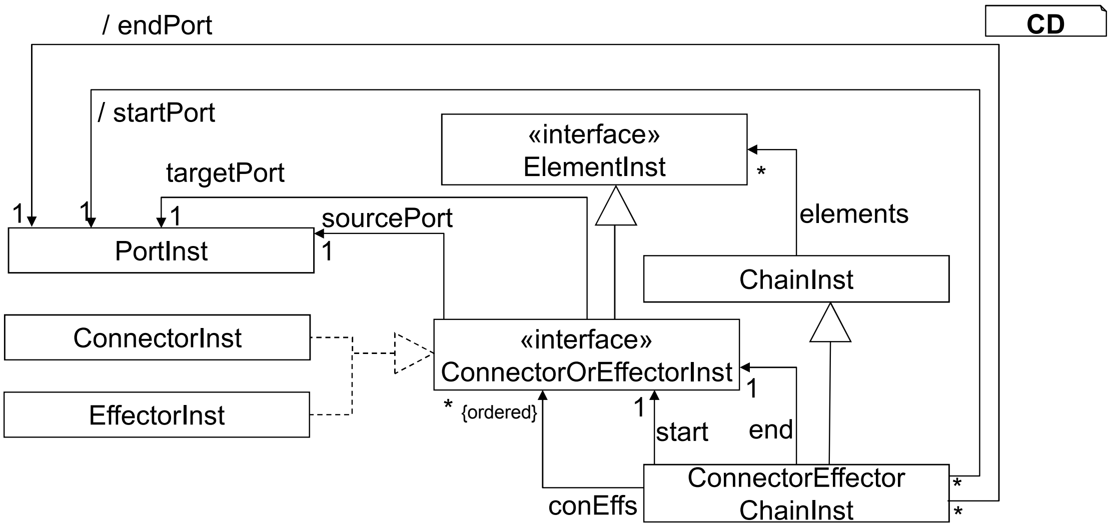
7.4.6. Abstract Effectors

The satisfaction relation of abstract effectors is similar to satisfaction relation of abstract connectors. First, a `ConnectorEffectorChainInst` is defined in a similar way as the `ConnectorChainInst` in Figure 7.28. The `ConnectorEffectorChainInst` in Figure 7.31 also extends `ChainInst`, it also has a `startPort`, `startComponent`, `endPort`, and an `endComponent` derived association. The only difference is that the `conEffs` chain contains effector instances and connector instances. The addition of effector instances enables to express data flow going through atomic components.

The satisfaction relation of abstract effectors is the same as the one of abstract connectors except that in line 33 in Figure 7.30 the `exists ConnectorChainInst` must be replaced by `exists ConnectorEffectorChainInst` and that the variable `ac` for abstract connector is replaced by `ae` for abstract effector in Figure 7.30.

7.4.7. Some Remarks

Chapter 4 (*Abstract Syntax of EmbeddedMontiArc*) and Section 7.3 (*Concrete and Abstract Syntax of EmbeddedMontiView*) introduced the formal definitions of component and connector models, their C&C instance structure, and C&C views by presenting the abstract syntax of these two




```
context ConnectorChainInst inv:
  elements.containsAll(connectors) &&
  connectors[0] == start && connectors[connectors.size - 1] == end &&
  forall i in {0 .. connectors.size - 1}:
    connectors[i].targetPort == connectors[i+1].sourcePort
```

```
context ConnectorChainInst inv:
  startPort == start.sourcePort && endPort == end.targetPort
```

```
context ComponentInstantiation inv:
  subs == typeif type instanceof Component then type.subs else {}
```

Figure 7.31.: Abstract syntax of `ConnectorEffectorChainInst` for *EmbeddedMontiArc*.

languages in class diagrams. This section formally defined the satisfaction relation between the component and connector models/instance structures and component and connector view artifacts in *OCL*, or by templates generating *OCL* code.

Please note, the class diagrams, esp. the textual CD4A syntax in Appendix B of the graphical class diagram representations of this chapter, is as formal as the tuple definitions of C&C models and C&C views presented by Maoz, Ringert, and Rumpe [MRR13, MRR14, Rin14, BMR+17a]: The translation of tuple structures to class diagram representations is straight forward; whereas the inverse translation of class diagrams to tuple structures is more challenging due to the missing inheritance features of tuple structures. This is also the main reason why this thesis uses the more powerful *UML* class diagram notation to formalize *EmbeddedMontiArc* and *EmbeddedMontiView*. In a first version of this thesis both languages were defined via tuples similar to Ringert [Rin14]; however, the tuple structures of *EmbeddedMontiArc* became very complex and hard to read, as *EmbeddedMontiArc* (including the powerful port type system with units) has much more language features than Ringert's *MontiArc* C&C model [Rin14, Definition 2.2 on p. 15], [Rin14, Definition 6.8 on p. 164f.].

The same holds for the specification of the satisfaction relation between *EmbeddedMontiView* and *EmbeddedMontiArc*: The formal *OCL* constraints in Figure 7.25, Figure 7.26, and Figure 7.28

satisfaction constraints under assumption of one loaded C&C model and one loaded C&C view:

```
OCL
context AComponentType inv PortsComplete:
  portsComplete implies exists ComponentType c: name ?== c.name &&
    c.ports.name == this.ports.name
```

```
OCL
context AComponentType inv Parameters:
  exists ComponentType c: name ?== c.name &&
    forall ap in {ap in parameters | !ap.underspec}:
      exists p in c.parameters:
        ap.name == p.name && ap.kind == p.kind && ap.type ?~~ p.type &&
        (forall t in ap.bindings.value.type: t.isCompatibleTo(p.type)) &&
        ap.dimension.min ?<= p.dimension && ap.dimension.max ?>= p.dimension
```

... (other OCL constraints)

"merging" of OCL constraints

one satisfaction constraint without any assumption:

```
OCL
context CnCModel cncm, CnCView cncv inv:
  cncm.satisfies(cncv) <=>
    forall act in cncv.aComponentTypes:
      // constraint PortsComplete
      (act.portsComplete implies exists c in cncm.componentTypes:
        act.name ?== c.name && c.ports.name == act.ports.name)
      && // constraint Parameters
      (exists c in cncm.componentTypes: act.name ?== c.name &&
         forall ap in {ap in act.parameters | !ap.underspec}:
           exists p in c.parameters:
             ap.name == p.name && ap.kind == p.kind && ap.type ?~~ p.type &&
             forall t in ap.bindings.value.type: t.isCompatibleTo(p.type))&&
              ap.dimension.min ?<= p.dimension &&
              ap.dimension.max ?>= p.dimension)
       && // ... (other OCL constraints)
```

Figure 7.32.: "Merging" of multiple *OCL* constraints with assumption that only one C&C model and one C&C view is loaded in *OCL* universe to one *OCL* constraints with no assumption.

plus the high-level function defined in Figure 7.30, define the same mathematical relations between C&C views and C&C models as the binary satisfaction relation defined by Maoz and Ringert et. al. (cf. [Rin14, Definition 3.8 on p. 36f]). The *OCL* definitions in this section assume that the C&C model and the C&C view, which are checked against each other, are the only available C&C model/view elements in the *OCL* universe. Thus, all C&C elements being available in expressions such as `exists ComponentType` (cf. l. 2 in Figure 7.25) belong to this one C&C model. This assumption enables splitting the satisfaction relation into multiple smaller *OCL* constraints, and it also only forces to write an *OCL* constraint generator for abstract connectors and abstract effectors. Then all (also the generated) *OCL* constraints can be merged to one large *OCL* constraint as shown in Figure 7.32 which defines the complete satisfaction relation between C&C models and C&C views.

The modular development of this satisfaction constraint enables easier understanding and multiple developers can easier work together where one developer is responsible for a set of small *OCL* constraints.

Defining the satisfaction relation in *OCL* has the additional advantage that the *OCL* framework presented in Chapter 6 can translate this specification to executable Java code.

7.5. Witnesses Based on Satisfaction-Relation between *EmbeddedMontiArc* and *EmbeddedMontiView*

C&C views (such as *EmbeddedMontiView* artifacts) document design decisions and relations between different elements in C&C models (such as *EmbeddedMontiArc* ones) [Rin14, p. 49]. Thus, every *EmbeddedMontiView* artifact is a specification which should be satisfied by an *EmbeddedMontiArc* model.

The previous sections presented the mathematical relation when an *EmbeddedMontiArc* model satisfies an *EmbeddedMontiView* artifact. However, the *EmbeddedMontiArc*/*EmbeddedMontiView* modeler is not only interested in a Boolean answer, he is also interested in why a C&C model satisfies the corresponding C&C view or why this relation is not satisfied. An additional case study with many interviews [BMR^{+}17a] unveiled that the modeler is also interested in receiving all tracing information between C&C models and their corresponding views.

Therefore, this section handles three witness kinds: The first subsection introduces witnesses justifying positive verification results by listing all needed *EmbeddedMontiArc* elements to satisfy a given *EmbeddedMontiView* element. The next section generates larger tracing witnesses by showing all elements in an *EmbeddedMontiArc* model which are satisfied by at least one element in an EmbeddedView artifact. The last subsection contains a small subset of elements of the *EmbeddedMontiArc* model violating a specific *EmbeddedMontiView* element, and thus, justifying a negative verification result. The concepts of the first and last subsections are similar to the ones presented by Maoz, Ringert, and Rumpe (e.g., cf. [Rin14, Subsection 4.3.1], [Rin14, Subsection 4.3.2]).

Every witness is itself a partially correct (corresponding to the textual syntax as well as to the formal constraints given in the textual model) *EmbeddedMontiArc* model, so that the modeler does not need to learn an additional modeling language. The witness is not a complete correct model, as e.g., ports are not connected or some components neither have input nor output ports.

7.5.1. Satisfaction Witnesses

This subsection extends Ringert's satisfaction witnesses [Rin14, MRR13] to support new language features of *EmbeddedMontiArc* and *EmbeddedMontiView*.

This subsection presents eleven rules for all features of the component and connector view language *EmbeddedMontiView*. Additionally, it explains how the corresponding local minimal witness looks like. The eleven rules to generate satisfaction witnesses are:

(1) **Hierarchy abstraction**
Similar to Ringert [Rin14, ll. 7-10 in Procedure 6], the witness contains all component instantiations in the view plus all component instantiations needed to satisfy rule 5 (connector-effector-chain witnesses may introduce additional component instances) and all parent component instances until their least common parent component instance (cf. [Rin14, ll. 5,6 in Procedure 6]).

(2) **Connectivity abstraction**
Similar to Ringert [Rin14, ll. 10-15 in Procedure 6], the witness contains the connector chains to satisfy all abstract connectors in the view. If an abstract connector can be satisfied by two connector chains, then the witness contains the shorter connector chain. If for an abstract connector already exists a connector chain in the witness satisfying this abstract connector, then no additional connector chain is added.

(3) **Incomplete interfaces**
Similar to Ringert [Rin14, ll. 16-26 in Procedure 6], the witness contains all ports including their data types. Port dimensions are only added if they have a corresponding satisfaction relation in the view; whereby the port dimension `1` is explicitly modeled when expressed in the C&C view. If the witness does not contain a dimension it is interpreted in *EmbeddedMontiArc* as one, and thus, only one port instance of the port array is needed to satisfy the view. The witness additionally contains the ports needed for all witness connector-chains of rule 2 or needed for all witness connector-effector-chain instances of rule 5.

(4) **Atomic, Direct and Complete Subcomponents as well as Interface Complete**
The witnesses for atomic components, which are also atomic in the view, contain their implementation body as proof. Witnesses for direct subcomponents are marked with a comment *direct* to show it. Witnesses for complete subcomponents contain the comment *subcomponent instances complete* to mark that the model does not contain any other subcomponent instance. Witnesses for components, which are marked as interface-complete in the view, contain a *(c)* comment as there is no other valid way to express in the witness that it is complete as the *EmbeddedMontiArc* model does not support this notation.

(5) **Data Flow Abstraction**
Similar to our C&C view case study paper [BMR+17a], the witness contains `connect` and `effect` statements (with only the needed index ranges) belonging to the connector-effector-chain-instances satisfying all abstract effectors in the view. If an abstract effector can be satisfied by two connector-effector-chain-instances, then the smaller (in terms of connector and effector elements) one is present in the view. Especially, unneeded feedback loops are not part of the witness.

(6) **Support of Component Types**
The witness contains all component types which are referenced by an component instance added by rule 1. If a component type in a view implements component interfaces (directly or indirectly), then the witness also implements the corresponding interfaces. The component type in a witness only contains the generic and configuration parameters which are needed to satisfy the view. The parameter order of the witness is the one of the C&C model.

(7) **Unit Kind Abstraction**
If the port in a view has only an abstract port type such as `Length`, then the witness creates a comment after the port's data type with the in the C&C view specified quantity. This adds the information to see directly that the unit kind abstraction has been verified.

(8) **Matrix Property and Dimension Abstraction**
All matrix properties and port type dimensions specified in a C&C view are directly added to the witness.

(9) **Port Array Abstraction**
As it is valid to omit ports in the witness [Rin14], the port array size is skipped and is only present if it is forced by the view. If the view also contains a maximum dimension, then this is marked by a comment to easily verify that the actual dimension is in the specified range.

(10) **Component Instance Array Abstraction**
Similar to port arrays: The witness contains the subcomponent instantiation array dimension, if it is also present in the corresponding view element; and C&C view ranges are added as comment.

(11) **Order of View elements and Order of Model Elements control Order of Witness elements**
Since the rules (1) to (10) massively depend on single view elements and for one view element several (even shortest) model elements may exist, the algorithm applies the following rule: the algorithm creates witness elements in the order of the view elements. This means that for a first view element `connect cmp -> cmp2` the connector-chain `cmp.portIn1 -> cmp2.portIn1` would be added to the witness; and for a second view element `connect cmp.portIn4 -> cmp2.portIn3` another connector chain is added to the witness. The second chain is also added to the witness as the first chain of the witness does not satisfy the second abstract connector. If the two abstract connect statements would be switched, then only one witness chain would be created, as the witness chain for `connect cmp.portIn4 -> cmp2.portIn3` already satisfies the abstract connector `connect cmp -> cmp2`.

Also for the first view element `connect cmp -> cmp2` the order of the model plays a role as `portIn1` comes before `portIn2` in both component instances, the algorithm takes `cmp.portIn1 -> cmp2.portIn1` and not `cmp.portIn2 -> cmp2.portIn2` even though both have the same length. Since the formal definitions of port instances and connector instances are sets, the witness contains no identical elements.

If for satisfying one view element multiple witness elements are generated (e.g., for a connector-effector-chain or a connector-chain), then the order of these elements is the same as the one in the textual model. Since *EmbeddedMontiArc* and *EmbeddedMontiView* are textual models, the order of the elements can be uniquely determined. How it is done is unimportant, it only must be unique. One possible solution would be to order the absolute paths of all textual models and then take the line and column number of the start position of the abstract syntax rule inside one textual model.

The implementation of the witness generation process can also generate the witnesses as *EmbeddedMontiView* artifact instead of an *EmbeddedMontiArc* model. The only difference when generating an *EmbeddedMontiView* witness is that in rule 4 the comments *direct* and *(c)* are keywords; additionally, the comment *subcomponent instances complete* is mapped to the keywords `instances (c)` and instead of showing the implementation body of an atomic component the `atomic` keyword is used. Generating both kinds of witnesses enables to further process witness models with the wanted toolchain (e.g., generate C&C model visualisation with navigation between component hierarchies or C&C view artifact one with all components in one picture).

Examples of Satisfaction Witnesses

The witness of the first simple car example covers rules (1), (2), (3), and (5). The witness of the second simple parking assistant example covers rules (3), (4), (6), (7), (9), and (10).

Rule (11) is followed by both witnesses, for rule (8) no extra example is presented as only properties form the C&C view to the witness are copied.

Simple Car Example Figure 7.33 shows an example C&C model of a car software component. This example model is an extension of the C&C model presented in the C&C views case study paper [BMR$^+$17a, Fig. 1]. This C&C model is composed of interior functions (cf. `InteriorFunctions` component) affecting the functionality of the car (i.e., electrical seat movements and seat heating) and of functions influencing the car's driving and exterior light behavior. This example is very basic, real car software consists of much more such functionalities. Examples of further functionalities are [Dai18c]: heated power side mirrors, automatic dimming of interior and exterior mirrors, automatic power fold-in for exterior mirrors, electrical movement of mirrors, two zone front and two zone back automatic air cooling and air heating system, rain-sensing windshield wipers, heated wipers, and much more. Thus, real-world component and connector models contain thousands of components interacting with each other.

The input port on the left side of this C&C model receive sensor data and wished user input. The components calculate the output (output ports on the right side) to control the actuators to achieve the desired driving behavior (e.g., defined maximal road speed considering distance to car in front) or user behavior (e.g., correct seat position and temperature).

The `ExteriorFunctions` subsystem controls the car's acceleration, brake, and light signals and consists of the two subcomponents `Driving` and `ALS` (Adaptive Light System). The component `Driving` is hierarchically decomposed into three components: `ADAS` (Advanced Driver Assistance System), `ParkAssist`, and `Switch` propagating outputs of `ADAS` when driving forward and outputs of `ParkAssist` when parking.

The C&C view CA1 shown in Figure 7.34 describes the ADAS component. This C&C view and the text explaining it is already published in our C&C view case study paper [BMR$^+$17a, Fig. 2]. This C&C view describes only the high-level ADAS functionality of the car software. The `ADAS` software component is inside the `ExteriorFunction` ones; it also receives inputs unmodified from `ExteriorFunction` one (left three abstract connectors from `ExteriorFunctions` to `ADAS`) and its `Acceleration` and `Brake` output values effect the corresponding output val-

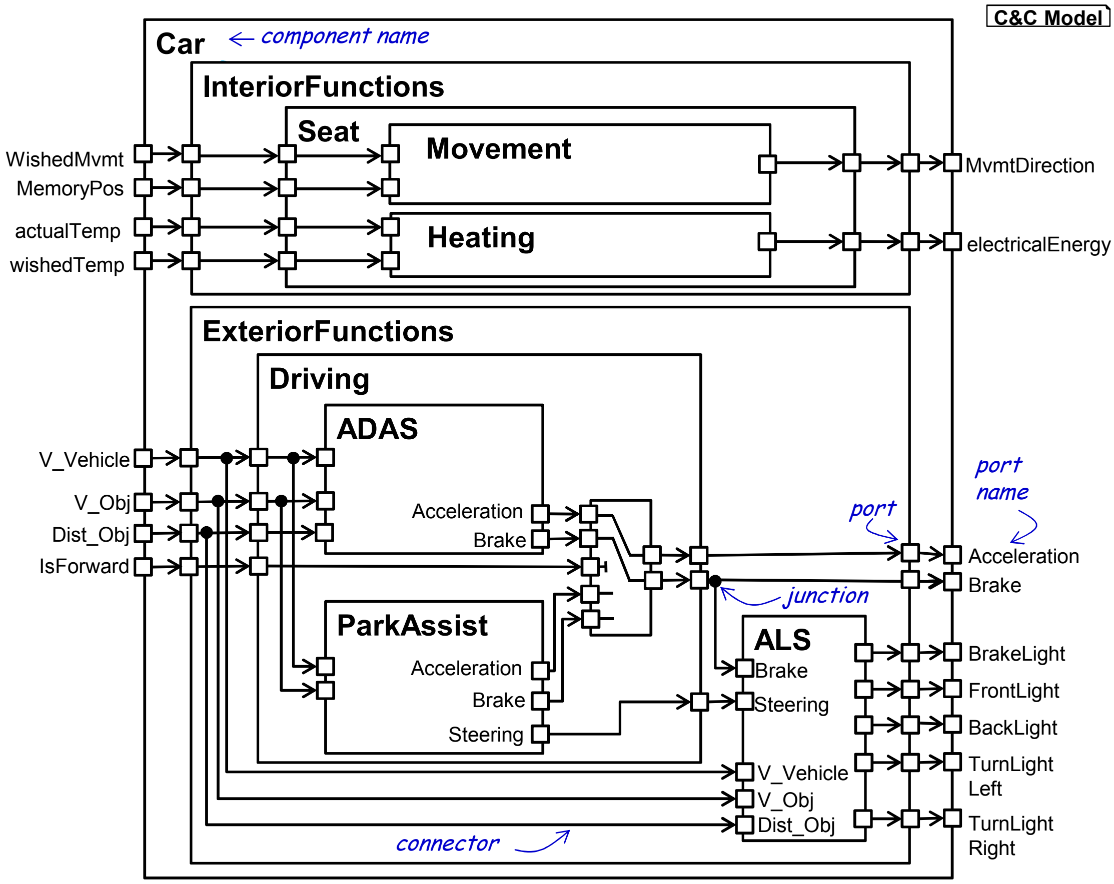


Figure 7.33.: Graphical representation of `Car` C&C model (enhanced example C&C views case study paper [BMR+17a, Fig. 1]).

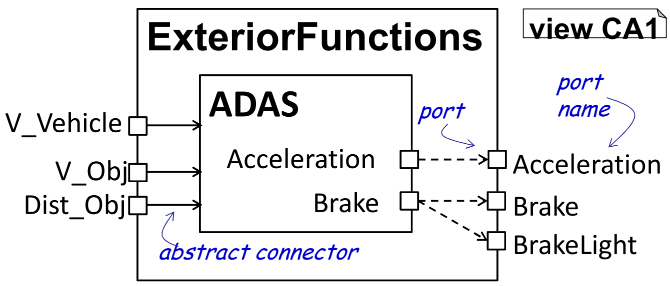


Figure 7.34.: Graphical representation of C&C view CA1 (copied from [BMR+17a, Fig. 2]).

ues of the `ExteriorFunction` component. The values of the `Brake` output port additionally effect the `BrakeLight` port: as harder the car brakes as brighter the braking light gets.

The C&C view in Figure 7.34 with two abstract components, eight abstract ports, three abstract connectors, and three abstract effectors is much smaller than the larger simple C&C model in

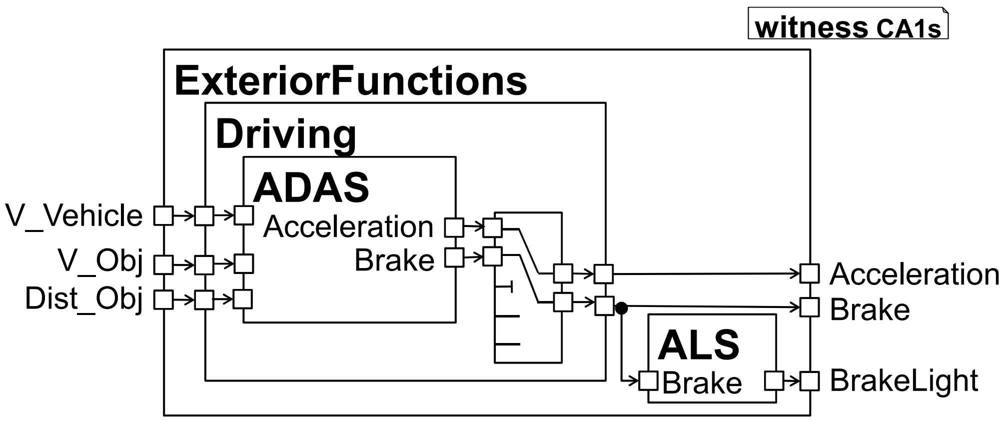


Figure 7.35.: Graphical representation of satisfaction witness for C&C model in Figure 7.33 satisfying C&C View in Figure 7.34 (copied from [BMR+17a, Fig. 3]).

Figure 7.33 with eleven components, 80 ports, 56 connectors, and 29 (in the graphic omitted) effectors. This much smaller nature of C&C views facilitates to focus on the communication between these two components by omitting all for this view unimportant information.

Figure 7.35 shows the graphical representation of the generated satisfaction witness for the C&C model and C&C view in Figure 7.33 and Figure 7.34.

Due to rule (1), the witness contains the least common parent component, i.e., `Exterior-Functions` in the C&C model. The least common parent component of `ExteriorFunc-tions` and `ADAS` is `ExteriorFunctions`; it is not `Car`. Further examples of the least common parent component are: The least common parent component of `ADAS` and `ParkAssist` is `Driving`; the least common parent component of `ADAS` and `ALS` is `ExteriorFunctions`; the least common parent component of `ADAS` and `Heating` is `Car`.

Additionally, rule (1) states that all components between the least common parent component, i.e., `ExteriorFunctions`, and the components matching the abstract ones in the view are part of the witness. Therefore, the witness contains additionally the component `Driving`.

Rule (2) specifies that the witness contains all connector chains to satisfy the abstract connectors, and all ports and components referenced by the connectors. This rule adds the nine ports - three incoming ports `V_Vehicle`, `V_Obj`, and `Dist_Obj` to each of the components `ExteriorFunctions`, `Driving`, and `ADAS` - to the witness. Additionally, this rule adds the six connectors between `ExteriorFunctions`, `Driving`, and `ADAS` as shown in the left part in Figure 7.35 to the witness.

Rule (3) adds the outgoing ports of `ADAS` and of `ExteriorFunctions` to the witness. Rule (4) is not applied, because the C&C view in Figure 7.34 does not have atomic, direct, complete markers for ports or subcomponent instantiations.

Rule (5) adds the elements to match the abstract effectors. The first abstract effector going from `Acceleration` port of `ADAS` to the same named port of `ExteriorFunctions` adds the `Switch` component, the most top input and output ports of the `Switch` component to the witness, and the `Acceleration` port of `Driving` as well as the connections from `ADAS`'s

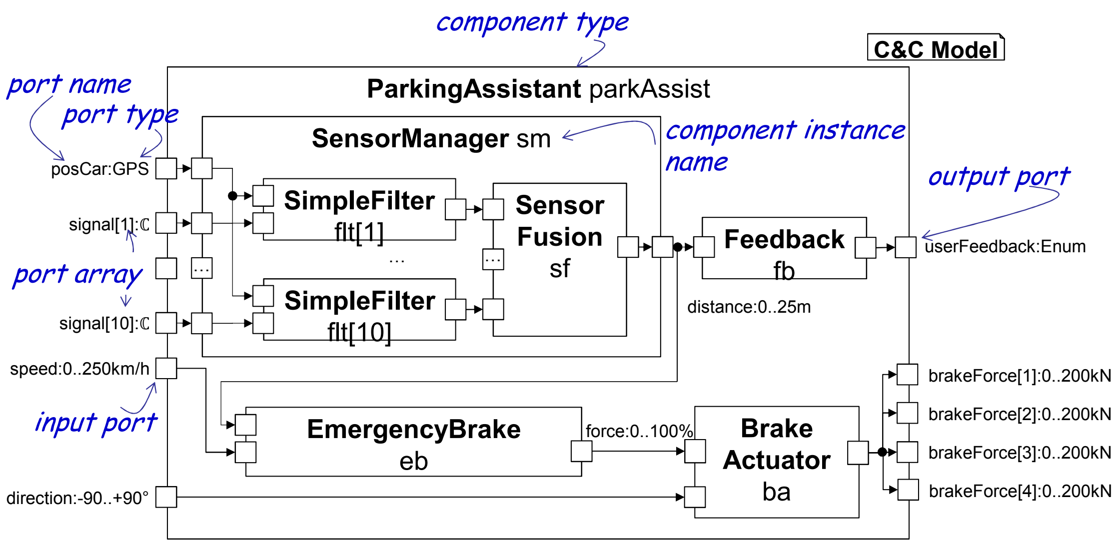


Figure 7.36.: Graphical Model of `ParkingAssistant` (adapted from [KRRvW17]). Some port types are omitted; cf. textual model in Figure 7.37.

`Acceleration` to `Switch`'s top input port, from top output port of `Switch` to the `Acceleration` port of `Driving`, and from `Acceleration` port of `Driving` to the same named port of `ExteriorFunctions`. The abstract effector from `ADAS`'s `Brake` output port to the `ExteriorFunctions` one adds the two ports below to the `Switch`, and also one output port to `Driving` plus the three connections following the schema described above. The abstract effector going from `ADAS`'s `Brake` to `ExteriorFunctions`' `BrakeLight` adds the `ALS` component plus the `Brake` and `BrakeLight` ports of the `ALS` component to the witness as well as the connections from `Driving`'s `Brake` port to `ALS`'s `Brake` port and from `ALS`'s `BrakeLight` port to `ExteriorFunctions`'s `BrakeLight` port.

Rule (6) adds no further information to the witnesses as no component type in the C&C model in Figure 7.33 extends any component interface. Rules (7), (8), (9), and (10) do not apply because no arrays of ports or component instantiations or unit/matrix port type exist in the C&C model.

Simple Parking Assistant Example Figure 7.36 and Figure 7.37 show an incomplete driver assistance software system. A slightly modified version of this C&C model is presented in our paper *Modeling Architectures of Cyber Physical Systems* [KRRvW17]. This driver assistance software system provides automated emergency braking and visual user feedback. The generic `ParkingAssistant` component (cf. l. 1) receive signals (cf. input ports on the left side in Figure 7.36) needed for component computations including the GPS position (cf. l. 2), steering angle of the vehicle (cf. l. 2), speed (cf. l. 3), as well as a port array for complex radar signals (cf. l. 3) containing in-phases and quadrature components for object movement detection. Output ports on the right hand side in Figure 7.36, represent the calculated results, i.e., user `feedback` (cf. l. 5) for the dashboard and a `brakeForce` array (cf. l. 6) controlling the car's four brakes.

```
1  component ParkingAssistant<N+ n> {                                          EMA
2    ports in  GPS posCar, (-90° : 0.1° : 90°) direction,
3              C signal[n],
4              (0 km/h : 0.2 km/h : 250 km/h) speed,
5          out UserFeedback feedback,
6              (0N:1N:200kN) brakeForce[4];
7    instance SensorManager<n>(SimpleFilter) sm;
8    instance BrakeActuator ba;
9    instance Feedback fb; instance EmergencyBrake eb;                            }
```

```
10 component SensorManager<N+ n> (Filter F) {                                  EMA
11   ports in  GPS posCar, C signal[n],
12         out (0m : 0.2m : 10m) mergedDistance;
13    instance F flt[n];
14    instance SensorFusion<n> sf;                                                 }
```

```
15 component SimpleFilter implements Filter<(0m : 0.2m : 10m)> {              EMA
16   implementation Math { /* skipped */ } }
```

```
17 component interface Filter<T is Length> {                                  EMA
18    ports in C signal, GPS posCar, out T distance;  }
```

```
19 component SensorFusion<N1 n> ( (-90°:90°)^{1,n} ) tilt = zeros(1,n)        EMA
20    ports in (0m : 0.2m : 10m) distance[n],
21          out  (0m : 0.2m : 10m) mergedDistance; }
```

```
22 component Feedback {                                                       EMA
23    ports in (0m : 25m) distance, out UserFeedback feedback; }
```

```
24 component EmergencyBrake { ports in (0 km/h:300 km/h) vehicleSpeed,        EMA
25        in (0m : 50m) obstacleDistance, out (0% : 100%) brakeIntensity; }
```

```
26 component BrakeActuator {                                                  EMA
27   ports in (-90° : 90°) carDirection, (0 : 1) brakeIntensity,
28         out (0N : 5N : 200 kN) brakeForce[4]; }
```

```
29 Main-Component-Instantiation: ParkingAssistant<10> parkAssist;             Main.txt
```

Figure 7.37.: *EmbeddedMontiArc* model of `ParkingAssistant`. Connections and implementation part of atomic components are skipped in textual model, cf. graphical model in Figure 7.36 for connections.

The behavior of the `ParkingAssistant` (i.e., its concrete computation) is decomposed into several subcomponents each handling one specific task: managing the sensor data (cf. `SensorManager` in l. 8), calculating overall emergency brake effort (cf. `EmergencyBrake` in l. 9) depending on the distance, assigning concrete brake forces to each wheel (cf. `BrakeActuator` in l. 8) relative to the car's direction, and creating user feedback (cf. `Feedback` in l. 9).

The tasks of the `SensorManager` are so complex that this component (cf. ll. 10-14) is further decomposed into filtering signals (cf. component interface `Filter` in ll.10, 13), and fusioning sensor data (cf. `SensorFusion` in l. 14).

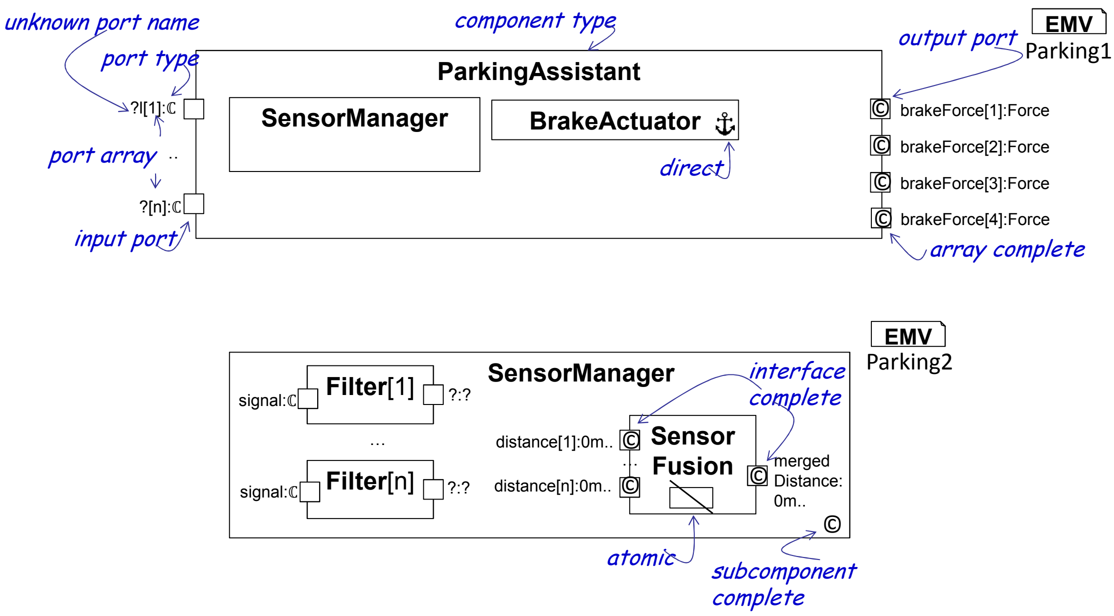


Figure 7.38.: Graphical design view of ParkingAssistant (graphical view is complete).

EmbeddedMontiArc does not support instantiating interfaces. Therefore, line 7 passes, in contrast to line d in Figure 7.39, the SimpleFilter component as configuration parameter to SensorManager, and line 13 instantiates this SimpleFilter component. The Simple-Filter component implements the component interface Filter (cf. l. 15), and thus, the SimpleFilter component has all the ports (cf. l. 18) of the Filter interface.

The connectors, depicted by solid arrow lines in Figure 7.36, represent directed data flows between subcomponents. The textual *EmbeddedMontiArc* model omits all connect statements as they are not needed for our witnesses later.

Figure 7.38 and Figure 7.39 show two parking assistant views. The first view has a generic abstract ParkingAssistant component with one abstract generic parameter n (cf. l. b). The abstract ParkingAssistant component has one unknown abstract input port (cf. l. c), which accepts complex numbers and has an array size of n, as well as the brakeForce abstract output port (cf. l. d), which emits values of the unit kind Force and has an exact array size of 4 (cf. exclamation mark in l. d). The abstract ParkingAssistant component has at least one abstract SensorManager subcomponent (cf. l. e) and one abstract BrakeActuator subcomponent (cf. l. f). Since in line e the abstract component parameter n is bound, the abstract component SensorManager has automatically at least one abstract generic parameter n. This is automatically derived even though the abstract SensorManager component is not modeled in the first view - this syntactic sugar enables textual views to concentrate only on important information of the specification; additionally, textual views do not need to carry around duplicated information in form of boiler plate code.

Since the second view is completely independent of the first one, it may be the case that the second view has redundant information as it is the case in line h where the second view also states that the SensorManager component has at least one abstract generic parameter n.

```
a  view Parking1 {                                                          EMV
b    component ParkingAssistant<n> {
c     ports in  C ?[n],
d           out Force brakeForce[!4]; // Unit Abstraction
e     instance SensorManager<n=n>;
f     instance direct BrakeActuator ba;                                   }   }
```

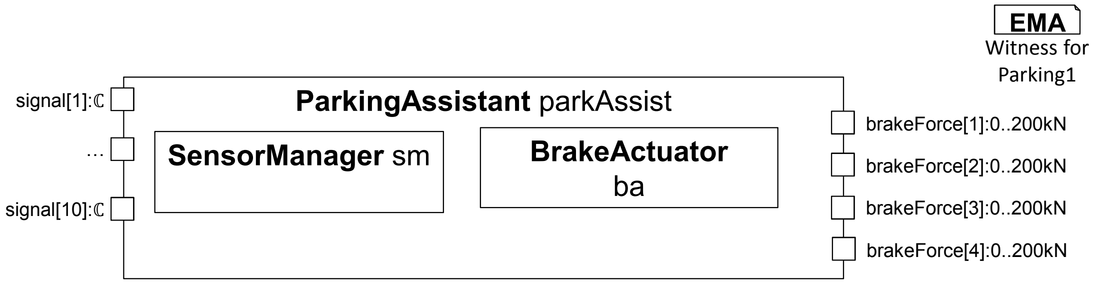
```
g  view Parking2 {                                                          EMV
h    component SensorManager<n>{
j      instances (c) Filter [n]; SensorFusion<n=n>;
k    }
l    atomic component SensorFusion<n> {
m      underspecification params Length maxDist, deltaDist;
n      ports (c) in   (0m:deltaDist:maxDist) distance[!n],
o                out  (0m:deltaDist:maxDist) mergedDistance;
p    }
q    component interface Filter {
r      ports  in  C signal,
s             out ? ?;                                                   }      }
```

Figure 7.39.: Textual *EmbeddedMontiView* model of Figure 7.38.

In line k, the second view adds additional information to `SensorManager` component type by forcing that a model that satisfies this view needs to have at least `n` subcomponents of the type (or a compatible type) `Filter` as well as one `SensorFusion` subcomponent type. Please note, C&C views can instantiate component interfaces which is not possible in C&C models. The `atomic` keyword in line l for the abstract component `SensorFusion` states that a satisfying `SensorFusion` component in a C&C model does not have any subcomponents. The abstract `SensorFusion` component also specifies an abstract generic parameter (cf. l. l). The under specification parameters `maxDist` and `deltaDist` (cf. l. m) specify that the port types of `distance` (cf. l. n) and `mergedDistance` (cf. l. o) are the same and that these port types are of unit kind length and that these port types start at `0m`. As the `ports` keyword is marked with an additional `(c)` in line n, the abstract `SensorFusion` component is interface-complete, meaning that a satisfying model must not contain more port names than these two specified ones. Line q defines an abstract component interface `Filter` having at least one complex typed abstract input port `signal` (cf. l. r) and one abstract output port with an unknown type and name (cf. l. s).

Figure 7.40 presents the generated witness to prove that the C&C model in Figure 7.36 satisfies the first C&C view in Figure 7.39. Line A in Figure 7.40 concretizes the abstract parameter `n` in line b in Figure 7.39 with the type information `N+` of line 1 in Figure 7.36. Line B is a witness for line c showing the port information of line 3. Line C is a witness for line d whereby the comment */* exact */* in the witness means that the array size of this port was exactly specified in the view. Line B does not contain this comment, because the view in line c only forces to have at least `n` port instances with a complex port type. Line C adds the type information of line 6 to the witness; the comment */* Force */* shows that the port type in the view is underspecified (cf. l. d).

Since in line e the instantiation for the abstract `SensorManager` subcomponent already binds the generic type parameter `n` to the abstract parameter `n` of the abstract `ParkingAssistant` component and it also binds the abstract `Filter` component interface, this information is also present in the witness in line D. And to show that the `SimpleFilter` is really a `Filter` as




```
A  component ParkingAssistant<N1 n> {
B    ports in C signal[n],
C            out (0N:1N:200kN) /* Force */ brakeForce[/* exact */ 4];
D    instance SensorManager<n>(SimpleFilter) sm;
E    instance /* direct */ BrakeActuator ba;                          }
```

```
F  component SensorManager<N+ n> (Filter F) { }
```

```
G  component BrakeActuator { }
```

```
H  component SimpleFilter implements Filter { }
```

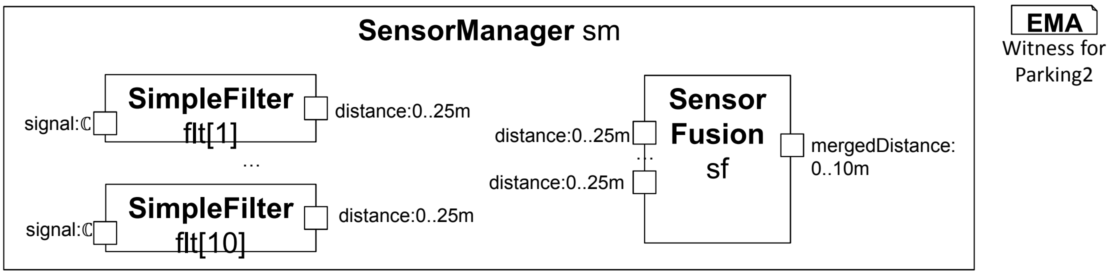
```
J  component interface Filter { }
```

Figure 7.40.: Graphical representation of *EmbeddedMontiArc* Witnesses (top part) and generated textual *EmbeddedMontiArc* witness for first view `Parking1` of Figure 7.39. The graphical and the textual witness shown in this figure are complete; both witnesses do not contain any connectors, because the C&C view in Figure 7.39 also does not contain any abstract connector or any abstract effector.

it is forced in line e, the witness also contains the lines H and J with the information of lines 15 and 17. Line E is the witness for lines f and 8; the */* direct */* comment indicates that this instantiation was directly forced in the view.

Figure 7.41 shows the second witness. Line A in Figure 7.41 corresponds to line h in Figure 7.39 and to line 10 in Figure 7.37. The configuration parameter `F` in line A is only present as this one is needed for subcomponent instantiation in line C, otherwise it would be omitted in the witness. The comment */* subcomponent instances complete */* in line B is added due to the `(c)` in line j. The subcomponent instantiations `flt` and `sf` (cf. lines C, and D) correspond to lines j, 13, and 14. Both (cf. ll. C, D) contain the parameter `n` as this one is present in the view.

The rest follows the same schema: Lines E, and F satisfy lines n, r, and s and map to lines 17 and 18. The generic parameter `T` (cf. line E) is only present as it is needed to match the unknown data type of the output port (cf. l. s). For the component `SensorFusion`, the underspecification parameters in line m force that the type of the two ports in lines n and o are the same and that they start at the interval `0m` as this shows the witness in lines H and J mapping to lines 20 and 21. The comment */* (c) */* in the witness in line H satisfies the `(c)` in line j and maps to lines 20 and 21. The */* exact */* comment satisfies the exclamation mark in line n




```
A  component SensorManager<N+ n> (Filter F) {                    EMA
B    /* subcomponent instances complete */
C    instance F flt[n];
D    instance SensorFusion<n> sf;                                   }
```

```
E  component interface Filter<T is Length> {                      EMA
F     ports in C signal, out T distance;                            }
```

```
G  component SensorFusion<N1 n> { /* (T) */                         EMA
H     ports /* (c) */ in  (0m : 0.2m : 10m) distance[/* exact */ n],
J           out (0m : 0.2m : 10m) mergedDistance;
K     implementation Math {}                                         }
```

Figure 7.41.: *EmbeddedMontiArc* witness for second view (Parking2) (this textual model is complete; the witness does not contain any connectors as the view does neither contain them).

and maps to line 20. Line K shows an empty implementation body to witness that the abstract component in the view has been marked as atomic in line l.

Every Satisfaction Witness satisfies its View If an *EmbeddedMontiArc* model M satisfies an *EmbeddedMontiView* V and W is the generated satisfaction witness, then W also satisfies V and the generated witness W' is the same as W.

This rule follows directly from the construction of the witnesses and this property is inherited from Ringert's C&C witness construction: "Interestingly, all witness are their own witness for satisfaction, when checked against the same C&C model." [Rin14, p. 57].

Satisfaction Witnesses may not be minimal The satisfaction witness is only locally minimal (minimal for one abstract connector, or for one abstract effector), but it may be already not minimal for two abstract connectors. Since the textual models of *EmbeddedMontiArc* and *EmbeddedMontiView* are finite, it would be possible to calculate a global minimal satisfaction witness (cf. discussion in Ringert [Rin14, Subsection 4.5.2 on p. 87]).

Ringert's witness generation algorithm does not handle abstract effectors, and therefore Ringert's witnesses are minimal in number of components. But the algorithm, specified in this thesis, creates for abstract effectors connector-effector-chain instances also containing component instances, and similar to abstract connectors (both using a breadth-first search), the algorithm does not create a global minimal for all effector chains.

```
EMV
1  view Redundancy {
2    component Cmp1 {
3       ports in ? port1;
4       instance Cmp2 cmp2;
5
6      connect this -> cmp2;
7      connect this.port1 -> cmp2; // Is 1.7 subsuming 1.8? It is not clear!
8      connect this -> cmp2.port2;
9      connect this.port1 -> cmp2.port2;
10   }
11   component Cmp2 {
12      ports in ? port2;                                          }  }
```

Figure 7.42.: Example redundant connectors.

Also as already discussed in rule (11), it might be possible to remove elements from the witness, and then the witness would still satisfy the view. At the first thought an easy sorting of abstract connectors from most concrete to most general may resolve this problem, then at least one abstract connector would not have multiple satisfaction chains in the witness. For example, line 9 in Figure 7.42 is more concrete than line 7 and line 7 is more concrete than line 6; and line 9 is more concrete than line 8 and line 8 is more concrete than line 6. However, line 7 is not more concrete than line 8 and line 8 is not more concrete than line 7. Thus, it is not always clear when an abstract connector is more general then another abstract connector, and so no fast algorithm (by only sorting elements) for generating a global minimal witness exists.

Even though we do not have minimalism for the generated satisfaction witnesses, we still have one very user-friendly property:

If V is a view, M a model satisfying V, W is the generated satisfaction witness, and V' is derived from V by adding view elements at the end (corresponding to the defined order in rule (11)) as well as M still satisfies V', then the generated satisfaction W' contains all the elements of W. This means that the witness structure does not completely change by adding new constraints to the view as long as the model still satisfies the view. This is especially useful when showing witnesses directly, e.g., during the textual creation of views for already existing models. This property would not be true, if we would create globally minimal witnesses or if we would sort the abstract connector or abstract effectors according to any metric.

7.5.2. Tracing Witnesses

The case study with Daimler AG [BMR+17a] (the next chapter provides more information about this case-study) unveiled two new application areas of C&C views: documentation and tracing. One drawback of satisfaction witnesses as presented in the previous subsection is that they show only elements needed to satisfy the satisfaction relation (e.g., only one connector chain for an abstract connector); even though many other model elements in the component and connector model or its derived component and connector instance structure would also satisfy the given view element (e.g., showing all connector chains satisfying one abstract connector). Tracing witnesses also enable to use *EmbeddedMontiView* as a convenient query language for *EmbeddedMontiArc* to find important information in very large models. This tracing information is useful for internal audits or scrum meetings.

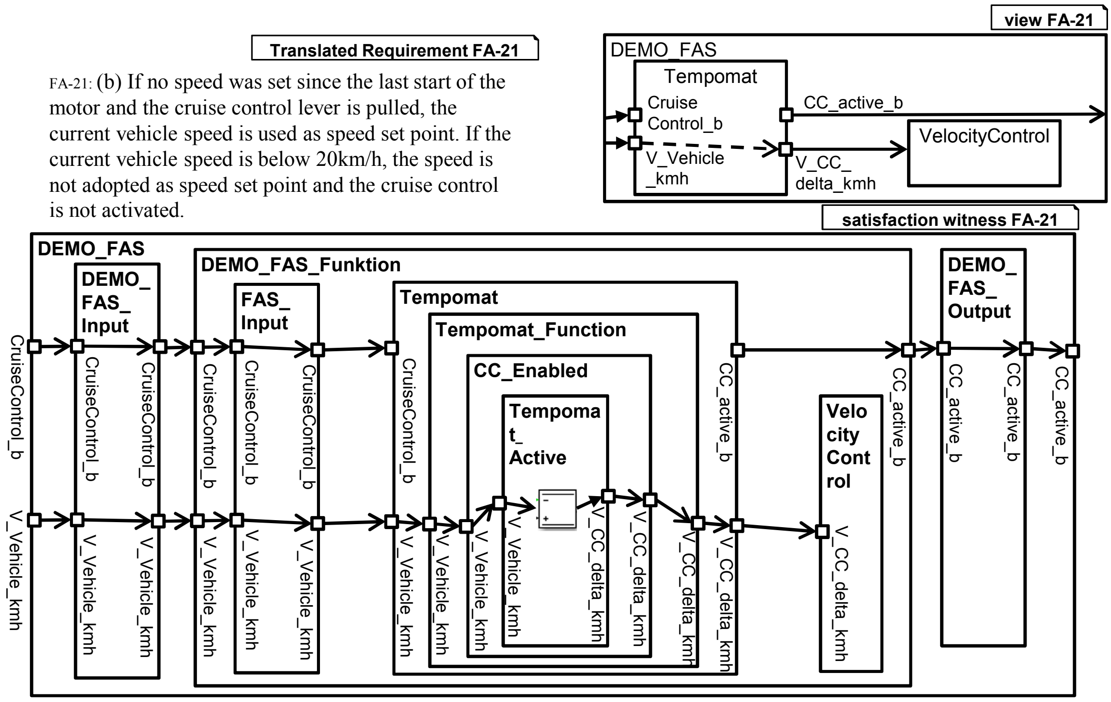


Figure 7.43.: Requirement, derived view and Satisfaction Witness (requirement and view copied from [BMR+17a]). Some witness components have the same port names for input and output ports - this is because the names in the graphical witness are the displayed names of the *Simulink* model (cf. Subsection 8.3.5).

Ringert [Rin14, Subsection 4.5.3 on p. 87] suggests an alternative representation inside the corresponding model. For tracing witnesses and for using *EmbeddedMontiView* as query language to find the right components, the author of this thesis thinks that this alternative representation of witnesses is well suited (esp., for graphical models such as *Simulink* or textual models having an automatic visualization algorithm as it is the case for *EmbeddedMontiArc* [Sch18]). The tracing witness generation applies rules (1) to (11) in Subsection 7.5.1 as far as possible; the only difference is that the tracing witness algorithm does not stop if it founds one witness satisfying a view element.

Figure 7.43 shows the satisfaction witness according to the view *FA-21*. Figure 7.44 displays the tracing witness of the same view. The two figures of the tracing witness show only the graphical representation of the tracing witness. Besides displaying the graphical representation of the tracing witness, it is also possible to copy the original C&C model and highlight all tracing elements in the witness. One highlighted *Simulink* model representing a tracing witness is available under[5]:

```
http://www.se-rwth.de/materials/cncviewscasestudy/ADASv4_FA21/webview.html
```

[5]If in *Simulink* a name started with an underscore, then this is ignored in the Figures as this is only a technical constraint and it is filtered out during our translation process [BMR+17a]. Our transformation resolves virtual busses, which are only used to group signals visually, into their own lines. Additionally, internal variables are also transformed to input and output ports; as it is done in Figure 7.44 for the variable `DEMO_FAS_V_CCSetValue` (in the figure the `DEMO_FAS` prefix has been skipped).

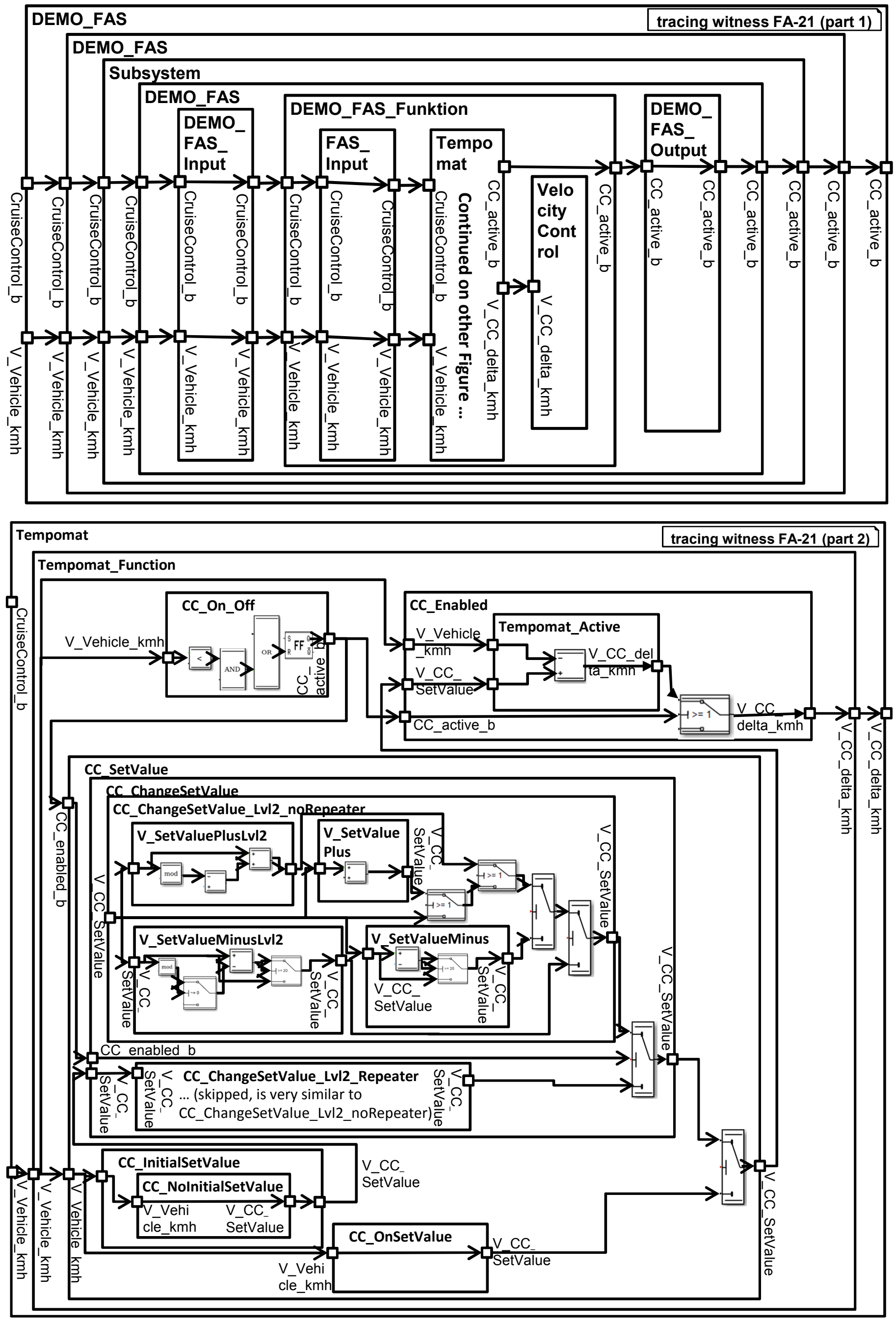


Figure 7.44.: Tracing Witness (the inner part of `Tempomat` shows nearly all connector-effector-chains going from `V_Vehicle_kmh` port to `V_CC_delta_kmh` port; this means this figure is the tracing witness for one abstract effector presented in the view in Figure 7.43. Connector-effector-chains of `CC_ChangeSetValue_Lvl2_Repeater` component are skipped.)

Since the complete model with over 650 blocks and more than 1 500 ports is too large, this thesis does not show any complete highlighted tracing witness of our case study. However, Section A.2 contains screenshots of some graphical layers to illustrate how highlighted tracing witnesses look like.

In contrast to Figure 7.43, the top part in Figure 7.44 contains the most-outside `DEMO_FAS` component[6]. The tracing witness contains all elements satisfying any element of the view and the most outside satisfies it the same way as the most inside one, because component contains relations in views are always indirect (unless otherwise stated with the `direct` keyword). The satisfaction witness uses according to rule (1) the least common parent to satisfy the hierarchy constraints, and this is already satisfied by the most inside `DEMO_FAS` component.

The tracing witness of the abstract effector going from `Tempomat`'s `V_Vehicle_kmh` port to `Tempomat`'s `V_CC_delta_kmh` port is presented in bottom part in Figure 7.44. In contrast to the satisfaction witness (cf. Figure 7.43) where only one shortest path is shown, the tracing witness shows all paths in the model going from the input port to the output port. This enables finding all functions (calculations, components and their interaction) which are involved in the relationship between the input and output port. The tracing witness in Figure 7.44 is not even complete (as the connector-effector-chains inside the `CC_ChangeSetValue_Lvl2_-Repeater` component are skipped). This means effects from one input to an output port may be very complex and the tracing witness helps understanding their relationships. The presence of all components between an abstract effector's source and target port helps to narrow down an error if the starting and end point is known.

7.5.3. Non-Satisfaction Witnesses

Similar to Ringert [Rin14, Subsection 4.3.3 on page 62], the verification algorithm creates its own C&C witness model for every non-satisfaction reason. This means for every C&C view element, there exists a rule how to create a non-satisfaction witness when this view element is not satisfied by a corresponding model. This subsection summarizes the rules for creating these witnesses. The rules are ordered according to *EmbeddedMontiView*'s main features in Section 7.1.

(i) **Hierarchy abstraction**
Similar to Ringert [Rin14, Table 4.10 on p. 66], the algorithm distinguishes between three different cases: (a) hierarchy is reverse; (b) two components are independent in the view, but not in the model; and (c) two components are not independent in the view but they are in the model.

(ii) **Connectivity abstraction**
Similar to Ringert [Rin14, Table 4.11 on p. 66], exists the case (a) where an abstract connection is present in the C&C view, but no connection chain is present in the model. As a concretization, our algorithm additionally supports the case where (b) an abstract connection is present in the view, and the direction of the connection chain is switched (goes from abstract connector's target port to its source port). Furthermore, the algorithm supports the case (c) where an abstract connection is present in the view, but only a

[6] The *Simulink* model contains three times a subsystem with the name `DEMO_FAS` - e.g., the most outside one is needed for configuring TargetLink

connector-effector-instance chain is present in the model (assume that case (b) is not true). This is a special case of Ringert's (a) [Rin14, Table 4.11 on p. 66]; however, we separated it to give better error messages indicating that the specification may used the wrong abstraction type. In addition, there exists the case (d) where the connection chain is present but too less indices in port or component arrays are connected, the case (e) where too many indices for ports are connected in the model, and case (f) where the wrong indices are connected.

(iii) **Incomplete interfaces**
The first three cases of Ringert [Rin14, Table 4.12 on p. 67] are also present in this algorithm: (a) A port with a given name is present in the view but not in the model, (b) a specified port has the wrong direction, and (c) a specified port has the wrong type. Additionally, case (d) the model does not contain enough different ports is present, since in *EmbeddedMontiView*'s abstract port instance can have no type and no name - Ringert [Rin14, Section 3.6 on p. 40ff.] supports only to skip the name or the type, but not both, because Ringert models C&C views as *MontiArc* models using the *MontiArcView* profile containing of stereotypes.

(iv) **Atomic, Direct and Complete Subcomponents as well as Interface Complete**
If a view contains an atomic abstract component and the corresponding component is not abstract in the C&C model, then the witness contains this component plus all direct subcomponents to prove that it is not atomic.

If in a view the abstract component instance A contains directly the abstract subcomponent instance B, and it is violated in the C&C model than the witness contains all components needed to model the hierarchy between A and B.

If in a view an abstract component is marked as subcomponent complete and the corresponding model has more subcomponents, then the C&C model witness contains only the additional subcomponents having no match in the C&C view.

If ports of an abstract component are marked as interface-complete and it is not the case in the corresponding C&C model, then the witness contains the corresponding component plus all ports which are in the C&C model and not mentioned in the C&C view to show that the model has too many ports.

(v) **Data Flow Abstraction**
Similar to connectivity abstraction we have the cases: (a) for an abstract effector does not exist a connector-effector chain, (b) for an abstract effector exists only an inverse connector-effector chain, (c) for an abstract effector exist only connector chains[7], (d) connector-effector chain is present but too less indices in port or component arrays are affected, (e) the view limits the effects of port indices in connector-effector chains and this limit is violated, and (f) the wrong port indices affect each other in the connector-effector chain.

[7]This is not an error, but it generates a warning to indicate that the too powerful abstract effector is used and an abstract connector would be more accurate.

(vi) **Support of Component Types**
Similar to Ringert [Rin14, Table 4.9], the algorithm throws an error if the (a) view contains a component type which is not present in the model. In Ringert the matching is done on the names as the names are the unique types. Additional to (a), the following new cases exists: (b) the component type is an interface in the view and there exists no compatible component type in the model; (c) if the component type in the view has parameters which are not present in the model; and (d) if the parameter type or values in the view are not compatible to the parameter type in the model.

(vii) **Unit Kind Abstraction**
If in a view a port type is constrained by a specific unit kind (e.g., `Length`), and the port type of the corresponding port in the model does satisfy this unit kind, than an witness containing the component with this port is generated.

(viii) **Matrix Property and Dimension Abstraction**
Similar to unit kind abstraction, this is also a kind of a special issue to incomplete interface abstraction as a given port also violates a constraint, e.g., the algebraic property, or the matrix dimension. Since one matrix can have multiple algebraic properties, (a) the witness contains only these properties which are violated. For the wrong dimension we separate between (b) dimensions are switched, as it often occurs when some modeler favors to work with column vectors and another with row vectors; and the other case (c) where the dimensions do not fit at all.

(ix) **Port Array Abstraction**
There are two cases for the non-satisfaction of port arrays: (a) the port array size is too small or (b) the port array size is forced with a limit using the exclamation mark. In both cases the non-satisfaction witness contains the component and the port including the dimension (dimension of one is explicitly modeled as `[1]`), only the textual message describing the cause of the mismatch differs for case (a) and (b).

(x) **Component Instance Array Abstraction**
The witness structure is similar to port array abstraction.

The bottom part of Figure 7.45 shows an example of a non-satisfaction witness. This negative non-satisfaction witness is generated by checking the C&C view in the top part of Figure 7.45 against the park assistant C&C model in Figure 7.36 on page 256. Similar to Ringert [Rin14, p. 65ff.], every non-satisfaction witness includes two parts: The C&C model showing the reason for violating the C&C view constraint plus a natural language description. The non-satisfaction witness is generated by the template *(ii) connectivity abstraction*.

The case study with Daimler AG unveiled that the natural language is often more helpful than the actual C&C witness model. Especially, non-satisfaction witnesses of abstract connectors (or even abstract effectors) are very large, since the non-satisfaction witness contains all possible connection chains starting at the source port of the abstract connector and ending at any target port being different than the one of the abstract connector.

Further examples of non-satisfaction witnesses are available in the C&C view case study paper [BMR+17a, Fig. 3] or in the bachelor thesis *Extension of the C&C View Language and its Verification for Embedded Systems* [Kah17b, Fig. 4.7 on p. 23], [Kah17b, Fig. A.9 on p. 46].

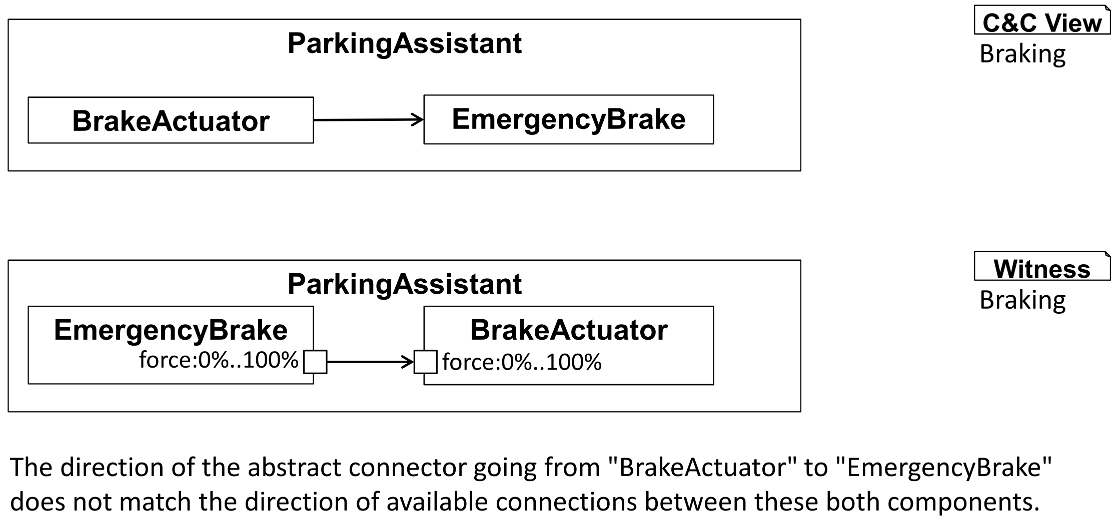


Figure 7.45.: Example of a non-satisfaction witness for the given C&C view and the ParkingAssistant C&C Model in Figure 7.36 on page 256.

Chapter 8.

Industrial Case Study on Component and Connector Views

The previous chapter introduced *EmbeddedMontiView*, a high-level design language for component and connector (C&C) models of embedded systems. This chapter presents a case study on component and connector views based on industrial-size *Simulink* models provided by Daimler AG. Most content of this case study (which was a team work together with Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and me) has already been published in our conference papers [BMR$^+$17a, BMR$^+$18].

The case study together with Daimler AG translated *Simulink* models to C&C models [Bru17b], and automotive domain experts modeled the C&C views in PowerPoint. Later, the graphical C&C views were manually transformed to textual *EmbeddedMontiView* models. For the 2017 case study, we also translated the textual witnesses, produced by the C&C views verification tool, to graphical representations in PowerPoint in order to discuss the C&C views verification results with the industrial partner. The aim of the 2017 industrial case study was to figure out scenarios where C&C views and its verification may support developers in industry.

Due to the master theses of Manual Schrick [Sch18] and Tayfun Özen [Oez18], our C&C views verification toolchain is able to generate the graphical representation of *EmbeddedMontiArc*, the language for C&C models and generated witnesses, automatically. In 2018 we executed a subsequent study to evaluate the complete toolchain including the graphical representation of the witnesses. The aim of the 2018 industrial case study was to analyze how helpful the generated graphical witnesses are and how much development time the C&C views verification toolchain may save developers in the scenarios identified in the 2017 case study.

8.1. Overview of Three Stages of Industrial Case Study

This section introduces the main research questions of the industrial case study applied in an automotive setting. The questions Q1 to Q4 are already discussed in our conference papers [BMR$^+$17a, BMR$^+$18].

- Q1 Which challenges in automotive contexts can be addressed by C&C views?
- Q2 How much effort do experts need to create C&C views and do experts miss any features of C&C views?
- Q3 Does C&C views verification work on existing automotive industry models and is its verification time for large C&C models reasonable?

Q4 Are the satisfaction witnesses of the C&C views verification of use for the engineers?

Q5 How helpful are the graphical representations of tracing witnesses of C&C views verification?

Q6 How much time need engineers with/without C&C views verification to detect important elements?

The case study execution has three stages: The first stage answers the first research question Q1. The second stage answers the research questions Q2 to Q4 based on the results of Q1. The research questions Q5 and Q6 were identified during the first part of the case study execution in the first half of 2017. Therefore, a separate case study part for these research questions was done at the end of 2018. This chapter refers the first stage as preliminary study, the second stage as main study, and the third stage as subsequent study. The preliminary study and the main study were executed in the first part in 2017, the subsequent study was executed in the second part in 2018.

The main task of the preliminary study was to find and interview automotive industrial partners to understand the general industrial development process in the automotive domain. Additionally, the aim of the preliminary study was to identify the most time consuming tasks which can be addressed by C&C views and their verification.

The main study was executed on four different evolution models of an advanced driver assistance system (ADAS), and an adaptive light system (ALS); both systems represent safety-critical, distributed control systems [PBKS07]. The main study was executed together with Daimler AG due to existing automotive research collaborations and the availability of models and requirements which can be made public available. We want to thank Daimler AG to provide us all these artifacts and to allow us to make them in a restricted way[1] available to the public. Two domain experts created together 50 C&C views based on 183 textual industrial requirements generated from *IBM Rational DOORS*. The most challenging part of the main study was to translate the five *Simulink* block diagrams to C&C models, so that these models can be verified against the created C&C views (cf. Section 7.4 for verification algorithm). The witnesses of the verification tool were large textual models and so hard to understand. Therefore, we manually created graphical C&C models in PowerPoint matching the textual witnesses. The linguistic output messages and the graphical C&C models have been showed to the two domain experts of this case study to evaluate the helpfulness of these witnesses according to the two identified challenges: evolution and traceability.

During the translation process of textual requirements to C&C views, the industrial partner identified a missing abstraction concept in C&C views. This was the hour of birth of the abstract effector (cf. Subsection 7.3.8). The main case study unveiled that domain experts can easily create C&C views based on given requirements to highlight implementation details in *Simulink* models. However, the industrial partner noticed that the generated satisfaction witnesses[2] did not contain all implementation elements. To address this issue, a new kind of positive witnesses - tracing witnesses - have been added to C&C views verification. Tracing witnesses enable even more accurate tracings between requirements and *Simulink* models.

[1] Daimler AG granted us the rights to upload web exports of the *Simulink* model to our homepage, so that reviewers are able to inspect them. However, we are not allowed to upload the executable *Simulink* models themselves.

[2] Tracing witness were invented in 2018 based on the feedback of the main case study.

The main study also showed that C&C views verification scales for industrial models, and the verification algorithm even returned the result immediately (average execution time of verification algorithm was always below two seconds in all our experiments).

The main study on C&C views helped the domain experts to discover several inconsistencies between requirements and their implementations (cf. [BMR+17a], Section 8.4, and Section A.3). The subsequent study was carried out more than one year later after the visualisation algorithm has been successfully implemented. The subsequent study evaluated whether the generated graphical tracing witnesses helped to identify all for a requirement important *Simulink* blocks.

The author of this thesis spend much effort to make all artifacts of both parts of this industrial case study executed in 2017 and 2018 public available in a convenient way by creating several web pages. The material is public available from *EmbeddedMontiArc*'s GitHub pages[3]. These materials include the web exports of the five *Simulink* models provided by Daimler AG, original textual requirements in German and an English translation, 55 textual and graphical C&C views inclusive a colored mapping to see which textual fragments resulted in what C&C view element, verification results, textual and graphical models of satisfaction and tracing witnesses, as well as many statistics about these two case study parts.

All three stages (i.e., preliminary, main, and subsequent study) of this industrial case study follow the guidelines of Runeson and Höst for conducting and reporting case studies in software engineering [RH08]. Specifically, each stage section defines research questions, the objective, theory, method, and selection strategy, as well as it presents hypotheses, case study execution, and results to answer these research questions.

8.2. Preliminary Study

The preliminary study investigated research question Q1: Which industrial contexts in automotive domain are relevant for C&C views and what challenges can C&C views address?

This question has been split into the following subquestions:

Q1a What industrial development processes in the automotive domain may C&C views address?
Q1b What industrial artifacts are public available for this industrial case study?
Q1c What documents are suited to create C&C views?

8.2.1. Execution of Preliminary Study

The **Objective** of the preliminary study explored industrial settings in automotive domain using C&C models; we skipped all development steps implementing C/C++ code directly. Of specific interest were the challenges developers are facing in daily life during the industrial development process to figure out where C&C views verification may assist developers.

Furthermore, one main aim was to find an industrial partner for the main study. Finding an industrial partner for our main study was not easy, because the participating industrial person needs experience in C&C modeling and the person needs to spend altogether two weeks of working time for our main study.

[3] `https://embeddedmontiarc.github.io/webspace/`

Additionally, the author of this thesis wants to have a comprehensible case study, and so the data plus development models should be made public available in a restricted way. This was an obstacle for many industrial partners, because anonymizing (replacing configuration parameters in models with random data) models and data also takes much time. Daimler AG spends six months working time to remodel the *Simulink* models to provide all research partners of SPES XT [Man15] a public industrial demonstrator containing the complexity and structure of real-world industrial models, but not containing any protected intellectual property anymore.

The **Theory** of the preliminary study is based on the ability of C&C views to express structural properties on C&C models (cf. Section 7.3), plus the automatic C&C views verification (cf. Section 7.4) with its intuitive witness generation (cf. Section 7.5).

The **Method** included the following activities: First, establishing contact with previous automotive partners of the Prof. Rumpe's chair; examples of partners are DSA [Mül18], VW [BBH+15a, BBH+14b], FEV [RSRS15, RRS+16, KMS+17], BMW [GHK+08b, KKRvW18, HKK+18, KKRvW18], Daimler [RSvW+15, BRRvW16, BRvW16, BMP+16], E-Go [RWT18a], and Thales Group [ZPK+11]. We want to explain the partners the aim of our study. Second, a two- to three-day long workshop should introduce the main concepts of C&C views and their verification to the chosen industrial partner based on already existing examples. The workshop should also be the starting point for many informal discussions to get some insights about the current development process and the challenges which could be addressed by C&C views and their verification.

As already mentioned the **Selection Strategy** of industrial contacts was based on former and current research collaborations of Bernhard Rumpe. The industrial case study was done in collaboration with Daimler AG, because the author of this thesis already worked together with Daimler AG, esp. with Bertram Vincent, in previous collaborations on evolution of *Simulink* models [RSvW+15, BMP+16] provided by Daimler AG, and therefore, the author of this thesis already had a very good understanding of the involved *Simulink/TargetLink* tool infrastructure as well as a pretty good overview of the models. Additionally, *Simulink* models look kind of similar to our C&C models. Furthermore, the public demonstrator models contain user-experience features selectable in car configurations at German car dealers; and so most readers of this thesis understand the underlying domain of these models. During the preliminary study, the author of this thesis could also inspect other domains of C&C models, i.e., engine control or battery charging; however, explaining these models require at least 20-pages of background material about electrical and mechanical engineering.

8.2.2. Results of Preliminary Study

The following subsections presents the identified challenges and it explains the two *Simulink* models provided by Daimler AG. This chapter skips the development process of Daimler AG as Subsection 2.1.1 explains it already in detail.

Based on the suggestion to improve the development process in Subsection 2.1.2, the answer of research question **Q1c** is that C&C views can be created based on textual requirements of these given *Simulink* models. The names of C&C views are the *IBM Rational DOORS*' identifiers of the corresponding requirements.

Identified Challenges

The main aim of the preliminary study is to find existing challenges in the context of Component and Connector models that C&C views verification can address or even solve. Vincent Bertram, an employee of Daimler AG, identified Traceability and Evolution as challenges to address for this C&C views case study.

Traceability/Documentation. Traceability creates links between artifacts impacting each other [BQ06]. Requirement traceability links domain model elements or code fragments to requirements they are implementing. In safety relevant domains requirement traceability is mandatory; many norms such as DO178C *Software Considerations in Airborne Systems and Equipment Certification*, ISO26262 *Road vehicles - Functional safety*, and IEC61508 *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems* provide guidelines for traceability.

At Daimler AG documentation and tracing of requirements is done for the following purposes [BMR+17a]:

1) **Preparing technical reviews** after implementing requirements; this is done once a week for sprint reviews. To create review documents, e.g., PowerPoint presentations, engineers need to locate relevant blocks and their interactions.
2) **Improving** (e.g., decrease memory usage or increase runtime performance) **the implementation** of a requirement. During this phase, the engineers needs to identify relevant blocks and information flows between them to modify the requirement's implementation.
3) **Testing** of user stories based on **requirements**. In this case, software testers need to find all relevant subsystems and ports according to the requirements of the user story in order to set up tests.

As already explained in Subsection 2.1.1 engineers, asked at Daimler AG, add tracing information manually (similar blocks as the `VERSION_INFO` one shown in the *Simulink* web export [Dai13g]) to *Simulink* subsystems. These blocks list the IDs of (and automatically link to) textual requirements implemented by the subsystem.

Evolution. Software evolution is the repeated change of software architectures and their implementations for various reasons [BR00]. For the industrial partner the evolution challenge is: What is the impact of changing user-experience or architectural requirements to the *Simulink* implementation? Based on this decision the amount of work (and thus, time and cost) can be estimated for software evolution.

According to our interviews, the following scenarios occur at Daimler AG [BMR+17a]:

1) **Adding or changing a requirement**. In this scenario, the engineer studies the existing implementation and analyzes what other requirements have impact on this implementation.
2) **Evolving the model**, e.g., to add new functionality. During this process, engineers check that the new implementation does not violate other requirements.
3) **Refactoring of models**. During time more and more features are added to a *Simulink* model, and so the architectural design of this model must be cleaned up, e.g., by splitting too large subsystems into multiple smaller ones, and thus, changing the names of signals. After a refactoring, engineers check whether the new *Simulink* model still satisfies all requirements.

The process description [Man13], created by an experienced employee at Daimler AG, states that engineers at Daimler AG manually determine whether a new set of requirements are (backward) compatible to the existing version. The author of this documents also states that the documentation of component dependencies is very complex (cf. [Man13, point 3 in Section 2.4]) and that updating (e.g., bug-fixing one component version) is very risky and not comprehensible if no good documentation exists (cf. [Man13, point 1 in Section 2.4]).

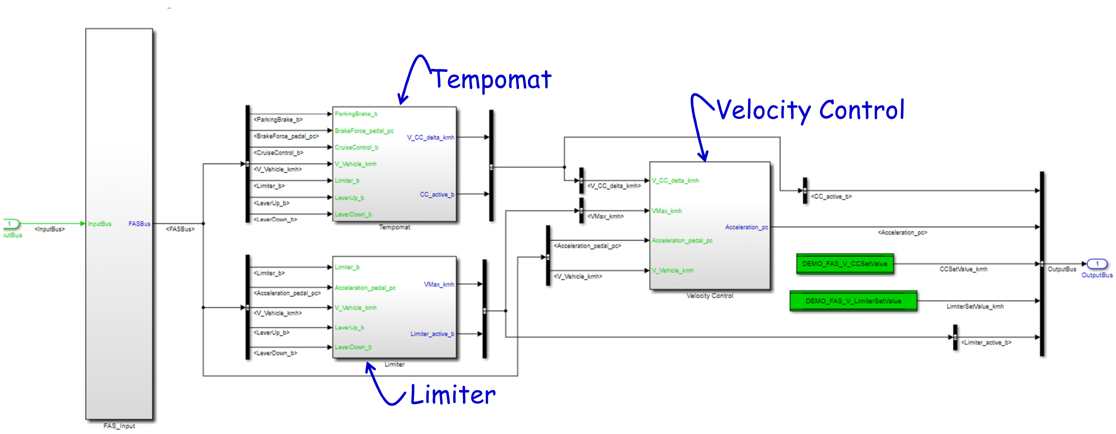


Figure 8.1.: Functional layer of ADAS version 1 (complete model is available from: [Dai13c])

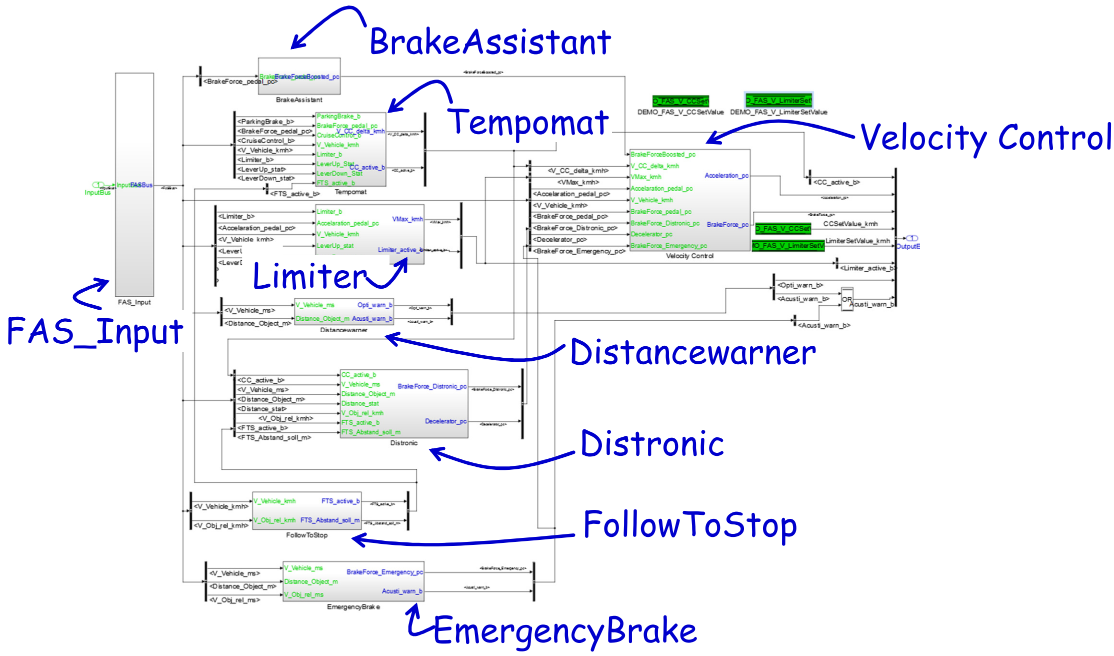


Figure 8.2.: Functional layer of ADAS version 4 (complete model is available from: [Dai13g])

LH-Version	Change Description	Editor	(Date)	Approved	(Date)
V02	Adding Two Stage CruiseControl Lever, Brake Assistant	A. Maisch	20.05.13	L. Muster	31.05.13
V03	Adding Tempomat (Cruise Control) with Brake Functionality, Sign Detection, Distance warner	B. Noch	16.06.13	L. Muster	20.06.13
V04	Adding Adaptive Cruise Control, Emergency Brake, Traffic Jam Following	F. Houdek	01.07.13	L. Muster	01.08.13

Figure 8.3.: Screenshot of change log of requirement document version 4 for ADAS by Daimler AG [Dai13k, p. 3].

Available Models

This paragraph introduces the two different industry models: The Advanced Driver Assistance System and the Adaptive Light System. The first one consists of four different *Simulink* models containing the four evolution steps.

Advanced Driver Assistance System (ADAS)

The main task of an ADAS is to assist the driver in the overall driving process to increase the general road safety. The four evolution models of the ADAS provided by Daimler AG receive as input (cf. left input ports in Figure 8.1 and Figure 8.2) the current sensor data of a vehicle such as current vehicle speed, detected speed sign, as well as current speed, and distance of detected objects in front of the vehicle. Based on this sensor data plus the current user input, the ADAS calculates the brake force and acceleration values of the current vehicle as well as optical and acoustical feedback signals. Examples of signals based on driver's input are: activating the parking brake, angle of acceleration and brake pedals, movement direction of cruise control lever, and (de)activation of cruise control by pressing a button.

Daimler AG gave us four different versions, as shown in Figure 8.3, of the ADAS system (text is borrowed from requirements of ADAS [Dai13k]):

1. ADASv1 is the oldest version of the ADAS system. This system has only the following user-experience features:
   a. Cruise control so that the vehicle automatically accelerates to reach the set speed of the driver, and
   b. Limiter lets the vehicle automatically brake if the car is getting to fast (e.g., when driving downhill decreasing the acceleration may not be enough).
2. ADASv2 extends the first version. This system has the following changes of user-experience features:
   a. The cruise control lever with three values down, neutral, and up is replaced by a two-stage lever having two values for down and up, and
   b. It adds brake assistant which automatically sets the brake force to 100% when the driver pushes the brake pedal hard enough.

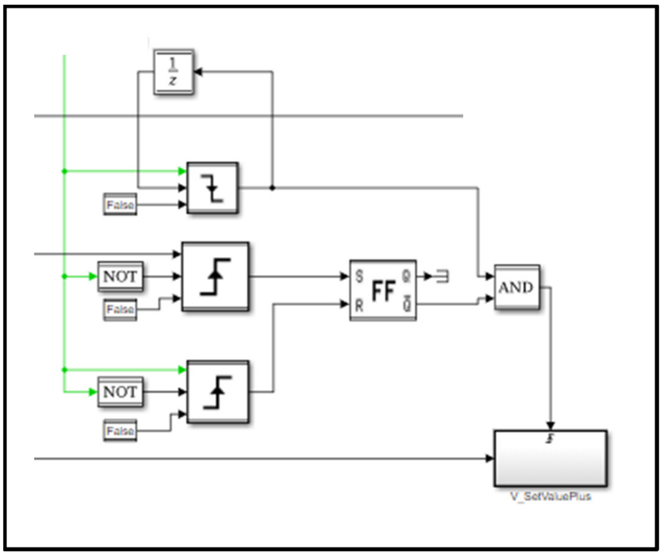
3. ADASv3 extends the second version and it adds the following user-experience features:
   a. Cruise control supports maintaining the speed-dependent safety distance automatically,
   b. Sign detection which is coupled with the limiter to avoid speeding,
   c. Distance warner notifies the driver with an optical and acoustical signal if the distance to the car in front is getting too close.
4. ADASv4 is the last and most complete ADAS system in our case study. This system extends version 3 with the following user-experience features:
   a. Traffic jam following enables the vehicle to accelerate from a standstill when the vehicle in front starts driving again,
   b. Distronic (also called adaptive cruise control) extends the cruise control and limiter so that the vehicle brakes until a full standstill if necessary, and
   c. Emergency brake assistant automatically brakes when the distance to the car in front is getting in a critical range.

Figure 8.1 and Figure 8.2 show the hierarchy levels of the two *Simulink* models (on hierarchy level 6) of ADASv1 and ADASv4, which contain the subsystems representing the above mentioned user-experience functions. Section A.1 contains more screenshots of the ADASv4 *Simulink* model.

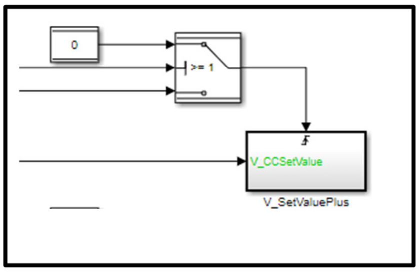
Table 8.4 presents statistics about the *Simulink* models provided by Daimler AG. The first row contains the number of *Simulink* blocks; it also includes all atomic blocks and subsystems, but it excludes inport and outport blocks as these are counted as ports. The second row contains the number of subsystems which are all blocks of the first row not being atomic ones. The third row reports the hierarchy depth of the models; the hierarchy depths says how deep subsystems are nested to describe complex functionalities. The fourth row lists the number of `VERSION_-INFO_BLOCK` blocks; engineers at Daimler AG added for every important functional block

	ADASv1	ADASv2	ADASv3	ADASv4	ALS
Blocks (not counting port blocks)	327	686	664	655	1065
SubSystem	122	211	203	195	184
Hierarchy Depth	12	12	12	13	10
VERSION_INFO_BLOCK	27	43	49	48	24
Total blocks	550	1050	1030	1044	1646
Ports (also counting ports of atomic blocks)	701	1454	1480	1513	2753
EnablePort/TriggerPort/ ActionPort	4/9/5	3/19/11	3/19/14	6/15/11	0/0/10
BusCreator/BusSelector	10/13	14/15	19/20	20/27	22/25
DataStoreMemory/DataStoreRead/ DataStoreWrite	2/7/9	2/10/19	2/10/19	2/8/15	0/0/0
UnitDelay (internal variables)	9	43	18	15	35

Table 8.4.: Statistics about the different *Simulink* models. More statistics about the *Simulink* subsystems are available from: [vW17b].



More Detailed Control Theory
(Excerpt of Subsystem
CC_ChangeSetValue_Lvl2_Repeater
of ADASv2)



Controller Part is replaced by Switch
(Excerpt of Subsystem
CC_ChangeSetValue_Lvl2_Repeater
of ADASv3 and ADASv4)

Figure 8.5.: Control theory details in ADASv2 compared to details in ADASv3 and ADASv4.

such an information block containing the copyright information and tracing information to its requirements. The fifth row contains the total number of blocks (without port blocks of atomic blocks). The sixth row lists the number of port blocks which is the sum of all inport and outport blocks of subsystems and atomic blocks; however special control-flow ports listed in the next row are not counted. The seventh row presents the numbers of `Enable`-, `Trigger`-, and `ActionPorts`; these ports model control-flow and so they cannot be 1:1 transformed to ports of a C&C model. The eighth row contains the numbers of bus creator and bus selector blocks; in *Simulink* these blocks are only there to graphically group signals to design a readable representation without too many crossing signal lines. The ninth row contains the number of global variables, and how often these variables are read and how often they are written. The last row lists the number of unit delay blocks which store the value of the previous time step, e.g., to examine whether the current value is larger or lower than the previous one.

ADASv2 has with 1 050 total blocks and 43 unit delay blocks more blocks than all the other ADAS versions even though it has less functionality. The reason is that ADASv3 and ADASv4 only contain simple placeholders for controllers regulating the increase or decrease of the car's velocity inside the tempomat's subsystem (probably due to intellectual property reasons). Figure 8.5 illustrates this fact. The left side shows an excerpt of the control part of the Repeater in ADAS version 2 (full model available from [Dai13d]). The right side shows an excerpt of the simplified model in ADASv4 where the controller has been replaced by a constant zero and a simple switch block (simplified model available from [Dai13e]).

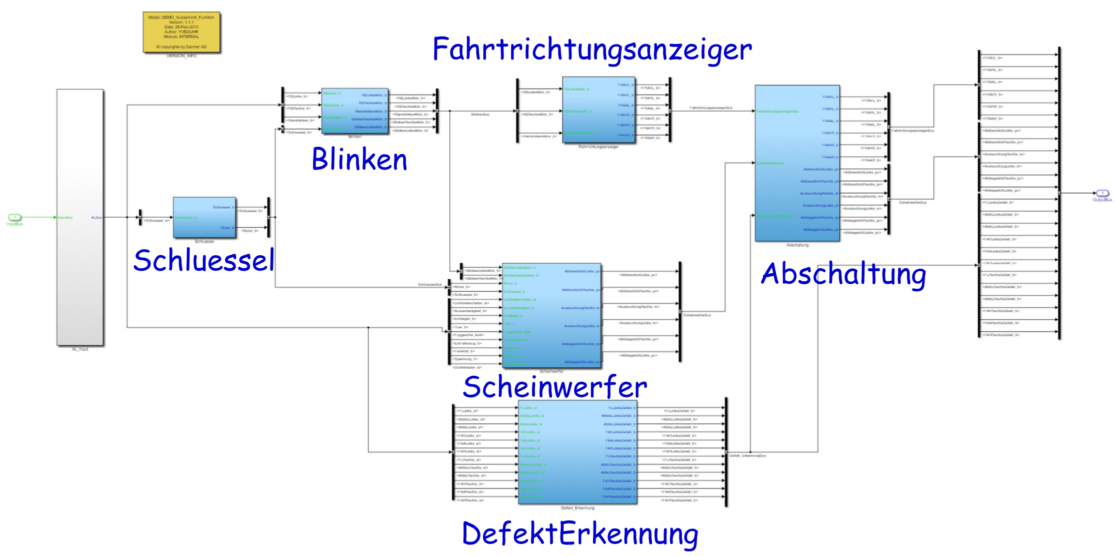


Figure 8.6.: Functional Overview Layer of Adaptive Light System (complete model is available under: [Dai13i]).

Adaptive Light System (ALS)

Adaptive light system controls adaptive high and low beam, turn signals as well as cornering and ambient light. Adaptive high and low beam adjust headlamps to the traffic situation and provides optimized illumination without dazzling others [Dai18b]. Cornering light illuminates the area to the side of a vehicle to take a look around the bend [Dai18b]. Ambient light welcomes the driver with an indirect light [Dai18b].

Figure 8.6 illustrates the hierarchy level containing the high-level (user-experience) functions. The ALS model contains only German names: `Schluessel` means vehicle key, it maps the current CAN value of the key status to two Boolean signals `Schluessel_b` (it is true if the key is present in the ignition) and `Motor_b` (it is true if the key is in the ignition and the key position is at motor running).

`Blinken` means flashing lights; the four input signals (from top to down) are left directional flashing (it is true if the driver moves the directional flashing lever down to indicate that the car should flash left), right directional flashing, hazard flashing, and key is present. The five output signals are left directional flashing is active (the value is true if the systems should activate left directional flashing - e.g., if the driver holds the directional flashing lever down shortly, the car flashes left only three times), right directional flashing is active, hazard flashing is active, flashing right is active (in the provided *Simulink* model flashing right may have value true even though directional flashing right has value false), and flashing left is active.

`Fahrtrichtungsanzeiger` means direction indicators; it receives as input signals left directional flashing is active, right directional flashing is active, and hazard flashing is active; and it emits the light status to the car lights: `FRAVL_b` is a short-form for the German word `Fahrtrichtungsanzeiger_vorne_links_b` and it represents the front left flashing light bulb; FRAHL_b is a short-form for Fahrtrichtungsanzeiger_hinten_links_b and it represents the back left flashing light bulb; FRAAL_b is a short-form for Fahrtrichtungsanzeiger_außen_links_b and it represents the light bulb in the left mirror; and the next three output signals are the equivalent light bulbs on the right side.

Scheinwerfer means head light. The 13 input signals are: flashing left is active, flashing right is active, motor is running, key is present, status of rotary light switch, external brightness, unlocked (true if the car is unlocked via vehicle remote control), door is open, vehicle speed, vehicle in front (true if a vehicle or a person is in front of this car or is in the visible area of the opposite lane), high beam is activated, vehicle voltage, and darkness switch (it is only available in armored vehicles such as the Mercedes-Benz S 600 Pullman Guard). The six output signals are: dimmed headlights left, dimmed headlights right, illumination light right, illumination light left, cornering light right, and cornering light left.

DefektErkennung means defect detection. It receives as input the status signal of all exterior light bulbs of the vehicle, and it produces for each status signal a Boolean signal representing whether the light bulb is defect. The Boolean signal is used to activate an optical signal in the driver's dashboard.

Abschaltung means switch-off. This subsystem receives as input all output values of the direction indicators[4], all output values of head light, and all Boolean output values of defect detection. If a light bulb is identified as defect, the subsystem sets its value to zero to switch it off to avoid further damage. The output signals are all output signals of the direction indicators subsystem plus all output signals of the head light subsystem.

Figure 8.7 shows the most important functions of the head light subsystem: Tagfahrlicht means daytime running lights, Umfeldbeleuchtung means ambient lights, AdaptivesFernlicht means adaptive high beam, Abbiegelicht means cornering light, and Ueberspannungsschutz means over voltage protection.

In contrast to the advanced driver assistance system model, the adaptive light system model contains functional safety blocks such as defect detection or over and sub voltage protection.

The last column in Table 8.4 lists statistics about the ALS model. This *Simulink* model has more blocks than any *Simulink* model of the different ADAS versions. However, the ALS *Simulink* model has less info blocks than even the smallest ADAS version. Vincent Bertram, our industrial partner at Daimler AG, reports that the info blocks in ALS implement more complex functionality than in any version of ADAS.

8.3. Main Study

The **Objective** of this main study is the evaluation of the improved development process for Daimler AG; Subsection 2.1.2 (esp., Figure 2.2 on page 23) presents this new process and how C&C high-level design models (i.e., C&C views) and automatic structural consistency checks for design (i.e., C&C views verification) are involved in this new improved process.

The studied **Case** is to observe how domain experts create C&C views based on given requirements (cf. Table 8.8 for number of available requirements), and to evaluate whether C&C views

[4] All the output values of the direction indicators are bundled to a signal bus and then passed to switch-off.

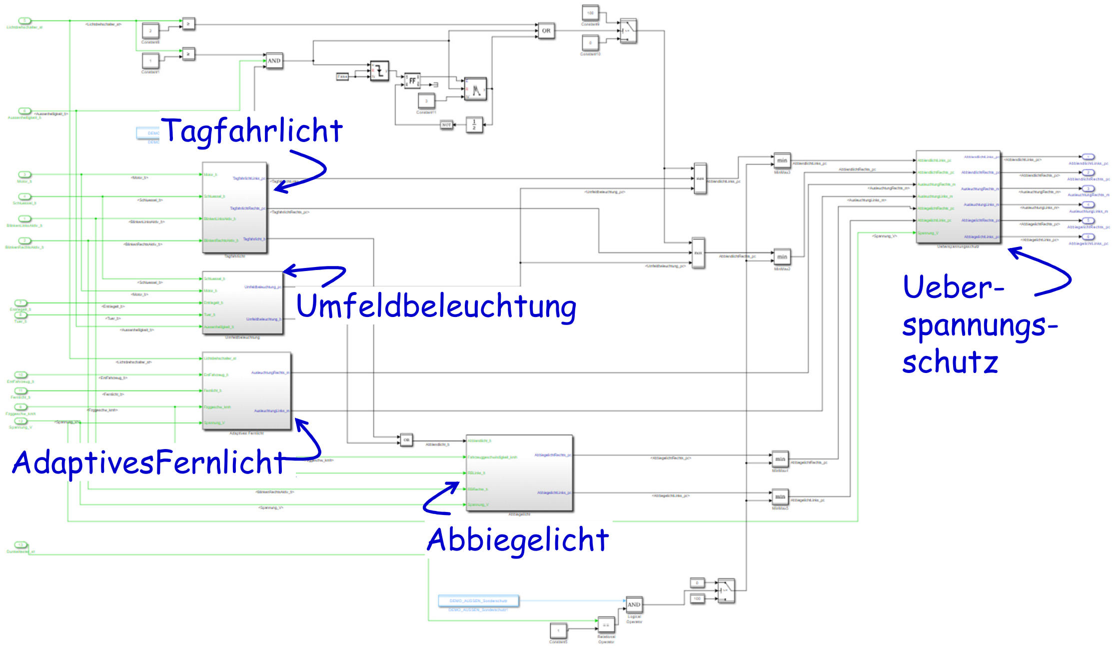


Figure 8.7.: Refinement of Head Light (German: Scheinwerfer) layer (complete model is available from: [Dai13j]).

	ADASv1	ADASv2	ADASv3	ADASv4	ALS
Requirements	33	Not available	Not available	68	82

Table 8.8.: Available requirements to create C&C views (copied from [BMR+17a, Table I]).

plus generated witnesses assist developers addressing the traceability and evolution challenges (cf. Subsection 8.2.2).

The **Theory** of the main study are the results of the preliminary study (cf. Subsection 8.2.2), the theory of the two languages *EmbeddedMontiArc* (cf. Chapter 3 and Chapter 4) and *EmbeddedMontiView* (cf. Section 7.3), C&C views satisfaction algorithm (cf. Section 7.4, and [MRR13, MRR14, Rin14]), as well as the witness generation algorithm (cf. Section 7.5). The tools to execute C&C views verification and to automatically generate witnesses are also part of the theory.

The **Method** is to create graphical representations of C&C views together with the industrial partner to collect his opinions during this process. The industrial partner should not start to model directly in *EmbeddedMontiView* to first focus on the concepts and not on the textual syntax of the modeling language; he should start modeling C&C views in PowerPoint. Later, we want to translate the first PowerPoint C&C views to the textual *MontiView* language (the predecessor of *EmbeddedMontiView* and the successor of *MontiArcView* language profile) together with the domain experts. Finally, the domain experts should create the textual files for the missing C&C views by themselves. Since C&C views are small, the domain experts can do this translation.

Q2	Can domain experts create C&C views with reasonable effort and are they missing any language features?
Q2a	How much knowledge/training of C&C views is necessary?
Q2b	How much knowledge about the provided models need domains to have in order to create C&C views?
Q2c	How long does it take to create a C&C view?
Q2d	What missing features would domain experts like to have in C&C views (verification)?
Q2e	Are there more preferable ways (with respect to methodology or tooling) to create C&C views?
Q3	Is C&C views verification applicable to automotive industry models?
Q3a	What is the effort to use industrial *Simulink* models as input for C&C views verification?
Q3b	Does the verification scale on industrial models?
Q4	Are the verification outputs of use for the engineers?
Q4a	What are the most useful elements in the representation of the witnesses?
Q4b	What elements are missing in the witnesses?

Table 8.9.: Overview of Research Questions in the Main Case Study (summary of [BMR$^+$17a]).

However, the textual witnesses produced by the verification tool are too large for the industrial partner[5], so the author of this thesis was chosen to create the graphical representations of these witnesses manually. The graphical representations of the witnesses are much easier to understand, and thus, showing them to the industrial partner is more promising to receive useful feedback. The subsequent study addresses the manual step of creating graphical witnesses later.

The **Selection** strategy is dominated by the four *Simulink* models of the ADAS system and the one *Simulink* model of the ALS system (cf. results of preliminary study in Subsection 8.2.2) as well as two requirement documents [Dai13a, Dai13b] (cf. [Dai13k] for English translation of original German documents) provided by Daimler AG. The number of requirements, shown in Table 8.8, represent the number of distinct *IBM Rational DOORS* requirement identifiers [TJ11]. Daimler AG did not provide us requirement documents of ADASv2 and ADASv3. Unfortunately, Daimler AG was not able to provide us informal design models nor traceability information within these *Simulink* models.

Table 8.9 shows an overview of all research questions of the main study. The next two sections try to answer them.

Our industrial partner at Daimler AG created all C&C views in the main study. The author of this thesis assisted the industrial partner in creating C&C views. This chapter refers to these persons as domain experts.

[5]It would take too much time (which was not available for the case study) to let the industrial partner create graphical witnesses based on the textual output.

8.3.1. Addressing Traceability

The hypotheses of traceability are:

1. Engineers, having good background and domain knowledge about the requirements and the implementation, are able to create C&C views based on given textual requirements.
2. Engineers need only a reasonable time to create a C&C view.
3. The modeled C&C views help engineers to understand relations between *Simulink* blocks and other requirements.
4. The graphical witnesses support engineers to trace down important *Simulink* elements for this requirement. As already mentioned earlier, the C&C views verification tool only creates textual witnesses and the graphical representation has been created manually.

To examine the first hypothesis our industrial partner, being unfamiliar with C&C views at the beginning of this case study, received papers [MRR13, MRR14] and additional materials[6] about C&C views. Additionally, the C&C views experts introduced the semantics of C&C views in a two-hour Skype session to the industrial partner. In a separate session, the experts created interactively some C&C views in PowerPoint together with the industrial partner.

Afterwards, the domain experts developed C&C views based on textual requirements and based on the *Simulink* models of ADAS and ALS. Our industrial modeled the C&C views in PowerPoint slides based on a given template. Since the first case study at beginning of 2017 only evaluated the methodology and usefulness of C&C views and it did not focus on tooling (support), the industrial partner did not model the C&C views directly in *EmbeddedMontiView*. Fifteen months later the tooling has been already optimized as there exists an IDE for *EmbeddedMontiArc* and for *EmbeddedMontiView* and a good layout algorithm to create graphical representations of textual *EmbeddedMontiArc* models/witnesses. The subsequent study evaluates the tooling, esp. the generated graphical representation of tracing witnesses, later.

We expected that our industrial partner was able to create C&C views for each requirement (Q2a, Q2d) based on the provided materials and Skype sessions. Furthermore, we expected that the domain experts do not need more than one hour (Q2c) to model a C&C view in PowerPoint and mark the important text parts in the requirement text.

Specifically, we asked the domain experts to create a C&C view for every ADASv1 and ADASv4 requirement. The author of this thesis worked with all the ADAS *Simulink* models more intensively at the end of 2015 and at beginning of 2016 in context of a collaboration research project together with Daimler AG. The industrial partner did not work with these models in detail at all.

In addition, we asked the domain experts to create C&C views for some requirements of the ALS *Simulink* model. The industrial partner had deep insight knowledge about the light system at this point of time. The author of this thesis did not really work with this large model before. As the ALS requirement document is with 82 requirements (cf. Table 8.8) the largest one and also the ALS *Simulink* model is with 1 646 blocks and 2 753 ports the most complex one, the industrial partner decided to focus only on the requirements related to sub- and over-voltages. This decision was made due to timing constraints, we could not create for each of the 82 requirements a C&C

[6] The extra material included a bachelor thesis [Kah17b] and its PowerPoint presentations as well as an explanatory video [Kah17a] about *EmbeddedMontiView*.

view, and that functional safety blocks (cf. Subsection 8.2.2) are only available in the ALS *Simulink* model.

During the creation of the C&C views, we measured the time it took domain experts to create C&C views based on given requirements and based on existing *Simulink* models (Q2c). Besides the requirement documents, the domain experts needed the *Simulink* models to identify the correct names of ports and component types. The domain experts could also create C&C views without inspecting *Simulink* models; however, this requires to create a mapping between names in requirement documents and signal names in *Simulink* models.

To examine the second hypothesis, we asked the domain experts to rate the effort to create C&C views (a) for models they did not work with before and (b) for models they did not inspect for more than a year (Q2b). During the C&C view creation process and directly after them, we interviewed the domain experts and asked them whether there would exist a more preferable way how to create or derive C&C views from (Q2e).

To examine the last two hypotheses, a two stage experiment has been set up. In a first step, the domain experts selected randomly ten different requirements of ADASv1 and ADASv4. For each of these ten requirements, the domain experts should highlight all important elements inside the *Simulink* model. This first step was done without using any C&C views. The domain experts executed the first step experiment before they created the C&C views for the requirements, because otherwise the creation of the C&C views could have impact how to interpret these requirements.

The second step works with C&C views and their verification. To examine the third hypothesis, we presented the graphical representation of C&C views to the domain experts and asked them whether they would now highlight different elements in the *Simulink* models.

To examine the fourth hypothesis, we showed the domain experts the graphical representation of the witnesses generated by the C&C views verification tool. Please note, the process involved a manual translation of the textual C&C witnesses to PowerPoint slides. Then we asked the domain experts how they interpret the difference between the graphical C&C witnesses and their perfect traceability *Simulink* models created in the first stage (Q4); esp. we wanted to know from the domain experts what are the most useful elements in C&C views (Q4a), and do the domain expert miss any elements in graphical C&C witnesses (Q4b).

8.3.2. Example of a Requirement, C&C View, and Graphical Witness

The top part of Figure 8.10 shows the translated text of the ADAS requirement `FA-6`. The prefix `FA` is an abbreviation of `Fahrerassistenzsystem` which is the German word for ADAS. The bottom part of Figure 8.10 shows the C&C view created by the domain experts according to this requirement. The requirement `FA-6` is part of the functions describing the `Adaptive Cruise Control` (cf. [Dai13k, Subsection 2.2.1]), which maps to the `Distronic` subsystem in the *Simulink* model. The colors in the text and in the C&C view show how the requirement names are mapped to *Simulink* signal names. The names in the if condition phrase are mapped to input ports, as the `Distronic` subsystem needs to read these values to produce the correct reaction. The vehicle word matches to the `DEMO_FAS` *Simulink* subsystem, because the ADAS (German short-form is `FAS`) is the most high-level software component of the vehicle in this *Simulink* model. The environment component (German `Umgebung`) is only present in the

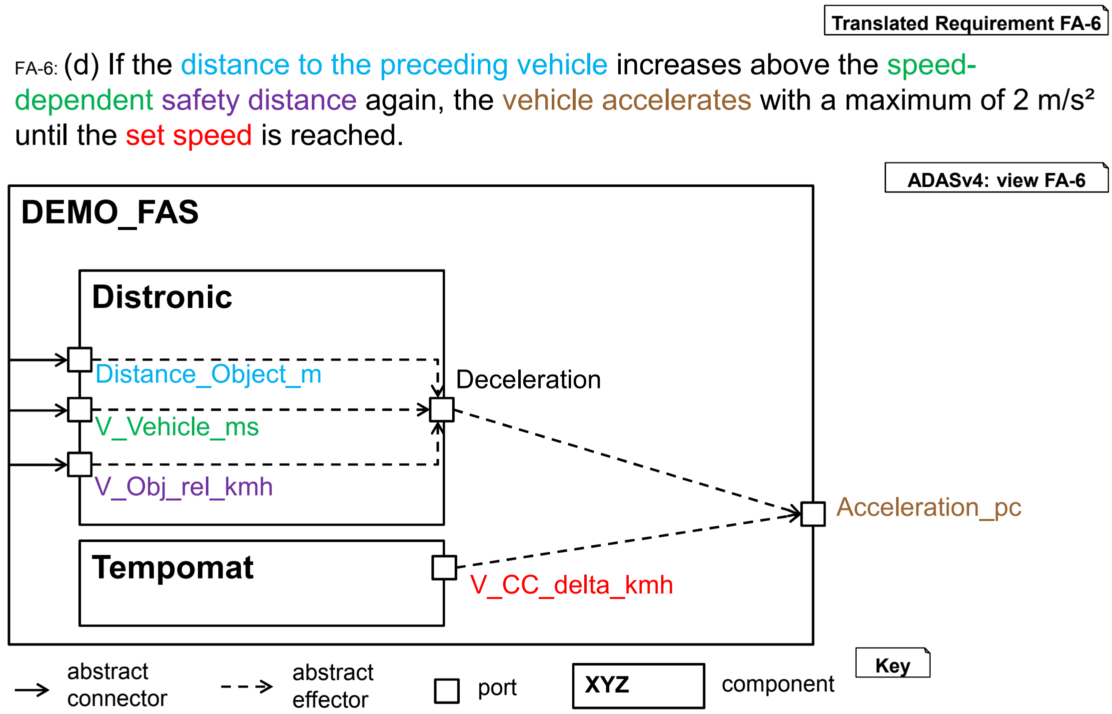


Figure 8.10.: Requirement FA-6 of unit `Distronic` of ADASv4 (top) and the view created for this requirement by the domain experts (bottom); copied from [BMR+17a, Fig. 5].

Simulink model to simulate the closed-loop of the ADAS system, but the environment component is not part of the ADAS system.

The solid arrows in Figure 8.10 represent abstract connectors. The left top abstract connector going from `DEMO_FAS` to the `Distance_Object_m` abstract port of the `Distronic` components states that the `DEMO_FAS` subsystem has an input port which delegates its value without modifying it to an input port of the `Distronic` subsystem having the signal name `Distance_Object_m`.

The dashed arrows in Figure 8.10 represent abstract effectors. The top right abstract effector going from the abstract port `Deceleration` of the `Distronic` component to `Acceleration_pc` of the `DEMO_FAS` component states that the output port with the signal name `Deceleration` of the `Distronic` subsystem influences the value of the output port with the signal name `Acceleration_pc` of the `DEMO_FAS` subsystem. Influence means that value of `Deceleration` may be modified by other atomic *Simulink* blocks.

The abstract port `Deceleration` is not mentioned in the FA-6 requirement. However, the domain experts included this abstract port in the C&C view as the deceleration value (100% deceleration means the car is not accelerating at all, 0% deceleration means that the car accelerates with its maximal acceleration) is a limiting factor of the vehicle's acceleration, and the domain experts meant that this port is crucial to understand the implementation of this requirement.

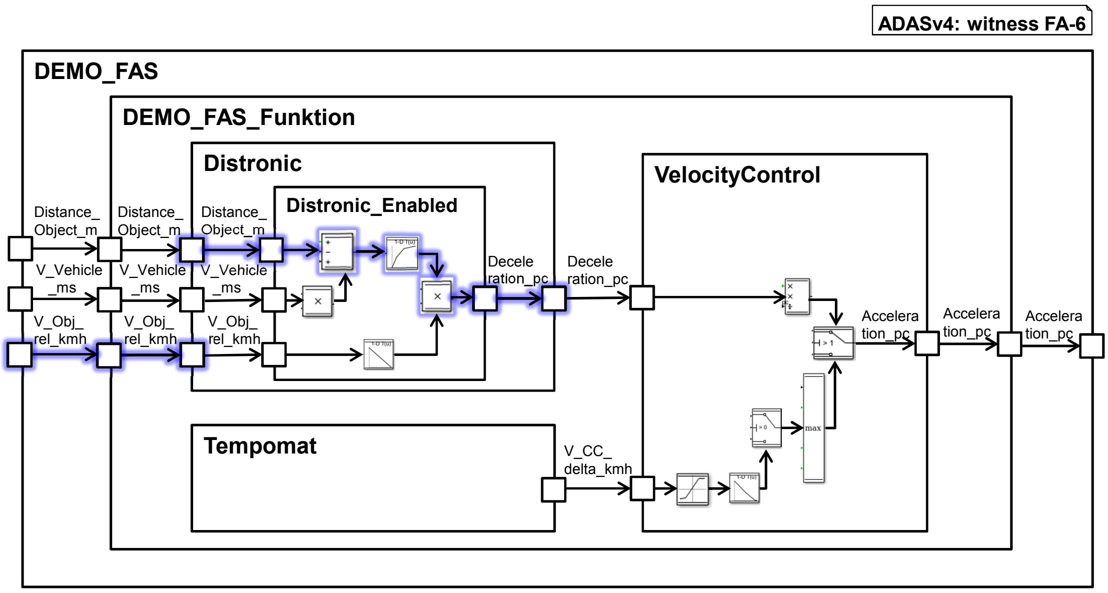


Figure 8.11.: Satisfaction Witness of view FA-6 (copied from [BMR+17a, Fig. 6]).

Figure 8.11 shows the generated satisfaction witness of the C&C view shown in Figure 8.10. The C&C views verification algorithm only creates textual output of witnesses; Figure 8.11 shows the graphical PowerPoint presentation which has been manually created based on the textual file. On average, the author of this thesis needed for each witness about one hour[7] to transform one textual witness into a graphical PowerPoint witness. This finding and the finding that textual witnesses are not really helpful, let to the decision to develop a layout algorithm and to redo parts of the main study in a new subsequent study using the generated graphical representations based on the layout algorithm.

The blue highlighted connectors in the bottom left part of Figure 8.11 belong to the connector chain of the witness representing the abstract connector going from `DEMO_FAS` (unknown port) to `Distronic`'s `V_Obj_rel_kmh` port in the C&C view. Additionally, Figure 8.11 highlights the witness elements (i.e., upper colored atomic blocks and signal lines in the *Simulink* model) belonging to the abstract effector starting at the `Distance_Object_m` port and ending at `Deceleration_pc` port of the `Distronic` subsystem.

Figure 8.11 shows all elements of the generated satisfaction witness, i.e., it contains all components (subsystems or atomic blocks), ports, and connectors so that all elements of the C&C view in Figure 8.10 are matched at least once. Please note, that the satisfaction witness shows for each abstract connector and abstract effector only the shortest path in the *Simulink* model (cf. Subsection 7.5.1).

[7]The witness of the view FA-6 is one of the smaller ones.

8.3.3. Design Decisions for Creating C&C Views

Every C&C view contains *Simulink* subcomponents mentioned in the requirement text. Additionally, every C&C view includes for every output port, mentioned in the requirement, the *Simulink* subsystem being the target of this output port. The same holds for input signals mentioned in the requirement. By adding the target subsystems, C&C views underline the component interaction between high-level *Simulink* subsystems to model dependencies between user-experience functions/requirements.

As already shown in Figure 8.10, nearly all textual requirements follow a trigger-action pattern (if-then sentences). Abstract effectors in C&C views mostly start from trigger ports and end at action ports.

The two domain experts tried to create for all requirements of ADASv1 and ADASv4 a C&C view if this was possible. They created 17 C&C views for ADASv1 and 26 C&C views for ADASv4. For ADASv2 and ADASv3 no separate requirement documents were available. Due to time restrictions (cf. Subsection 8.3.1) the domain experts created only 7 C&C views for the ALS focusing on functional safety blocks.

Table 8.12 and Table 8.13 show the numbers of components, connectors, effectors, and ports of the C&C views created by the domain experts. Table 8.12 lists the sizes of C&C views for ADASv1, and Table 8.13 lists the sizes of C&C views for ADASv4. This thesis skips the sizes of C&C views for the ALS as the domain experts modeled only 7 out of 82 requirements.

The last two rows in Table 8.12 and Table 8.13 demonstrate that the average and median size of C&C views of ADASv1 and ADASv4 are about the same. This result is surprising when considering that the *Simulink* model of ADASv4 is about twice as complex according to the total number of blocks (cf. Table 8.4) than the *Simulink* model of ADASv1.

8.3.4. Addressing Evolution

The hypotheses of the evolution challenge are:

1. C&C views verification helps to identify violations of existing requirements due to architecture updates.
2. The generated witnesses assist developers to locate and fix violations in *Simulink* models.

To examine the first hypothesis, the domain experts analyzed the *Change of Documentation* table (cf. Figure 8.3) in the requirement document of ADASv4 to figure out which requirements were updated in which ADAS version. All C&C views belonging to not updated requirements of ADASv1 (i.e., all other ADAS versions such as ADASv2, ADASv3, and ADASv4) should be satisfied by all four ADAS versions.

To examine the second hypothesis, ADASv1 to ADASv3 are tested against the C&C views belonging to the requirement documentation of ADASv4. The generated witnesses by the not-satisfied C&C views should link to the *Simulink* elements which are missing as these feature are only introduced in ADASv4. For the experiment of the first hypothesis, domains experts identified five C&C views (FA-29, FA-23, FA-24, FA-35, and FA-36) related to the cruise control lever which ADASv2 should not satisfy. These five C&C views are invalid as the two input ports of `Tempomat`'s `LeverUp_b` and `LeverDown_b` are only valid for the one-stage cruise control

ViewsADAS1.pptx	FA-	Components	Connectors	Effectors	Ports	Sum
Slide 1	14	4	2	0	0	6
Slide 2	19	2	2	1	2	7
Slide 3	27	3	2	1	2	8
Slide 4	28	3	4	3	4	14
Slide 5	29	3	5	1	5	14
Slide 6	25	3	3	1	4	11
Slide 7	22	3	4	0	5	12
Slide 8	23	3	3	1	3	10
Slide 9	24	3	3	1	3	10
Slide 10	30	3	2	1	3	9
Slide 11	26	2	3	2	3	10
Slide 12	33	2	2	1	2	7
Slide 13	38	3	2	1	2	8
Slide 14	34	3	3	1	3	10
Slide 15	35	3	3	1	3	10
Slide 16	36	3	3	1	3	10
Slide 17	37	2	2	1	2	7
Average		2.82	2.82	1.06	2.88	9.59
Median		3	3	1	3	10

Table 8.12.: View Sizes for ADASv1 (copied from [vW17a]).

ViewsADAS4.pptx	FA-	Components	Connectors	Effectors	Ports	Sum
Slide 1	15	9	8	1	1	19
Slide 2	3	4	5	3	5	17
Slide 3	4	4	9	6	9	28
Slide 4	5	3	2	10	7	22
Slide 5	6	3	2	6	6	17
Slide 6	99	2	2	4	3	11
Slide 7	86	2	2	6	4	14
Slide 8	84	3	3	6	5	17
Slide 9	19	2	2	1	2	7
Slide 10	20	3	3	0	3	9
Slide 11	21	3	4	1	4	12
Slide 12	22	3	5	1	5	14
Slide 13	23	3	4	2	3	12
Slide 14	24	3	1	2	5	11
Slide 15	25	3	3	1	3	10
Slide 16	26	3	3	1	3	10
Slide 17	27	3	2	1	3	9
Slide 18	28	2	3	2	3	10
Slide 19	30	2	2	1	2	7
Slide 20	31	3	2	1	2	8
Slide 21	32	3	3	1	3	10
Slide 22	65	3	3	1	3	10
Slide 23	67	3	3	1	3	10
Slide 24	35	2	2	1	2	7
Slide 25	77	4	2	2	3	11
Slide 26	75	2	0	1	2	5
Average		3.08	3.08	2.42	3.62	12.19
Median		3	3	1	3	10.5

Table 8.13.: View Sizes for ADASv4 (copied from [vW17a]).

lever, which was updated to a two-stage one in ADASv2 (cf. Figure 8.3). However, ADASv2 did not satisfy any of the 17 C&C views of ADASv1, because the signal names of ADASv1 and ADASv2 differ. For example, `CC_active_b` has been changed to `CC_enabled_b` in ADASv2, similar `Limiter_active_b` was modified to `Limiter_enabled_b` in ADASv2. The error message of the verification algorithm *No match for port "CC_active_b" of component "Tempomat"* helped us to quickly locate the problem.

However, updating the signal names in the *Simulink* model of ADASv2 to the signal names used in ADASv1 caused only positive verification results even though the domain experts identified five C&C views that should fail. Further investigations unveiled that the two-stage cruise control lever is only available in ADASv3 and ADASv4; these two models do not satisfy these five C&C views. Hence, C&C views verification with its generated witnesses located a mismatch between *Simulink* models and the requirement change history.

To examine the second hypothesis, the domain experts should locate and explain which features are available in ADASv4 but not in ADASv3. Based on this information, we could identify the C&C views which should be satisfied by ADASv4, but not by ADASv3. Applying the *Simulink* model of ADASv3 against the C&C views of ADASv4 showed the same name mismatch as the one between ADASv1 and ADASv2. Fixing this name issue, the C&C view verification failed exactly on the five identified C&C views (FA-15, FA-4, FA-5, FA-99, and FA-84) describing the emergency brake and follow to stop features being only available in ADASv4.

8.3.5. Translating *Simulink* Block Diagrams to *EmbeddedMontiArc*

All models provided by Daimler AG for the main study were *Simulink* block diagrams. At a first look, the graphical layouts of *Simulink* block diagrams are very close to the graphical representations of *EmbeddedMontiArc* models: *Simulink* subsystems and atomic blocks map to *EmbeddedMontiArc* components, *Simulink* in- and outport blocks map to *EmbeddedMontiArc* ports, and *Simulink* signal lines map to *EmbeddedMontiArc* connectors.

However, *Simulink* also contains many special model elements:

1. Enabled subsystems [The18n, pp. 10-11 to 10-19]
2. If-Then-Else Blocks [The18n, p. 10-33]
3. Merge Blocks [The18n, p. 10-6]
4. Triggered subsystem [The18n, pp. 10-21 to 10-25]
5. Data Store, Data Store Read, Data Store Write [The18n, pp. 42-125 to 42-131]
6. Goto block, From block [The18n, p. 63-3]

The four ADAS *Simulink* versions use the first three kinds of *Simulink* model elements to express variability. Figure 8.14 illustrates the feature diagram model extracted from the ADAS version 4 *Simulink* model. The *Simulink* models provided by Daimler AG use pure::variants [pur14] to activate via constant values in combination with enabled subsystems or if-else constructions one specific variant in the 150% product line [HKM+13] model. The ALS uses the first three kinds of *Simulink* model elements to express conditional execution for different vehicle modes (cf. mode transition diagrams [BBR+05], and component modes for dynamic reconfiguration [HKR+16]). Example of a mode transition in the ALS is the switch between normal mode to sub or overvoltage modes at runtime according to the current battery voltage; e.g., in sub voltage mode the car turns off the adaptive high beam light to save battery (cf. [Dai13h]).

The triggered subsystems are kind of a special case of the enabled subsystems, but the five *Simulink* models use the triggered subsystems to react on user events instead of modeling variability. A prominent example of a triggered subsystem is the subsystem reacting on the event when the driver pulls up the cruise control lever: neutral position of the cruise control lever has value zero, and pulled up position of the cruise control lever has value one or two; a rising triggered subsystem is only executed if the value at time step t is higher than the value at the

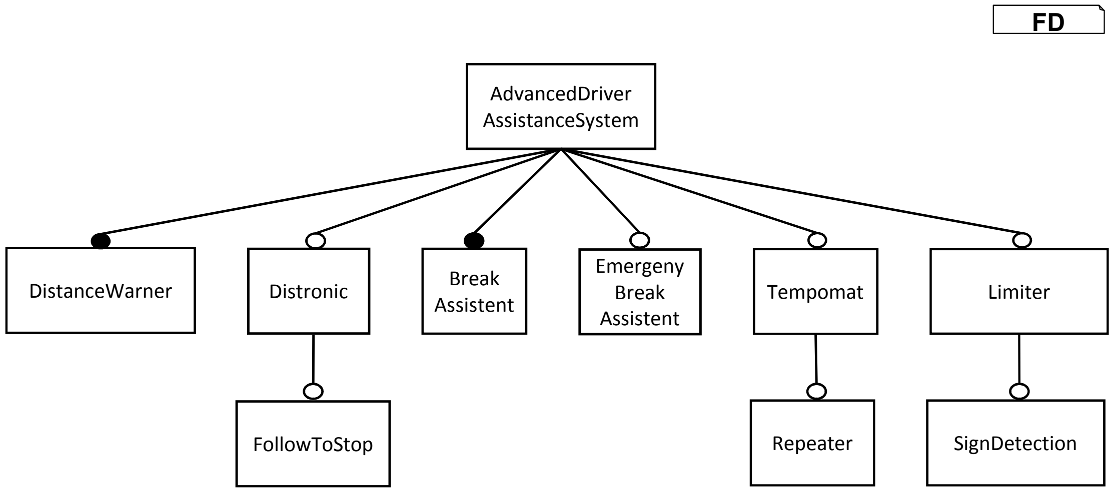


Figure 8.14.: Feature diagram model extracted from ADAS version 4 *Simulink* model. (Feature diagram created by Christoph Schulze and Michael von Wenckstern.)

previous time step $t - 1$. The falling triggered subsystem is only executed when the driver pulls down the cruise control lever.

The fifth kind of *Simulink* model elements introduce global variables to communicate between different *Simulink* subsystems. The `data store` block defines a global variable, the `data store read` block reads the value of the global variable, and the `data store write` block updates the global variable to a new value.

The last kind of *Simulink* model elements creates a connection between *Simulink* subsystems without drawing a signal line. A value written in a `Goto` block with a specific label, can be read by a `From` block with the same label [The18h]. Developers use this kind of communication to reduce the effort to pass a value through many subsystem hierarchies, and to have not too many cross-cutting signal lines resulting in unreadable *Simulink* models. However, using this signal exchange pattern hides communications between subsystems.

To perform the C&C views case study on the *Simulink* industry models provided by Daimler AG, we developed a converter tool translating *Simulink* models to *MontiArcLight* ones (these models are similar to the C&C instance structure presented in Section 4.3) [Bru17b]; *MontiArcLight* models are later converted to *EmbeddedMontiArc* ones. The next paragraphs shortly introduce the main ideas behind the algorithms to translate *Simulink* models to *MontiArcLight* models. Later *MontiArcLight* models, storing all information of *Simulink* elements such as type of blocks in stereotypes, are translated to C&C models as defined in Chapter 3 and Chapter 4. The bachelor thesis of Stefan Brunecker [Bru17b, Bru17a] contains implementation details of the *Simulink* converter tool.

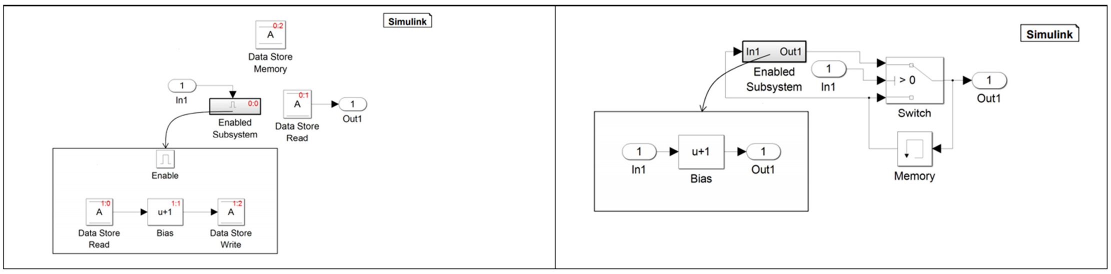


Figure 8.15.: Example how an enabled subsystem with global variables is removed (left side is a copy of [Bru17b, Figure 4.4] and right side is a copy of [Bru17b, Figure 4.5]).

Model references

Simulink block diagrams may contain model reference blocks referencing to *Simulink* library blocks. The model reference block matches the component type instantiation in *EmbeddedMontiArc*. The library block is the component type in *EmbeddedMontiArc*.

Simulink specific blocks

As stated above, *Simulink* models may have special blocks that cannot be directly mapped to components in a C&C model. Thus, first all special blocks are transformed to behavior equivalent subsystems including only standard blocks and connectors in *Simulink*. Figure 8.15 shows an example how an `Enabled Subsystem`, containing the global variable `A` inside, is transformed to standard blocks (global variable `A` is removed). This transformation is complex, as the execution order of the special blocks must be considered (e.g., in what order are variables written and read). The combination of variables with conditional or reconfiguration ports is very difficult, as variables are not always updated (e.g., *Simulink* elements inside an enabled subsystem are only executed when its corresponding enabled port receives true as input signal) and normal subsystems are executed every time (meaning that all internal variable inside this subsystem are always updated). Therefore, the converter tool generates a suitable a reset mechanism (cf. loop around unit-delay block in the right part in Figure 8.15) for global variables used inside these special blocks. Second, these standard blocks are translated to C&C models. The first translation of *Simulink* specific blocks to C&C equivalent subsystems and atomic blocks is done in *Simulink* to easier test this transformation step: the original and the transformed *Simulink* models are black box tested with the same input values (using *Simulink* signal builder component [The18n, p. 62-119]), and the test succeeded when the transformed model produced the same output values as the original one.

Signal buses

Since *Simulink* is a visual modeling language without any automatic layout mechanism, engineers use (even nested) signal buses to group signal lines going from one subsystem to another one. Figure 8.16 shows an example hierarchy of the ALS using buses to graphically group signal lines to avoid many cross-cutting lines in the graphical representation. Since these buses exist

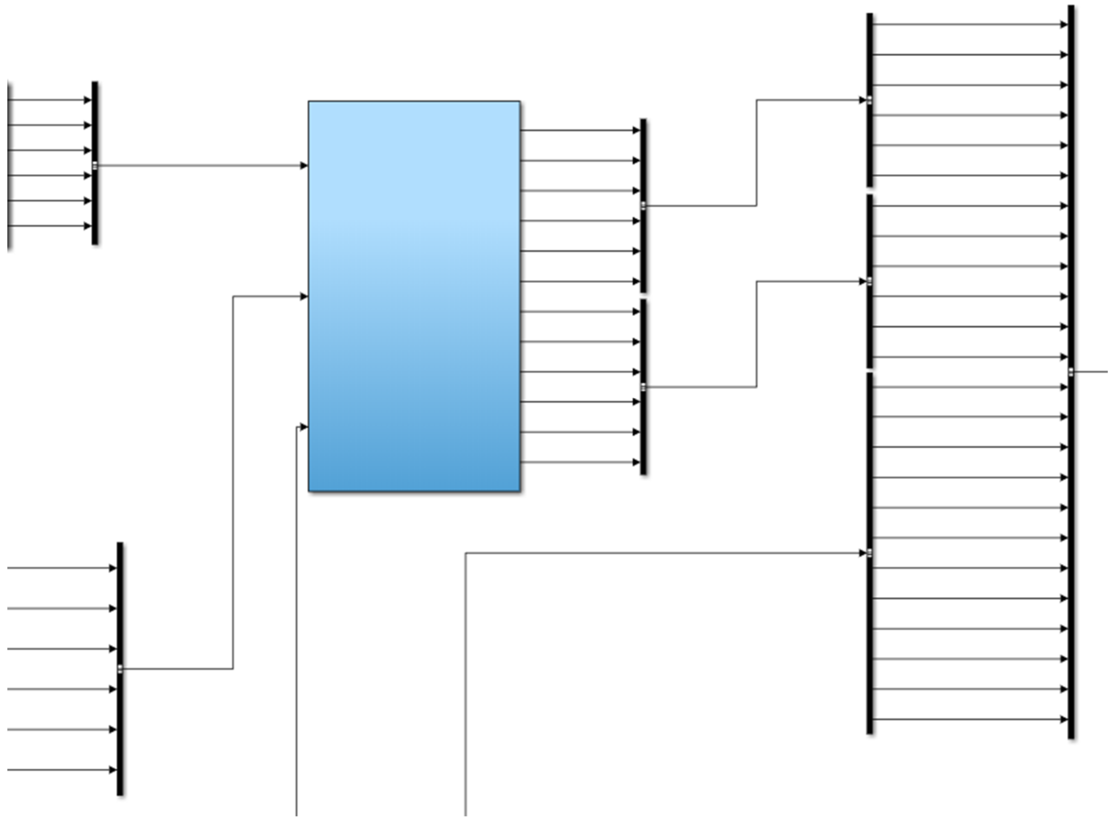


Figure 8.16.: Screenshot showing the usage of nested buses to group signal lines graphically; model available from [Dai13i]

only for representation purposes, the *Simulink* translator removes all `Bus Creator` and `Bus Selector` blocks as well as the it connects the subsystems' output and input ports directly with each other. If this translation step would not exist, the translated model would never satisfy an abstract connector between two user-experience subsystems as always a `Bus Creator` or a `Bus Selector` block would be between the abstract connector; thus the C&C views would only contain abstract effectors.

Translation results

Table 8.17 shows the sizes of C&C models resulting from our automated translation. This translation increases the size of all models as shown in the third and sixth row. The factor how much the number of components and ports are increased depends on the number of specific *Simulink* elements being present in the *Simulink* models. The translation increases the number of ports and components only slightly for the adaptive light system (last column), since this model has no `DataStoreMemory`, `DataStoreRead`, or `DataStoreWrite` blocks (cf. Table 8.4). The slightly increase results from the `ActionPorts`. ADASv2 and ADASv3 have the most

	ADASv1	ADASv2	ADASv3	ADASv4	ALS
Simulink Blocks (not counting port blocks)	327	686	664	655	1 065
C&C Components	639	2 309	2 278	1 396	1 086
Blocks Increased by Factor	1,95	3,37	3,43	2,13	1,02
Simulink Ports (also counting ports of atomic blocks)	701	1 454	1 480	1 513	2 753
C&C Ports	1 528	9 009	8 981	3 596	3 193
Ports Increased by Factor	2,18	6,20	6,07	2,38	1,15

Table 8.17.: Translation Results (partially copied from [BMR+17a, Table I]).

significant increase, and this is due to the many `DataStoreRead`, `DataStoreWrite`, and `UnitDelay` blocks in combination with the special ports such as `EnablePort`, `Trigger-Port`, and `ActionPort` (cf. Figure 8.15).

The translation tool does not create a minimal *EmbeddedMontiArc* model based on a given *Simulink* model. Figure 8.18 shows the *Simulink* hierarchy `V_SetValuePlusLvl2` of ADASv4. Figure 8.19 shows the graphical representation of the translated *EmbeddedMontiArc* model. As described in the previous part of this subsection, the *EmbeddedMontiArc* model replaces the write statement of the global variable `DEMO_FAS_V_CCSetValue` with output ports. However, the validation tool creates for each write-read pair of global variables one communication path (component-connector- chain) to handle special blocks such as the `Trigger` port shown in Figure 8.18 (cf. Figure 8.15 for more details). Therefore, Figure 8.19 contains seven output ports for the `DEMO_FAS_V_CCSetValue` global variable.

The translation tool could be improved to do a further control-flow graph analysis to merge parts of communication paths together as long as it is possible; this would decrease the number of ports and connectors in the translated *EmbeddedMontiArc* model. Therefore, the `Increased by Factors` rows in Table 8.17 must be considered with caution.

8.4. Results of Main Study

This section summarizes the results of the main study. Subsection 8.4.1 to Subsection 8.4.3 present the results to answer research questions Q2 to Q4. Subsection 8.4.4 presents the outcomes addressing traceability and evolution.

8.4.1. Feasibility and Effort to Create C&C Views

This case showed that engineers can create for many (but not all) requirements C&C views to capture design decisions of *Simulink* implementations. For UI-related requirements, e.g., *AL-72: The rotary light switch has the following positions: Off; Auto (automatic position); Exterior light on*; as well as extra-functional requirements, e.g., *FA-53: The safety classification of the system speed control is ASIL B* no C&C views could be created. The *Simulink* model covers only functional requirements, and thus, UI-related ones are not present. The ASIL B safety

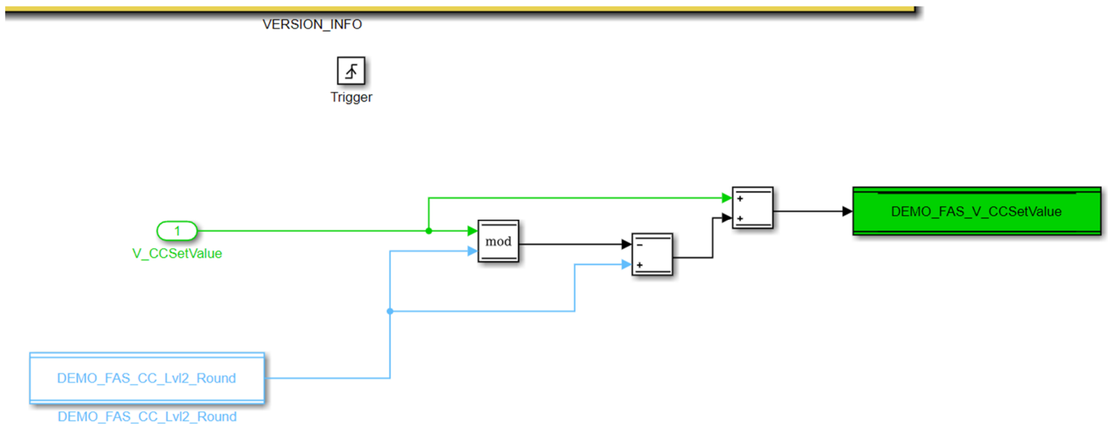


Figure 8.18.: Screenshot of *Simulink* hierarchy `V_SetValuePlusLvl2` of ADASv4. `DEMO_FAS_CC_Lvl2_Round` is a constant bounded at compile time. `DEMO_FAS_V_CCSetValue` is a `Data Store Write` block to save the value of the last sum component into the global variable `DEMO_FAS_V_CCSetValue`.

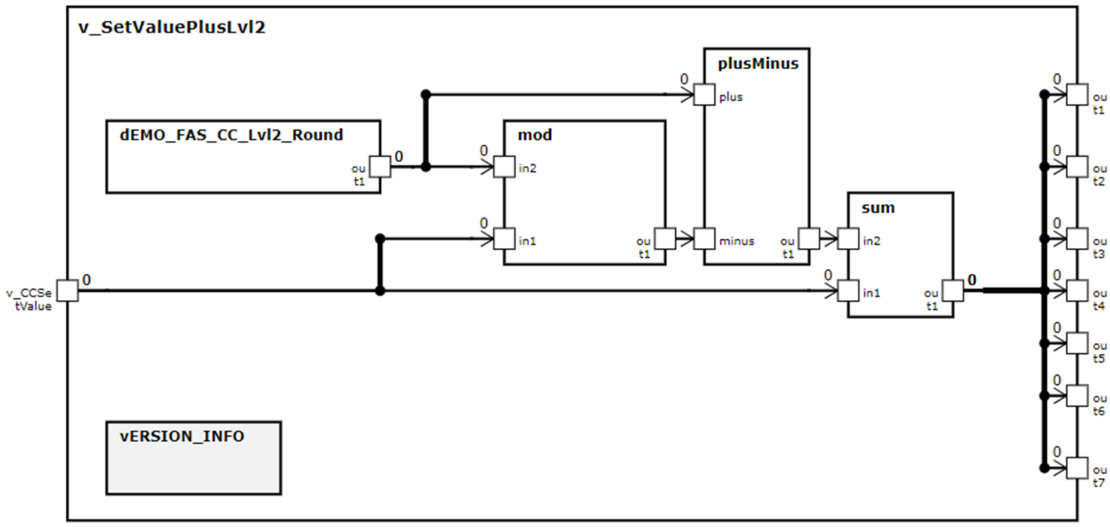


Figure 8.19.: Screenshot of the graphical representation of the *EmbeddedMontiArc* model, which has been automatically generated by the *Simulink* one shown in Figure 8.18.

requirement can also not be verified with the *Simulink* model alone, because hardware-specific failures needed for a fault-tree analysis are additional necessary. In future, an extension of the C&C views verification algorithm, also considering behavioral properties, would enable to verify these safety requirements. UI-related requirements are in the opinion of the author of this thesis out of scope for C&C views verification.

For ADASv1 the domain experts created 17 C&C views covering 21 out of 33 requirements. Sometimes one C&C view contains multiple requirements. For example, the C&C view FA-14 of ADASv1 shown in Figure 8.20 includes the requirements FA-15 and FA-16. For ADASv4

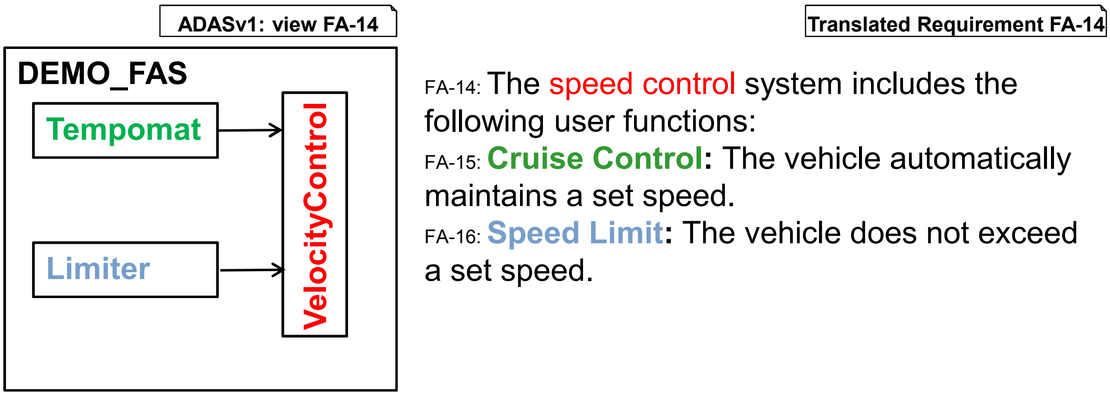


Figure 8.20.: Example of C&C view covering multiple requirements (copied from [BvW17, slide 1]).

the domain experts created 26 C&C views covering 50 out of 68 requirements. This means, the domain experts created C&C views for 70% (71 out of 101) of the requirements belonging to ADASv1 and ADASv4.

The domain experts needed on average half an hour to model one C&C view in PowerPoint; answering question **Q2c**. As training the two-day workshop and modeling some C&C views together with the industrial partner as well as providing existing materials (including one video) about C&C views and their verification was enough; thus we conclude that within one-week domain experts are able to learn the C&C views modeling techniques - answering **Q2a**.

For the domain experts it was really helpful to have a basic understanding of the domain of the models to understand the textual requirements and the *Simulink* models. The answer of the industrial partner to question **Q2b** was that it is helpful if the domain expert knows *Simulink* to understand the specific *Simulink* blocks (cf. Subsection 8.3.5) in the *Simulink* models provided by Daimler AG and the domain expert should have a basic understanding of the requirement domain; but the domain expert may not have been worked with the *Simulink* models before.

As already mentioned before, we extended the C&C views language with the abstract effector concept to be able to model the requirements; the industrial partner also wants to model conditional abstraction connectors and effectors (more information in Section 8.6) - this answers **Q2d**.

As an alternative way the industrial partner wants to create C&C views in a graphical manner by removing elements in the *Simulink* model. The supposed advantage of this workflow is that the industrial partner does not need to learn a new tool and it is less error-prone as no signal name typos could occur. This answers **Q2e**; however, we had no time to create such a tooling, and to evaluate whether domain experts are really faster with this proposed methodology.

8.4.2. Technical Applicability

Our first opinion that *Simulink* models are very similar to C&C models must be corrected during the case study. The development of the transformation tool involved two bachelor theses

Model	Average Positive Verification Time	Average Negative Verification Time	Average Time to Create One Positive Satisfaction Witness	Average Time to Create All Negative Non-Satisfaction Witnesses
ADASv1	62 ms	61 ms	27 ms	42 ms
ADASv2	1 809 ms	1 174 ms	615 ms	6 443 ms
ADASv3	1 459 ms	1 303 ms	566 ms	5 928 ms
ADASv4	404 ms	506 ms	114 ms	963 ms
ALS	218 ms	126 ms	82 ms	175 ms

Table 8.21.: Average verification and witness generation time (copied from [BMR+17a, Table III]). Individual verification times are available in Subsection A.4.1. Time measured on Windows 7 Professional notebook with 4 cores plus hyper-threading.

[Ern16, Bru17b]; transforming *Simulink* models to C&C ones needed much more technical effort than estimated.

The most complex task in the *Simulink* transformation tool was to analyze control-flow graphs of *Simulink* models. The control flow graphs depend not only on the visible elements in the *Simulink* models, but also on the specified settings defining how often blocks are executed. All this information must be considered to eliminate global variables and replace them by connectors. But at the end, our transformation tool was finally able to translate all given *Simulink* models to equivalent C&C models. The transformation tool only supports the 83 *Simulink* block diagram elements needed for this industrial case study. All supported *Simulink* block diagrams are available from [vW17b].

The answer to research question **Q3a** is the following: C&C views verification can now be applied to *Simulink* models. However, the transformation tool must be extended when supporting additional *Simulink* libraries such as *SimBiology* [She10], *Signal Processing Toolbox* [PI04], or *DSP System Toolbox* [KK00].

To answer question **Q3b** whether the C&C views verification scales, we measured the running time of our C&C views verification tool. Table 8.21 lists the average times for C&C views verification, split into positive and negative results; as well as it also reports the time needed for positive and negative witness generation.

These first two columns in Table 8.21 demonstrate that C&C views verification is very fast and scales up to real-size industry models. Generating all negative non-satisfaction witnesses is the most time consuming task (cf. last column for ADASv2 and ADASv3) as for every C&C view element, which is not satisfied by the *Simulink* model, its own non-satisfaction witness is generated[8].

[8] For abstract effectors not satisfying the *Simulink* model our verification tool does not generate a negative non-satisfaction witness, because the witnesses would contain too many possible elements. This case study also showed that the natural error message text is the most useful part of the non-satisfaction witness.

8.4.3. Helpfulness of Witnesses

During the execution of the main study, being part of the first experiment in beginning of 2017 [BMR+17a], we manually translated the textual output of our tool to graphical C&C witnesses in PowerPoint to analyze the results with the domain experts. Due to a master thesis extension [Oez18], it is now possible to visualize the textual model as a graphical result; the subsequent study evaluates these generated graphical representations.

Table 8.22 and Table 8.23 present the sizes of generated positive satisfaction witnesses of ADASv1 and ADASv4. Even though the median (cf. last rows in Table 8.12 and Table 8.13) for the view sizes of ADASv1 and ADASv4 are nearly the same, the median (cf. last rows in Table 8.22 and Table 8.23) of connectors and ports of the witness size of ADASv4 is about 10%-20% larger than the witness size of ADASv1. The reason for the difference of the median witness sizes is probably the fact that the *Simulink* model of ADASv4 is two and a half times larger than the one of ADASv1. The median of the component sizes is the same for witnesses of ADASv1 and ADASv4. The satisfaction witness in this experiment contains all components up to the main component instance instead of only up to the least common parent one; this change was made to easier locate the witness elements in the large *Simulink* models.

The large graphical positive satisfaction witnesses were no obstacle for the industrial partner, because he is used to work with very large graphical *Simulink* models with hundreds of subsystems and thousands of signal lines. The domain experts found all element kinds (i.e., components, ports, connectors, and effectors) of the witnesses useful; answering **Q4a** for positive satisfaction witnesses.

Comparing the manual colored *Simulink* models with the generated witnesses, we figured out that *Simulink* (version R2016a) only supports to color all signal lines going from one source port to all other destination ports. Thus, the domain experts could not colorize the *Simulink* models as they preferred to, and these models contain too many highlighted lines.

Nevertheless, the domain experts missed some *Simulink* blocks and signal lines (they do not mean the accidental colored lines mentioned in the sentences above) in the generated witness. This was also the reason to introduce the tracing witness kind in Subsection 7.5.2. The subsequent study tries to answer **Q2b** finally whether tracing witnesses still miss important *Simulink* elements.

During the execution of the evolution challenge, we presented the domain experts negative non-satisfaction witnesses; these witnesses did not contain elements for abstract effectors. The natural language description explaining in one sentence why a *Simulink* model does not satisfy a C&C view is also part of the C&C witness, and this natural description was according to the domain experts the most useful information for non-satisfaction witnesses. The domain experts had so deep knowledge about the *Simulink* models that they were able to locate the problem directly in *Simulink* after reading the natural-language description. Therefore, we conclude that the natural language descriptions for non-satisfaction witnesses are the most useful elements; answering **Q4a** for negative witnesses.

WitnessesADAS1.pptx	FA-	Components	Connectors	Ports	Sum
Slide 1	14	9	2	4	15
Slide 2	19	20	25	32	77
Slide 3	27	23	26	35	84
Slide 4	28	27	45	57	129
Slide 5	29	17	33	40	90
Slide 6	25	21	26	34	81
Slide 7	22	11	17	21	49
Slide 8	23	25	31	42	98
Slide 9	24	25	31	42	98
Slide 10	30	13	14	18	45
Slide 11	26	18	31	38	87
Slide 12	33	15	18	22	55
Slide 13	38	19	21	27	67
Slide 14	34	24	38	49	111
Slide 15	35	27	34	46	107
Slide 16	36	28	35	48	111
Slide 17	37	15	18	22	55
Average		19,82	26,18	33,94	79,94
Median		20	26	35	84

Table 8.22.: Size of positive Satisfaction Witnesses for ADASv1.

WitnessesADAS4.pptx	FA-	Components	Connectors	Ports	Sum
Slide 1	15	18	30	39	87
Slide 2	3	25	48	62	135
Slide 3	4	39	77	105	221
Slide 4	5	25	54	69	148
Slide 5	6	26	46	63	135
Slide 6	99	27	46	60	133
Slide 7	86	17	34	42	93
Slide 8	84	20	34	46	100
Slide 9	19	19	23	30	72
Slide 10	20	11	11	14	36
Slide 11	21	17	27	33	77
Slide 12	22	17	33	40	90
Slide 13	23	27	37	51	115
Slide 14	24	20	25	34	79
Slide 15	25	27	33	46	106
Slide 16	26	27	33	46	106
Slide 17	27	15	16	22	53
Slide 18	28	18	31	38	87
Slide 19	30	19	23	30	72
Slide 20	31	20	22	29	71
Slide 21	32	25	40	53	118
Slide 22	65	29	36	50	115
Slide 23	67	30	37	52	119
Slide 24	35	15	18	22	55
Slide 25	77	16	21	26	63
Slide 26	75	12	11	14	37
Slide 27	24B	30	79	100	209
Slide 28	25B	69	91	138	298
Slide 29	26B	72	94	145	311
Slide 30	30B	22	26	36	84
Slide 31	75B	16	15	22	53
Average		24,84	37,13	50,23	112,19
Median		20	33	42	93

Table 8.23.: Size of positive Satisfaction Witnesses for ADASv4; C&C views of slide 27 to slide 31 are created during subsequent study (cf. Section 8.5).

8.4.4. Results from Addressing the Identified Challenges

Traceability

This main study showed that C&C views and the generated satisfaction witnesses assist engineers to collect traceability information between textual *IBM Rational DOORS* requirements and *Simulink* model implementations. Furthermore, C&C views verification helps to identify mismatches between requirement documents and different *Simulink* model implementations. For example, C&C views verification unveiled typos in signal names in the *Simulink* models as well as inconsistencies of encoded types in signal names (e.g., ending `_b` stands for Boolean type, `_stat` for an integer range representing an enumeration, `_m` for meter, and `_kmh` for kilometer per hour).

The generated satisfaction witnesses were useful to locate the high-level user-experience subsystems in the *Simulink* models. The manual inspection of the graphical representation of the satisfaction witnesses also identified that the `Limiter` subsystem has not been updated according to the requirement FA-68 when replacing the one-stage cruise control lever by a two-stage one. Figure A.22 on page 332 shows the C&C view containing the abstract effector `LeverDown_-stat -> VMax_kmh` which is not behaviorally satisfied by `Limiter_SetValue` *Simulink* subsystem in Figure A.23 (cf. Figure A.24 to see that the `Tempomat` subsystem satisfies a similar requirement by containing two subsystems `SetValueMinus` and `SetValueMinusLvl2` to decrease the value by N or to the next ten's place).

However, the domain experts, esp. the industrial partner, needed a complete mapping from a requirement to all *Simulink* blocks and their interaction to fully satisfy the traceability requirement. However, for every element in a C&C view the satisfaction witness contains only the smallest number of *Simulink* elements to demonstrate its satisfaction. Thus, the satisfaction witness helps to assist engineers in the traceability challenge, but it does not completely solve it. The subsequent study evaluates whether the graphical representation of the tracing witness satisfies the expectations of the domain experts.

Evolution

C&C views and their verification confirmed our two evolution related hypothesis:

1. C&C views verification is able to check whether all structural properties of the evolved *Simulink* model still satisfies all unchanged requirements.
2. C&C views verification is able to verify that structural changes, related to requirement updates, of the architecture design have been implemented in the *Simulink* model.

During the evaluation of the first hypothesis, we located inconsistencies of signal names in different *Simulink* versions. The evaluation of the second hypothesis (cf. case study execution in Subsection 8.3.4) unveiled that the *Simulink* model of ADASv2 does not implement the two-stage cruise control lever in contrast to its requirement document (cf. changelog in Figure 8.3).

Please remind that C&C views, as introduced in Chapter 7, only describe structural properties. Therefore, C&C views verification is not able to verify any behavioral properties of an implementation. To verify behavioral properties of a system other methodologies and verification tools (e.g., LTL [MR15], CTL [BK08], underspecification automata [Rum96]) exist. The analyzes of behavior models of *Simulink* systems is much more complex, because the semantics of a *Simulink* block diagram depends heavily on the specified solver settings (cf. [The18g]). Our paper *Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution* [RSvW^{+}15] formalizes the semantics of *Simulink* models using the fixed-step discrete solver. Thus, C&C views verification cannot address all evolution challenges [MMR10].

Nevertheless, the experiments helped to identify inconsistencies between requirement documentations and *Simulink* models. And most important, the evolution study reused all C&C views created for traceability. Hence, there are no additional expenses for industry to execute C&C views verification for model evolution, if C&C views are used in combination for generating tracing information/links.

8.5. Subsequent Study

The preliminary (cf. Section 8.2) and the main study (cf. Section 8.3 and Section 8.4) together with Daimler AG unveiled that textual C&C models and textual C&C witnesses are hard to comprehend. Furthermore, creating graphical representations[9] based on textual witnesses is a tedious and time consuming work (the author of this thesis needed on average one hour to manually translate a textual witness into a graphical representation). Therefore, an algorithm generating graphical C&C representations of textual *EmbeddedMontiArc* models has been developed (cf. [Sch18]). The subsequent study, explained in this section, tries to answer the two research questions:

Q5: How helpful are graphical representations of tracing witnesses of C&C views verification?
Q6: How much time need engineers with/without C&C views verification to detect important elements?

Execution of Subsequent Study

For the execution of the subsequent study we used the same framework as for the preliminary and main study. The **Objective** of the subsequent study is the evaluation of graphical representations of tracing witnesses for the traceability challenge. The subsequent study should evaluate whether the tracing witness addresses the drawbacks of the satisfaction witness according to the traceability challenge (cf. Subsection 8.4.4).

The studied **Case** is to figure out what tracing information domain experts still miss in graphical representations of the traceability witnesses. Furthermore, the studied case should evaluate graphical representations themselves (e.g., is the representation intuitive to the domain experts). Additionally, the studied case should evaluate how much working time engineers save for the traceability challenge (cf. Subsection 8.2.2) when providing C&C views with their generated graphical tracing witnesses to engineers.

[9]The graphical representation does not mean the PowerPoint slides which are presented during talk sessions. In the first study, the textual witness has been first completely modeled in PowerPoint. This graphical PowerPoint witness was very large for the complex industrial size *Simulink* model. In a next step, the first graphical witness has been remodeled (e.g., splitting up the model into several slides and hiding some port or component names) to be presentable. The aim of the subsequent study is to skip the first PowerPoint modeling step completely to directly create presentable PowerPoint slides based on the generated and easier to understand graphical representation.

The **Theory** of the subsequent study are the theory (cf. Section 8.3) and the results of the main study (cf. Section 8.4), the new tracing witness kind (cf. Subsection 7.5.2), as well as the layout algorithm (cf. Subsection 8.5.1) to automatically create graphical representations of textual *EmbeddedMontiArc* models.

The **Method** is to generate textual tracing witnesses (instead of satisfaction ones) with the algorithm presented in Section 7.4 and to use the layout algorithm to generate graphical representations. We show the graphical representations to the domain experts to receive feedback. To measure the saved time, in a first step domain experts should trace down all *Simulink* blocks of a given requirement without C&C views, and in a second step they should trace down all blocks of a given requirement with C&C views and with the graphical representation of the generated tracing witnesses; the difference is the saved time.

As **Selection** strategy we divided the 26 requirement groups (one requirement groups are the requirements belonging to one C&C view) of ADASv4 among the three domain experts participating on the subsequent study. Due to the time consuming task in inspecting all tracing witnesses, which are expected to be larger than the satisfaction ones, the evaluation is only done on the larger ADAS model.

8.5.1. Generation of Graphical Representations of C&C Models and Witnesses

This subsection presents the requirements, and the high-level steps of the algorithm to generate graphical representations of C&C models based on textual *EmbeddedMontiArc* models. Furthermore, this subsection compares the generated graphical representation of ADASv4 against the *Simulink* subsystem provided by Daimler AG.

Requirements for Graphical C&C Models

The generated graphical C&C representations should be similar to the ones of the visual *Simulink* models and to C&C model representations of existing papers of our chair (e.g., cf. [MRR13, MRR14, Rin14, Hab16, Wor16]). Thus, the generated graphical representation should satisfy the following requirements (some of this requirements are already published in [Sch18, Chapter 2]):

- V1: Components are displayed as rectangles.
- V2: The size of components may vary, e.g., based on number of ports.
- V3: The component name/type is inside the component rectangle and it is in the top part.
- V4: Ports are displayed as squares.
- V5: All ports have the same size.
- V6: All input ports are on the left side of a component.
- V7: The text of an input port belonging to a subcomponent/outer component is right/left of the input port square.
- V8: All output ports are on the right side of a component.
- V9: The text of an output port belonging to a subcomponent/outer component is left/right of the output port square.
- V10: Connectors consist of horizontal and vertical lines.

V11: The head of every connector is a right arrow.
V12: Connectors start and end with a horizontal line.
V13: Junctions of connectors are displayed as small filled circles.
V14: Connectors having the same source and target component instance are combined as buses (similar to *Simulink*).
V15: Bus creators and bus selectors have unique indices to match the ingoing and outgoing signals of buses; the graphical order may not match.
V16: Connectors should go from left to right whenever possible; only feedback-loops should go from right to left.
V17: All elements (except of connector lines) should not overlap each other.
V18: The number of cross-cutting lines should be as small as possible.
V19: All graphical elements should be inside the canvas' bounds; port names of the outer component should also be visible (via scrolling).
V20: The size of the canvas should be as small as possible.
V21: The output should be static (client-side) HTML pages. Each hierarchy layer is one HTML page.
V22: The URL of the HTML page maps to the full-qualified name of the displayed outer component. This way readable links to a specific hierarchy presentation can be shared via e-mail.
V23: The HTML page should contain a navigation bar to navigate to direct and indirect parent components.
V24: Ports and all atomic components have a white background color.
V25: Non-atomic components have a light-gray background color.
V26: Clicking into a non-atomic component opens the graphical representation of this clicked subcomponent.

Since the graphical representation of one hierarchy layer may become very complex, and thus also large, the algorithm should support four different abstractions for each layer:
V27 - Simplified: The graphical representation contains only components with simple instance names and connections between components.
V28 - No Port Names: The graphical representation consists only of components with simple instance names, ports without names and without types, and connections between ports.
V28 - Standard: The graphical representation contains components with simple instance names, ports with names, and connections between ports.
V29 - Extended: The graphical representation consists of components with component types in bold font and simple instance names in normal font, ports with names and types, connections between ports, as well as tag information for components, ports, and connections in small font.

V30: The URLs of the four different graphical representations are unique to share them.
V31: All graphical elements in all hierarchy levels and all graphical abstractions have unique identifiers in the generated HTML page. The unique identifiers correlate with the full-qualified name of the visualized textual C&C element.

The extended representation (cf. V29) is also used for the interactive simulator, where the simulator replaces the text of port data types with their current port values. The interactive simulator updates the values in the graphical representation via JavaScript using the unique identifiers (cf. V31).

Algorithm Creating Graphical C&C Models

This paragraph summarizes the algorithm how to generate the graphical representation of C&C models. A more complete description of the algorithm is available in the master thesis *Visualisation of Textual Component and Connector Models* [Sch18, Chapter 5] supervised by this author.

The main steps of this algorithm are (summary of [Sch18, Chapter 5]):

1. The algorithm creates for each component instance in the abstract syntax (presented in Chapter 4) of the textual *EmbeddedMontiArc* model a new directed graph. Each graph contains nodes of all direct subcomponent instances plus one left and one right node representing the borders of the current component instance. An edge between nodes exists when ports of the corresponding subcomponent instances are connected.
2. Since the dataflow should go from left to right, the algorithm separates the graph in a set of paths. Every edge is part of exactly one path, two paths share at least two nodes, and a shared node must not be a middle node in both paths.
3. The algorithm inserts temporary nodes (nodes not matching any component instance) for feedback loops (cycles), buses (new nodes are inserted for bus creator and bus selector), and to avoid parted components.
4. The algorithm permutes over the set of paths to find a "readable" vertical ordering of the paths. A fitness function evaluates the readability of a current permutation result to minimize the number of parted nodes, edge crossings, the number of edge bends, and the length of vertical edge segments. The permutation's heuristic is based on simulated annealing[10] [Cha96] with a logarithmic temperature function.
5. After the fourth step, the layout (row and column position) of the component instances (node positions) is fixed. This step merges two paths to one path if they do not have nodes in the same column.
6. A component may be represented by multiple nodes and these nodes may not be row neighbors, so the algorithm switches the paths in a way so that the component nodes are not parted anymore.
7. The algorithm assigns for each component instance hierarchy coordinates for the elements inside this hierarchy based on the final set of paths calculated in step 6. The width of a component is based on the length of its component instance name, its component type name, and the names of its ports. The height of a component is based on the positions of the input and output ports.
8. The layout after step 7 is a table-based layout where all components are below or next each other. Due to busses and feedback-loops the space between the component columns may

[10]The algorithm of simulated annealing is inspired by the annealing process in metallurgy heating and cooling down material to reduce the number of dislocated atoms and thus to change the plastic deformation of the material [Sch18].

become pretty large. Therefore, the algorithm moves all components (with their ports and their port connections) inside the same row as far as possible to the left.

9. Based on final coordinate positions, the algorithm generates for each layer an HTML and SVG file using *FreeMarker* templates. Every generated HTML file contains a navigation bar and the SVG image. Each graphical element in the SVG file has a unique identifier which can be derived from the full-qualified name of its port or component instance, the identifiers of connector elements are based on the full-qualified name of the unique target port.

The layout algorithm also supports a mode to generate more *Simulink* equivalent graphical representations. This mode replaces the component instance names, component type names, and port names of atomic components of specific component types with text fragments being more similar to the visual representation of the corresponding atomic *Simulink* block. For example, the component type `GreaterEquals` is replaced by `>=` and the instance name of this atomic component is skipped in the graphical representation. Another example is the `And` component type, the layout algorithm skips the component instance name and the port names (as they are irrelevant). This mode should make the graphical representation as intuitive as possible for the domain experts participating in the subsequent study. A nice side effect of these text replacements is that the sizes of these special components becomes much smaller, which results in a better readable graphical representation.

Evaluation of Graphical C&C Models

Figure 8.24 shows a screenshot of the `CC_On_Off` subsystem of the ADASv4 *Simulink* model provided by Daimler AG. Figure 8.25 shows a screenshot of the generated graphical representation of the translated *EmbeddedMontiArc* model. Figure 8.25 shows the standard view (cf. requirement V28) containing the same information as the *Simulink* model. Figure A.9 on page 323 shows the simplified view (cf. requirement V27); and Figure A.10 on page 324 shows the extended view (cf. requirement V29) containing additional information hidden in *Simulink* dialog boxes.

First, both graphical representations are good readable as both have less cross-cutting lines. The *Simulink* model (layout is created manually) contains no cross-cutting signal lines at all, and the generated one consists of only one cross-cutting connector (connector starting at input port `CruiseControl_b` and going to the `OR` block intersects the connector going from `limiter_b` to `NOT`). Furthermore, the data-flow in both graphical representations goes only from left to right. However, the concrete representation differs; as the graphical representation of Figure 8.25 is generated on a textual model containing no information about the graphical position of the corresponding *Simulink* blocks. Therefore, a 1:1 matching of both graphical representations is not so easy possible.

Still, the domain experts stated that the generated graphical representation shown in Figure 8.25 is very comprehensible for them; the industrial partner also liked the simplified representation (cf. Figure A.9 on page 323) containing component instances and their communication to receive a first overview. The extended representation (cf. Figure A.10) was not so useful for the industrial partner, because the component and port data types are already encoded in component instance names or in component port names; and the larger image size of the extended representation caused more scrolling activities. Interviews of the author of this thesis with other

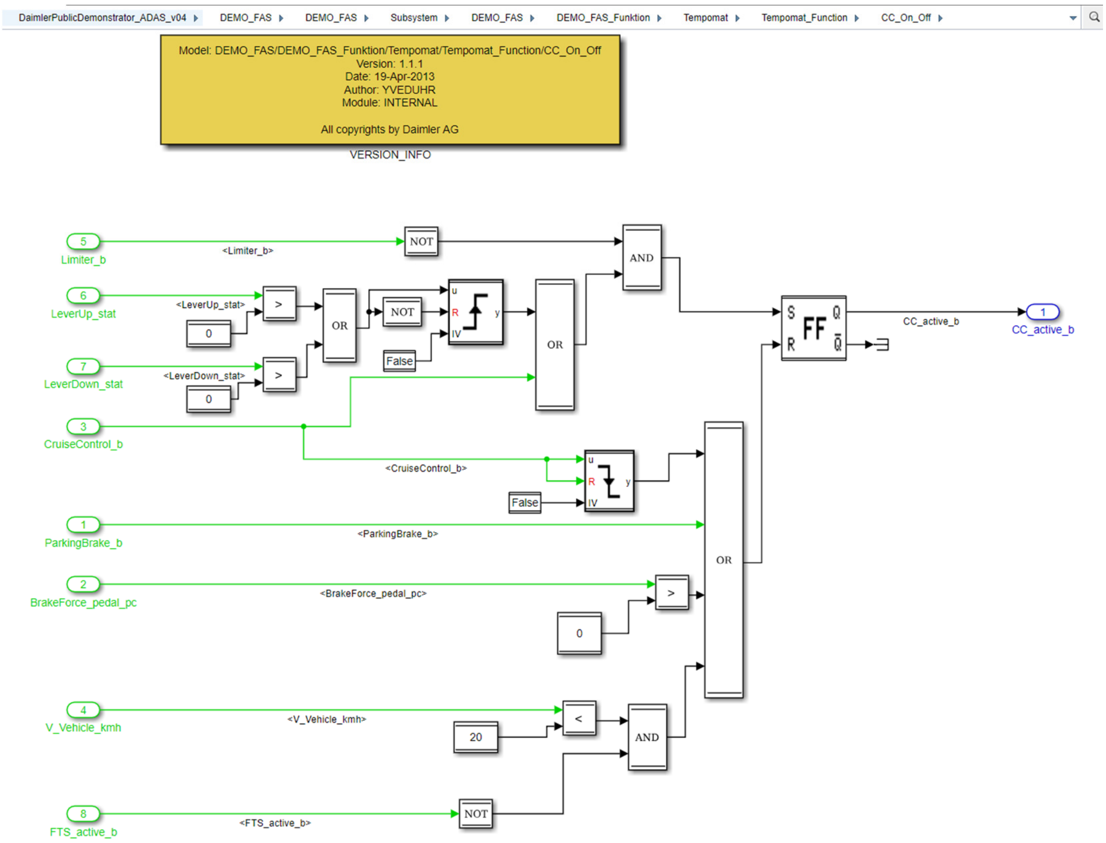


Figure 8.24.: Screenshot of CC_On_Off subsystem in *Simulink* (cf [Dai13f]).

EmbeddedMontiArc developers unveiled the need for the extended representation. The other developers created among others PacMan, ImageCluster, SuperMario, or a racing car controller in *EmbeddedMontiArc*; all these models have in common that data types and port names do not correlate at all with port names or component instance names.

Section A.1 lists more screenshots comparing the manual layout of *Simulink* models against the automatically generated layout of our visualisation tool. This section also demonstrates that the simplified representation creates much smaller layouts which are very helpful to receive an overview between the communication of component instances.

Our visualisation tool creates 1129 SVG and HTML files[11] for the graphical representation of the ADASv1 C&C model with its 639 component instances; the running time to create all these files is 5 minutes and 35 seconds. For the C&C model of ADASv4, having 1396 component instances, our visualisation tool generated 1865 SVG and HTML files in 20 minutes and 26 seconds. However, our visualisation tool is also capable to just update the visualisation of one hierarchy level, e.g., when changing just one *EmbeddedMontiArc* file, and this needs only about one second. Developers need to generate the complete graphical representation only once; later

[11] For each non-atomic component instance four HTML and four SVG files are generated.

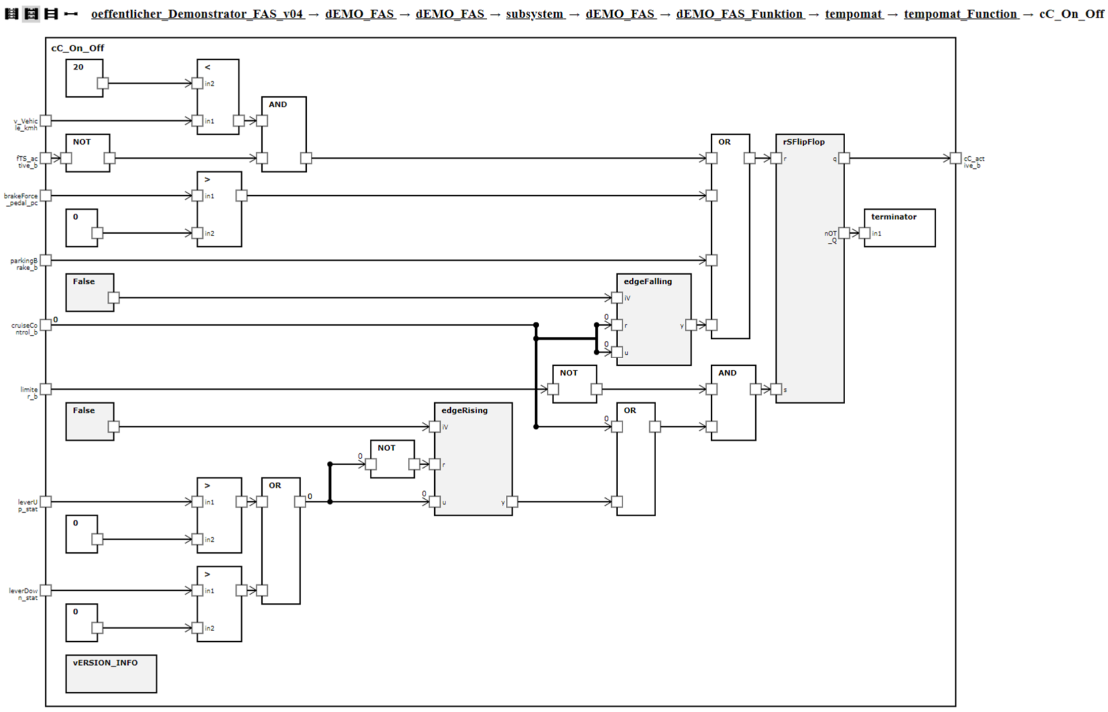


Figure 8.25.: Screenshot of generated graphical representation of translated *EmbeddedMontiArc* model which shows the port names and the component names (layer 3). This version shows similar information as the *Simulink* models.

this graphical representation can be updated incrementally based the textual changes in a few seconds (depending how many hierarchy layers have been modified). Hence, the runtime of our visualisation tool is still capable for industry.

When generating only one view, e.g., the standard one described in requirement V28, the web visualisation for ADASv1 needs less than 3 minutes with our tool. The time to create only one out of four views is not divided by four, as tasks like starting the program (cold-start time of JVM), parsing all textual models and creating the symbol table as well as transforming *EmbeddedMontiArc* models to C&C instance structures is only done once and not for every view representation kind. Executing the same task in *Simulink* also needs about 3 minutes (ca. 30s to start *MATLAB*, ca. 60s to load the *Simulink* model, and ca. 90s to generate the web export of ADASv1 for all subsystems and library components); and *Simulink* does not need to calculate any layout information as it uses the graphical layout created by the user. Thus, we conclude that the performance of our tool generating the graphical representation as web files is similar to the performance of the most used industrial tool *Simulink*.

The performance of the complete C&C views verification toolchain is important to illustrate that C&C views verification may be integrated into the industrial development process (cf. Subsection 2.1.2 and Subsection 2.2.2) to improve quality.

8.5.2. Results of Subsequent Study

The three domain experts of the subsequent study needed 2-5 hours to identify all *Simulink* models being related to one requirement without the usage of C&C views; each domain expert analyzed only five requirements, because this task was so time intensive[12]. At least two hours are needed to analyze communications between blocks belonging to the ADAS system and the closed-loop involving the environment. Five hours are needed when the domain experts must analyze the data-flow between subsystems via global variables to identify side-effects. In contrast, given the graphical representation of the tracing witnesses, the domain experts only needed 30 minutes to identify all relevant elements in the *Simulink* model. They needed these 30 minutes, because the tracing witness is generated based on the *EmbeddedMontiArc* model and this differs (due to special *Simulink* blocks, cf. Subsection 8.3.5) from the *Simulink* one; therefore, the domain experts still needed some time to match the graphical witness components to the correct *Simulink* subsystems/blocks. Together with the 30 minutes to create a C&C view (cf. Subsection 8.4.1), the domain experts spend all together about one hour to trace down one *Simulink* requirement using C&C views and the graphical tracing witness. This is a very good result when considering the 2-5 hours they needed without using C&C views. For the 26 C&C views of ADASv4 this saves about 7 days of working time; answering research question **Q6**.

Table 8.26 and Table 8.27 list the sizes of the generated satisfaction witnesses. The average sum and the average difference (last two columns in the penultimate line) show that tracing witnesses are about 40% / 60% larger than the satisfaction witnesses of ADASv1 / ADASv4. This confirms the impression of the industrial partner that satisfaction witnesses skip many elements according to the traceability challenge. This statistic also demonstrates that a graphical representation of tracing witnesses containing all elements in one layer similar to the graphical representation of satisfaction witnesses as shown in Figure 8.11 is not suited, because tracing witnesses with a median value of 46 components (cf. third column in last row in ADASv4) are too large.

First, we generated only graphical representations of tracing witnesses. However, the domain experts complained that the layout of these graphical witness representations differ too much; this is caused by the automatic layout process. Therefore, we showed the domain experts an alternative representation of witnesses, which highlights all elements of the tracing witness in the complete graphical representation. This way, the graphical representations of all witnesses of different requirements have the same layout, and the highlighted parts show all elements addressed by this requirement. Even though the graphical representations of the highlighted witnesses are much larger[13], the domain experts preferred this highlighted version. In general, the domain experts liked the graphical representations of highlighting the tracing witnesses inside the ADASv4 *EmbeddedMontiArc* model. Only some graphical representations of components at deeper hierarchy levels, e.g., `CC_SetValue` hierarchy level of colorized tracing witness of

[12] We repeated the manual task to trace down requirements (cf. Subsection 8.3.1 where it has been done the first time), because the main study did not exclude very similar requirements (e.g., `Limiter` and `Tempomat` requirements, as well as only evolved requirements from ADASv1 to ADASv4). The subsequent study excluded similar requirements as it evaluates the average time is when just tracing down one requirement at a time (e.g., to create a new test or to create a presentation to present the implementation during a sprint review meeting).

[13] 46 components (median size of components in witnesses) vs 1396 ones (number of components in ADASv4)

TracingADAS1.pptx	FA-	Components	Connectors	Ports	Sum	Difference to Satisfaction
Slide 1	14	9	2	4	15	0
Slide 2	19	38	70	90	198	121
Slide 3	27	37	46	65	148	64
Slide 4	28	48	90	117	255	126
Slide 5	29	30	56	72	158	68
Slide 6	25	21	26	34	81	0
Slide 7	22	11	17	21	49	0
Slide 8	23	41	54	76	171	73
Slide 9	24	42	55	78	175	77
Slide 10	30	13	14	18	45	0
Slide 11	26	38	79	101	218	131
Slide 12	33	15	18	22	55	0
Slide 13	38	26	35	48	109	42
Slide 14	34	33	64	81	178	67
Slide 15	35	41	60	85	186	79
Slide 16	36	45	67	95	207	96
Slide 17	37	15	18	22	55	0
Average		29,59	45,35	60,53	135,47	55,53
Median		33	54	72	158	67

Table 8.26.: Size of positive Tracing Witnesses for ADASv1; the difference in the last column is the difference between the sum in this table and the sum in Table 8.22.

TracingADAS4.pptx	FA-	Components	Connectors	Ports	Sum	Difference to Satisfaction
Slide 1	15	18	39	51	108	21
Slide 2	3	47	98	129	274	139
Slide 3	4	81	167	224	472	251
Slide 4	5	48	108	140	296	148
Slide 5	6	71	144	193	408	273
Slide 6	99	78	157	210	445	312
Slide 7	86	25	52	65	142	49
Slide 8	84	46	100	133	279	179
Slide 9	19	19	23	30	72	0
Slide 10	20	11	11	14	36	0
Slide 11	21	45	84	108	237	160
Slide 12	22	45	90	115	250	160
Slide 13	23	65	114	157	336	221
Slide 14	24	53	84	117	254	175
Slide 15	25	53	92	124	269	163
Slide 16	26	53	92	124	269	163
Slide 17	27	46	91	118	255	202
Slide 18	28	41	82	107	230	143
Slide 19	30	20	25	33	78	6
Slide 20	31	23	27	37	87	16
Slide 21	32	54	100	135	289	171
Slide 22	65	42	61	87	190	75
Slide 23	67	46	68	97	211	92
Slide 24	35	16	20	25	61	6
Slide 25	77	17	23	29	69	6
Slide 26	75	51	97	125	273	236
Slide 27	24B	94	189	253	536	327
Slide 28	25B	129	211	304	644	346
Slide 29	26B	138	227	329	694	383
Slide 30	30B	24	30	42	96	12
Slide 31	75B	55	101	133	289	236
Average		50,13	90,55	122,19	262,87	150,68
Median		46	91	118	255	160

Table 8.27.: Size of positive Tracing Witnesses for ADASv4; the difference in the last column is the difference between the sum in this table and the sum in Table 8.23.

FA-26, were not so comprehensible for the domain experts; because the graphical representation of the *EmbeddedMontiArc* model differs much from the *Simulink* one due to switch components added during the translation process. This answers research question **Q5**. Section A.2 presents some graphical representations of C&C views, satisfaction and tracing witnesses, as well as the graphical representation highlighting witnesses. All graphical representations of C&C views and all positive satisfaction and tracing witnesses are online available from *EmbeddedMontiArc*'s GitHub pages[14]. The author of this thesis explicitly allows to reuse these materials and results for further case studies.

The evaluation of the domain experts in this subsequent study also unveiled that some C&C views of ADASv4[15] were not detailed enough to trace down all *Simulink* elements implementing one requirement. For this reason, the domain experts updated the C&C views of the requirements FA-24, FA-25, FA-26, FA-30, FA-65, FA-67, and FA-75. As already mentioned in Subsection 8.4.4 the `Limiter` subsystem of ADASv4 does not implement the two-stage cruise control lever, thus the updated C&C views of FA-65 and FA-67 are not satisfied, and so neither a satisfaction nor a tracing witness is generated. For all other C&C views, the satisfaction and tracing witnesses are available from *EmbeddedMontiArc*'s GitHub pages; the extension views and the witnesses end with a `B` such as `FA-24B`.

The average time to check positive satisfaction and to generate the positive satisfaction witness is below half a second (cf. Table A.37) per C&C view. Surprisingly, the verification time and the generation of the larger tracing witness is also below half a second (cf. Table A.38). Generating the graphical representation of satisfaction and tracing witnesses needs on average below 10 seconds per textual witness (cf. Table A.37); however, large tracing witnesses may need about one minute (cf. FA-26b in Table A.38).

Highlighting graphical elements needs on average about 2-5 seconds per textual witness, because our implementation loads the already generated HTML and SVG files and just modifies via JSOUP [Hou13] the line color attribute in the DOM (document object model of websites). The mapping from the abstract syntax of a witness to the DOM elements is straight forward, as all DOM elements in the graphical representation have an identifier encoding the full-qualified name of the C&C instance structure of the *EmbeddedMontiArc* model satisfying the view (cf. requirement V31).

Using the engineers' preferred representation by highlighting the tracing witness elements in the C&C model, the execution of the complete toolchain (tracing verification, textual witness generation, and highlighting the graphical representation) needs about 3 to 6 seconds per C&C view. This means the engineers receives the tracing results of a C&C view nearly instantly. Most important, the fast execution time of the entire toolchain does not interrupt the workflow of developers. Section A.4 contains the measured runtime for all C&C views.

8.6. Additional Observations and Desired Extensions

Interviews during the main study in 2017 unveiled that engineers at Daimler AG create *Simulink* models with manually highlighted blocks or manually deleted elements to present only important information (slices) when discussing requirement implementations or analyzing defects. For this purpose, the University of Ulm developed for Daimler AG a tool to highlight effect chains. C&C views verification enables validating the existence of effect chains automatically, e.g., during nightly builds, and tracing witnesses also support to highlight (if it is present) effect chains. The automatic generated graphical representations of witnesses even supports generating

[14] `https://embeddedmontiarc.github.io/webspace/`

[15] Due to time reasons we only extended the C&C views of ADASv4.

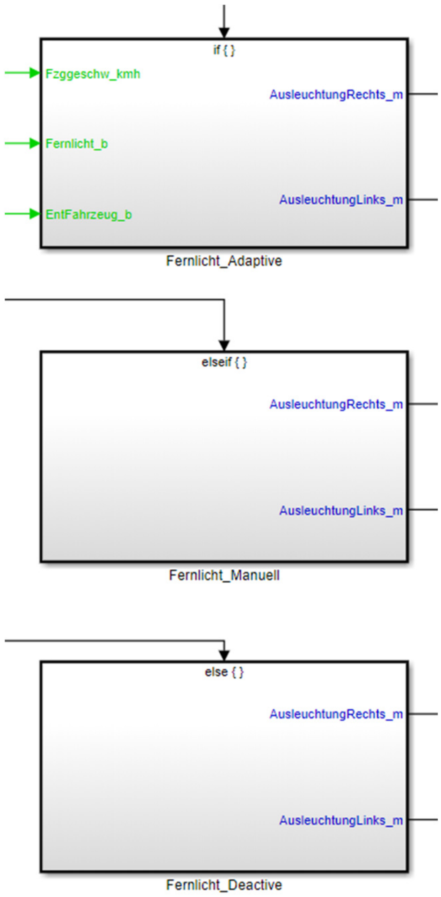


Figure 8.28.: Different Modes of High Beam subsystem; cf. [Dai13h].

different graphical overview levels (cf. Subsection 8.5.1). However, our toolchain still cannot generate defect slices (or its highlighting in *Simulink*) full-automatically, because our verification algorithm does not support conditional abstract effectors. For defect tracing it is important to look at effects occurring only under special conditions such as at highways where the speed is larger than 60 km/h. Our algorithm would now find all effects between the corresponding input and output ports and then the modeler needs to remove manually all for this situation unimportant component-connector chains.

This case study also figured out that requirements describing different modes of subsystems are not so well suited for C&C view verification on *Simulink* models. In the presented examples the modes are modeled via enabled subsystems or via if/else subsystems as shown in Figure 8.28. C&C views verification can check that the subsystems (also the modes) are contained in *Simulink* models, but C&C views verification does not support modes as modeling feature to specify that only one of the subsystems should become active. For this feature, the *EmbeddedMontiView* language, and thus also the C&C views verification and witness generation algorithms, could be extended with a Modi mechanism as it exists in *MontiArcAutomaton* [HKR+16].

8.7. Threats to Validity

This industrial case study is based on *Simulink* models released by Daimler AG for evaluation and demonstration purposes. Therefore, the ADAS and ALS *Simulink* models are simplified by removing subsystems containing intellectual property as well as by removing *AUTOSAR* integration frames. Thus, it is possible that these removed elements would require additional C&C views (verification) features. However, it is neither allowed to describe these features in public publications, nor to make this models public available; thus, there exists no way to address this thread.

We were not able to evaluate the use of C&C views on pure C&C industrial models, because *EmbeddedMontiArc* and its tooling is not used by industry (yet). Hence, all three studies in this chapter evaluated *Simulink* models; *Simulink* block diagrams are close to C&C models, however, *Simulink* supports additional communication paradigms and control-flow modeling. To mitigate this threat, Subsection 8.3.5 explained in detail how *Simulink* block diagrams were automatically translated to C&C models. Black-box tests ensured that *Simulink* block diagrams and the created *EmbeddedMontiArc* models have the same behavior.

It is important to note that all three studies (i.e., preliminary, main, and subsequent study) in this chapter worked on existing models and on existing requirements. Furthermore, all materials used for this industrial case study are online available as structured websites for further investigations and to enable replication of this industrial study.

8.8. Similar Studies

The C&C views approach for requirement is not completely new. This technique has already been investigated by Grönniger, Hartmann, Krahn, Kriebel, and Rumpe in the paper *View-Based Modeling of Function Nets* [GHK$^+$07]. However, the authors of the function net paper state: "While the results of smaller case-studies are promising, a detailed evaluation of the method with an example of realistic size still needs to be carried out" [GHK$^+$07, Conclusion]. This industrial case study addressed their last point by evaluating the C&C views approach on five industrial size models with real-world requirements. Furthermore, our industrial case study showed that C&C views may support engineers in industry.

The paper *Modeling Variants of Automotive Systems using Views* [GKPR08] by Grönniger, Krahn, Pinkernell, and Rumpe uses views to model variants and features. The advanced driver assistant system (ADAS) also contains different features (cf. feature diagram in Figure 8.14), and the C&C views derived from the requirements belong to different features. The evolution challenge in our case study used C&C views to figure out whether any features were accidentally removed during model evolution or whether all required features are implemented; this industrial case study unveiled that the two-stage cruise control lever was added to requirements in ADASv2, but it was only implemented in ADASv3.

In 2014, Broy figured out during his study that description and verification of requirements are one of the biggest challenges: "To capture the requirements right (i.e. complete and consistent) is the basis for the development of software. The study participants report that the description and the verifiability of the requirements and the reconciliation OEM - supplier are currently the biggest challenges they face in the requirements analysis" [BKKS14]. The interviews with our industrial partner confirmed this statement. Furthermore, Broy's survey unveiled: "Study participants report that up to 15 000 requirements per function has become a normal value". Therefore, we can conclude that the system used in this case study with its 150 requirements (ADASv4 plus ALS requirements) is one of the smaller ones in industry. Addressing the traceability challenge in this case study manually was already a lot of work; we needed two to five hours of work for each requirement. With the support of C&C views, this work can be reduced by two hours per requirement; for the 15 000 requirements (mentioned by Broy) this would save about 15 person years of work, and thus, also a lot of money.

Another study by Mäder and Egyed unveiled that "subjects with traceability performed on average 24% faster on a given task and created on average 50% more correct solutions - suggesting that traceability not only saves effort but can profoundly improve software maintenance quality" [ME15]. Thus, C&C views may help to improve the quality of software. Indeed, our case study identified a missing component in ADASv4 for requirement FA-67, and wrong connected signals in ALS (cf. appendix A3). Hence, our case study confirmed the observation by Mäder and Egyed.

8.9. Summary of Industrial Case Study

Maoz et. al. [MRR13, MRR14] already suggested C&C views and its verification as formal and intuitive structural specification of C&C models. This chapter described the experience in applying C&C views verification to address traceability and evolution in an industrial automotive setting at Daimler AG. Besides conceptional questions whether C&C views verification may (and in what context) help engineers (cf. preliminary and main study in Section 8.2 to Section 8.4 based on the industrial case study paper [BMR+17a]), this thesis also evaluated the usefulness of the tooling around C&C views verification (adapted witness generation, generating graphical representations, and highlighting witnesses directly in C&C models) and how much time engineers may actually save when integrating this tooling into their development process (cf. subsequent study in Section 8.5).

Even though this case study has only been applied with one automotive company, we know based on other industrial collaborations [KKRvW18, HKK+18, RRS+16] that traceability is for many other automotive OEMs or suppliers a very important issue and that the links and the verification between the design model (cf. SMARDT level 2 in Figure 2.3 on page 29) and the logical model (cf. SMARDT level 3 in Figure 2.3) are yet done manually, and thus, time consuming and error-prone.

Although these three studies focused on the automotive domain, the evaluation results of these studies on C&C views verification may also relate to other industrial companies in embedded or cyber-physical domains, e.g., avionics [FG12], wind power systems [AB17, BPB17] robotics, assembly and production systems [BKL+18], or telecommunication [HB06].

This industrial case study showed that C&C views verification with its tooling to automatically generate a graphical representation for tracing witnesses supports engineers to address the traceability challenge. It also unveiled that C&C views verification with its natural language error description helps developers to locate accidently broken requirements during the evolution challenge.

The overall effort to develop the algorithmic concepts plus all prototype implementations and documentations of the used toolchain to execute these preliminary, main, and subsequent studies involved about 65 person months of working time. Examples of the developed algorithmic concepts are: translating *Simulink* models to C&C models, adapting the C&C views verification algorithm of Ringert [Rin14] to support abstract effectors, adapting the witness generation process to generate tracing witnesses, and automatically generate graphical representations of textual *EmbeddedMontiArc* models and witnesses as well as highlighting existing C&C models in an efficient way.

Chapter 9.

Summary and Conclusion

This thesis aims to improve the development process of software systems engineering for embedded and cyber-physical systems. Therefore, this thesis provides domain specific languages to develop component and connector (C&C) models, as well as to specify structural design decisions and extra-functional properties of C&C models in an efficient, agile, and intuitive way. Section 9.1 shortly summarizes the main results of this thesis. Section 9.2 concludes the contribution of this PhD thesis.

9.1. Main Results

The goal of this thesis was to improve the software development process of large and complex C&C models for embedded and cyber-physical systems; esp. in the automotive domain. The approach of this thesis follows the current model-based development process of large car manufactures, and it addresses the engineering challenges traceability and evolution. Main achievements of this work is to provide automatic consistency checks between requirements, high-level design models, functional C&C models, as well as extra-functional properties.

Chapter 3 and Chapter 4 introduced the new functional C&C modeling language *EmbeddedMontiArc*. *EmbeddedMontiArc* is a textual domain-specific language which enables an efficient development of the logical/functional layer of embedded systems by providing the following language features:

(i) SI unit system to model physical quantities used in the interaction between components and its environment.
(ii) Component types including component interfaces to enable reusability and model architectural flexibility.
(iii) Arrays of component instantiations and ports to avoid copy and paste.
(iv) Powerful component libraries with parameters for port types, array dimensions, and components themselves to support configuration of internal/external components.
(v) Comfortable name- and index-based connection patterns to increase readability and to speed up modeling of large C&C architectures.
(vi) Component types as configuration parameters enables flexible product-line modeling.
(vii) Strict type system supporting algebraic matrix types detects model inconsistencies during compile time to reduce time-consuming error analysis due to wrongly connected components.
(viii) First level integration for black-box unit and integration tests to increase product quality.

Chapter 5 presented a tagging mechanism, which improves the systems engineering process by providing a non-invasive way to enrich C&C models with consistent extra-functional properties. Advantages of this new introduced methodology compared to existing annotation-, comment, or stereotype-based solutions are:

(ix) C&C models are not polluted with many different extra-functional properties.
(x) C&C models may have different sets of extra-functional properties (e.g., different hardware deployments).
(xi) Strict separation of concerns allows multiple domain experts to decorate a C&C model with different extra-functional properties in parallel by just focusing on their domain.
(xii) Tagging mechanism is typed (e.g., with physical units) to prevent careless mistakes (e.g., slipping in lines or typos).
(xiii) Tag schemas supports definition of new extra-functional properties in an efficient way.
(xiv) Tagging mechanism includes consistent meta- and table-based tags.

The *OCL* framework, introduced in Chapter 6, improves the development process by enabling efficient (in very less lines of *OCL* code) definition of context condition, company specific guideline rules for C&C models, as well as consistency rules for many kinds of extra-functional properties. Our *OCL* framework improves the systems engineering process in the following way:

(xv) Definition of company specific guideline rules in very few lines of code.
(xvi) Automatic generation of expressive error messages.
(xvii) Providing a mathematical framework to specify consistency rules on extra-functional properties.
(xviii) Automatic generation of positive consistency and negative inconsistency witnesses based on defined consistency rules on extra-functional properties.
(xix) *OCL* rules depend only on internal structure of *EmbeddedMontiArc* specified via class diagrams; thus, no knowledge about *EmbeddedMontiArc*'s Java implementation is needed.
(xx) Verification and witness generation is very fast and scales up to industry-size models.

The C&C design language *EmbeddedMontiView*, elucidated in Chapter 7, improves the C&C development process by providing an intuitive and formal way how to specify concrete architectural design decisions for one specific embedded and cyber-physical system. The specification language *EmbeddedMontiView* extends the concrete syntax of *EmbeddedMontiArc* with intuitive underspecification concepts. Therefore, no knowledge about the abstract syntax or any theoretical complex theory such as SMT theory is necessary to formulate concrete design decisions. Furthermore, the specific syntax of the C&C design language enables to automatically generate positive satisfaction witnesses to explain why a logical architecture (C&C model) satisfies a design specification. Additionally, non-satisfaction witnesses with its natural language description of errors and the model elements causing the incompatibility between logical architecture and elements of design specifications enables to locate and understand these inconsistencies. *EmbeddedMontiView* supports the development process with the following features:

(xxi) Intuitive specification of structural properties of C&C models.
(xxii) Fast verification algorithm to check whether a logical C&C architecture satisfies all design specifications.

(xxiii) Generation of non-satisfaction witnesses to locate inconsistencies between specification and architecture.
(xxiv) Generation of positive satisfaction witnesses to explain why an architecture satisfies a specific design specification.
(xxv) Generation of positive tracing witnesses to trace down all architectural elements belonging to a design specification.

Chapter 8 evaluated the improved development process together with Daimler AG during an industrial case study. To increase acceptance of the developed methodology in industry, our toolchain has been extended by the following features:

(xxvi) Automatic generation of graphical representations of textual C&C models with four different abstraction levels.
(xxvii) Highlighting all important elements of an architecture according to a C&C view.
(xxviii) C&C view verification tool supports besides *EmbeddedMontiArc* also *Simulink* files as C&C model input.

The industrial case study in Chapter 8 also proofed that the methodologies presented in this thesis scale for real-world industry models and that the improved methodology saved much working time of developers for the traceability challenge.

9.2. Conclusion

This thesis presented novel modeling languages for the development of embedded and cyber-physical systems.

The *EmbeddedMontiArc* language family defines the structure and behavior of C&C systems to model the functional and logical layer of cyber-physical systems in an efficient, agile, and intuitive way. *EmbeddedMontiArc* supports to define modular and reusable architectures by introducing a comprehensive component type system as well as arrays of port and component instantiations.

In contrast to most graphical C&C modeling languages in industry, *EmbeddedMontiArc* is a textual one. The textual nature of *EmbeddedMontiArc* enables a seamless integration into existing DevOps lifecycle platforms such as *GitLab* or *GitHub*. These platforms support among others versioning, branching, merging of *EmbeddedMontiArc* artifacts, as well as they provide powerful difference tools, an issues (tickets) system, continuous integration tests, code analyses and code reviews linking to line numbers of *EmbeddedMontiArc* artifacts.

EmbeddedMontiArc itself is completely model-based developed by empowering *MontiCore*'s grammar format to define the concrete syntax (cf. *EmbeddedMontiArcParsing* grammar) and its internal structure (cf. *EmbeddedMontiArcTooling* and *CnCInstanceStructure* grammars). The context conditions of *EmbeddedMontiArc* as well as the two transformations between the abstract syntax of these three *EmbeddedMontiArc* grammars are specified via *OCL* in a model-based way.

The tagging mechanism of this thesis also uses a model-based approach to define new kinds of extra-functional properties via tag schemas and to enrich C&C models with these extra-functional properties via tag models. All valued tags in tag schemas are typed which enables to check concrete extra-functional property values to reduce modeling failures. Constraints about C&C

models enriched with a specific extra-functional property type are also specified via *OCL* in a model-based manner. This thesis also presented techniques to validate whether an enriched *EmbeddedMontiArc* model satisfies all extra-functional property constraints. Additionally, the result of this validation is a positive or a negative consistency witness model to help the developer to understand the validation result.

The C&C view language *EmbeddedMontiView* specifies structural properties of *EmbeddedMontiArc* artifacts in a model-based, expressive and intuitive way. The strength of C&C views is the ability to describe abstract relations between different hierarchy levels. *EmbeddedMontiView* provides abstractions/underspecification for hierarchy, connectivity, interface completeness, data flow, component types, port types with units, as well as arrays of ports and component instantiations.

The development of *EmbeddedMontiView* also uses a model-based approach: The concrete and abstract syntax of this language are defined by an *MontiCore* grammar; and its context conditions as well as the satisfaction relation between *EmbeddedMontiArc* and *EmbeddedMontiView* are specified via *OCL*.

For the extra-functional consistency checks as well as for the verification of the C&C views satisfaction relation we implemented a prototype. This prototype and its integration in an industrial C&C development process has been evaluated during a case study together with Daimler AG. All results of this industrial case study have been uploaded to the GitHub pages[1] of *EmbeddedMontiArc* to make all these artifacts public available for further exploration and research.

The author of this thesis believes that our work provides promising results to improve the model-based development process of embedded and cyber-physical systems in industry.

[1] `https://embeddedmontiarc.github.io/webspace/`

Appendix A.

Appendix for Industrial Case Study

A.1. Screenshots of Visualisation

This section contains additional screenshots of graphical representations generated by our visualisation tool. It also shows some *Simulink* models. Please note that the *Simulink* model and the translated C&C model do not match 1:1; cf. Subsection 8.3.5 for details. Therefore, the graphical representation of the C&C models may have more ports as well as more connections to show the communication between components which exchange data in *Simulink* via global variables.

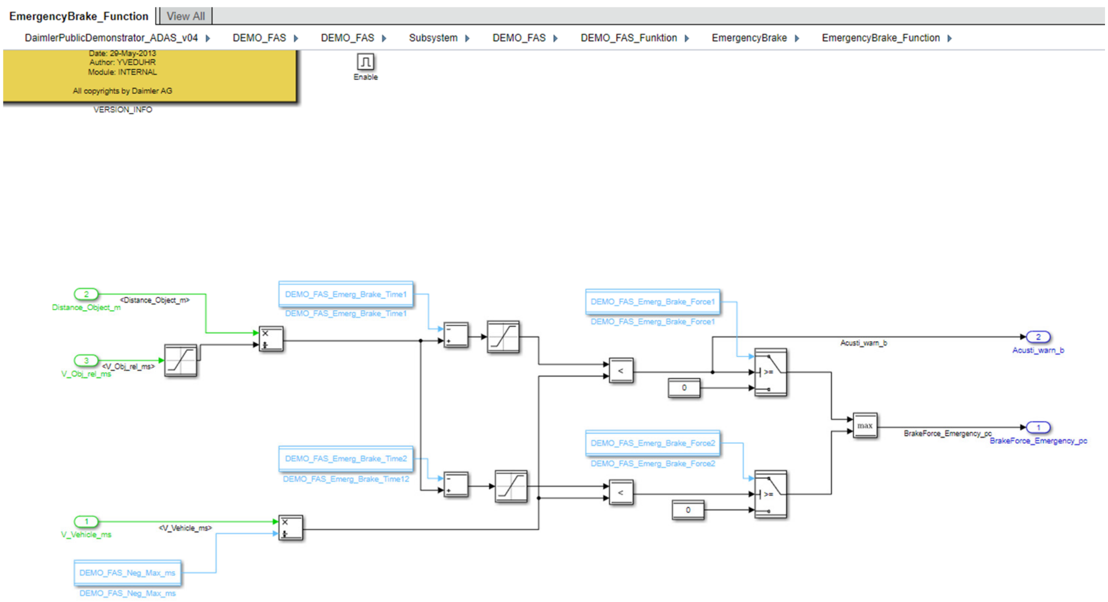


Figure A.1.: Screenshot of `EmergencyBrake_Function` subsystem in ADASv4 *Simulink* model.

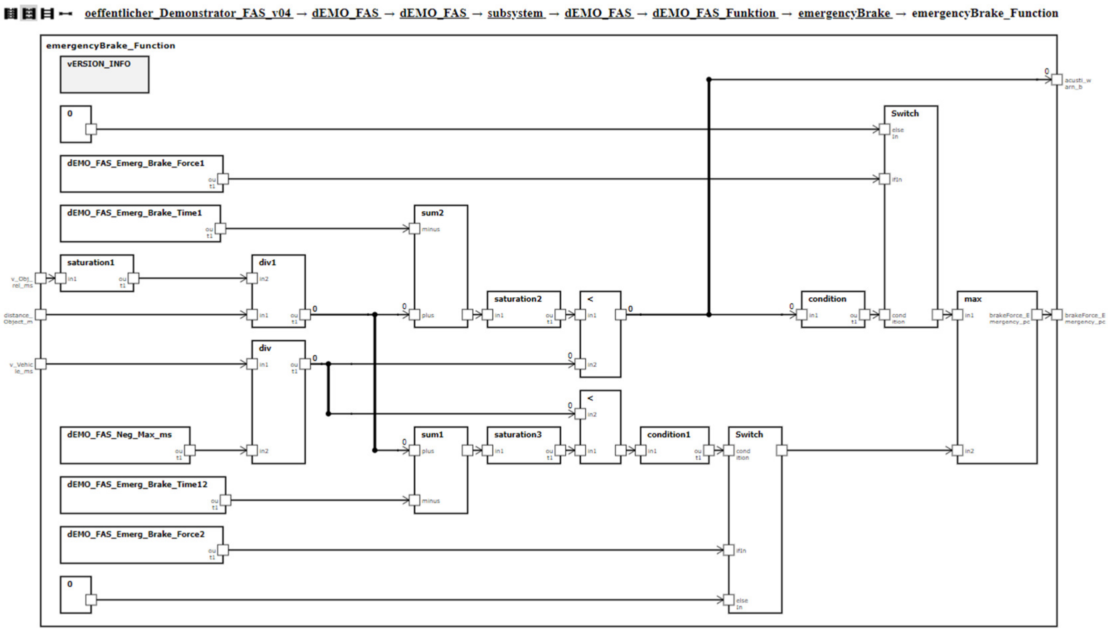


Figure A.2.: Screenshot of `EmergencyBrake_Function` hierarchy in the graphical representation of ADASv4 *EmbeddedMontiArc* model (normal view kind); equivalent representation to Figure A.1.

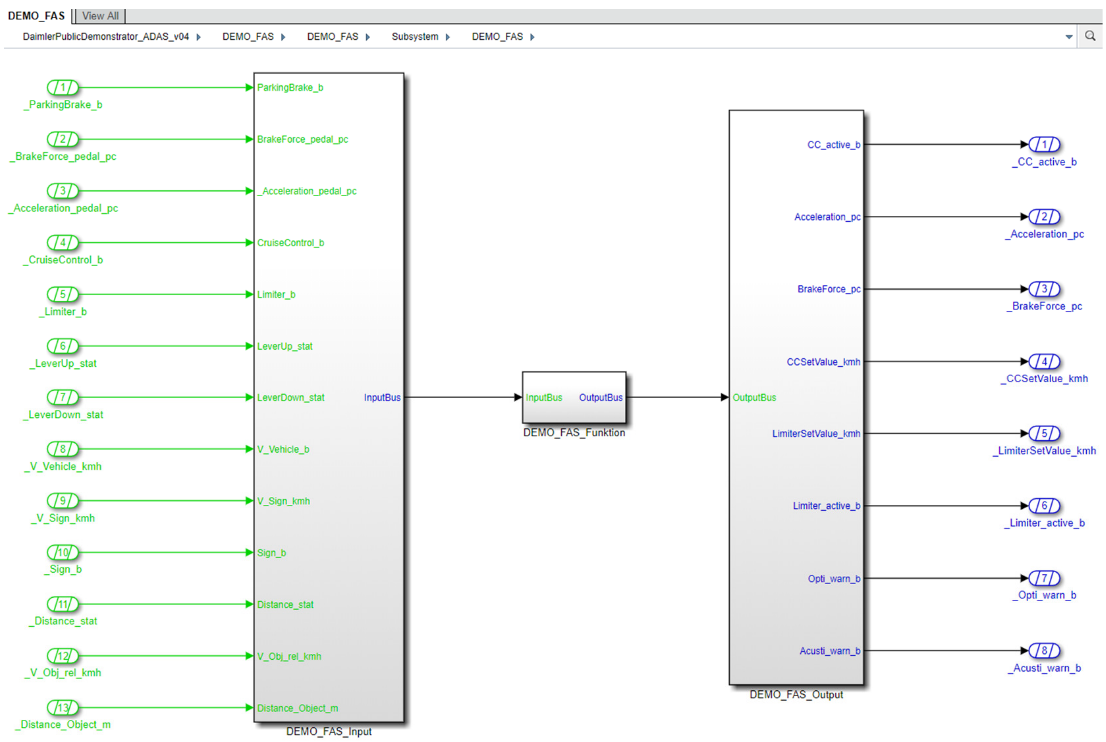


Figure A.3.: Screenshot of `DEMO_FAS` subsystem in ADASv4 *Simulink* model.

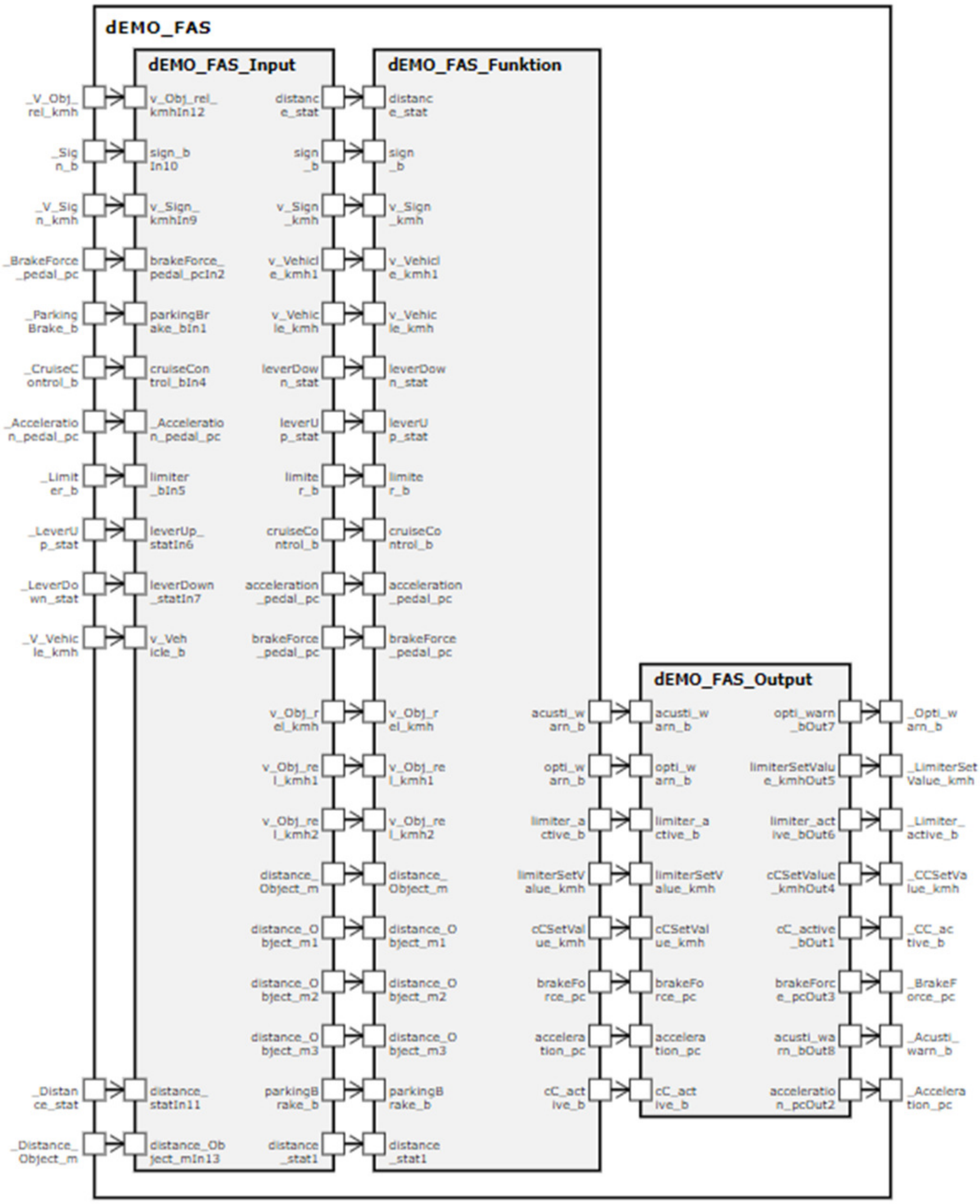


Figure A.4.: Screenshot of `DEMO_FAS` hierarchy in the graphical representation of ADASv4 *EmbeddedMontiArc* model (normal view kind); equivalent representation to Figure A.3.

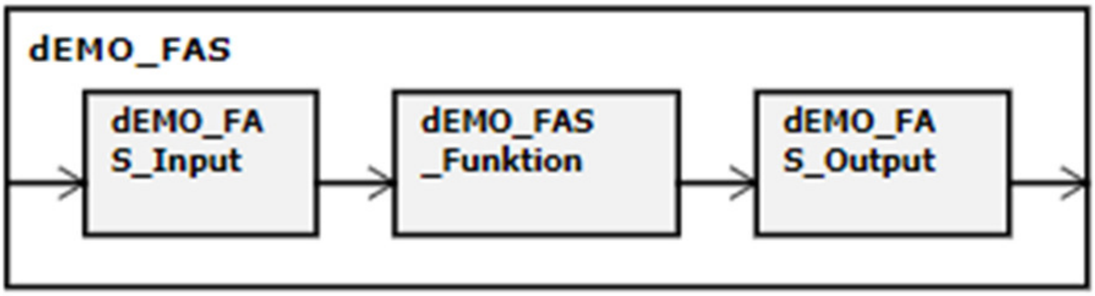


Figure A.5.: Simplified view of Figure A.4. It is much smaller to focus on the main component interactions.

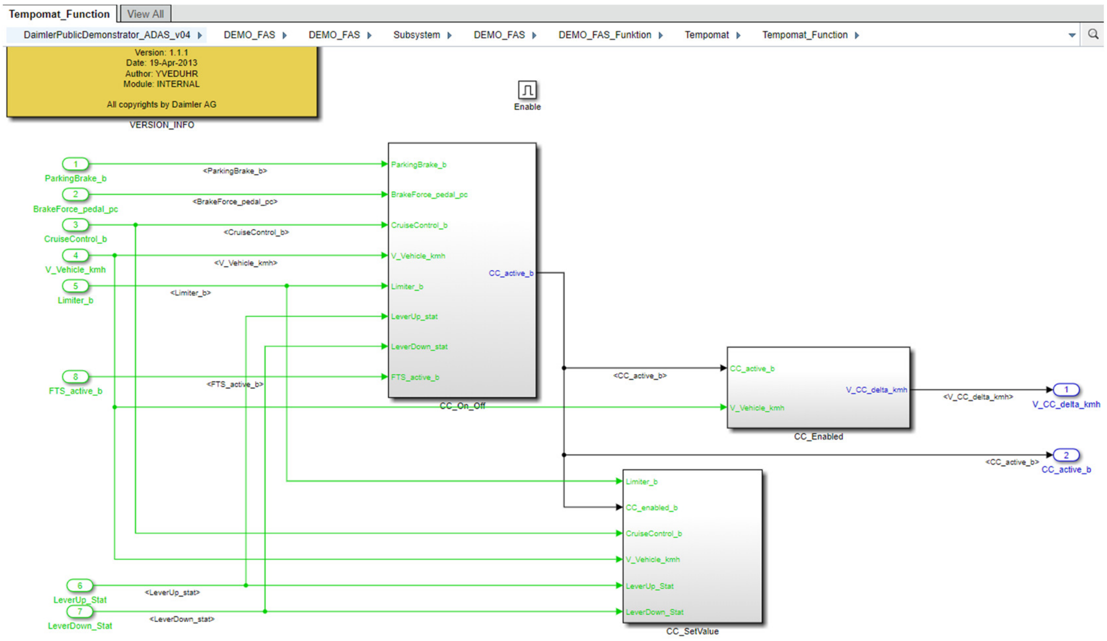


Figure A.6.: Screenshot of `Tempomat_Function` subsystem in ADASv4 *Simulink* model.

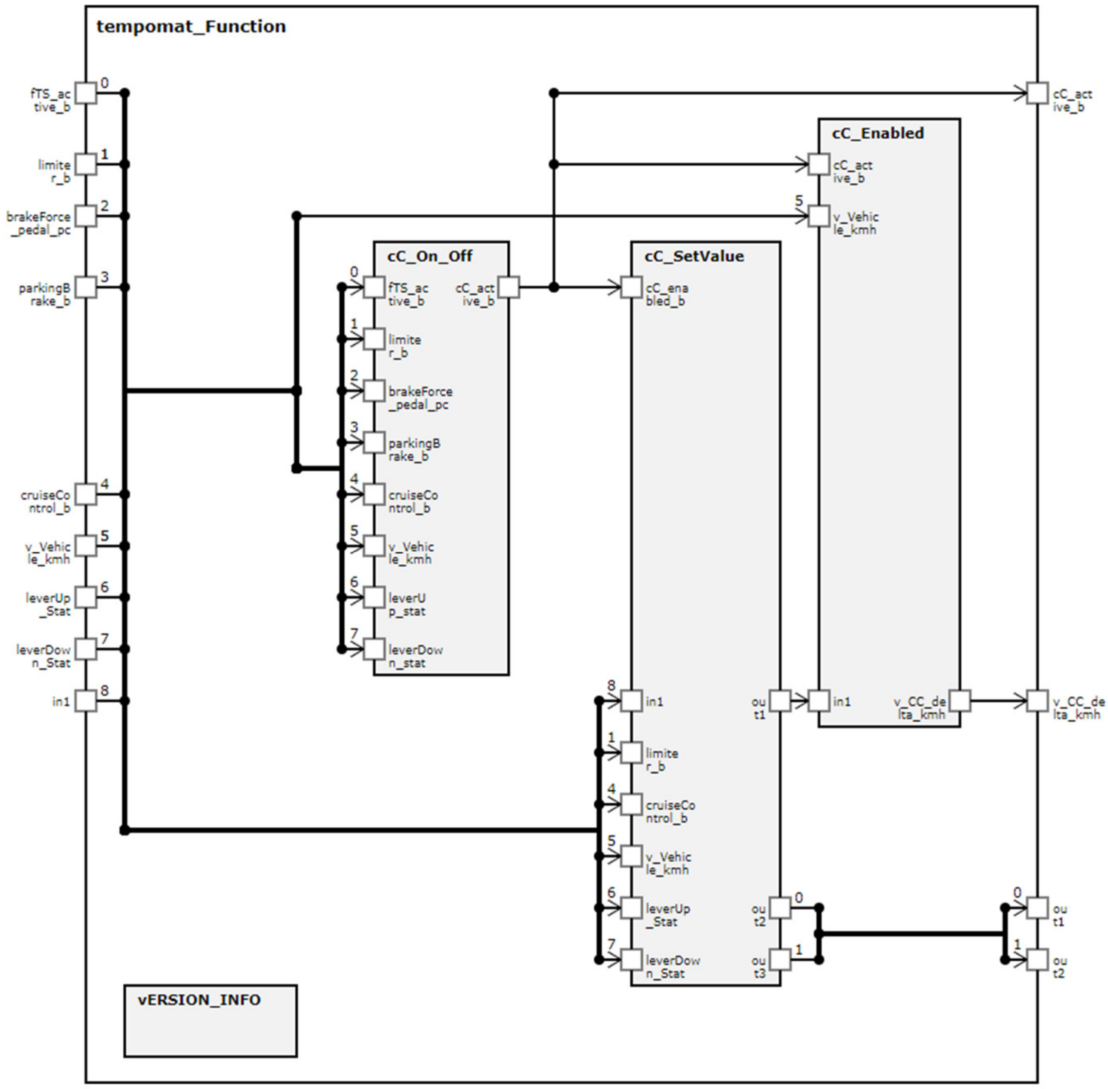


Figure A.7.: Screenshot of `Tempomat_Function` hierarchy in the graphical representation of ADASv4 *EmbeddedMontiArc* model (normal view kind); equivalent representation to Figure A.6.

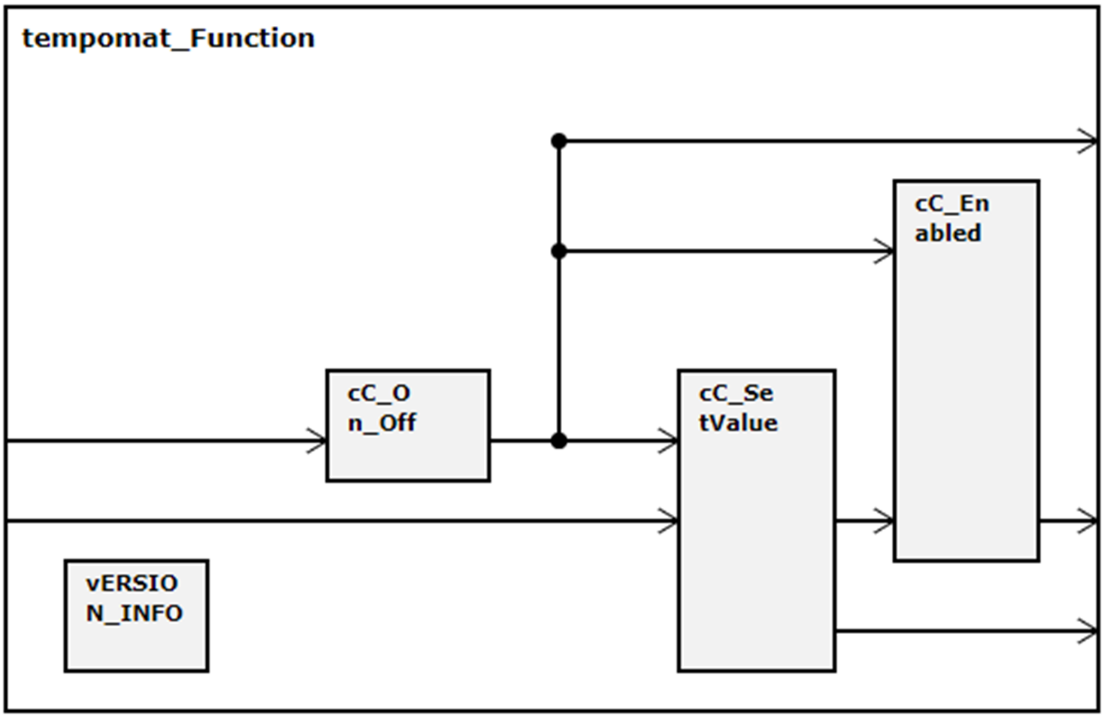


Figure A.8.: Simplified view of Figure A.7. It is much smaller to focus on the main component interactions.

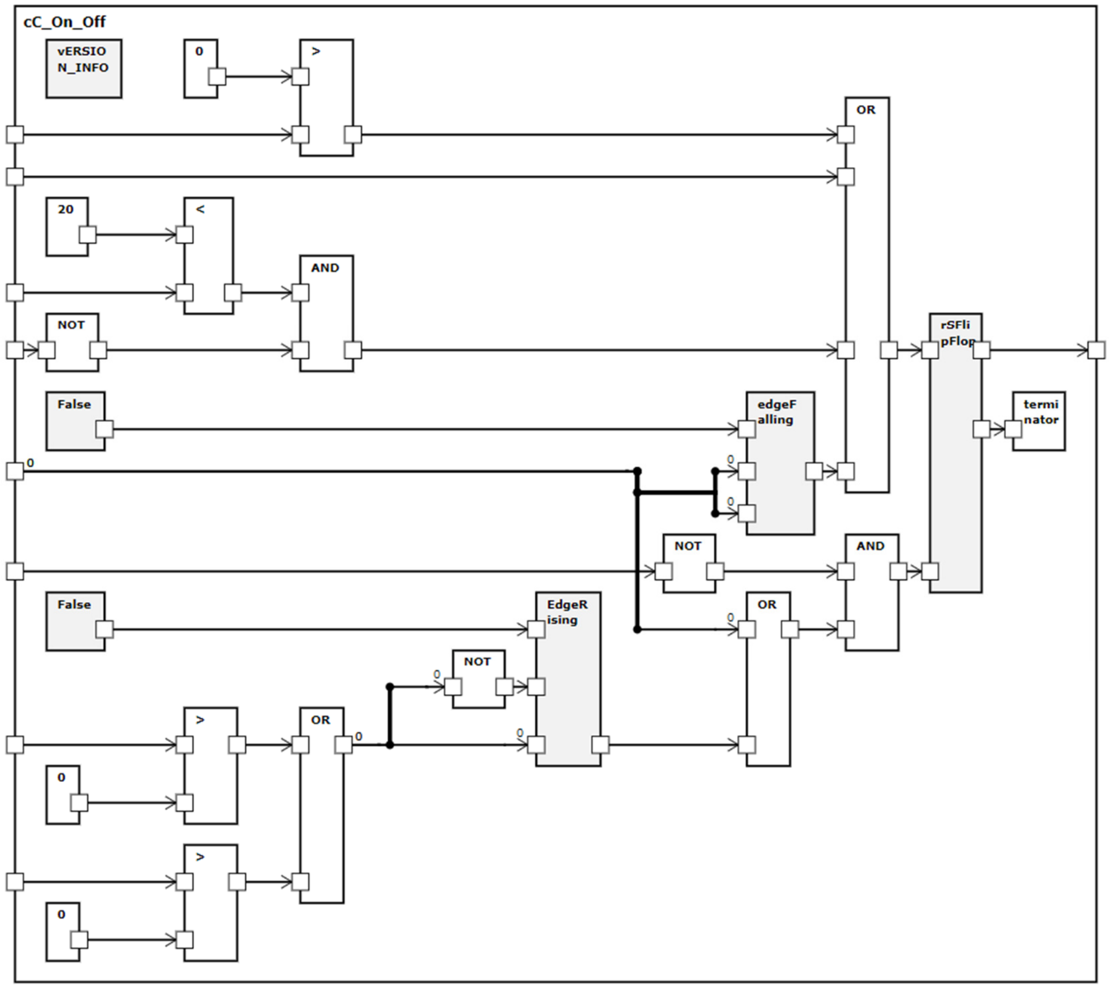


Figure A.9.: Screenshot showing the No Port Names (cf. requirement V28) representation of `CC_On_Off`. It skips the port names to focus on the component communication. Corresponding *Simulink* model is displayed in Figure 8.24.

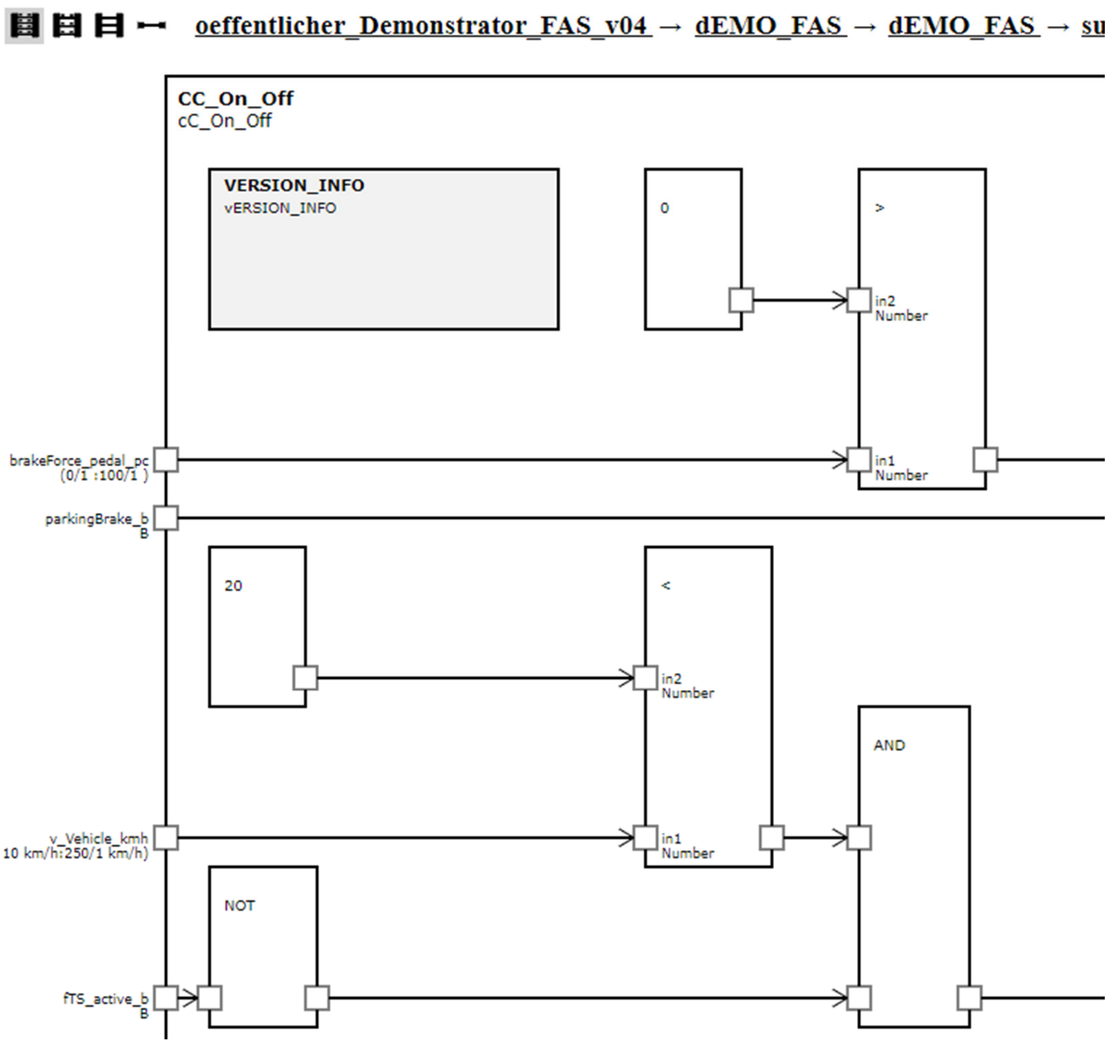
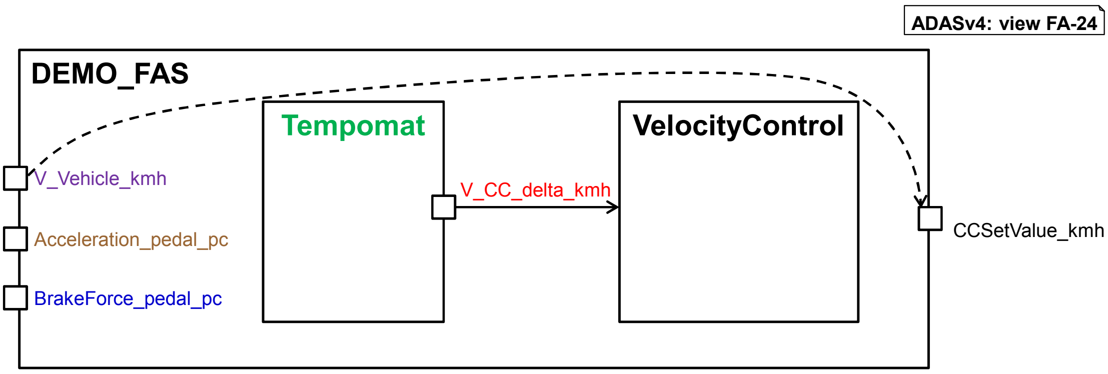


Figure A.10.: Excerption of a screenshot representing the full graphical information of the `CC_-On_Off` *EmbeddedMontiArc* models. It additionally shows the component type (cf. bold text in components) and the port data type (cf. second text line of ports). Corresponding *Simulink* model is displayed in Figure 8.24.

A.2. Screenshots of Graphical Representation of C&C Views, Satisfaction/Tracing Witnesses, and Highlighted Witnesses

The graphical representation of C&C views is manually created in PowerPoint. The graphical representation of all witnesses is generated.

A.2.1. FA-24



Translated Requirement FA-24

FA-24: As long as the cruise control is activated, the vehicle maintains the current vehicle speed of without the driver having to press the accelerator or the brake pedal.

Figure A.11.: C&C view of requirement FA-24 of ADASv4.

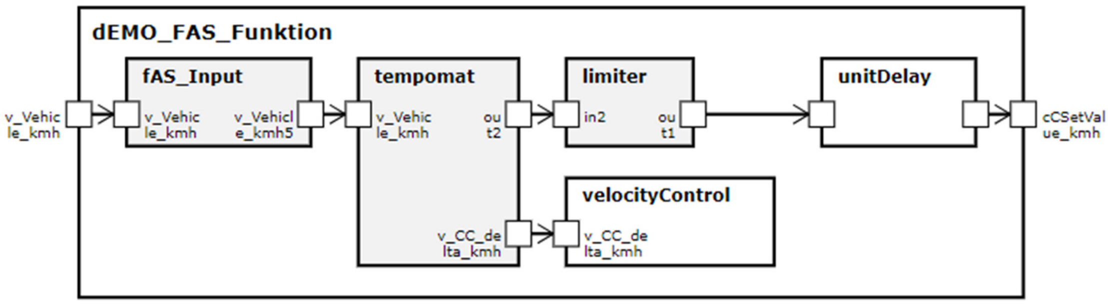


Figure A.12.: Generated graphical representation of `DEMO_FAS_Funktion` layer of satisfaction witness for C&C view of FA-24 shown in Figure A.11. The component type of the component instance `dEMO_FAS_Funktion` is `DEMO_FAS_Funktion`; just capitalize the first letter of the component instance.

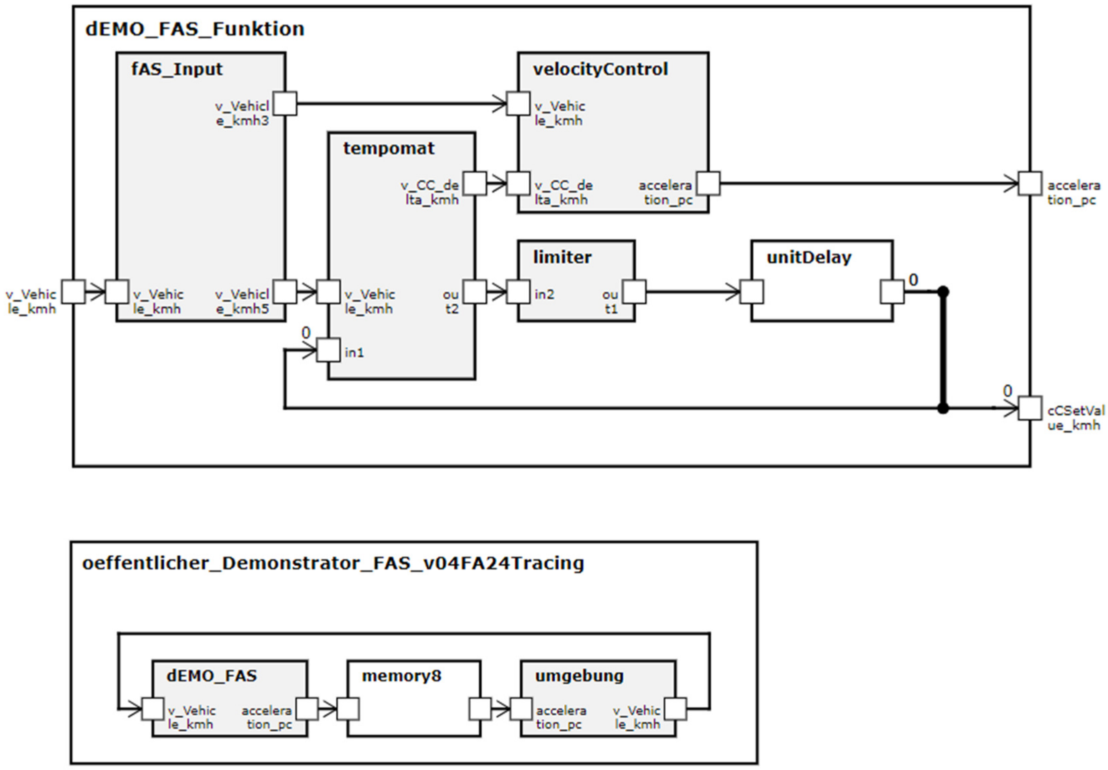


Figure A.13.: Generated graphical representation of `DEMO_FAS_Funktion` layer (top) and main component layer (bottom) of tracing witness for C&C view of FA-24 shown in Figure A.11. Tracing includes feedback over environment (German: Umgebung).

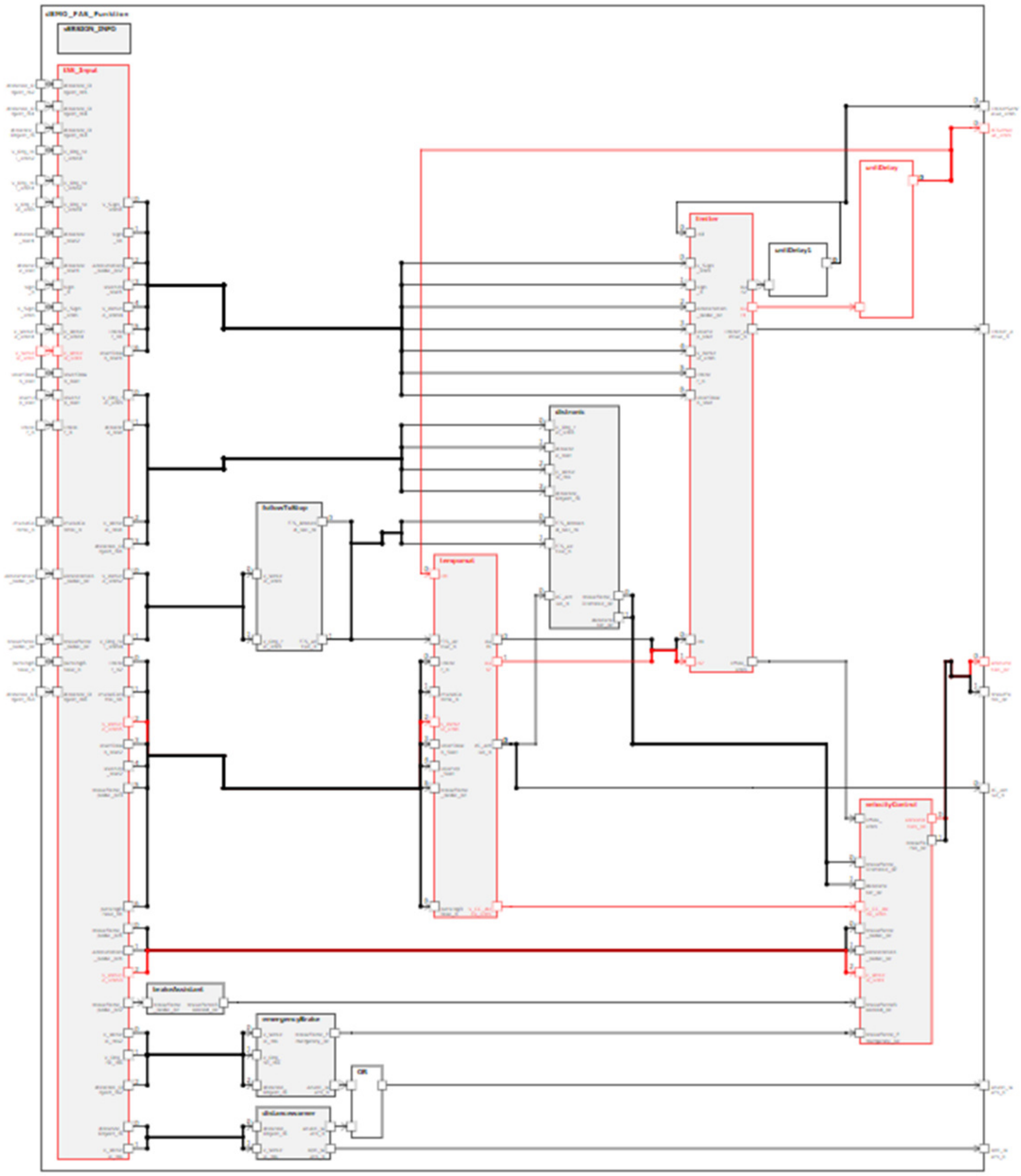


Figure A.14.: Generated graphical representation of `DEMO_FAS_Funktion` layer highlighting the tracing witness of FA-24 shown in the top part of Figure A.13.

A.2.2. FA-4

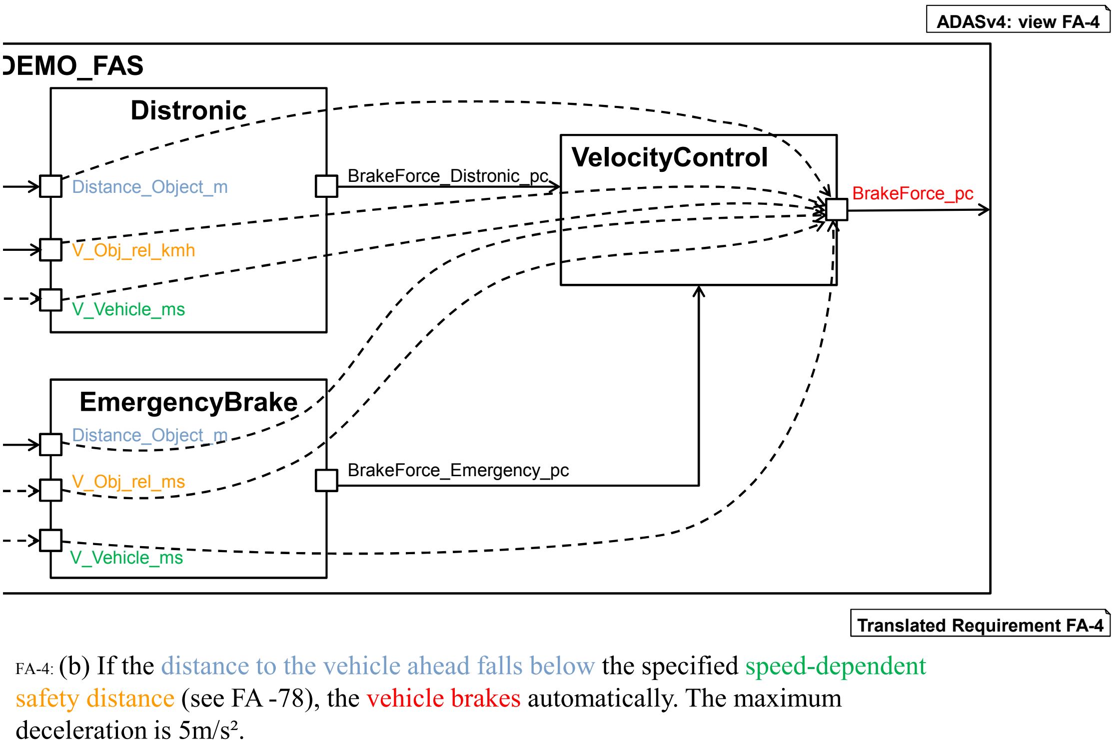


Figure A.15.: C&C view of requirement FA-4 of ADASv4.

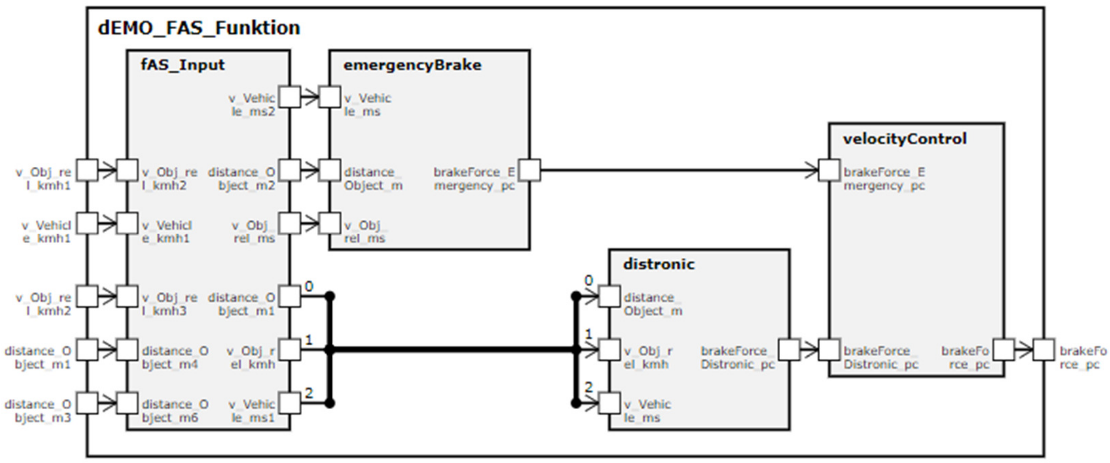


Figure A.16.: Generated graphical representation of `DEMO_FAS_Funktion` layer of satisfaction witness for C&C view of FA-4 shown in Figure A.15.

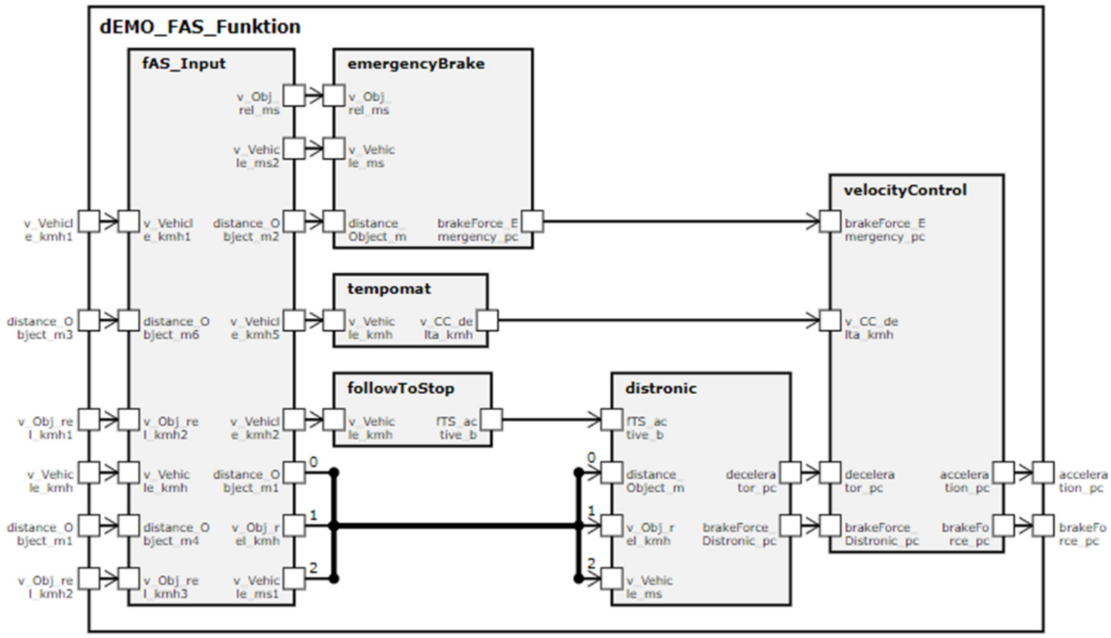
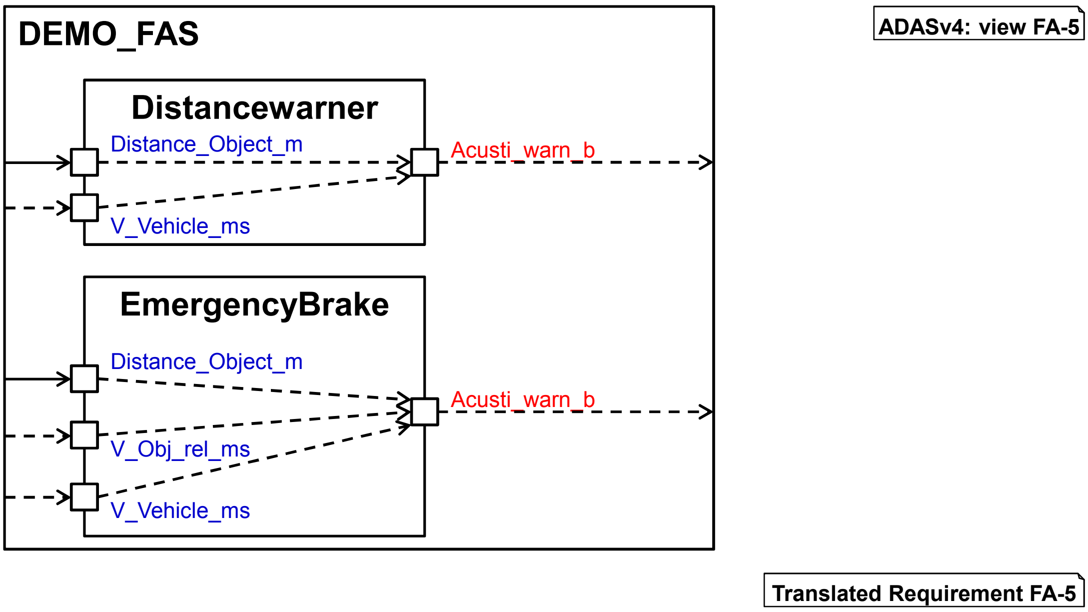


Figure A.17.: Generated graphical representation of `DEMO_FAS_Funktion` layer of tracing witness for C&C view of FA-4 shown in Figure A.15. Tracing includes feedback over environment (German: Umgebung).

A.2.3. FA-5



FA-5: (c) If the maximum deceleration of 5 m/s² is insufficient to prevent a collision with the vehicle ahead, the vehicle warns the driver by two acoustical signals (0.1 seconds long with 0.2 seconds pause between) and by this demands to intervene.

Figure A.18.: C&C view of requirement FA-5 of ADASv4.

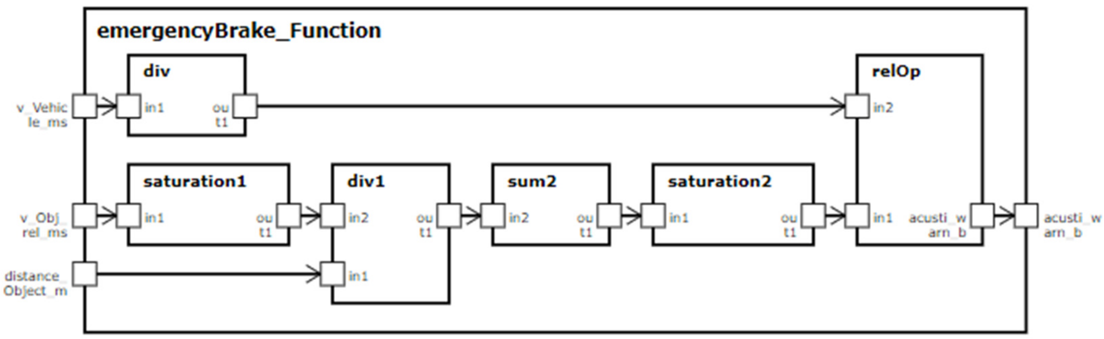


Figure A.19.: Generated graphical representation of `EmergencyBrake_Function` layer of satisfaction witness for C&C view of FA-5 shown in Figure A.18.

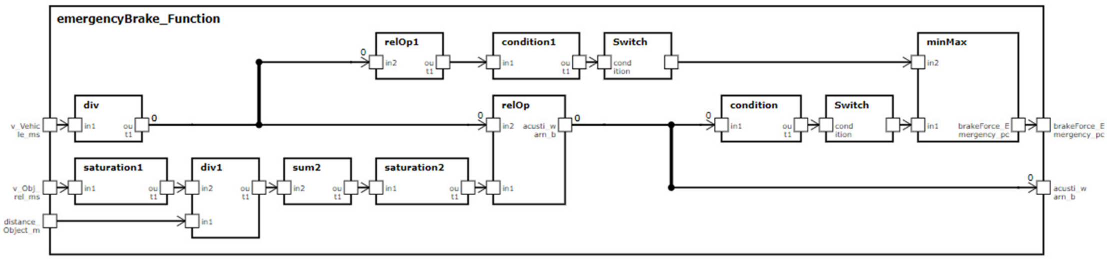


Figure A.20.: Generated graphical representation of `EmergencyBrake_Function` layer of tracing witness for C&C view of FA-5 shown in Figure A.18.

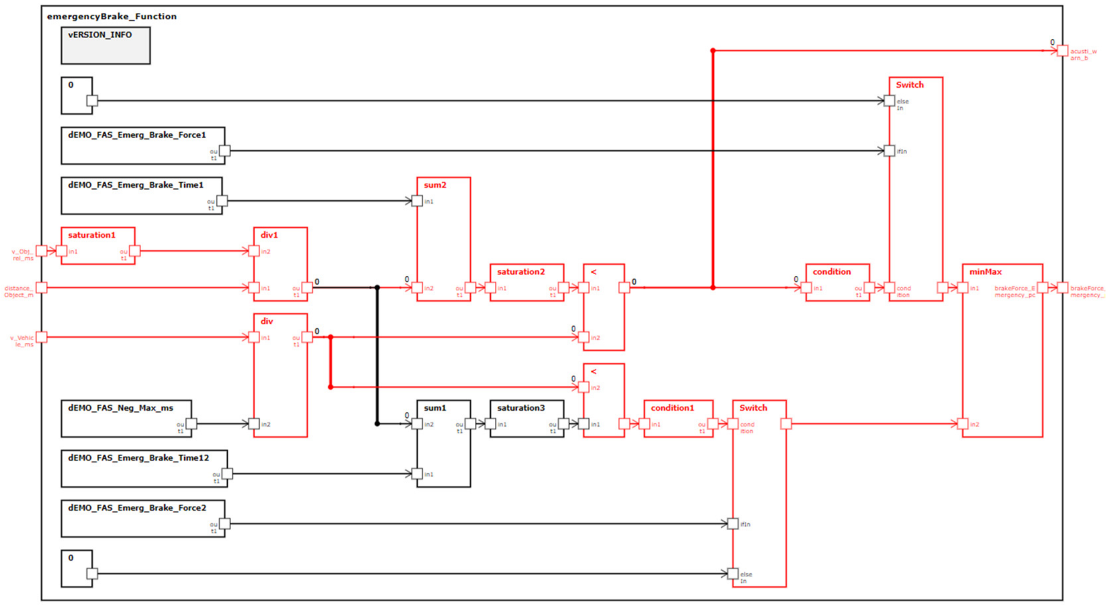
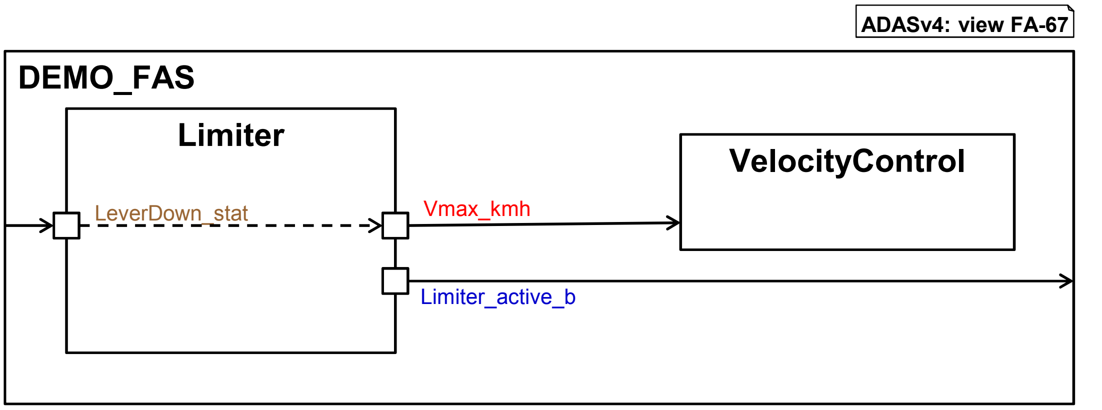


Figure A.21.: Generated graphical representation of `EmergencyBrake_Function` layer highlighting the tracing witness of FA-5 shown Figure A.20.

A.3. Identified Errors during Case Study

A.3.1. ADAS



FA-67: If the driver presses the speed limiting lever downwards within the first resistance stage and speed limit function is activated, the speed limit is decreased by N.
FA-68: If the driver presses the speed limiting lever downwards beyond the first resistance stage (i.e. beyond the pressure point) and speed limit function is activated, the speed limit is decreased to the next ten's place (e.g. starting speed limit 57 km/h → target speed limit 50 km/h).

Figure A.22.: C&C view (top) and translated requirement (bottom) of FA-76 of ADASv4.

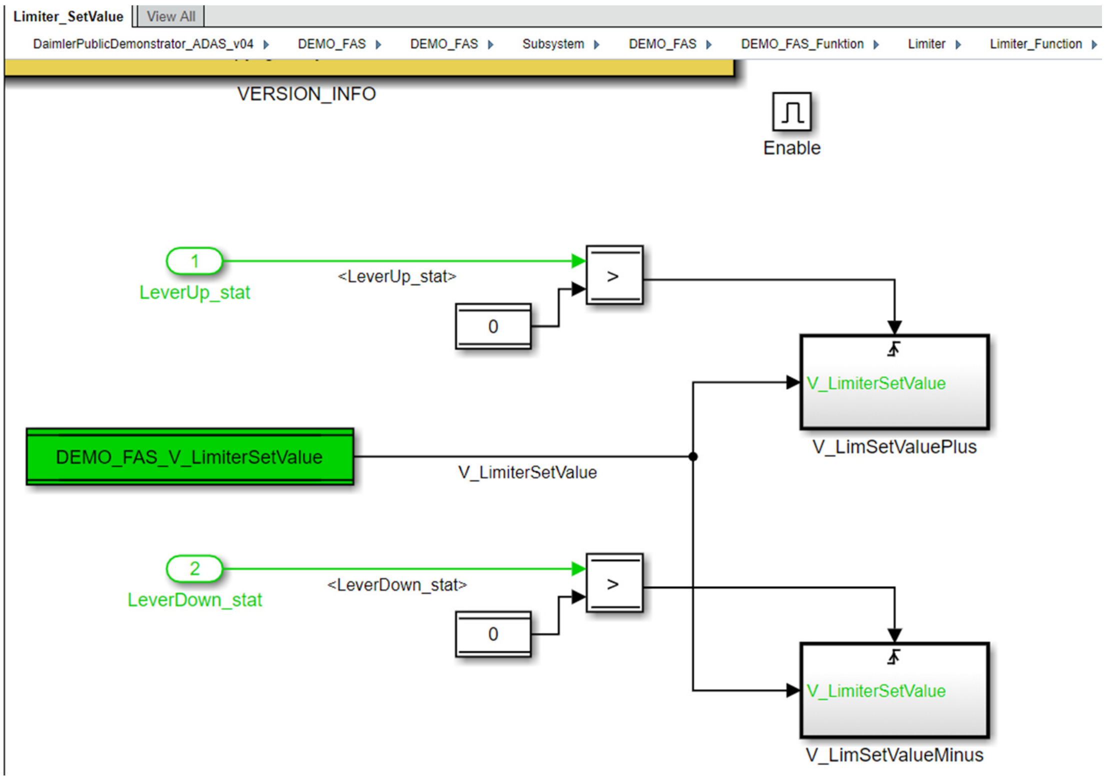


Figure A.23.: Screenshot of `Limiter_SetValue` *Simulink* subsystem to show that the limiter has not been updated when introducing the two-stage cruise control lever (cf. only one `SetValueMinus` subsystem for `FA-67`, but no extra subsystem for `FA-68`).

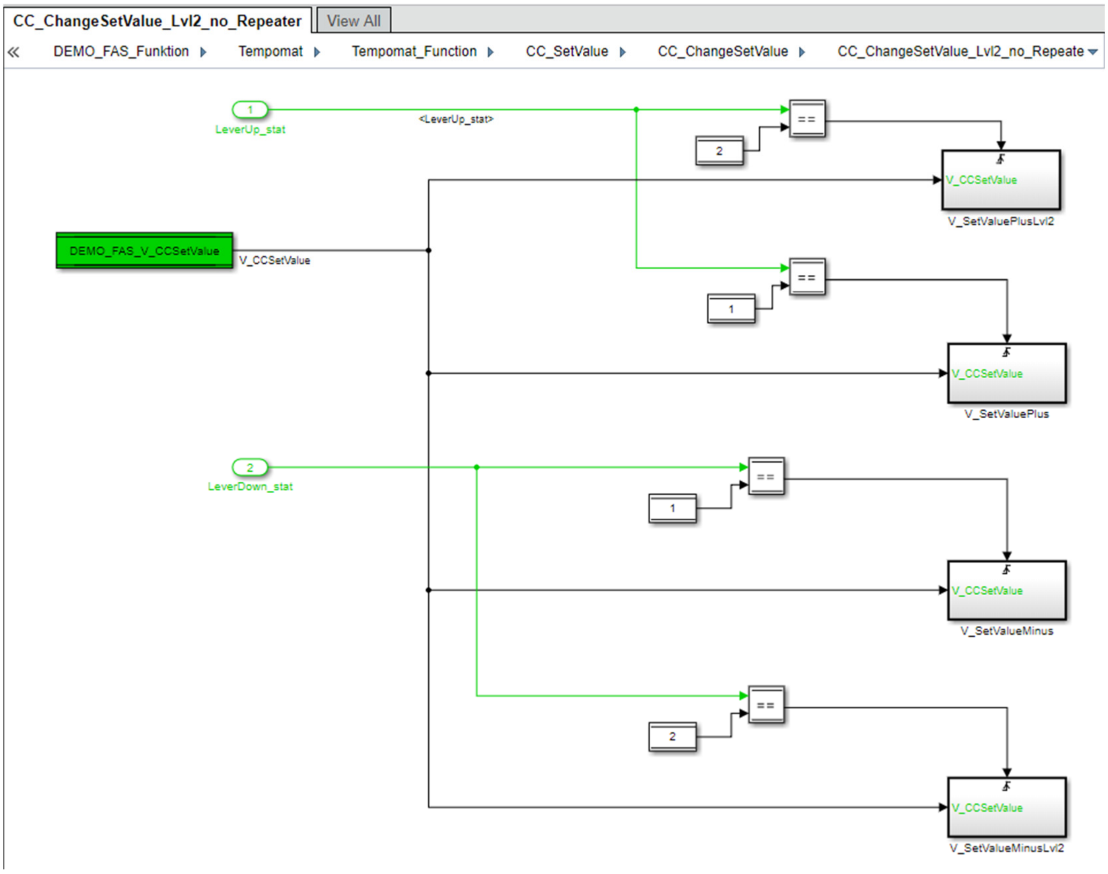


Figure A.24.: Screenshot of `CC_ChangeSetValue_Lvl2_no_Repeater` showing that tempomat has been updated to react on two-stage cruise control lever (cf. two subsystems `SetValueMinus` and `SetValueMinusLvl2` to decrease the value by N or to the next ten's place).

A.3.2. ALS

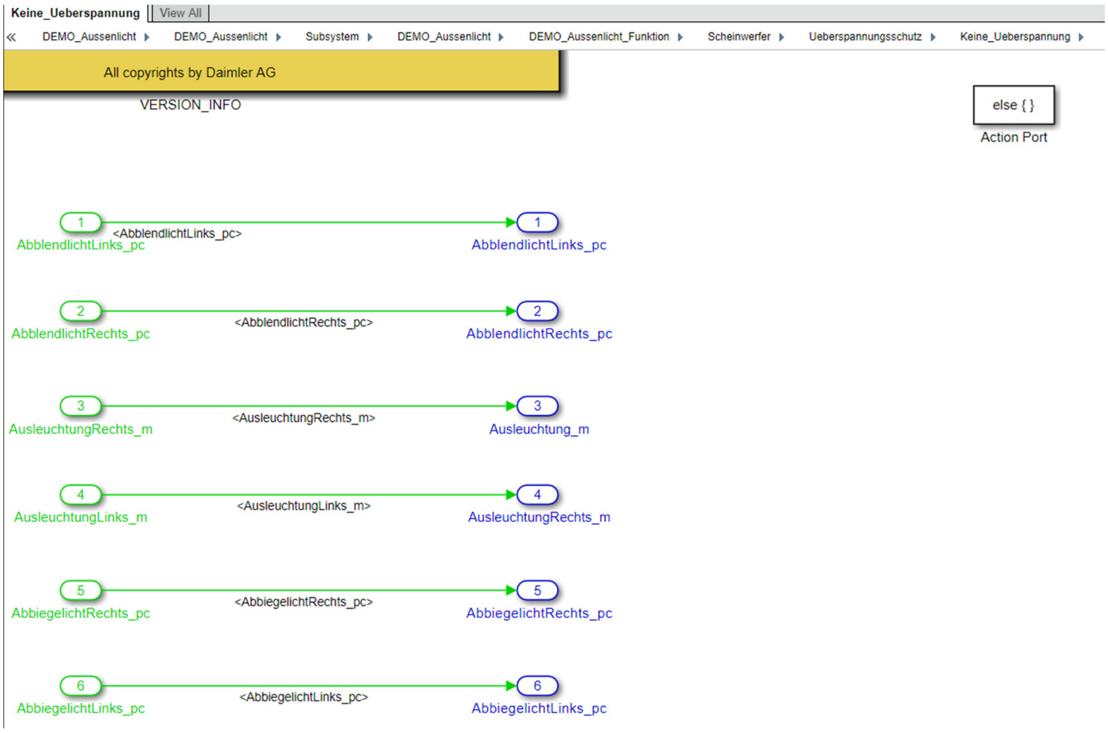


Figure A.25.: Screenshot of `No_Over_Voltage_Protection` (German: `Keine_Ueberspannung`) hierarchy in ALS *Simulink* model. The left ambient light inport (number 4) is connected with the right ambient light outport (also number 4).

A.4. Statistics about Running Time of Verification Tool

This section presents time measurements about the running time of the verification tool. Subsection A4.1 presents the time values for the verification tool only generating textual satisfaction or non-satisfaction witnesses. Subsection A4.2 lists the time values for the complete C&C views toolchain including the verification and generation of textual satisfaction and tracing witnesses, as well as the times for highlighting the textual witnesses in the graphical representation of the C&C model, and times for generating a graphical representation for the textual witnesses.
The measured times may vary when executing the experiment on the same hardware multiple times due to influences of the operating systems (e.g., when Windows starts background tasks). However, the time values are very good indicator for the speed of the C&C views verification toolchain applied on an industrial-size C&C models and on C&C views based on real-world requirements.

A.4.1. Running Time of Verification Tool Generating Textual (Non-)Satisfaction Witnesses

The measurements for the case study in 2017 were executed on a laptop with Windows 7 Professional and 4 cores plus hyperthreading.

ADAS version 1

	negative verification time	time to create all negative non-satasfaction Witnesses
Non-Satisfying Views Set v1		
Non-Satisfying Views Set v4		
FA15	214 651 142 ns	151 924 564 ns
FA22	75 192 753 ns	123 679 721 ns
FA23	72 515 288 ns	16 476 476 ns
FA25	71 263 115 ns	35 377 395 ns
FA26	51 887 733 ns	50 549 191 ns
FA3	67 194 220 ns	61 162 790 ns
FA4	48 949 636 ns	33 771 752 ns
FA5	3 034 360 ns	3 866 858 ns
FA6	24 875 658 ns	2 144 788 ns
FA65	80 260 418 ns	86 210 806 ns
FA67	82 986 206 ns	85 039 677 ns
FA75	1 224 778 ns	4 597 768 ns
FA77	141 342 545 ns	3 084 964 ns
FA84	39 853 787 ns	3 429 301 ns
FA86	1 351 859 ns	1 876 927 ns
FA99	1 518 892 ns	3 931 160 ns
Total time for Set v1 and v4	978 102 390 ns	667 124 138 ns
Average time for Non-Satisfaction	61 131 399 ns	41 695 259 ns
	61,13 ms	**41,70 ms**

Table A.26.: Negative verification time and to time create textual (non-)satisfaction witnesses of ADASv1.

	positive verification time	time to create one positive satisfaction Witness
Satisfying Views Set v1		
FA14	195 863 988 ns	34 083 749 ns
FA19	27 734 614 ns	26 476 734 ns
FA22	99 359 950 ns	10 094 619 ns
FA23	65 071 119 ns	31 149 457 ns
FA24	66 861 677 ns	14 321 414 ns
FA25	69 930 660 ns	21 510 278 ns
FA26	13 740 416 ns	16 866 852 ns
FA27	54 928 180 ns	15 015 798 ns
FA28	106 497 067 ns	29 039 674 ns
FA29	87 832 429 ns	27 316 843 ns
FA30	86 955 414 ns	21 063 209 ns
FA33	5 297 858 ns	15 179 787 ns
FA34	63 692 246 ns	96 834 298 ns
FA35	63 318 992 ns	29 512 615 ns
FA36	62 849 855 ns	26 889 940 ns
FA37	4 113 031 ns	21 370 260 ns
FA38	74 604 144 ns	25 108 894 ns
Satisfying Views Set v4		
FA19	25 497 368 ns	32 920 230 ns
FA20	116 344 371 ns	11 562 145 ns
FA21	78 295 600 ns	22 479 370 ns
FA24	74 878 854 ns	52 393 395 ns
FA27	76 756 161 ns	20 273 325 ns
FA28	20 979 503 ns	21 447 878 ns
FA30	4 334 854 ns	14 906 218 ns
FA31	55 162 177 ns	14 867 029 ns
FA32	62 788 977 ns	64 877 453 ns
FA35	4 371 761 ns	17 167 054 ns
Total time for Set v1 and v4	1 668 061 266 ns	734 728 518 ns
Average time for Satisfaction	61 780 047 ns	27 212 167 ns
	61,78 ms	**27,21 ms**

Table A.27.: Positive time and to time create textual (non-)satisfaction witnesses of ADASv1.

ADAS version 2

	positive verification time	time to create one positive satisfaction Witness
Satisfying Views Set v1		
FA14	3142729789 ns	183291277 ns
FA25	1803037175 ns	514979271 ns
FA27	2187723534 ns	219687987 ns
FA30	2111294589 ns	716196668 ns
FA38	1323611843 ns	250141159 ns
Satisfying Views Set v4		
FA23	2286551187 ns	1523150544 ns
FA24	1942190415 ns	1453652874 ns
FA27	1943854271 ns	410080236 ns
FA31	1332652901 ns	384827899 ns
FA75	12525530 ns	499375246 ns
Total time for Set v1 and v4	18 086 171 234 ns	6 155 383 161 ns
Average time for Satisfaction	1 808 617 123 ns	615 538 316 ns
	1.808,62 ms	**615,54 ms**

Table A.28.: Positive time and to time create textual (non-)satisfaction witnesses of ADASv2.

	negative verification time	time to create all negative non-satasfaction Witnesses
Non-Satisfying Views Set v1		
FA19	296 804 468 ns	12 010 391 016 ns
FA22	1 871 029 270 ns	6 322 232 589 ns
FA23	2 224 769 731 ns	12 137 403 014 ns
FA24	1 837 631 914 ns	6 439 046 858 ns
FA26	218 727 266 ns	6 649 886 311 ns
FA28	1 785 386 146 ns	6 642 069 269 ns
FA29	1 604 410 874 ns	3 590 740 467 ns
FA33	29 440 704 ns	4 352 925 845 ns
FA34	960 024 077 ns	4 932 671 849 ns
FA35	1 092 647 073 ns	4 986 885 480 ns
FA36	937 912 254 ns	2 683 010 480 ns
FA37	29 060 981 ns	4 940 373 986 ns
Non-Satisfying Views Set v4		
FA15	5 106 570 120 ns	2 697 026 366 ns
FA19	291 553 028 ns	8 704 530 651 ns
FA20	1 607 261 079 ns	6 177 247 290 ns
FA21	1 925 449 123 ns	6 235 629 503 ns
FA22	1 917 484 453 ns	11 075 575 900 ns
FA25	2 099 563 509 ns	16 477 961 992 ns
FA26	2 144 198 838 ns	15 700 983 779 ns
FA28	249 046 507 ns	11 732 586 773 ns
FA3	2 153 140 209 ns	10 983 738 498 ns
FA30	31 836 611 ns	8 724 166 664 ns
FA32	1 193 269 468 ns	4 594 818 113 ns
FA35	32 525 667 ns	8 684 050 343 ns
FA4	1 059 863 056 ns	6 658 849 ns
FA5	12 455 521 ns	3 689 552 ns
FA6	238 179 506 ns	2 035 209 ns
FA65	1 302 069 985 ns	13 387 441 824 ns
FA67	1 165 265 091 ns	7 091 391 095 ns
FA77	2 281 286 812 ns	4 633 801 734 ns
FA84	1 023 672 187 ns	2 045 862 ns
FA86	7 977 987 ns	2 625 720 ns
FA99	7 198 375 ns	2 079 344 ns
Total time for Set v1 and v4	38 737 711 890 ns	212 607 722 225 ns
Average time for Non-Satisfaction	1 173 870 057 ns	6 442 658 249 ns
	1.173,87 ms	**6.442,66 ms**

Table A.29.: Negative verification time and to time create textual (non-)satisfaction witnesses of ADASv2.

ADAS version 3

	positive verification time	time to create one positive satisfaction Witness
Satisfying Views Set v1		
FA14	2 799 473 563 ns	180 125 651 ns
FA25	1 612 764 018 ns	383 784 612 ns
FA27	1 990 565 521 ns	201 657 236 ns
FA30	1 559 489 421 ns	450 883 331 ns
FA38	1 183 900 051 ns	210 071 637 ns
Satisfying Views Set v4		
FA23	2 201 448 350 ns	867 542 896 ns
FA24	2 030 827 948 ns	2 633 941 372 ns
FA27	1 817 309 887 ns	450 453 765 ns
FA31	1 302 620 165 ns	353 799 817 ns
FA6	969 451 707 ns	631 071 001 ns
FA75	11 312 547 ns	275 801 755 ns
FA86	34 760 249 ns	153 027 586 ns
Total time for Set v1 and v4	17 513 923 427 ns	6 792 160 659 ns
Average time for Satisfaction	1 459 493 619 ns	566 013 388 ns
	1.459,49 ms	**566,01 ms**

Table A.30.: Positive time and to time create textual (non-)satisfaction witnesses of ADASv3.

	negative verification time	time to create all negative non-satasfaction Witnesses
Non-Satisfying Views Set v1		
FA19	346 088 549 ns	9 219 592 867 ns
FA22	1 811 755 964 ns	5 336 977 482 ns
FA23	1 631 465 183 ns	11 928 356 401 ns
FA24	1 667 889 288 ns	4 171 003 549 ns
FA26	246 795 184 ns	5 591 848 817 ns
FA28	1 696 936 191 ns	5 639 255 210 ns
FA29	1 691 816 780 ns	5 197 700 203 ns
FA33	56 535 344 ns	4 752 727 309 ns
FA34	1 002 402 376 ns	2 663 982 480 ns
FA35	1 143 568 376 ns	6 861 807 862 ns
FA36	1 109 897 833 ns	3 800 018 035 ns
FA37	55 627 129 ns	4 811 308 895 ns
Non-Satisfying Views Set v4		
FA15	6 775 152 270 ns	22 387 674 ns
FA19	232 850 068 ns	7 901 431 837 ns
FA20	1 932 733 868 ns	5 376 298 212 ns
FA21	2 138 170 451 ns	9 666 516 121 ns
FA22	2 125 506 804 ns	9 791 245 598 ns
FA25	2 017 120 634 ns	5 136 009 683 ns
FA26	2 089 346 374 ns	9 380 971 314 ns
FA28	306 923 057 ns	9 505 684 049 ns
FA3	1 594 087 585 ns	8 637 332 244 ns
FA30	50 118 102 ns	8 104 505 233 ns
FA32	1 361 075 811 ns	8 245 492 027 ns
FA35	64 710 041 ns	8 145 681 204 ns
FA4	1 034 351 229 ns	3 177 422 ns
FA5	38 338 701 ns	1 829 366 ns
FA65	1 314 264 494 ns	8 316 544 734 ns
FA67	1 268 583 215 ns	8 309 564 376 ns
FA77	2 461 610 695 ns	7 238 676 798 ns
FA84	1 106 678 178 ns	2 457 165 ns
FA99	5 952 671 ns	2 159 627 ns
Total time for Set v1 and v4	40 378 352 445 ns	183 762 543 794 ns
Average time for Non-Satisfaction	1 302 527 498 ns	5 927 823 993 ns
	1.302,53 ms	**5.927,82 ms**

Table A.31.: Negative verification time and to time create textual (non-)satisfaction witnesses of ADASv3.

ADAS version 4

	negative verification time	time to create all negative non-satasfaction Witnesses
Non-Satisfying Views Set v1		
FA23	674 355 613 ns	1 219 144 657 ns
FA24	665 441 255 ns	1 268 736 931 ns
FA29	526 630 449 ns	1 213 924 037 ns
FA35	332 354 982 ns	551 305 973 ns
FA36	332 523 917 ns	560 949 338 ns
Non-Satisfying Views Set v4		
Total time for Set v1 and v4	2 531 306 216 ns	4 814 060 936 ns
Average time for Non-Satisfaction	506 261 243 ns	962 812 187 ns
	506,26 ms	**962,81 ms**

Table A.32.: Negative verification time and to time create textual (non-)satisfaction witnesses of ADASv4.

	positive verification time	time to create one positive satisfaction Witness
Satisfying Views Set v1		
FA14	750 137 735 ns	83 790 168 ns
FA19	101 756 998 ns	102 683 096 ns
FA22	493 050 081 ns	38 427 353 ns
FA25	615 667 871 ns	124 281 266 ns
FA26	88 579 318 ns	108 265 176 ns
FA27	519 148 994 ns	94 486 332 ns
FA28	537 741 722 ns	134 598 088 ns
FA30	908 990 151 ns	175 116 961 ns
FA33	75 323 259 ns	91 646 019 ns
FA34	606 336 503 ns	307 998 685 ns
FA37	38 877 466 ns	65 917 697 ns
FA38	345 615 988 ns	68 773 228 ns
Satisfying Views Set v4		
FA15	2 479 638 783 ns	103 626 316 ns
FA19	99 658 249 ns	87 273 498 ns
FA20	441 415 750 ns	61 662 746 ns
FA21	389 932 852 ns	51 534 263 ns
FA22	471 465 229 ns	109 261 283 ns
FA23	447 579 590 ns	157 227 748 ns
FA24	514 867 028 ns	488 289 847 ns
FA25	591 947 363 ns	122 951 855 ns
FA26	605 106 779 ns	120 453 979 ns
FA27	462 655 123 ns	84 400 845 ns
FA28	85 422 444 ns	91 928 719 ns
FA3	515 579 675 ns	118 809 146 ns
FA30	37 260 789 ns	68 875 579 ns
FA31	338 551 924 ns	68 851 989 ns
FA32	337 564 188 ns	192 374 189 ns
FA35	38 049 152 ns	66 391 780 ns
FA4	389 051 652 ns	104 377 391 ns
FA5	79 931 300 ns	70 942 748 ns
FA6	288 750 384 ns	180 870 638 ns
FA65	336 929 159 ns	86 338 269 ns
FA67	317 479 964 ns	55 333 776 ns
FA75	5 914 241 ns	77 955 447 ns
FA77	637 599 344 ns	100 381 169 ns
FA84	301 520 566 ns	83 706 842 ns
FA86	22 324 514 ns	66 766 556 ns
FA99	36 093 465 ns	134 107 263 ns
Total time for Set v1 and v4	15 353 515 593 ns	4 350 677 950 ns
Average time for Satisfaction	404 039 884 ns	114 491 525 ns
	404,04 ms	**114,49 ms**

Table A.33.: Positive time and to time create textual (non-)satisfaction witnesses of ADASv4.

A.4.2. Running Time of Toolchain Including Verification and Generation of Graphical Representation

The satisfaction time addressed in columns in this subsection represents the time for executing the verification and the time for generating the textual witness. The values in this column in this subsection represents the sum of the values of the columns positive/negative verification time plus time to create (non-)satisfaction witnesses in the previous subsection A4.1. The values may also differ, as the two case study in the beginning of 2017 and at the end of 2018 are executed on different hardware. However, the changes are not dramatically and below one second, so that the developer executing these tools will not notice any difference.

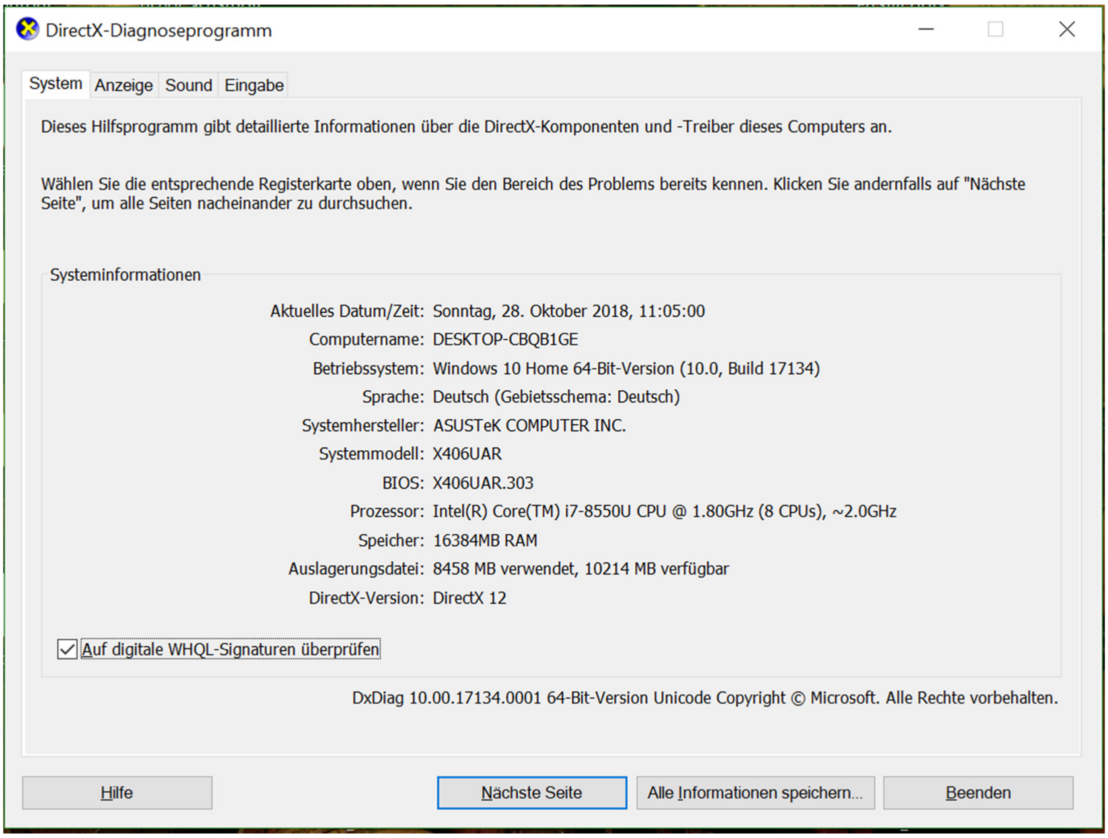


Figure A.34.: Screenshot of PC configuration executing the measurements for the C&C views verification toolchain including generation of graphical representations.

Times for Satisfaction Witnesses of ADASv1

FA-	Satisfaction	Coloring	Layouting	Tooling Option 1 (Satisfaction + Coloring)	Tooling Option 2 (Satisfaction + Layouting)
14	217,00 ms	480,00 ms	1 312,00 ms	697,00 ms	1 529,00 ms
19	192,00 ms	1 312,00 ms	5 344,00 ms	1 504,00 ms	5 536,00 ms
22	158,00 ms	753,00 ms	3 846,00 ms	911,00 ms	4 004,00 ms
23	190,00 ms	2 211,00 ms	7 009,00 ms	2 401,00 ms	7 199,00 ms
24	148,00 ms	1 817,00 ms	6 842,00 ms	1 965,00 ms	6 990,00 ms
25	148,00 ms	1 582,00 ms	5 245,00 ms	1 730,00 ms	5 393,00 ms
26	81,00 ms	2 402,00 ms	5 591,00 ms	2 483,00 ms	5 672,00 ms
27	118,00 ms	1 802,00 ms	5 792,00 ms	1 920,00 ms	5 910,00 ms
28	141,00 ms	2 247,00 ms	8 402,00 ms	2 388,00 ms	8 543,00 ms
29	136,00 ms	4 469,00 ms	5 711,00 ms	4 605,00 ms	5 847,00 ms
30	92,00 ms	739,00 ms	2 974,00 ms	831,00 ms	3 066,00 ms
33	80,00 ms	1 613,00 ms	3 790,00 ms	1 693,00 ms	3 870,00 ms
34	101,00 ms	1 298,00 ms	5 136,00 ms	1 399,00 ms	5 237,00 ms
35	122,00 ms	2 611,00 ms	7 570,00 ms	2 733,00 ms	7 692,00 ms
36	96,00 ms	2 190,00 ms	8 252,00 ms	2 286,00 ms	8 348,00 ms
37	119,00 ms	874,00 ms	4 507,00 ms	993,00 ms	4 626,00 ms
38	82,00 ms	1 292,00 ms	8 012,00 ms	1 374,00 ms	8 094,00 ms
Average	130,65 ms	1 746,59 ms	5 607,94 ms	**1 877,24 ms**	**5 738,59 ms**

Table A.35.: Time measurement of Running Time of Satisfaction Verification Toolchain; evaluated on ADASv1.

Times for Tracing Witnesses of ADASv1

FA-	Satisfaction	Coloring	Layouting	Tooling Option 1 (Satisfaction + Coloring)	Tooling Option 2 (Satisfaction + Layouting)
14	192,00 ms	307,00 ms	1 487,00 ms	499,00 ms	1 679,00 ms
19	138,00 ms	3 004,00 ms	10 668,00 ms	3 142,00 ms 10	806,00 ms
22	126,00 ms	794,00 ms	3 025,00 ms	920,00 ms	3 151,00 ms
23	119,00 ms	3 761,00 ms	12 775,00 ms	3 880,00 ms	12 894,00 ms
24	119,00 ms	5 912,00 ms	13 773,00 ms	6 031,00 ms	13 892,00 ms
25	106,00 ms	1 270,00 ms	5 848,00 ms	1 376,00 ms	5 954,00 ms
26	113,00 ms	2 954,00 ms	12 778,00 ms	3 067,00 ms	12 891,00 ms
27	138,00 ms	2 711,00 ms	11 723,00 ms	2 849,00 ms	11 861,00 ms
28	118,00 ms	6 069,00 ms	16 356,00 ms	6 187,00 ms	16 474,00 ms
29	99,00 ms	2 833,00 ms	10 291,00 ms	2 932,00 ms	10 390,00 ms
30	84,00 ms	602,00 ms	3 170,00 ms	686,00 ms	3 254,00 ms
33	76,00 ms	1 015,00 ms	4 243,00 ms	1 091,00 ms	4 319,00 ms
34	96,00 ms	2 881,00 ms	9 564,00 ms	2 977,00 ms	9 660,00 ms
35	107,00 ms	4 053,00 ms	12 076,00 ms	4 160,00 ms	12 183,00 ms
36	95,00 ms	3 820,00 ms	14 466,00 ms	3 915,00 ms	14 561,00 ms
37	65,00 ms	1 004,00 ms	4 297,00 ms	1 069,00 ms	4 362,00 ms
38	84,00 ms	2 118,00 ms	7 697,00 ms	2 202,00 ms	7 781,00 ms
Average	110,29 ms	2 653,41 ms	9 072,76 ms	**2 763,71 ms**	**9 183,06 ms**

Table A.36.: Time measurement of Running Time of Satisfaction Verification Toolchain; evaluated on ADASv1.

Times for Satisfaction Witnesses of ADASv4

FA-	Satisfaction	Coloring	Layouting	Tooling Option 1 (Satisfaction + Coloring)	Tooling Option 2 (Satisfaction + Layouting)
3	293,00 ms	4 469,00 ms	9 548,00 ms	4 762,00 ms	9 841,00 ms
4	277,00 ms	4 282,00 ms	11 413,00 ms	4 559,00 ms	11 690,00 ms
5	177,00 ms	2 963,00 ms	7 846,00 ms	3 140,00 ms	8 023,00 ms
6	261,00 ms	2 548,00 ms	6 996,00 ms	2 809,00 ms	7 257,00 ms
15	911,00 ms	1 461,00 ms	4 227,00 ms	2 372,00 ms	5 138,00 ms
19	192,00 ms	1 257,00 ms	5 283,00 ms	1 449,00 ms	5 475,00 ms
20	296,00 ms	848,00 ms	2 554,00 ms	1 144,00 ms	2 850,00 ms
21	258,00 ms	1 549,00 ms	5 185,00 ms	1 807,00 ms	5 443,00 ms
22	290,00 ms	1 817,00 ms	5 744,00 ms	2 107,00 ms	6 034,00 ms
23	302,00 ms	1 842,00 ms	6 658,00 ms	2 144,00 ms	6 960,00 ms
24	288,00 ms	1 732,00 ms	5 253,00 ms	2 020,00 ms	5 541,00 ms
24b	349,00 ms	2 201,00 ms	10 847,00 ms	2 550,00 ms	11 196,00 ms
25	330,00 ms	2 178,00 ms	7 068,00 ms	2 508,00 ms	7 398,00 ms
25b	541,00 ms	7 473,00 ms	22 431,00 ms	8 014,00 ms	22 972,00 ms
26	304,00 ms	1 717,00 ms	7 050,00 ms	2 021,00 ms	7 354,00 ms
26b	480,00 ms	8 699,00 ms	23 103,00 ms	9 179,00 ms	23 583,00 ms
27	274,00 ms	860,00 ms	3 158,00 ms	1 134,00 ms	3 432,00 ms
28	159,00 ms	1 730,00 ms	5 559,00 ms	1 889,00 ms	5 718,00 ms
30	142,00 ms	1 810,00 ms	5 092,00 ms	1 952,00 ms	5 234,00 ms
30b	192,00 ms	1 704,00 ms	5 815,00 ms	1 896,00 ms	6 007,00 ms
31	241,00 ms	1 455,00 ms	4 732,00 ms	1 696,00 ms	4 973,00 ms
32	263,00 ms	1 671,00 ms	5 638,00 ms	1 934,00 ms	5 901,00 ms
35	143,00 ms	1 097,00 ms	4 051,00 ms	1 240,00 ms	4 194,00 ms
65	228,00 ms	2 643,00 ms	7 824,00 ms	2 871,00 ms	8 052,00 ms
67	221,00 ms	2 772,00 ms	8 537,00 ms	2 993,00 ms	8 758,00 ms
75	141,00 ms	694,00 ms	2 729,00 ms	835,00 ms	2 870,00 ms
75b	206,00 ms	771,00 ms	3 743,00 ms	977,00 ms	3 949,00 ms
77	364,00 ms	1 763,00 ms	3 638,00 ms	2 127,00 ms	4 002,00 ms
84	220,00 ms	1 684,00 ms	5 756,00 ms	1 904,00 ms	5 976,00 ms
86	145,00 ms	1 324,00 ms	5 348,00 ms	1 469,00 ms	5 493,00 ms
99	159,00 ms	2 555,00 ms	4 663,00 ms	2 714,00 ms	4 822,00 ms
Average	278,94 ms	2 308,68 ms	7 015,77 ms	**2 587,61 ms**	**7 294,71 ms**

Table A.37.: Time measurement of Running Time of Satisfaction Verification Toolchain; evaluated on ADASv4.

Times for Tracing Witnesses of ADASv4

				Tooling Option 1 (Satisfaction	Tooling Option 2 (Satisfaction
3	343,00 ms	3 959,00 ms	16 631,00 ms	4 302,00 ms	16 974,00 ms
4	329,00 ms	6 846,00 ms	25 534,00 ms	7 175,00 ms	25 863,00 ms
5	238,00 ms	3 384,00 ms	14 566,00 ms	3 622,00 ms	14 804,00 ms
6	322,00 ms	4 203,00 ms	20 163,00 ms	4 525,00 ms	20 485,00 ms
15	920,00 ms	1 147,00 ms	5 347,00 ms	2 067,00 ms	6 267,00 ms
19	163,00 ms	1 413,00 ms	5 017,00 ms	1 576,00 ms	5 180,00 ms
20	260,00 ms	511,00 ms	2 594,00 ms	771,00 ms	2 854,00 ms
21	377,00 ms	3 602,00 ms	19 448,00 ms	3 979,00 ms	19 825,00 ms
22	384,00 ms	3 537,00 ms	14 246,00 ms	3 921,00 ms	14 630,00 ms
23	369,00 ms	35 684,00 ms	17 250,00 ms	36 053,00 ms	17 619,00 ms
24	366,00 ms	8 297,00 ms	16 155,00 ms	8 663,00 ms	16 521,00 ms
24b	404,00 ms	7 021,00 ms	30 572,00 ms	7 425,00 ms	30 976,00 ms
25	335,00 ms	4 148,00 ms	16 317,00 ms	4 483,00 ms	16 652,00 ms
25b	668,00 ms	14 911,00 ms	54 820,00 ms	15 579,00 ms	55 488,00 ms
26	339,00 ms	4 154,00 ms	17 351,00 ms	4 493,00 ms	17 690,00 ms
26b	642,00 ms	17 148,00 ms	55 368,00 ms	17 790,00 ms	56 010,00 ms
27	357,00 ms	2 605,00 ms	12 407,00 ms	2 962,00 ms	12 764,00 ms
28	217,00 ms	3 535,00 ms	12 569,00 ms	3 752,00 ms	12 786,00 ms
30	148,00 ms	1 370,00 ms	5 981,00 ms	1 518,00 ms	6 129,00 ms
30b	212,00 ms	2 114,00 ms	6 204,00 ms	2 326,00 ms	6 416,00 ms
31	235,00 ms	2 298,00 ms	5 883,00 ms	2 533,00 ms	6 118,00 ms
32	327,00 ms	4 159,00 ms	13 435,00 ms	4 486,00 ms	13 762,00 ms
35	154,00 ms	1 566,00 ms	4 510,00 ms	1 720,00 ms	4 664,00 ms
65	236,00 ms	3 983,00 ms	12 946,00 ms	4 219,00 ms	13 182,00 ms
67	232,00 ms	4 114,00 ms	14 226,00 ms	4 346,00 ms	14 458,00 ms
75	205,00 ms	2 892,00 ms	15 920,00 ms	3 097,00 ms	16 125,00 ms
75b	306,00 ms	3 168,00 ms	22 674,00 ms	3 474,00 ms	22 980,00 ms
77	379,00 ms	1 208,00 ms	3 995,00 ms	1 587,00 ms	4 374,00 ms
84	285,00 ms	3 245,00 ms	13 226,00 ms	3 530,00 ms	13 511,00 ms
86	190,00 ms	1 520,00 ms	10 010,00 ms	1 710,00 ms	10 200,00 ms
99	263,00 ms	7 816,00 ms	22 351,00 ms	8 079,00 ms	22 614,00 ms
Average	329,19 ms	5 340,58 ms	16 377,94 ms	**5 669,77 ms**	**16 707,13 ms**

Table A.38.: : Time measurement of Running Time of Tracing Toolchain; evaluated on ADASv4.

Appendix B.

Class Diagram in CD4A Syntax

The CD4A syntax is the nearly the same as the one of Roth [Rot17]. The CD4A language has been extended with the `read-only` keyword for associations to mark that the association as read-only. Additionally, the CD4A language uses only the EBNF context conditions[1]; hence, interfaces may have non-static fields.

```
classdiagram EmbeddedMontiArc {

/////////////////////////////////////////////////
//// below specific for EmbeddedMontiArc only
/////////////////////////////////////////////////

  // Figure 4.4
  interface PortType extends Type {
    boolean isCompatibleTo(PortType pt); // Figure 4.11
  }
  interface PortValue extends Value;
  class Unit {
    double prefix;
  }
  association [1] Unit (baseUnit) <-> Quantity [1];

  interface Value;
  interface Type extends Value; // Figure 4.7
  interface Quantity extends Type;

  read-only association Value -> (type) Type [1];
  read-only association PortValue -> (type) PortType [1];

  // Figure 4.6
  class Tensor implements PortValue;
  class EnumItem implements PortValue;
```

[1] https://git.rwth-aachen.de/monticore/cd4analysis/cd4analysis/tree/9baf060e2d94065b90772049b2ae353621de5990/src/main/java/de/monticore/umlcd4a/cocos/ebnf

```
class Boolean implements PortValue {
  boolean value;
}
class Matrix extends Tensor;
class Vector extends Matrix;
class Number extends Vector <<Quantity = "Any">> {
  double value; // Figure 3.18
  boolean isPlusInf;
  boolean isMinusInf;
}
class NaturalNumber extends Number;
association Tensor -> (elements) Number [*] <<ordered>>;
association Tensor (tensorOfRows) -> (rows) NaturalNumber
    ↪ [1];
association Tensor (tensorOfCols) -> (cols) NaturalNumber
    ↪ [1];
association Tensor (tensorOfDepth) -> (depth) NaturalNumber
    ↪  [1];
association Tensor -> Quantity [1];
association Number -> Unit [1];

// Figure 3.18
interface AlgebraicProperty;
class NumericType implements PortType;
association min NumericType -> Number [1];
association max NumericType -> Number [1];
association res NumericType -> Number [0..1];
association NumericType -> Quantity [1];
association NumericType -> (algebraicProperties)
    ↪ AlgebraicProperty [*];
class EnumType implements PortType;
association [1] EnumType <-> (items) EnumItem [*];
class BooleanType implements PortType;

class Struct implements PortValue;
class StructItem {
  String name;
}
association Struct [[name]] -> (item) StructItem [1];
association / Struct -> (items) StructItem [*];
association StructItem -> (value) PortValue [1];
class StructTypeItem {
  String name;
```

```
}
association StructTypeItem -> (type) PortType [1];
class StructType implements PortType {
  String name;
}
association StructType [[name]] -> (item) StructTypeItem
    ↪ [1];
association / StructType -> (items) StructTypeItem [*];
association Struct -> (type) StructType [1];
association StructItem -> (type) StructTypeItem [1];

// Figure 4.7
enum ParameterKind {
  CONFIG, GENERIC;
}
interface Parameter extends ComponentElement { //
    ↪ Figure 6.8
  String name;
  ParameterKind kind;
}
association Parameter -> (dimension) NaturalNumber [1];

interface ParameterBinding;
association ParameterBinding -> Range [1];
read-only association ParameterBinding -> Parameter [1];

// Figure 4.8
class GeneralTypeParameter implements Parameter;
association GeneralTypeParameter -> (defaultValue) PortType
    ↪  [0..1];
class GeneralParameterBinding implements ParameterBinding;
association GeneralParameterBinding -> (value) Type [1];
association GeneralParameterBinding -> (parameter)
    ↪ GeneralTypeParameter [1];

class QuantityParameter implements Parameter;
association QuantityParameter -> (defaultValue) Quantity
    ↪ [0..1];
class QuantityParameterBinding implements ParameterBinding;
association QuantityParameterBinding -> (value) Quantity
    ↪ [1];
association QuantityParameterBinding -> (parameter)
    ↪ QuantityParameter [1];
```

```

class NumericTypeParameter extends NumericType implements Parameter;
association NumericTypeParameter -> Quantity [1];
association NumericTypeParameter -> (defaultValue) NumericType [0..1];
class NumericTypeParameterBinding implements ParameterBinding;
association NumericTypeParameterBinding -> (value) NumericType [1];
association NumericTypeParameterBinding -> (parameter) NumericTypeParameter [1];

class TensorParameter extends Tensor implements Parameter;
association TensorParameter -> (type) NumericType [1];
association TensorParameter -> (defaultValue) Tensor [0..1];
class TensorParameterBinding implements ParameterBinding;
association TensorParameterBinding -> (value) Tensor [1];
association TensorParameterBinding -> (parameter) TensorParameter [1];

class NaturalNumberParameter extends NaturalNumber implements Parameter;
association NaturalNumberParameter -> (type) NumericType [1];
association NaturalNumberParameter -> (defaultValue) NaturalNumber [0..1];
class NaturalNumberParameterBinding implements ParameterBinding;
association NaturalNumberParameterBinding -> (value) NaturalNumber [1];
association NaturalNumberParameterBinding -> (parameter) NaturalNumberParameter [1];

class EnumTypeParameter extends EnumItem implements Parameter;
association EnumTypeParameter -> (type) EnumType [1];
association EnumTypeParameter -> (defaultValue) EnumItem [0..1];
class EnumTypeParameterBinding implements ParameterBinding;
association EnumTypeParameterBinding -> (value) EnumItem [1];
```

```
association EnumTypeParameterBinding -> (parameter) EnumTypeParameter [1];

class BooleanTypeParameter extends Boolean implements Parameter;
association BooleanTypeParameter -> (type) BooleanType [1];
association BooleanTypeParameter -> (defaultValue) Boolean [0..1];
class BooleanTypeParameterBinding implements ParameterBinding;
association BooleanTypeParameterBinding -> (value) Boolean [1];
association BooleanTypeParameterBinding -> (parameter) BooleanTypeParameter [1];

class StructTypeParameter extends Struct implements Parameter;
association / StructTypeParameter -> (type) StructType [1];
association StructTypeParameter -> (defaultValue) Struct [0..1];
class StructTypeParameterBinding implements ParameterBinding;
association StructTypeParameterBinding -> (value) Struct [1];
association StructTypeParameterBinding -> (parameter) StructTypeParameter [1];

class ComponentParameter extends BoundComponentType implements Parameter;
association / ComponentParameter -> (type) ComponentType [1];
association ComponentParameter -> (defaultValue) BoundComponentType [0..1];
class ComponentParameterBinding implements ParameterBinding;
association ComponentParameterBinding -> (value) BoundComponentType [1];
association ComponentParameterBinding -> (parameter) ComponentParameter [1];

// Figure 4.10
class BoundComponentType implements Value;
```

```
association BoundComponentType -> (values) ParameterBinding [*];
association BoundComponentType -> (type) ComponentType [1];

class ComponentInstantiation extends BoundComponentType implements ComponentElement { // Figure 6.8
  String name;
}
association ComponentInstantiation -> (dimension) NaturalNumber [1];

interface ComponentType extends Type {
  String name; // Figure 4.15
}
class ComponentInterface implements ComponentType;
class Component implements ComponentType, ComponentElement; // Figure 6.8
association Component -> (implements) BoundComponentType [*];

// Figure 4.11
enum PortDirection {
  IN , OUT;
}
class Port implements ComponentElement { // Figure 6.8
  String name;
  PortDirection direction;
}
association Port -> (type) PortType [1];
association Port -> (dimension) NaturalNumber [1];

class PortInstantiation;
association [*] PortInstantiation (portInstantiations) -> Port [1]; // Figure 6.19
association PortInstantiation -> (sub) ComponentInstantiation [0..1];
association PortInstantiation -> (portIndices) Range [1];
association PortInstantiation -> (subIndices) Range [0..1];

class Connector;
association Connector (startCon) -> (sourcePort) PortInstantiation [1];
```

```
association Connector (endCon) -> (targetPort) PortInstantiation [1];

class Range;
association start Range -> NaturalNumber [1];
association end Range -> NaturalNumber [1];
association step Range -> NaturalNumber [1];

// Figure 4.13
class Effector;
association sourceIndex Effector -> Range [1];
association targetIndex Effector -> Range [1];
association Effector (startEff) -> (sourcePort) Port [1];
association Effector (endEff) -> (targetPort) Port [1];

// Figure 4.15
association [1] ComponentType -> (ports) Port [*] <<ordered>>;
association ComponentType [[name]] -> (port) Port [1];
association ComponentType -> (parameters) Parameter [*] <<ordered>>;
association [0..1] Component (parent) <-> (subs) ComponentInstantiation [*];

// Figure 4.16
class CnCModel {
 boolean satisfies(CnCView cncv); // Figure 7.32
}
association CnCModel -> (depends) CnCLibrary [*];
association CnCModel -> (main) ComponentInstantiation [1];
association / CnCModel -> (effectors) Effector [*];
association / CnCModel -> (connectors) Connector [*];
association / CnCModel -> (componentTypes) ComponentType [*];

class CnCLibrary;
association / CnCLibrary -> Effector [*];
association / CnCLibrary -> Connector [*];
association [0..1] CnCLibrary <-> ComponentType [*];

// Figure 4.17
class ComponentInst implements ElementInst {
 String fullName;
```

```
}
association ComponentInst -> ComponentInstantiation [1];
association / ComponentInst (componentInsts) -> Component [1];
association ComponentInst -> (params) TensorParameterBinding [*];
association [0..1] ComponentInst (parent) <-> (subs) ComponentInst [*];
association ComponentInst -> (ports) PortInst [*];
class CnCInstanceStructure;
association CnCInstanceStructure (cis) -> (main) ComponentInst [1];
association [1] CnCInstanceStructure <-> CnCModel [*];

class ConnectorInst implements ElementInst;
association ConnectorInst (startCon) -> (sourcePort) PortInst [1];
association ConnectorInst (endCon) -> (targetPort) PortInst [1];
association ConnectorInst -> Connector [1];

class PortInst implements ElementInst {
  String fullName;
  PortDirection direction;
}
association PortInst -> PortInstantiation [1];
association / PortInst -> Port [1];
association PortInst -> (type) PortType [1];

class EffectorInst implements ElementInst;
association EffectorInst (startEff) -> (sourcePort) PortInst [1];
association EffectorInst (endEff) -> (targetPort) PortInst [1];
association EffectorInst -> Effector [1];

// Figure 4.18
interface ElementInst;
class ChainInst;
association ChainInst -> (elements) ElementInst [*];
read-only association start ChainInst -> ElementInst [1];
read-only association end ChainInst -> ElementInst [1];
```

```
association / PortInst (invInfluencee) -> (influencee)
    ↪ PortInst [*];
association / PortInst (invInfluencer) -> (influencer)
    ↪ PortInst [*];
association / PortInst (invSender) -> (sender) PortInst
    ↪ [0..1];
association / PortInst (invReceiver) -> (receiver) PortInst
    ↪  [*];

association / ComponentInst (invSender) -> (sender)
    ↪ ComponentInst [*];
association / ComponentInst (invReceiver) -> (receiver)
    ↪ ComponentInst [*];

// Figure 3.18

// support for units comes with the jscience library
// http://jscience.org/api/javax/measure/quantity/Quantity.
    ↪ html
class Acceleration implements Quantity;
class Angle implements Quantity;
class QuantityOfSubstance implements Quantity;
class AngularAcceleration implements Quantity;
class AngularVelocity implements Quantity;
class Area implements Quantity;
class CatalyticActivity implements Quantity;
class DataQuantity implements Quantity;
class DataRate implements Quantity;
class Dimensionless implements Quantity;
class Duration implements Quantity;
class DynamicViscosity implements Quantity;
class ElectricCapacitance implements Quantity;
class ElectricCharge implements Quantity;
class ElectricConductance implements Quantity;
class ElectricCurrent implements Quantity;
class ElectricInductance implements Quantity;
class ElectricPotential implements Quantity;
class ElectricResistance implements Quantity;
class Energy implements Quantity;
class Force implements Quantity;
class Frequency implements Quantity;
class Illuminance implements Quantity;
class KinematicViscosity implements Quantity;
```

```
class Length implements Quantity;
class LuminousFlux implements Quantity;
class LuminousIntensity implements Quantity;
class MagneticFlux implements Quantity;
class MagneticFluxDensity implements Quantity;
class Mass implements Quantity;
class MassFlowRate implements Quantity;
class Money implements Quantity;
class Power implements Quantity;
class Pressure implements Quantity;
class RadiationDoseAbsorbed implements Quantity;
class RadiationDoseEffective implements Quantity;
class RadioactiveActivity implements Quantity;
class SolidAngle implements Quantity;
class Temperature implements Quantity;
class Torque implements Quantity;
class Velocity implements Quantity;
class Volume implements Quantity;
class VolumetricDensity implements Quantity;
class VolumetricFlowRate implements Quantity;

association NumericType (numericTypeOfRows) -> (rows)
    ↪ NaturalNumber [1];
association NumericType (numericTypeOfCols) -> (cols)
    ↪ NaturalNumber [1];
association NumericType (numericTypeOfDepth) -> (depth)
    ↪ NaturalNumber [1];

class Diagonal implements AlgebraicProperty;
class Symmetric implements AlgebraicProperty;
class Invertible implements AlgebraicProperty;

// Figure 6.7
association / Component -> (subDefs) Component [*];

// Figure 6.8
interface ComponentElement {
  String name;
}

association Component -> (innerComponents) Component [*];
association / Component -> (innerElements) ComponentElement
    ↪  [*];
```

```
////////////////////////////////////////////////
// EXTENSION VIA TAGGING (uses merging of class diagrams)
////////////////////////////////////////////////

 // Figure 6.13
 class Traceable extends Boolean;
 association Component -> Traceable [1];
 association ComponentInst -> Traceable [1];

 // Figure 6.14
 class NumberPower extends Number <<Quantity = "Power">>;
 class MaxPower extends NumberPower;
 association Component -> MaxPower [*];
 association ComponentInst -> MaxPower [*];

 // Figure 6.19
 class EncryptionCollection;
 enum EEncryption {
  AES, RSA, DES, DES3;
 }
 association Port -> (encryption) EncryptionCollection [*];
 association EncryptionCollection -> (elements) EEncryption [*];

 // Figure 6.16
 enum EAuth {
  Pin, Voice, FaceID, Finger;
 }
 class Auth;
 association Auth -> (value) EAuth [1];
 association Connector -> (auth) Auth [*];
 association ConnectorInst -> (auth) Auth [*];

 // Figure 6.17
 class Cert extends String;
 association Port -> (cert) Cert [*];
 association ComponentInst -> (cert) Cert [*];

 // Figure 6.20
 class Encryption;
 association Encryption -> (value) EEncryption [1];
```

```
association PortInst -> (encryption) Encryption [*];
association Encryption -> (decryptPower) NumberPower [1];
association Encryption -> (encryptPower) NumberPower [1];

// Figure 6.21
class EncryptPower {
  Encryption encryption;
}
association EncryptPower -> (encrypt) NumberPower [1];
association EncryptPower -> (decrypt) NumberPower [1];
association Component [[encryption]] -> (encryptPower)
    ↪ EncryptPower [*];

// Figure 6.22
enum EAsil {
  QM, ASIL_A, ASIL_B, ASIL_C, ASIL_D;
}
class Asil;
association Asil -> (value) EAsil [1];
association Component -> (asil) Asil [*];

// Figure 6.23
class NumberDuration extends Number <<Quantity = "Duration
    ↪ ">>;
class Wcet extends NumberDuration;
association Component -> (wcet) Wcet [*];

// Figure 6.25
class Threads extends NaturalNumber;
association ComponentInst -> (threads) Threads [*];

//////////////////////////////////////////////////
// Syntactic Sugar Diagram (added here in same CD,
// so that OCL does not need different packages)
// 'Component'' in slide matches to 'ComponentSugar'
// same is for the rest
//////////////////////////////////////////////////

 // Figure 6.34
 // this association is only needed to express the
    ↪ transformations
 association [1] Component (componentScope) <-> (
    ↪ definedConnectors) Connector [*];
```

```

interface ComponentTypeSugar;
association [1] ComponentTypeSugar (componentType) -> (
    ↪ ports) PortSugar [*] <<ordered>>;

class ComponentSugar implements ComponentTypeSugar;
association [1] ComponentSugar (componentScope) <-> (
    ↪ definedConnectors) ConnectorSugar [*];
association [0..1] ComponentSugar (parent) <-> (subs)
    ↪ ComponentInstantiationSugar [*];

class PortSugar {
  String name;
}
association PortSugar -> (direction) PortDirection [0..1];
association PortSugar -> (type) PortType [1];
association PortSugar -> (dimension) NaturalNumber [0..1];

class PortInstantiationSugar {
  boolean indexBased;
  boolean nameBased;
}
association PortInstantiationSugar -> (port) PortSugar [1];
association PortInstantiationSugar -> (sub)
    ↪ ComponentInstantiationSugar [0..1];
association PortInstantiationSugar -> (portIndices)
    ↪ RangeSugar [0..1];
association PortInstantiationSugar -> (subIndices)
    ↪ RangeSugar [0..1];
// one PortInstantiationSugar can have zero to two
    ↪ RangeSugars (cf. portIndices, subIndices),
// but one RangeSugar belongs to one PortInstantiationSugar
association RangeSugar -> (portInstantiation)
    ↪ PortInstantiationSugar [1];

class ComponentInstantiationSugar;
association ComponentInstantiationSugar -> (dimension)
    ↪ NaturalNumber [0..1];

class ConnectorSugar;
association ConnectorSugar -> (sourcePort)
    ↪ PortInstantiationSugar [1];
```

```
association ConnectorSugar -> (targetPort)
    ↪ PortInstantiationSugar [1];

class RangeSugar {
  boolean all;
}
association RangeSugar -> (start) NaturalNumber [0..1];
association RangeSugar -> (end) NaturalNumber [0..1];
association RangeSugar -> (step) NaturalNumber [0..1];

////////////////////////////////////////////////
//// classes for EmbeddedMontiView language, they are all
//// merged in this class diagram, so that OCL does not
//// need different packages
//// classes of EmbeddedMontiView are not complete -> only
//// the once differ of EmbeddedMontiArc are listed below
////////////////////////////////////////////////

  // Figure 7.4
  class ADimension;
  association ADimension -> (min) NaturalNumber [1];
  association ADimension -> (max) NaturalNumber [1];

  interface AType extends AValue { // Figure 7.11
    boolean isCompatibleTo(Type t);
  }

  interface AParameter extends AType {
   boolean underspec;
   String name;
   ParameterKind kind;
  }
  association / AParameter -> (type) AType [0..1];

  class APort {
   PortDirection direction;
  }
  association APort -> (name) String [0..1];
  association APort -> (dimension) ADimension [0..1];
  association APort -> (type) APortType [0..1];

  interface AComponentType extends ATypeOrAInstantiation { //
    ↪  Figure 7.20
```

```
  boolean portsComplete;
  boolean atomic; // Ch07_SatisfactionPort
}
association AComponentType -> (name) String [0..1];
association AComponentType -> (parameters) AParameter [*];
association AComponentType -> (ports) APort [*];

interface APortType extends AType;

// Figure 7.11
class AComponentInstantiation implements
    ↪ ATypeOrAInstantiation { // Figure 7.20
  boolean direct;
}
association AComponentInstantiation -> (name) String [0..1];
    ↪
association AComponentInstantiation -> (dimension)
    ↪ ADimension [0..1];
association AComponentInstantiation -> (values)
    ↪ AParameterBinding [*];
association AComponentInstantiation -> (type)
    ↪ AComponentType [0..1];

interface AParameterBinding;
association [*] AParameterBinding (bindings) -> (parameter)
    ↪  AParameter [1];
association AParameterBinding -> (range) ARange [1];
association AParameterBinding -> (value) AValue [1];

interface AValue;
read-only association AValue -> (type) AType [0..1];

class AComponentInterface implements AComponentType;

class AComponent implements AComponentType {
  boolean instComplete;
  boolean atomic;
}
association / AComponent -> (name) String [0..1];
association AComponent -> (implements)
    ↪ AComponentInstantiation [*];
association AComponent (parent) -> (subs)
    ↪ AComponentInstantiation [*];
```

```

// Figure 7.17
class ANumericType implements APortType;
association ANumericType -> (quantity) Quantity [0..1];
association ANumericType -> (rows) NaturalNumber [0..1];
association ANumericType -> (cols) NaturalNumber [0..1];
association ANumericType -> (depth) NaturalNumber [0..1];

interface APortValue extends AValue;
read-only association APortValue -> (type) APortType [1];

// Figure 7.20
class ARange extends Range {
  boolean all;
  boolean notSpecified;
}

class AConnector;
association AConnector -> (sourcePort) APortInstantiation
    ↪ [1];
association AConnector -> (targetPort) APortInstantiation
    ↪ [1];

class APortInstantiation;
association APortInstantiation -> (portIndices) ARange
    ↪ [0..1];
association APortInstantiation -> (cmpNavIndices) ARange
    ↪ [*] <<ordered>>;
association APortInstantiation -> (port) APort [0..1];
association APortInstantiation -> (cmpNav)
    ↪ ATypeOrAInstantiation [*] <<ordered>>;
interface ATypeOrAInstantiation;

// Figure 7.21
class AEffector;
association AEffector -> (sourcePort) APortInstantiation
    ↪ [1];
association AEffector -> (targetPort) APortInstantiation
    ↪ [1];

// Figure 7.23
class CnCView;
```

```
 association CnCView -> (aComponentTypes) AComponentType [*];
 association CnCView -> (aComponentInstantiations) AComponentInstantiation [*];
 association CnCView -> (aConnectors) AConnector [*];
 association CnCView -> (aEffectors) AEffector [*];

 // Figure 7.24
 association Type -> (name) String [1]; // a type has a short name which can be derived

 // Figure 7.26
 association AParameter -> (dimension) ADimension [0..1];

 // Figure 7.28
 class ConnectorChainInst extends ChainInst;
 association / ConnectorChainInst -> (start) ConnectorInst [1];
 association / ConnectorChainInst -> (end) ConnectorInst [1];
 association ConnectorChainInst -> (connectors) ConnectorInst [*] <<ordered>>;
 association / [*] ConnectorChainInst -> (startPort) PortInst [1];
 association / [*] ConnectorChainInst -> (endPort) PortInst [1];
 association / ComponentInstantiation -> (subs) ComponentInstantiation [*];
 association / Component -> (allSubs) ComponentInstantiation  [*];

/////////////////////////////////////////////////
//// support for standard types and operations in OCL
/////////////////////////////////////////////////

 class Class;

 class Object;
 class Collection {
  boolean containsAll(Collection c);
  boolean contains(Collection c);
  int size();
  boolean isEmpty();
```

```
  Collection addAll(Collection c);
  Collection retainAll(Collection c);
  Set asSet();
  Collection flatten(); // see http://mbse.se-rwth.de/book1/
     ↪ index.php?c=chapter3-3#x1-560003.3.6
  Collection listPartitions(int length); // see https://
     ↪ github.com/dpaukov/combinatoricslib#7-list-
     ↪ partitions
  boolean areCompatibleTo(Collection algebraicProperties);
     ↪ // boolean Collection<AlgebraicProperty>::
     ↪ areCompatibleTo(Collection<AlgebraicProperty>
     ↪ algebraicProperties)
}

class List extends Collection {
    boolean nonEmpty();
    List addAll(List c);
    List add(Object o);
    int indexOf(Object o);
}

class Set extends Collection {
    Set addAll(Set c);
    List asList();
    Set add(Object o);
}

class Optional {
  Set asSet(); // Optional.empty => {} and Optional.of(X) =>
     ↪  { X }
  boolean isAbsent();
  boolean isPresent();
}

class Map {
  int size();
}

class Date;
class Time {
    static Time now();
    boolean lessThan(Time that);
}
```

```

  class Integer extends Number;
  class Double extends Number;
  class Float extends Number;
  class Long extends Number;
  class Character;
  class String {
    boolean contains(String s);
    String replaceAll(String s1, String s2);
    String replace(String s1, String s2);
    boolean endsWith(String s);
    int length();
  }

  class Math {
    static double abs(double v);
  }
}
```

Listing B.1: Merged class diagram in CD4A syntax of all graphical class diagram representations of this PhD thesis.

Appendix C.

Other Material

C.1. *MontiCore* 5 grammar for C&C instance structure

MC5

```
grammar ComponentAndConnectorInstanceStructure
                                              extends embeddedMontiArc.Types {
  CnCInstanceStructure = main:ComponentInst;
  interface ElementInst;
  ComponentInst implements ElementInst =
    "cmp-i" NameWithDollar
    "(" params:(MParameterBinding || ",")* ")"
    "{" bodyElements:ElementInst* "}"
  ;
  MParameterBinding =
    Type Name ("[" dimension:PositiveNumber "]")
    "=" Value
  ;
  PortInsts implements ElementInst =
    "port-i" (PortInst || ",")+ ";"
  ;
  PortInst = direction:("in" | "out") Type NameWithDollar;

  EffectorInst implements ElementInst =
    "eff-i" sourcePort:NameWithDollar "->" targetPort:NameWithDollar ";"
  ;
  ConnectorInst implements ElementInst =
    sourcePort:NameWithDollarAndDot "->" targetPort:NameWithDollarAndDot ";"
  ;
}
```

Figure C.1.: *MontiCore* 5 grammar for C&C instance structure. This grammar is no official modeling grammar. It is only a test grammar to validate the transformation from *EmbeddedMontiArc*'s C&C abstract syntax to the C&C instance structure abstract syntax.

This C&C instance structure language as shown in Figure C.1 is no official modeling language to create C&C models. It is only a test language to verify the transformation from C&C models to C&C instance structures, because this way the expected test result can be formulated in a convenient way.

C.2. Operator Priority in *OCL*

Table C.2.: *OCL/P* Operator Priority. Higher priority binds stronger. It is incomplete, the table shows only operators needed in *OCL* expressions in this PhD thesis. The priority order is the same as in [Rum11, Tabelle 3.12]; however, the actual priority numbers do not fit as new operators have been introduced. Section 6.5 discusses the differences between the *OCL* versions of this PhD thesis and the one of Rumpe [Rum11].

PRIORITY	EXPRESSION TYPE	EXAMPLE
17	numbers, literals	**7 m/s^2**, **“normal text”**, **true**
16	qualified primary	**component**, **component.ports[0]**, **x****
15	parentheses sets	(2+3)*4 {1, 2, 3}, {x*x | x in {1 .. 7 } }
14	function call	**method1**(3, 3)
13	collection prefixes	**min** {1, 2}, **sum** List{ 1, 1, 7 .. 19}, **or** List{true, false}, **intersection** set1
12	logical not	**!**cond1
11	multiplication, division	2*3, sum/size
10	plus, minus	3+4, a-2
9	greater/smaller (equals) optional greater/smaller (equals) instanceof in, isin	1 > 2, 2 >= 3, 3 < a, b <= c opValue **?>** 2, opValue **?>=** 3, opValue **?<** 4, opValue **?>=** 5 value **instanceof** NumericType comp **in** Component, 1 **isin** {1, 2}
8	elvis operator	opValue **?:** defaultValue
7	equals/not equals similar/not similar optional (not) equals optional (not) similar	x **==** 1, x **!=** y p **~~** q, p **!~** r opValue **?==** x, opValue **?!=** y opValue **?~~** r, opValue **?!~** s
6	logical and	cond1 **&&** cond2
5	logical or	cond1 **||** cond2
4	implies	a > b && b > c **implies** a > c
3	if and only if	atomic **<=>** subs == {}
2	type if	**typeif** ct **instanceof** Component **then** ct.subs else {}
1	for all exists	**forall** p in ports**:** ports.direction == IN **exists** Component c**:** c.atomic

C.3. Material to Chain Instances

The four longest chain instances of the C&C instance structure presented in Figure 4.19 are:

- $\texttt{chainInst}_{\texttt{signal\$1} \rightarrow \texttt{filter\$1.distance}}$ = {cmp-i SensorProcessing, port-i SensorProcessing.signal$1, SensorProcessing.signal$1 -> SensorProcessing.filter$1.signal, cmp-i SensorProcessing.filter$1, port-i SensorProcessing.filter$1.signal, eff-i SensorProcessing.filter$1.signal -> SensorProcessing.filter$1.distance, port-i SensorProcessing.filter$1.distance}
- $\texttt{chainInst}_{\texttt{signal\$2} \rightarrow \texttt{filter\$2.distance}}$ = {cmp-i SensorProcessing, port-i SensorProcessing.signal$2, SensorProcessing.signal$2 -> SensorProcessing.filter$2.signal, cmp-i SensorProcessing.filter$2, port-i SensorProcessing.filter$2.signal, eff-i SensorProcessing.filter$2.signal -> SensorProcessing.filter$2.distance, port-i SensorProcessing.filter$2.distance}
- $\texttt{chainInst}_{\texttt{posCar} \rightarrow \texttt{filter\$1.distance}}$ = {cmp-i SensorProcessing, port-i SensorProcessing.posCar, SensorProcessing.posCar -> SensorProcessing.filter$1.posCar, cmp-i SensorProcessing.filter$1, port-i SensorProcessing.filter$1.posCar, eff-i SensorProcessing.filter$1.posCar -> SensorProcessing.filter$1.distance}
- $\texttt{chainInst}_{\texttt{posCar} \rightarrow \texttt{filter\$2.distance}}$ = {cmp-i SensorProcessing, port-i SensorProcessing.posCar, SensorProcessing.posCar -> SensorProcessing.filter$2.posCar, cmp-i SensorProcessing.filter$2, port-i SensorProcessing.filter$2.posCar, eff-i SensorProcessing.filter$2.posCar -> SensorProcessing.filter$2.distance}

Bibliography

[AB14] Jamie Ayre and Jenna Beaucage. AdaCore's CodePeer Static Analysis Tool Earns Qualification for Software Verification in Avionics, Railway. `https://www.adacore.com/press/codepeer-earns-qualification`, 2014. Accessed: 2018-07-31.

[AB17] Markus Andres and Michael Bockmair. Ausführbare Spezifikation - Berechnungen von komplexen Systemzusammenhängen. In *Tag des Systems Engineering*, pages 167–170. Carl Hanser Verlag, 2017.

[ABGM09] Aldeida Aleti, Stefan Bjornander, Lars Grunske, and Indika Meedeniya. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on*, pages 61–71. IEEE, 2009.

[acc18] accellera SYSTEMS INITIATIVE. SystemC Verification Working Group (VWG). `http://www.accellera.org/activities/working-groups/systemc-verification`, 2018. Accessed: 2018-07-31.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 334–367, Berlin, Heidelberg, 2002. Springer-Verlag.

[ADEM14] Marcos Arjona, Carolina Dania, Marina Egea, and Antonio Maña. Validation of a Security Metamodel for the Development of Cloud Applications. In *OCL@ MoDELS*, pages 33–42, 2014.

[ADS+06] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing Safe, Reliable Systems Using Scade. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods*, pages 115–129, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[AFBL14] Albert Albers, Jan Fischer, Matthias Behrendt, and Dirk Lieske. Messung und Interpretation der Wirkkette eines akustischen Phänomens im Antriebsstrang eines Elektrofahrzeugs. *ATZ-Automobiltechnische Zeitschrift*, 116(3):68–75, 2014.

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AJB00] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *nature*, 406(6794):378, 2000.

[AK13] P. M. Asprion and G. F. Knolmayer. Assimilation of Compliance Software in Highly Regulated Industries: An Empirical Multitheoretical Investigation. In *2013 46th Hawaii International Conference on System Sciences*, pages 4405–4414, Jan 2013.

[All97] Robert J Allen. A Formal Approach to Software Architecture. Technical report, Carnegie-Mellon Univ Pittsburgh Pa School Of Computer Science, 1997.

[Alt16] AJ Alt. Kotlin: The Good, The Bad, and The Ugly. `https://medium.com/keepsafe-engineering/kotlin-the-good-the-bad-and-the-ugly-bf5f09b87e6f`, 2016. Accessed: 2018-07-31.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, volume 2097 of *CEUR Workshop Proceedings*, pages 75–79. CEUR-WS.org, May 2018.

[Arc18] ArchUnit. Unit test your Java architecture. `https://www.archunit.org/`, 2018. Accessed: 2018-07-31.

[ASM04] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 225–249, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Aßm08] Uwe Aßmann. ArchJava – A Java Extension for Architecture. `http://st.inf.tu-dresden.de/files/teaching/ss10/cbse/08b-arch-java.pdf`, 2008. Accessed: 2018-07-31.

[ASZT07] Ghiath Al-Sammane, Mohamed H Zaki, and Sofiène Tahar. A symbolic methodology for the verification of analog and mixed signal designs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 249–254. EDA Consortium, 2007.

[AUT06] AUTOSAR. UML Profile for AUTOSAR. Technical Report Version 1.0.1, AUTOSAR, 2006.

[AUT09] AUTOSAR. Explanation of Error Handling on Application Level. Technical Report Version 1.0.0 final (part of AUTOSAR Release 4.0), AUTOSAR, 2009.

[AUT16] AUTOSAR. Overview of Functional Safety Measures in AUTOSAR. Technical Report AUTOSAR CP Release 4.3.0, AUTOSAR, 2016.

[AUT17] AUTOSAR. AUTOSAR Feature Model Exchange Format. Technical Report Document Identification No 606 (AUTOSAR CP Release 4.3.1), AUTOSAR, 2017.

[AVKB14] Ahmed Ahmed, Paola Vallejo, Mickaël Kerboeuf, and Jean-Philippe Babau. CdmCL, a Specific Textual Constraint Language for Common Data Model. In *OCL@ MoDELS*, pages 63–72, 2014.

[AVT+15] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In *ACES-MB*, 2015.

[Bag10] Anya Helene Bagge. Language description for front end implementation. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, page 9. ACM, 2010.

[BBD+16] Mira Balaban, Phillipa Bennett, Khanh-Hoang Doan, Geri Georg, Martin Gogolla, Igal Khitron, and Michael Kifer. A Comparison of Textual Modeling Languages: OCL, Alloy, FOML. In *OCL@ MoDELS*, pages 57–72, 2016.

[BBH+13] Christian Berger, Delf Block, Christian Hons, Stefan Kühnel, André Leschke, Bernhard Rumpe, and Torsten Strutz. Meta-metrics for simulations in software engineering on the example of integral safety systems. In *Proceedings des 14. Braunschweiger Symposiums AAET 2013, Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, 2013.

[BBH+14a] Christian Berger, Delf Block, Sönke Heeren, Christian Hons, Stefan Kühnel, André Leschke, Dimitri Plotnikov, and Bernhard Rumpe. Simulations on Consumer Tests: A Systematic Evaluation Approach in an Industrial Case Study. In *Intelligent Transportation Systems Conference (ITSC'14)*, pages 1474–1480. IEEE, 2014.

[BBH+14b] Christian Berger, Delf Block, Sönke Heeren, Christian Hons, Stefan Kühnel, André Leschke, Dimitri Plotnikov, and Bernhard Rumpe. Simulations on Consumer Tests: Systmatic Evaluation of Tolerance Ranges by Model-Based Generation of Simulation Scenarios. In *30. VDI/VW-Gemeinschaftstagung: Fahrerassistenzsysteme und Integrierte Sicherheit*, volume 2223 of *VDI-Berichte*, pages 403–418. VDI Wissensforum GmbH, 2014.

[BBH+15a] Christian Berger, Delf Block, Christian Hons, Stefan Kühnel, André Leschke, Dimitri Plotnikov, and Bernhard Rumpe. Large-Scale Evaluation of an Active Safety Algorithm with EuroNCAP and US NCAP Scenarios in a Virtual Test Environment–An Industrial Case Study. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 2280–2286. IEEE, 2015.

[BBH+15b] Christian Berger, Delf Block, Christian Hons, Stefan Kühnel, André Leschke, Dimitri Plotnikov, and Bernhard Rumpe. Simulations on Consumer Tests: A Systematic Evaluation Approach in an Industrial Case Study. *Intelligent Transportation Systems Magazine (ITSM)*, 7(4):24–36, October 2015.

[BBKR09] Arne Bartels, Christian Berger, Holger Krahn, and Bernhard Rumpe. Qualitätsgesicherte Fahrentscheidungsunterstützung für automatisches Fahren auf Schnellstraßen und Autobahnen. In *Proceedings des 10. Braunschweiger Symposiums: Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, pages 341–353, 2009.

[BBR+05] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Núria Mata, Robert Sandner, and Dirk Ziegenbein. Automodenotations, methods, and tools for model-based development of automotive software. Technical report, SAE Technical Paper, 2005.

[BBVB+01] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. `https://moodle2016-17.ua.es/moodle/pluginfile.php/80324/mod_resource/content/2/agile-manifesto.pdf`, 2001. Accessed: 2018-07-31.

[BCC+08] Tomas Bures, Jan Carlson, Ivica Crnkovic, Séverine Sentilles, and Aneta Vulgarakis Feljan. ProCom - the Progress Component Model Reference Manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Industrial Software Engineering, Malardalen University, Vasteras, Sweden, June 2008.

[BCD+16] Achim D Brucker, Jordi Cabot, Gwendal Daniel, Martin Gogolla, AS Herrera, Frank Hilken, Frédéric Tuong, Edward D Willink, and Burkhart Wolff. Recent Developments in OCL and Textual Modelling. In *Proceedings of International Workshop on OCL and Textual Modeling (OCL 2016)*, volume 1756, pages 157–165. CEUR-WS. org, 2016.

[BCvDV11] E. Bouwers, J. P. Correia, A. v. Deursen, and J. Visser. Quantifying the Analyzability of Software Architectures. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 83–92, June 2011.

[BD95] Adolf-Peter Bröhl and Wolfgang Dröschel. Das V-Modell. *München, Wien: Oldenburg-Verlag*, 1995.

[BDBK10] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SCV: SystemC Verification Library*, pages 189–205. Springer US, Boston, MA, 2010.

[BDV+16] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 84–89. ACM, 2016.

[BEK+18] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.

[BEM93] Noureddine Belkhatir, Jacky Estublier, and Walcelio L Melo. Software process model and work space control in the Adele system. In *Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the*, pages 2–11. IEEE, 1993.

[BEP+18] Inga Blundell, Jochen Martin Eppler, Konstantin Perun, Abigail Morrison, Dimitri Plotnikov, Bernhard Rumpe, and Guido Trensch. Reengineering NestML with Python and MontiCore, July 2018.

[Ber18] Berkeley EECS Department. Ptolemy II Frequently Asked Questions. `https://ptolemy.berkeley.edu/ptolemyII/ptIIfaq.htm#simulink`, 2018. Accessed: 2018-07-31.

[Bet13] Lorenzo Bettini. Implementing Java-like Languages in Xtext with Xsemantics. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1559–1564, New York, NY, USA, 2013. ACM.

[Beu05] Cedric Beust. The Perils of Duck Typing. `http://beust.com/weblog/2005/04/15/the-perils-of-duck-typing/`, 2005. Accessed: 2018-07-31.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE, 2001.

[BGG56] Richard Bellman, Irving Glicksberg, and Oliver Gross. On the "bang-bang" control problem. *Quarterly of Applied Mathematics*, 14(1):11–18, 1956.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHJ16] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52, May 2016.

[BHK+07] Delf Block, Sönke Heeren, Stefan Kühnel, André Leschke, Bernhard Rumpe, and Vladislavs Serebro. Simulations on Consumer Tests: A Perspective for Driver Assistance Systems. In *Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems*, ES4CPS '14, pages 38:38–38:43, New York, NY, USA, 2007. ACM.

[BHK+16] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. View-based model-driven software development with ModelJoin. *Software & Systems Modeling*, 15(2):473–496, 2016.

[BHSV+96] Robert K Brayton, Gary D Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, et al. VIS: A system for verification and synthesis. In *International conference on computer aided verification*, pages 428–432. Springer, 1996.

[BI96] Barry W. Boehm and Hoh In. Identifying Quality-Requirement Conflicts. *IEEE Software*, 13(2):25–35, 1996.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BK11] Mira Balaban and Michael Kifer. Logic-based model-level software development with F-OML. In *International Conference on Model Driven Engineering Languages and Systems*, pages 517–532. Springer, 2011.

[BKH+13] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The BRICS component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1758–1764. ACM, 2013.

[BKKS14] Manfred Broy, Sascha Kirstan, Helmut Krcmar, and Bernhard Schätz. What is the benefit of a model-based design of embedded software systems in the car industry? In *Software Design and Development: Concepts, Methodologies, Tools, and Applications*, pages 310–334. IGI Global, 2014.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications based on Product Process Resource Models. In *International Conference on Industrial Automation, Robotics and Control Engineering (IARCE '18)*. IEEE, September 2018.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKS08] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. *Scoot: A Tool for the Analysis of SystemC Models*, volume 4963 of *Lecture Notes in Computer Science*, pages 467–470. Springer, 2008.

[BKU14] Denis Buzdalov, Alexey Khoroshilov, and Alexander Ugnenko. AADL Core Questions. `https://wiki.sei.cmu.edu/aadl/images/5/5f/201402-ispras-aadl-proposals.pdf`, 2014. Accessed: 2018-07-31.

[BMP+16] Vincent Bertram, Peter Manhart, Dimitri Plotnikov, Bernhard Rumpe, Christoph Schulze, and Michael von Wenckstern. Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 109–116. Bonner Köllen Verlag, March 2016.

[BMR+17a] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Component and Connector Views in Practice: An Experience Report. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 167–177. IEEE, September 2017.

[BMR+17b] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Supplementary Material: Component and Connector Views in Practice: An Experience Report. `http://www.se-rwth.de/materials/cncviewscasestudy/`, 2017. Accessed: 2018-07-31.

[BMR+18] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Component and Connector Views in Practice: An Experience Report (extended abstract). In *Software Engineering und Software Management 2018 (SE'18)*, volume P-279 of *GI-Edition–Lecture Notes in Informatics (LNI)*, pages 97–99. Bonner Köllen Verlag, March 2018.

[BNP+04] Rémi Bastide, David Navarre, Philippe Palanque, Amélie Schyn, and Pierre Dragicevic. A model-based approach for real-time embedded multimodal systems in military aircrafts. In *Proceedings of the 6th international conference on Multimodal interfaces*, pages 243–250. ACM, 2004.

[Bor06] Moritz Borgmann. Matrix Taxonomy & Matrix Properties. `https://www.nari.ee.ethz.ch/teaching/ha/handouts/linalg3p.pdf`, 2006. Accessed: 2018-07-31.

[BP06] Nancy L Bayer and Lisa Pappas. Accessibility testing: Case history of blind testers of enterprise software. *Technical Communication*, 53(1):32–38, 2006.

[BPB17] Alexander Blumör, Gerhard Pregitzer, and Martin Bothen. Werkzeuge für die Entwicklung mechatronischer Systeme mit Methoden des MBSE. In *Tag des Systems Engineering*, pages 191–202. Carl Hanser Verlag, 2017.

[BQ06] M. F. Bashir and M. A. Qadir. Traceability Techniques: A Critical Study. In *IEEE International Multitopic Conference*, pages 265–268, Dec 2006.

[BR00] Keith H. Bennett and Václav Rajlich. Software maintenance and evolution: a roadmap. In Anthony Finkelstein, editor, *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 73–87. ACM, 2000.

[BR05] Manfred Broy and Andreas Rausch. Das neue v-modell® xt. *Informatik-Spektrum*, 28(3), 2005.

[Bro05] David Brown. Electronic government and public administration. *International Review of Administrative Sciences*, 71(2):241–254, 2005.

[Bro08] Manfred Broy. Architecture Based Specification and Verification of Embedded Software Systems (Work in Progress). In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, pages 1–13, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BRR+10] Christian Berger, Holger Rendel, Bernhard Rumpe, Carsten Busse, Thorsten Jablonski, and Fabian Wolf. Product line metrics for legacy software in practice. In *International Systems and Software Product Line Conference (SPLC '10)*, 2010.

[BRRvW16] Vincent Bertram, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Extendable Toolchain for Automatic Compatibility Checks. In *International Workshop in OCL and Textual Modeling (OCL'16)*, pages 49–56. ACM/IEEE, October 2016.

[BRRW13] Christian Berger, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Generative Softwareentwicklung zur Optimierung der Konstruktion eingebetteter

Softwaresysteme am Beispiel einer Lenkungssteuerung. In *Tagungsband AALE 2013, 10. Fachkonferenz, Das Forum für Fachleute der Automatisierungstechnik aus Hochschulen und Wirtschaft*, 2013.

[BRS00] Peter Braun, Martin Rappl, and Jörg Schäuffele. Softwareentwicklung für steuergerätenetzwerke–eine methodik für die frühe phase. In *VDI-Berichte. NR. 1547*, 2000.

[BRS09] Alexej Beresnev, Bernhard Rumpe, and Frank Schoven. Automated testing of graphical models in heterogeneous test environments. In *6th International Workshop on Intelligent Transportation*, 2009.

[Bru17a] Stefan Brunecker. Transformation Tool for Simulink Models to MontiArc Models, 2017.

[Bru17b] Stefan Brunecker. *Transforming Simulink Models to MontiArc Models*. Bachelor Thesis at RWTH Aachen University, 2017.

[BRvW16] Vincent Bertram, Bernhard Rumpe, and Michael von Wenckstern. Encapsulation, Operator Overloading, and Error Class Mechanisms in OCL. In *International Workshop in OCL and Textual Modeling (OCL'16)*, pages 17–32. ACM/IEEE, 2016.

[BRW16] Arvid Butting, Bernhard Rumpe, and Andreas Wortmann. Embedding Component Behavior DSLs into the MontiArcAutomaton ADL. In *Globalization of Modeling Languages Workshop (GEMOC'16)*, volume 1731 of *CEUR Workshop Proceedings*, October 2016.

[BS12] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[BTC13] Sagar Behere, Martin Törngren, and De-Jiu Chen. A reference architecture for cooperative driving. *journal of Systems Architecture*, 59(10):1095–1112, 2013.

[BvW17] Vincent Bertram and Michael von Wenckstern. C&C Views of ADAS version 1. `http://www.se-rwth.de/materials/cncviewscasestudy/ViewsADASv1.pdf`, 2017. Accessed: 2018-07-31.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[BWHK12] Ed Blankenship, Martin Woodward, Grant Holliday, and Brian Keller. *Professional team foundation server 2012*. John Wiley & Sons, 2012.

[Cab12] Jordi Cabot. Object Constraint Language (OCL) tutorial. `https://modeling-languages.com/ocl-tutorial/`, 2012. Accessed: 2018-07-31.

[Cab14] Jordi Cabot. Clarifying concepts: MBE vs MDE vs MDD vs MDA. `https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/`, 2014. Accessed: 2018-07-31.

[Car96] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[CBCP02] Geoff Coulson, Gordon S Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *CAV*, 2014.

[CCF55] Abraham Charnes, William W Cooper, and Robert O Ferguson. Optimal estimation of executive compensation by linear programming. *Management science*, 1(2):138–151, 1955.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[CCLS11] Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, and Séverine Sentilles. Evolution management of extra-functional properties in component-based embedded systems. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, pages 93–102. ACM, 2011.

[CCW+05] Po-Han Chen, Lu Cui, Caiyun Wan, Qizhen Yang, Seng Kiong Ting, and Robert L.K. Tiong. Implementation of IFC-based web server for collaborative building design between architects and structural engineers. *Automation in Construction*, 14(1):115 – 128, 2005.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CFJ+10] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark-Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. *11 The EAST-ADL Architecture Description Language for Automotive Embedded Software*, pages 297–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[CG14] Srdjan Capkun and Frank Gürkaynak. Using Verilog for Testbenches (Adapted from Digital Design and Computer Architecture, David Money Harris & Sarah L. Harris, 2007 Elsevier). `http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/14_Verilog_Testbenches.pdf`, 2014. Accessed: 2018-07-31.

[CGL+03] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. Documenting software architectures: views and beyond. In *Proceedings of the 25th International Conference on Software Engineering*, pages 740–741. IEEE Computer Society, 2003.

[Cha96] Matthew Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Visualization'96. Proceedings.*, pages 127–131. IEEE, 1996.

[Cha18] ChangeVision, Inc. Astah Quick Start Guide. `http://astah.net/tutorial/pro/quick-start-guide`, 2018. Accessed: 2018-07-31.

[Che16] Chih-Hong Cheng. autoCode4 integrated inside Ptolemy II (ver. 11.0.devel). `https://youtu.be/ImSHmsnUyeA?t=34s`, 2016. Accessed: 2018-07-31.

[CKK01] Eun Sook Cho, Min Sun Kim, and Soo Dong Kim. Component metrics to measure component quality. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 419–426. IEEE, 2001.

[Cla18a] Clarity. ARCADIA a tooled method to Define, Analyse, Design & Validate System, Software, Hardware Architectures. `http://polarsys.org/capella/resources/Datasheet_Arcadia.pdf`, 2018. Accessed: 2018-07-31.

[Cla18b] Clarity. Ecosystem for the Model Based Systems Engineering Solution Capella. `http://www.clarity-se.org/`, 2018. Accessed: 2018-07-31.

[CM78] Joseph P. Cavano and James A. McCall. A Framework for the Measurement of Software Quality. *SIGSOFT Softw. Eng. Notes*, 3(5):133–139, January 1978.

[CMST03] J. A. Cooley, J. L. Mineweaser, L. D. Servi, and E. T. Tsung. Software-based erasure codes for scalable distributed storage. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, pages 157–164, April 2003.

[CN15] David Cutting and Joost Noppen. An extensible benchmark and tooling for comparing reverse engineering approaches. *International Journal on Advances in Software*, 8(1&2):115–124, 2015.

[Con12] Mirko Conrad. Verification and Validation According to ISO 26262: A Workflow to Facilitate the Development of High-Integrity Software. In *Embedded Real Time Software and Systems (ERTS2 2012)*, 2012.

[CRJ12] Ravi Chugh, Patrick M Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *ACM SIGPLAN Notices*, volume 47, pages 231–244. ACM, 2012.

[CSM+79] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering*, SE-5(2):96–104, March 1979.

[CV16] Daniel Calegari and Marcos Viera. On the Functional Interpretation of OCL. In *OCL@ MoDELS*, pages 33–48, 2016.

[Dai13a] Daimler AG. Original Requirement Documentation of ADAS version 1. `http://www.se-rwth.de/materials/cncviewscasestudy/SystemClusterSPESDemonstratorStufe1_220813.pdf`, 2013. Accessed: 2018-07-31.

[Dai13b] Daimler AG. Original Requirement Documentation of ADAS version 4. `http://www.se-rwth.de/materials/cncviewscasestudy/SystemClusterSPESDemonstratorStufe4_220813.pdf`, 2013. Accessed: 2018-07-31.

[Dai13c] Daimler AG. Simulink Web Export of Subsystem ADASv1 DEMO_FAS_Funktion. `http://www.se-rwth.de/materials/cncviewscasestudy/ADASv1/webview.html#6`, 2013. Accessed: 2018-07-31.

[Dai13d] Daimler AG. Simulink Web Export of Subsystem ADASv2 CC_ChangeSetValue_Lvl2_Repeater. `http://www.se-rwth.de/materials/cncviewscasestudy/ADASv2/webview.html#96`, 2013. Accessed: 2018-07-31.

[Dai13e] Daimler AG. Simulink Web Export of Subsystem ADASv4 CC_ChangeSetValue_Lvl2_Repeater. `http://www.se-rwth.de/materials/`

cncviewscasestudy/ADASv4/webview.html#107, 2013. Accessed: 2018-07-31.

[Dai13f] Daimler AG. Simulink Web Export of Subsystem ADASv4 CC_On_Off. http://www.se-rwth.de/materials/cncviewscasestudy/ADASv4/webview.html#97, 2013. Accessed: 2018-07-31.

[Dai13g] Daimler AG. Simulink Web Export of Subsystem ADASv4 DEMO_FAS_Funktion. http://www.se-rwth.de/materials/cncviewscasestudy/ADASv4/webview.html#6, 2013. Accessed: 2018-07-31.

[Dai13h] Daimler AG. Simulink Web Export of Subsystem ALS Adaptives_Fernlicht. http://www.se-rwth.de/materials/cncviewscasestudy/ExteriorLightSystem/webview.html#70, 2013. Accessed: 2018-07-31.

[Dai13i] Daimler AG. Simulink Web Export of Subsystem ALS DEMO_Aussenlicht_Funktion. http://www.se-rwth.de/materials/cncviewscasestudy/ExteriorLightSystem/webview.html#11, 2013. Accessed: 2018-07-31.

[Dai13j] Daimler AG. Simulink Web Export of Subsystem ALS Scheinwerfer. http://www.se-rwth.de/materials/cncviewscasestudy/ExteriorLightSystem/webview.html#45, 2013. Accessed: 2018-07-31.

[Dai13k] Daimler AG. Translated Requirement Documentation of ADAS version 4 (translated by TU Munich). http://www.se-rwth.de/materials/cncviewscasestudy/SystemClusterSPESDemonstratorStufe4_220813_en.pdf, 2013. Accessed: 2018-07-31.

[Dai18a] Daimler AG. Investor Relations Release 1. Februar 2018. https://www.daimler.com/dokumente/investoren/nachrichten/kapitalmarktmeldungen/daimler-mercedes-benz-ir-release-de-20180201.pdf, 2018. Accessed: 2018-07-31.

[Dai18b] Daimler AG. Mercedes-Benz Innovations. https://www.mercedes-benz.com/en/mercedes-benz/innovation, 2018. Accessed: 2018-07-31.

[Dai18c] Daimler AG. S-Class. https://www.mbusa.com/vcm/MB/DigitalAssets/Digital_Brochures/2018_S_Class/2018-S-Class.pdf, 2018. Accessed: 2018-07-31.

[Dal18] Baran Dalgic. EmbeddedMontiArc with ROS connector for Gazebo simulator. https://youtu.be/DNtrR6mxxsk, 2018. Accessed: 2018-07-31.

[Dat17] DataStax. Apache Cassandra 2.0: Configuring data consistency. `https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html`, 2017. Accessed: 2018-07-31.

[Dat18] Refsnes Data. JavaScript Objects. `https://www.w3schools.com/js/js_objects.asp`, 2018. Accessed: 2018-07-31.

[Dau07] JM Dautelle. JScience: Java tools and libraries for the advancement of science. Technical report, Technical report, http://www. jscience. org/, accessed 2007, 2007.

[DCB+15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 25–36. ACM, 2015.

[DDE+17] Jens Dankert, Christian Dernehl, Lutz Eckstein, Stefan Kowalewski, Evgeny Kusmenko, and Bernhard Rumpe. RapidCoop - Robuste Architektur durch geeignete Paradigmen für Kooperativ Interagierende Automobile. In *Automatisiertes und Vernetztes Fahren (AAET'17)*, February 2017.

[DEISS09] Ernesto Damiani, Nabil El Ioini, Alberto Sillitti, and Giancarlo Succi. Ws-certificate. In *Services-I, 2009 World Conference on*, pages 637–644. IEEE, 2009.

[Den76] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 2019.

[DHJ+08] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, pages 603–612. ACM, 2008.

[Die17] Erik Dietrich. Marker Interface Isn't a Pattern or a Good Idea. `https://blog.ndepend.com/marker-interface-isnt-pattern-good-idea/`, 2017. Accessed: 2018-07-31.

[DJP04] Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau. Applying CLP to Predict Extra-Functional Properties of Component-Based Models. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming*, pages 454–455, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[dLdCGR06] R. de Lemos, P. A. de Castro Guerra, and C. M. F. Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, March 2006.

[DNCH10] H. Dong, B. Ning, B. Cai, and Z. Hou. Automatic Train Control System Development and Simulation for High-Speed Railways. *IEEE Circuits and Systems Magazine*, 10(2):6–18, Secondquarter 2010.

[DPGA10] Massimiliano Di Penta, Daniel M German, and Giuliano Antoniol. Identifying licensing of jar archives using a code-search approach. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 151–160. IEEE, 2010.

[DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[DSW+03] Werner Damm, Christoph Schulte, Hartmut Wittke, Marc Segelken, Uwe Higgen, and Michael Eckrich. Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. *GI Jahrestagung (1)*, 34:340–344, 2003.

[DvdHT01] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A highly-extensible, XML-based architecture description language. In *Conference on Software Architecture*, 2001.

[DVM+05] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. Boosting re-use of embedded automotive applications through rich components. *Proceedings of Foundations of Interface Technologies*, 2005, 2005.

[DW02] O. Das and C. M. Woodside. Modeling the coverage and effectiveness of fault-management architectures in layered distributed systems. In *Proceedings International Conference on Dependable Systems and Networks*, pages 745–754, June 2002.

[DW09] Birgit Demuth and Claas Wilke. Model and object verification by using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia*, pages 687–690, 2009.

[EA15] Philippe Esling and Carlos Agon. Introduction to SCADE). `https://www-master.ufr-info-p6.jussieu.fr/2015/spip.php?action=acceder_document&arg=19736&cle=173fc1e70493c48df87b6e10bce8d808a7738a0d&file=pdf/Cours_1.pdf`, 2015. Accessed: 2018-07-31.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[ecm18] ecma INTERNATIONAL. ECMA-262, 9th edition, June 2018 ECMAScript 2018 Language Specification. `http://www.ecma-international.org/ecma-262/9.0/index.html`, 2018. Accessed: 2018-07-31.

[ECSG09] Huascar Espinoza, Daniela Cancila, Bran Selic, and Sébastien Gérard. Challenges in combining SysML and MARTE for model-based design of embedded systems. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 98–113. Springer, 2009.

[EDG+05] Huáscar Espinoza, Hubert Dubois, Sébastien Gérard, Julio L. Medina Pasaje, Dorina C. Petriu, and C. Murray Woodside. Annotating UML models with non-functional properties for quantitative analysis. In *MoDELS Int. Workshops*, pages 79–90, 2005.

[Edw00] Stephen H. Edwards. Chapter 3: Semantics. `http://www.cs.sfu.ca/CourseCentral/383/dma/notes/chapter3_semantics.pdf`, 2000. Accessed: 2018-07-31.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for Java. In *ACM SIGPLAN Notices*, volume 48, pages 112–121. ACM, 2012.

[EJL+03] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[EMO99] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica-a language for physical system modeling, visualization and interaction. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 630–639. IEEE, 1999.

[EMOW07] Vina Ermagan, Jun-ichi Mizutani, Kentaro Oguchi, and David Weir. Towards Model-Based Failure-Management for Automotive Software. In *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, SEAS '07, pages 8–, Washington, DC, USA, 2007. IEEE Computer Society.

[ERA09] Markus Eisenhauer, Peter Rosengren, and Pablo Antolin. A development platform for integrating wireless devices and sensors into ambient intelligence systems. In *Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops' 09. 6th Annual IEEE Communications Society Conference on*, pages 1–3. IEEE, 2009.

[Ern16] David Ernst. *Transformation von MontiArc-Modellen zu Kontrollflussgraphen*. Bachelor Thesis at RWTH Aachen University, 2016.

[Est10] Esterel Technologies. SCADE Suite Design Notations Quick Reference. `http://www.esterel-technologies.com/wp-content/uploads/2013/02/SCADE-Reference-card.pdf`, 2010. Accessed: 2018-07-31.

[Est14] Esterel Technologies. Technical Data Sheet SCADE Suite(R) 17.0. `http://www.esterel-technologies.com/wp-content/uploads/2014/02/SCADE-Suite-Technical-Datasheet.pdf`, 2014. Accessed: 2018-07-31.

[EVDSV+13] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[EWSG94] Steven D Eppinger, Daniel E Whitney, Robert P Smith, and David A Gebala. A model-based method for organizing tasks in product development. *Research in engineering design*, 6(1):1–13, 1994.

[FALW14] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wąsowski. Sound Merging and Differencing for Class Diagrams. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, pages 63–78, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[FB11] Bent Flyvbjerg and Alexander Budzier. Why your IT project may be riskier than you think. *Harvard Business Review*, 89(9):601–603, 2011.

[Fei05] Peter Feiler. Product Line Modeling With AADL. `http://www.aadl.info/aadl/downloads/papers/ProductLines.pdf`, 2005. Accessed: 2018-07-31.

[Fei10] Peter Feiler. SAE AADL V2: An Overview. `https://wiki.sei.cmu.edu/aadl/images/7/73/AADLV2Overview-AADLUserDay-Feb_2010.pdf`, 2010. Accessed: 2018-07-31.

[FG07] Lidia Fuentes and Nadia Gámez. Adding Aspects to xADL 2.0 for Software Product Line Architectures. In *VaMoS*, pages 87–96, 2007.

[FG12] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[Fir04] Donald Firesmith. Engineering safety requirements, safety constraints, and safety-critical requirements. *Journal of Object technology*, 3(3):27–42, 2004.

[FKSH09] Jan Friedrich, Marco Kuhrmann, Marc Sihling, and Ulrike Hammerschall. *Das v-modell xt*. Springer, 2009.

[Flo02] Gert Florijn. *Revjava-design critiques and architectural conformance checking for java software*. White Paper. SERC, the Netherlands, 2002.

[FLV03] Peter H Feiler, Bruce Lewis, and Steve Vestal. The SAE avionics architecture description language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. Technical report, ARMY AVIATION AND MISSILE COMMAND REDSTONE ARSENAL AL, 2003.

[FLV06] P. H. Feiler, B. A. Lewis, and S. Vestal. The SAE Architecture Analysis amp; Design Language (AADL) a standard for engineering performance critical systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211, Oct 2006.

[FMS11] Sanford. Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. The MK/OMG Press. Elsevier Science, 2011.

[FOW01] Clemens Fischer, Ernst-Rüdiger Olderog, and Heike Wehrheim. A CSP view on UML-RT structure diagrams. In *International Conference on Fundamental Approaches to Software Engineering*, pages 91–108. Springer, 2001.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, pages 37–54, May 2007.

[Gac16] Gary Gack. Core Set of Effectiveness Metrics for Software and IT. `https://www.isixsigma.com/methodology/metrics/core-set-effectiveness-metrics-software-and-it/`, 2016. Accessed: 2018-07-31.

[Gar12] F Garret. LabVIEW-Multisim Co-Simulation with Variants and Hierarchical Blocks (Part 1). `https://forums.ni.com/t5/National-Instruments-Circuit/LabVIEW-Multisim-Co-Simulation-with-Variants-and-Hierarchical/ba-p/3490092`, 2012. Accessed: 2018-07-31.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1):27 – 34, 2007. Special issue on Experimental Software and Toolkits.

[GD07] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[GG18] Sergey Gershtein and Anna Gershtein. The Complete List of Measurement Units Supported. `https://www.convert-me.com/en/unitlist.html`, 2018. Accessed: 2018-07-31.

[GHK+07] Hans Grönninger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based modeling of function nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4)*, 2007.

[GHK+08a] Hans Grönninger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhart, and Bernhard Rumpe. Modelling automotive function nets with views for features, variants, and modes. In *4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08b] Hans Grönninger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhart, and Bernhard Rumpe. View-centric modeling of automotive logical architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, 2008.

[Gho10] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GM03] Jürgen Gausemeier and Stefan Moehringer. New Guideline Vdi 2206-a Flexible Procedure Model for the Design of Mechatronic Systems. In *DS 31: Proceedings of ICED 03, the 14th International Conference on Engineering Design, Stockholm*, 2003.

[GMPO09] Roy Grønmo, Birger Møller-Pedersen, and Gøran K Olsen. Comparison of three model transformation languages. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 2–17. Springer, 2009.

[GMR15] Timo Greifenberg, Klaus Müller, and Bernhard Rumpe. Architectural Consistency Checking in Plugin-Based Software Systems. In *European Conference on Software Architecture Workshops (ECSAW'15)*, pages 58:1–58:7. ACM, 2015.

[GMW00a] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, 68:47–68, 2000.

[GMW00b] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, 2000.

[Gör17] Philipp Görick. *Matrix Properties in Type Systems of Component and Connector Based Languages*. Bachelor Thesis at RWTH Aachen University, 2017.

[Gos12] Andreas Goser. MATLAB Answers: Clean up Simulink block diagram. `https://de.mathworks.com/matlabcentral/answers/30016-clean-up-simulink-block-diagram`, 2012. Accessed: 2018-07-31.

[Gre07] Aaron Greenhouse. An Overview of AADL Properties. `https://wiki.sei.cmu.edu/aadl/index.php/An_Overview_of_AADL_Properties#Unit_Types`, 2007. Accessed: 2018-07-31.

[Gri84] Richard Grigonis. FIFTH-GENERATION COMPUTERS. In *Digital Deli: The Comprehensive, User-Lovable Menu of Computer Lore, Culture, Lifestyles and Fancy*, 1984. Accessed: 2018-07-31.

[Gri06] Robert Grimm. Better extensibility through modular syntax. In *ACM SIGPLAN Notices*, volume 41, pages 38–51. ACM, 2006.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[Gru07] Lars Grunske. Early quality prediction of component-based systems - A generic framework. *Journal of Systems and Software*, 80(5):678–686, 2007.

[Gül14] Tim Gülke. *Erweiterung des Anforderungsmanagement-Fokus*. Aachener Informatik-Berichte, Software Engineering, Band 18. Shaker Verlag, Oktober 2014.

[GV99] Abhijit Ghosh and Ranga Vemuri. Formal verification of synthesized analog designs. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 40–45. IEEE, 1999.

[GV06] Holger Giese and Alexander Vilbig. Separation of non-orthogonal concerns in software architecture and design. *Software & Systems Modeling*, 5(2):136–169, 2006.

[HA+03] Qingfeng He, Annie I Antón, et al. A framework for modeling privacy requirements in role engineering. In *Proc. of REFSQ*, volume 3, pages 137–146, 2003.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Hal18] Philipp Haller. Modeling a Controler for Super Mario Bros. `https://youtu.be/LZ3rp8KgdHI`, 2018. Accessed: 2018-07-31.

[HB06] Matthias Heindl and Stefan Biffl. Risk management with enhanced tracing of requirements rationale in highly distributed projects. In *Proceedings of the 2006 international workshop on Global software development for the practitioner*, pages 20–26. ACM, 2006.

[HBB+94] Wolfgang Hesse, Georg Barkow, Hubert von Braun, Hans-Bernd Kittlaus, and Gert Scheschonk. Terminologie der Softwaretechnik, Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 1: Begriffssystematik und Grundbegriffe. *Informatik Spektrum*, 17(1):39–47, 1994.

[HBL99] Pieter H. Hartel, Michael J. Butler, and Moshe Levy. *The Operational Semantics of a Java Secure Processor*, pages 313–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[HDP14] Myron Hecht, Emily Dimpfl, and Julia Pinchak. Automated Generation of Failure Modes and Effects Analysis from SysML Models. In *Software Reliability*

Engineering Workshops (ISSREW), 2014 IEEE International Symposium on, pages 62–65. IEEE, 2014.

[Hei18] Malte Heithoff. Modeling PacMan with EmbeddedMontiArc. `https://youtu.be/f7YKCsSB_Tg`, 2018. Accessed: 2018-07-31.

[Hel18] Alexander Hellwig. Connecting an EMAM Component to ROS with the EMAM2Middleware generator. `https://youtu.be/uKNIzIeMcy8`, 2018. Accessed: 2018-07-31.

[HF10] Florian Hölzl and Martin Feilkas. *13 AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems*, pages 317–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[HH18] Philipp Haller and Malte Heithoff. A Case Study of the Component and Connector Modeling Language EmbeddedMontiArc. `http://www.se-rwth.de/staff/vonwenckstern/A.Case.Study.of.the.Component.and.Connector.Modeling.Language.EmbeddedMontiArc-Seminar.pdf` , 2018. Accessed: 2018-07-31.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating domain-specific transformation languages for component & connector architecture descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, 2015.

[HHW99] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 174–183, New York, NY, USA, 1999. ACM.

[Hil00] Rich Hilliard. Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems. *IEEE, http://standards. ieee. org*, 12(16-20):2000, 2000.

[HJLGP99] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac. UMLAUT: An Extendible UML Transformation Framework. In *14th IEEE International Conference on Automated Software Engineering*, page 275. IEEE, 1999.

[HK12] Abdulhamed Moh Suliman Hwas and Reza Katebi. Wind turbine control using PI pitch angle controller. In *IFAC Conference on Advances in PID Control (PID'12)*, 2012.

[HKK+18] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, Georg Strobl, and Michael von Wenckstern. Model-Based Development of Self-Adaptive Autonomous Vehicles using the

SMARDT Methodology. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 163 – 178. SciTePress, January 2018.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+11a] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR+11b] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Towards a Family-based Analysis of Applicability Conditions in Architectural Delta Models. In *Variability for You Workshop (VARY)*, IT University Technical Report Series TR-2011-144, pages 43–52, 2011.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, volume 9839 of *LNCS*, pages 175–182. Springer, December 2016.

[HM02] Jan Heering and Marjan Mernik. Domain-specific languages for software engineering. In *Prec. of the 35th Hawaii International Conference on System Sciences*, 2002.

[HMPO+08] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *2008 12th International Software Product Line Conference*, pages 139–148, Sept 2008.

[Ho18] Don Ho. Notepad++. `https://notepad-plus-plus.org/`, 2018. Accessed: 2018-07-31.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[Hou13] Pete Houston. *Instant jsoup How-to*. Packt Publishing Ltd, 2013.

[HR00] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuf. Technical Report MCS00-16, Weizmann Institute, Rehovot, Israel, 2000.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, volume 2010-01 of *Informatik-Bericht*, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, pages 181–192, 2012.

[HRRW17] Katrin Hölldobler, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Advances in Modeling Language Engineering. In *International Conference on Model and Data Engineering*, LNCS 10563, pages 3–17. Springer, October 2017.

[HRvW17] Malte Heithoff, Bernhard Rumpe, and Michael von Wenckstern. Anforderungsverikation von Komponenten- und Konnektormodellen am Beispiel Autonom Fahrender Autos. *GI Softwaretechnik-Trends*, 37(2), May 2017.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW16] Robert Heim, Bernhard Rumpe, and Andreas Wortmann. Extending Architecture Description Languages With Exchangeable Component Behavior Languages. In *Conference on Software Engineering & Knowledge Engineering (SEKE'16)*, pages 1–6. KSI Research Inc., Fredericton, Canada, June 2016.

[HSSG16] Frank Hilken, Marcel Schuster, Karsten Sohr, and Martin Gogolla. Integrating UML/OCL Derived Properties into Validation and Verification Processes. In *OCL@ MoDELS*, pages 89–104, 2016.

[HWP15] Adolfo Sánchez-Barbudo Herrera, Edward D Willink, and Richard F Paige. An OCL-based Bridge from Concrete to Abstract Syntax. In *OCL@ MoDELS*, pages 19–34, 2015.

[IBM13] IBM. Engineering Safety Critical Automotive Systems Complying with ISO 26262. `https://www.ibm.com/developerworks/community/files/form/anonymous/api/library/9e66f5de-701e-4994-9291-75646d558240/document/1b24c0ea-2bef-4316-a171-6904ad5839e9/media`, 2013. Accessed: 2018-07-31.

[IBM18] IBM. IBM Rational Rhapsody Architect for Software. `https://www.ibm.com/de-de/marketplace/architect-for-software`, 2018. Accessed: 2018-07-31.

[Ilo18a] Petyo Ilov. Simulating several Cars. `https://youtu.be/OFXWg8o3ni8`, 2018. Accessed: 2018-07-31.

[Ilo18b] Petyo Ilov. *Software architecture of distributed multi-user simulation of autonomously driving vehicles*. Master Thesis at RWTH Aachen University, 2018.

[INC07] INCOSE, SE. Vision 2020 (INCOSE-TP-2004-004-02), 2007.

[Ins15] Institut Superieur de l'Aeronautique et de l'Espace. The Architecture Analysis and Design Language: an overview. `http://www.openaadl.org/downloads/tutorial_models15/part1_introducing_aadl.pdf`, 2015. Accessed: 2018-07-31.

[Int11] International Organization for Standardization. ISO 26262-6:2011 Road vehicles – Functional safety – Part 6: Product development at the software level. `https://www.iso.org/standard/51362.html`, 2011. Accessed: 2018-07-31.

[ITC+18] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones. In *European Conference on Computer Vision*, pages 288–314. Springer, 2018.

[JB16] Frédéric Jouault and Olivier Beaudoux. Efficient OCL-based Incremental Transformations. In *OCL@ MoDELS*, pages 121–136, 2016.

[JCJ+11] Seo-Hyun Jeon, Jin-Hee Cho, Yangjae Jung, Sachoun Park, and Tae-Man Han. Automotive hardware development according to ISO 26262. In *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, pages 588–592. IEEE, 2011.

[JH05] Anjali Joshi and Mats PE Heimdahl. Model-based safety analysis of simulink models using SCADE design verifier. In *International Conference on Computer Safety, Reliability, and Security*, pages 122–135. Springer, 2005.

[JH17] Arne N. Johanson and Wilhelm Hasselbring. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering*, 22(4):2206–2236, Aug 2017.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 128–138, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[JSH13] Reiner Jung, Christian Schneider, and Wilhelm Hasselbring. Type Systems for Domain-specific Languages. In Stefan Wagner and Horst Lichter, editors, *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 26. Februar - 1. März 2013 in Aachen*, volume 215 of *LNI*, pages 139–154. GI, 2013.

[Jur15] Benajmin Jurke. Compile-time numerical unit dimension checking in C++11. `https://benjaminjurke.com/content/articles/2015/compile-time-numerical-unit-dimension-checking/`, 2015. Accessed: 2018-07-31.

[JYP$^+$17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[Kah17a] Fabian Kahlert. Component and Connectors Views: Definition, Verification, Witnesses. `https://youtu.be/bzbKWqcYcxA`, 2017. Accessed: 2018-07-31.

[Kah17b] Fabian Kahlert. *Extension of the C&C View Language and its Verification for Embedded Systems*. Bachelor Thesis at RWTH Aachen University, 2017.

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *International Conference on Objects, Components, Models and Patterns*, pages 175–194. Springer, 2009.

[Kaz14] Dmitry A. Kazakov. UNITS OF MEASUREMENT FOR ADA: version 3.8. `http://www.dmitry-kazakov.de/ada/units.htm`, 2014. Accessed: 2018-07-31.

[KBFS12] Simon Frederick Königs, Grischa Beier, Asmus Figge, and Rainer Stark. Traceability in Systems Engineering - Review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Engineering Informatics*, 26(4):924 – 940, 2012.

[KDH+13] Carsten Kolassa, David Dieckow, Michael Hirsch, Uwe Creutzburg, Christian Siemers, and Bernhard Rumpe. Objektorientierte Graphendarstellung von Simulink-Modellen zur einfachen Analyse und Transformation. In *Tagungsband AALE 2013, 10. Fachkonferenz, Das Forum für Fachleute der Automatisierungstechnik aus Hochschulen und Wirtschaft*, 2013.

[KH08] Moran Kupfer and Irit Hadar. Understanding and Representing Deployment Requirements for Achieving Non-Functional System Properties. In *The 1st International Workshop on Non-functional System Properties in Domain-Specific Modeling Languages, Toulouse, France*. Citeseer, 2008.

[KK00] Nasser Kehtarnavaz and Mansour Keramat. *DSP system design: using the TMS320C6000*. Prentice Hall PTR, 2000.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR09] Matt Kaufmann, Jacob Kornerup, and Mark Reitblatt. Formal Verification of LabVIEW Programs Using the ACL2 Theorem Prover. In *ACL2*, 2009.

[KKRvW18] Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Michael von Wenckstern. Finding Inconsistencies in Design Models and Requirements by Applying the SMARDT Process. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18)*, Univ. Hamburg, April 2018.

[KLH+02] Antti Karola, Hannu Lahtela, Reijo Hänninen, Rob Hitchcock, Qingyan Chen, Stephen Dajka, and Kim Hagström. BSPro COM-Server?interoperability between software tools using industrial foundation classes. *Energy and Buildings*, 34(9):901 – 907, 2002. A View of Energy and Bilding Performance Simulation at the start of the third millennium.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMS+17] Stefan Kriebel, Vincent Moyses, Georg Strobl, Johannes Richenhagen, Phillip Orth, Stefan Pischinger, Christoph Schulze, Timo Greifenberg, and Bernhard Rumpe. The Next Generation of BMW's Electrified Powertrains: Providing Software Features Quickly by Model-Based System Design. In *26th Aachen Colloquium Automobile and Engine Technology*, October 2017.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KPMS01] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd. What accuracy statistics really measure [software estimation]. *IEE Proceedings - Software*, 148(3):81–85, June 2001.

[KPP06a] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, volume 20062, page 200, 2006.

[KPP06b] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (EOL). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer, 2006.

[KPP06c] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Towards using OCL for instance-level queries in domain specific languages. *OCL for (Meta-) Models in Multiple Application Domains*, pages 26–37, 2006.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[Kra18] Guido Kramann. Der Windup-Effekt bei Reglern mit begrenzten Stellgrößen. `http://www.kramann.info/62_Regelungssysteme/11_Stabilitaet/03_Windup/`, 2018. Accessed: 2018-07-31.

[KRGDP18] Dimitris Kolovos, Louis Rose, Antonio Garcia-Dominguez, and Richard Paige. The Epsilon Book. `http://www.eclipse.org/epsilon/doc/book/`, 2018. Accessed: 2018-07-31.

[KRR15] Carsten Kolassa, Holger Rendel, and Bernhard Rumpe. Evaluation of Variability Concepts for Simulink in the Automotive Domain. In *System Sciences Conference (HICSS'15)*, pages 5373–5382. IEEE, 2015.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[KRRvW18] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. Supplementary Material: HighEmbeddedMontiArc: Textual Modeling Alternative to Simulink. `http://www.se-rwth.de/materials/embeddedmontiarc/`, 18. Accessed: 2018-07-31.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRRvW18] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc: Textual Modeling Alternative to Simulink. In *Proceedings of MODELS 2018. Workshop EXE*, October 2018.

[KRSvW18] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Supplementary Material: Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. `http://www.se-rwth.de/materials/ema_compiler/`, 18. Accessed: 2018-07-31.

[KRSvW18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*. IEEE, October 2018.

[KRSvW18b] Evgeny Kusmenko, Bernhard Rumpe, Ievgen Strepkov, and Michael von Wenckstern. Teaching Playground for C&C Language EmbeddedMontiArc. In *Proceedings of MODELS 2018. Workshop ModComp*, October 2018.

[Kru95] Philippe Kruchten. Architectural blueprints?the "4+ 1" view model of software architecture. *Tutorial Proceedings of Tri-Ada*, 95:540–555, 1995.

[KRV07] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KSRvW18] Evgeny Kusmenko, Igor Shumeiko, Bernhard Rumpe, and Michael von Wenckstern. Fast Simulation Preorder Algorithm. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 256 – 267. SciTePress, January 2018.

[KTB+07] Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, Sabine Rittmann, and Martin Wechs. COLA – The Component Language. Technical report, Technical University Munich, 2007.

[Kug12] Stefan Kugele. *Model-Based Development of Software-intensive Automotive Systems*. PhD thesis, Technical University Munich, 2012.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[Kur16] Kimio Kuramitsu. Fast, flexible, and declarative construction of abstract syntax trees with PEGs. *Journal of Information Processing*, 24(1):123–131, 2016.

[Kur17] Don Kurelich. Mentor Keynote: Systems of Systems ? What?s the Story? `http://go.mentor.com/4PLF7`, 2017. Accessed: 2018-07-31.

[Lee04] Edward Lee. Concurrent Models of Computation for Embedded Software. `https://ptolemy.berkeley.edu/projects/embedded/concurrency/lectures/ExtendingPtolemyII.pdf`, 2004. Accessed: 2018-07-31.

[Lee08] Edward A Lee. Cyber physical systems: Design challenges. In *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[Lee13] Edward Lee. Ptolemy Tutorial - Creating Directors. `https://youtu.be/Xo9N9mPIiiA`, 2013. Accessed: 2018-07-31.

[Lei17] Antonio Leiva. *Kotlin for Android Developers*. LeanPub, 2017.

[LEK13] Andrea Leitner, Wolfgang Ebner, and Christian Kreiner. Mechanisms to Handle Structural Variability in MATLAB/Simulink Models. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse*, pages 17–31, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Lem16] Jürgen Lemke. *C++-Metaprogrammierung : Eine Einführung in die Präprozessor- und Template-Metaprogrammierung*. Springer, 2016.

[Leo16] David Leonard. *Technetium: Productivity Tracking for Version Control Systems*. Master Thesis at The City College of New York, 2016.

[LGC14] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):12, 2014.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Trans. Software Eng.*, 21(4):336–355, 1995.

[LKD+03] Heiko Ludwig, Alexander Keller, Asit Dan, Richard King, and Richard Franck. A Service Level Agreement Language for Dynamic Electronic Services. *Electronic Commerce Research*, 3(1):43–59, Jan 2003.

[LM12] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.

[Log07] Francesco Logozzo. Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verification of Java Classes. In *VMCAI*, 2007.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[Lor17] Mike Lorang. Evolutionary Tuning of PID Controllers. `https://youtu.be/7llpVLklnPY`, 2017. Accessed: 2018-07-31.

[LS11] Thomas Leveque and Séverine Sentilles. Refining extra-functional property values in hierarchical component models. In *CBSE*, 2011.

[LSLGG+11] Thierry Le Sergent, Alain Le Guennec, Sebastien Gerard, Yann Tanguy, and Francois Terrier. Using SCADE System for the Design and Integration of Critical Systems. Technical report, SAE Technical Paper, 2011.

[LTW14] Delphine Longuet, Frédéric Tuong, and Burkhart Wolff. Towards a Tool for Featherweight OCL: A Case Study On Semantic Reflection. In *OCL@ MoDELS*, pages 43–52, 2014.

[Luc96] David Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Technical Report CSL-TR-96-705, Stanford University, 1996.

[Luk16] Wojtek Lukaszuk. Summary of 'Clean code' by Robert C. Martin. `https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29`, 2016. Accessed: 2018-07-31.

[LWL04] Yi-zhi D. Candidate Liang, Yan-zhang Wang, and Yun-fei Liu. The formal semantics of an UML activity diagram. *Journal of Shanghai University (English Edition)*, 2004.

[LZPH09] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In *International Conference on Reliable Software Technologies*, pages 237–250. Springer, 2009.

[MA02] Daniel A Menasce and Virgilio AF Almeida. *Capacity Planning for Web Services: metrics, models, and methods*. Prentice Hall PTR Upper Saddle River, NJ, 2002.

[Man13] Peter Manhart. Systematische Rekonfiguration eingebetteter softwarebasierter Fahrzeugsysteme auf Grundlage formalisierbarer Kompatibilitätsdokumentation und merkmalbasierter Komponentenmodellierung. In Stefan Kowalewski and Bernhard Rumpe, editors, *Software Engineering 2013: Fachtagung des GI-Fachbereichs Softwaretechnik, 26. Februar - 2. März 2013 in Aachen*, volume 213 of *LNI*, pages 301–317. GI, 2013.

[Man15] Peter Manhart. Schlussbericht SPES_XT : Software Plattform Embedded Systems 2020. Technical Report DOI:10.2314/GBV:87150491X, Daimler AG, 2015.

[Mar09] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[MBNJ09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE software*, 20(5):14–18, 2003.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A framework for generating domain-specific property languages. In *International Conference on Software Language Engineering*, pages 1–20. Springer, 2014.

[ME15] Patrick Mäder and Alexander Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441, 2015.

[Meh17a] Ferdinand Mehlan. Defining Semantics of Extra-Functional-Properties in Component and Connector Models with OCL, 2017.

[Meh17b] Ferdinand Mehlan. *Verification of Non-Functional Properties on Component and Connector Models*. Bachelor Thesis at RWTH Aachen University, 2017.

[Men18a] Mentor. Mentor Capital Design. `https://www.mentor.com/products/electrical-design-software/capital/design`, 2018. Accessed: 2018-07-31.

[Men18b] Mentor. Mentor Capital HarnessXC: Rapidly create detailed, manufacturing-ready harness designs. `https://www.mentor.com/products/electrical-design-software/capital/harness-xc`, 2018. Accessed: 2018-07-31.

[Men18c] Mentor. Mentor Capital Integrator: Rules-driven vehicle wiring synthesis. `https://www.mentor.com/products/electrical-design-software/capital/integrator`, 2018. Accessed: 2018-07-31.

[Men18d] Mentor. Mentor Capital Publisher: Create service documentation direct from existing wiring diagrams and other data sources. `https://www.mentor.com/products/electrical-design-software/capital/capital-publisher`, 2018. Accessed: 2018-07-31.

[Mer18] Mercurial community. Mercurial: free, distributed source control management tool. `https://www.mercurial-scm.org/`, 2018. Accessed: 2018-07-31.

[Mey86] Bertrand Meyer. def programming - quotes about coding. `http://www.defprogramming.com/quotes-by/bertrand-meyer/`, 1986. (Year of quote is guessed, it is the year of the first release of Eiffel). Accessed: 2018-07-31.

[MFZ+09] Cem Mengi, Christian Fuß, Ruben Zimmermann, Ismet Aktas, et al. Model-driven Support for Source Code Variability in Automotive Software Engineering. In *1st MAPLE Workshop*, pages 44–50, 2009.

[MH03] Vincent Massol and Ted Husted. *Junit in action*. Manning Publications Co., 2003.

[Mik17] Clark Mike. JDepend - Github. `https://github.com/clarkware/jdepend`, 2017. Accessed: 2018-07-31.

[MMM02] Tomoko Matsumura, Akito Monden, and Ken-ichi Matsumoto. A method for detecting faulty code violating implicit coding rules. In *Proceedings of the international workshop on Principles of software evolution*, pages 15–21. ACM, 2002.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, 2010.

[MMR14] Jose Miguel, David Mauricio, and Glen Rodriguez. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications*, 5(6):31–53, 2014.

[MMR+17] Shahar Maoz, Ferdinand Mehlan, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. OCL Framework to Verify Extra-Functional Properties in Component and Connector Models. In *Proceedings of MODELS 2017. Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'17)*, CEUR 2019, September 2017.

[MMY10] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. Packrat parsers can handle practical grammars in mostly constant space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 29–36. ACM, 2010.

[MNR+13] Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Konzepte zur erweiterung des spes meta-modells um aspekte der variabilitäts-und deltamodellierung. In *Software Engineering (Workshops)*, pages 283–292, 2013.

[Mod05] Modelica, Association. Modelica language specification. *Linköping, Sweden*, 2005.

[Mod17] Modelica, Association. Modelica - A Unified Object-Oriented Language for Systems Modeling: Language Specification. Technical Report Version 3.4, Modelica, 2017.

[Mok18] Armin Mokhtarian. Modeling an Autopilot for Self-Driving Cars with EmbeddedMontiArc. `https://youtu.be/i4DWrKFC9j4`, 2018. Accessed: 2018-07-31.

[Moo20] E Moors. On the reciprocal of the general algebraic matrix. *Bull. Amer. Math. Soc.*, 26:394–395, 1920.

[MPRS17] S. Maoz, N. Pomerantz, J. O. Ringert, and R. Shalom. Why is My Component and Connector Views Specification Unsatisfiable? In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 134–144, Sept 2017.

[MR13] Shahar Maoz and Bernhard Rumpe. Crosscutting Structural Views for Component and Connector Models. Proposal I-80-407.2-2013, Proposal for Application for Research Grant Regular Program, 2013.

[MR15] Shahar Maoz and Jan Oliver Ringert. GR (1) synthesis for LTL specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 96–106. ACM, 2015.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRvW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, volume 1723 of *CEUR Workshop Proceedings*, pages 19–24, October 2016.

[MSN11] Pedram Mir Seyed Nazari. *Architektur Alignment von Java Systemen*. Diploma Thesis at RWTH Aachen University, 2011.

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MT00] Nenad Medvidovic and Richard N Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering*, 26(1):70–93, 2000.

[MT10] Nenad Medvidovic and Richard N Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 471–472. ACM, 2010.

[Mül18] Klaus Müller. *Modellbasierte Unterstützung der Software Evolution im industriellen Kontext*. Aachener Informatik-Berichte, Software Engineering, Band 35. Shaker Verlag, July 2018.

[MVF00] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Advanced Research in Asynchronous Circuits and Systems, 2000.(ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 52–59. IEEE, 2000.

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

[Nag99] M Nagl. *Die Software-Programmiersprache ADA 95*. Vieweg-Verlag, Wiesbaden, 1999.

[Nat98] National Instruments. BridgeView and LabView: G Programming Reference Manual. Technical Report 321296B-01, National Instruments, 1998.

[Nat09] National Instruments. Automatische Bereinigung von LabVIEW-Blockdiagrammen. http://www.ni.com/tutorial/7386/de/, 2009. Accessed: 2018-07-31.

[Nat17a] National Instruments. LabView 2017 Help: Using Matrices. http://zone.ni.com/reference/en-XX/help/371361P-01/lvconcepts/using_matrices/, 2017. Accessed: 2018-07-31.

[Nat17b] National Instruments. Prove It Works: Using the Unit Test Framework for Software Testing and Validation. http://www.ni.com/white-paper/8082/en/, 2017. Accessed: 2018-07-31.

[Nin97] J. Q. Ning. Component-based software engineering (CBSE). In *Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies*, pages 34–43, June 1997.

[NLM18] Daye Nam, Youn Kyu Lee, and Nenad Medvidovic. EVA: a tool for visualizing software architectural evolution. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 53–56. ACM, 2018.

[No 18] No Magic. MagicDraw 19.0 LTR Documentation. https://docs.nomagic.com/display/MD190/User+Guide, 2018. Accessed: 2018-07-31.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, volume 1118 of *CEUR Workshop Proceedings*, pages 15–24, 2013.

[OAS08] OASIS. Solution Deployment Descriptor Specification 1.0 - OASIS Standard. http://docs.oasis-open.org/sdd/v1.0/os/sdd-spec-v1.0-os.pdf, 2008. Accessed: 2018-07-31.

[OBM05] Ciaran O'Reilly, David Bustard, and Philip Morrow. The War Room Command Console: Shared Visualizations for Inclusive Team Coordination. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 57–65, New York, NY, USA, 2005. ACM.

[Oez18] Ahmet Tayfun Oezen. *Evaluierung von Komponenten- und Konnektoren-Views*. Master Thesis at RWTH Aachen University, 2018.

[OLB16] Flavio Oquendo, Jair Leite, and Thaís Batista. *Software Architecture in Action*. Springer, 2016.

[OMB03] Ciaran O'Reilly, Philip Morrow, and David Bustard. Lightweight prevention of architectural erosion. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 59–64. IEEE, 2003.

[OMG05] OMG. 2.0 Specification. *Object Management Group, Final Adopted Specification*, 2005.

[OMG08] OMG. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2 (convenience document without change bars). Technical Report OMG Document Number: ptc/2008-06-09, OMG Group, 2008.

[OMG11] OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Technical Report Version 1.1, OMG Group, 2011.

[OMG15] OMG. OMG Systems Modeling Language (OMG SysML). Technical Report Version 1.4, OMG Group, 2015.

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 27(5):58–68, December 1993.

[Ora99] Oracle Corporation. Java Naming Conventions. `http://www.oracle.com/technetwork/java/codeconventions-135099.html`, 1999. Accessed: 2018-07-31.

[Ora17a] Oracle Corporation. ORACLE Java Documenation: Setting an Application's Entry Point. `https://docs.oracle.com/javase/tutorial/deployment/jar/appman.html`, 2017. Accessed: 2018-07-31.

[Ora17b] Oracle Corporation. ORACLE Java Documenation: Signing and Verifying JAR Files. `https://docs.oracle.com/javase/tutorial/deployment/jar/signindex.html`, 2017. Accessed: 2018-07-31.

[Ora17c] Oracle Corporation. ORACLE Java Documenation: Signing JAR Files. `https://docs.oracle.com/javase/tutorial/deployment/jar/signing.html`, 2017. Accessed: 2018-07-31.

[Ora17d] Oracle Corporation. ORACLE Java Documenation: Understanding Signing and Verification. `https://docs.oracle.com/javase/tutorial/deployment/jar/intro.html`, 2017. Accessed: 2018-07-31.

[Ora17e] Oracle Corporation. ORACLE Java Documenation: Working with Manifest Files: The Basics. `https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html`, 2017. Accessed: 2018-07-31.

[Ora17f] Oracle Corporation. ORACLE Java Documentation: Class SimpleDateFormat. `https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html`, 2017. Accessed: 2018-07-31.

[Ora17g] Oracle Corporation. ORACLE Java Documentation: Using Package Members. `https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html`, 2017. Accessed: 2018-07-31.

[OVM15] Marc Opijnen, Nico Verwer, and Jan Meijer. Beyond the experiment: the extendable legal link extractor. In *Workshop on Automated Detection, Extraction and Analysis of Semantic Information in Legal Texts, part of: International Conference on Artificial Intelligence and Law (ICAIL)*, 2015.

[Pag18] Bruno Pages. BOUML: Online HTML reference manual for release 7.6. `https://www.bouml.fr/doc/index.html`, 2018. Accessed: 2018-07-31.

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[PBKS07] A Pretschner, M Broy, I Krüger, and T Stauner. Software Engineering for Automotive Systems: A Roadmap. In *Future of Software Engineering*, 2007.

[PCSF08] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control*. O'Reilly Media, Inc., 2008.

[Pen55] Roger Penrose. A generalized inverse for matrices. In *Mathematical proceedings of the Cambridge philosophical society*, volume 51, pages 406–413. Cambridge University Press, 1955.

[PFR01] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison Wesley, First Edition December, 2001.

[PHP87] Daniel Pilaud, N Halbwachs, and JA Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[PI04] John G Proakis and Vinay K Ingle. *A self-study guide for digital signal processing*. Pearson Prentice Hall, 2004.

[Plo13] Michael Plotke. Kernel (image processing). `https://en.wikipedia.org/wiki/Kernel_(image_processing)`, 2013. Accessed: 2018-07-31.

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project*. Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.

[PMPdK15] Georgios Plataniotis, Qin Ma, Erik Proper, and Sybren de Kinderen. Traceability and modeling of requirements in enterprise architecture from a design rationale perspective. In *Research Challenges in Information Science (RCIS), 2015 IEEE 9th International Conference on*, pages 518–519. IEEE, 2015.

[PMRT18] Maria Pittou, Panagiotis Manolios, Jan Reineke, and Stavros Tripakis. Checking multi-view consistency of discrete systems with respect to periodic sampling abstractions. *Science of Computer Programming*, 167:1–24, 2018.

[PNR04] Sebastian Pavel, Jacques Noyé, and Jean-Claude Royer. Dynamic Configuration of Software Product Lines in ArchJava. In Robert L. Nord, editor, *Software Product Lines*, pages 90–109, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Pol18] PolarSys. LET YOURSELF BE GUIDED WITH ARCADIA: A comprehensive methodological and tool-supported model-based engineering guidance. `http://polarsys.org/capella/arcadia.html`, 2018. Accessed: 2018-07-31.

[Pou94] J. S. Poulin. Measuring software reusability. In *Proceedings of 1994 3rd International Conference on Software Reuse*, pages 126–138, Nov 1994.

[PPK+11] Pablo Parra, Oscar R. Polo, Martin Knoblauch, Ignacio Garcia, and Sebastian Sanchez. MICOBS: Multi-platform Multi-model Component Based Software Development Framework. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*, CBSE '11, pages 1–10, New York, NY, USA, 2011. ACM.

[PPL14] Branko Perisic, Ana Perisic, and Marko Lazic. The Broker Solution Example of Domain Specific Languages Orchestration Framework. In *ERK 2014, IEEE Conference*, 2014.

[PPS+03] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-Based Test Case Generation for Smart Cards. *Electronic Notes in Theoretical Computer Science*, 80(Supplement C), 2003.

[PPW+05] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401. ACM, 2005.

[PRRJ06] Nachiketh R Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Transactions on mobile computing*, 5(2):128–143, 2006.

[PSAK04] Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, and Stefan Kriebel. Model-based testing for real. *International Journal on Software Tools for Technology Transfer*, 5(2-3), 2004.

[PSSK14] Sebastian B.M. Patzelt, Leonie K. Schaible, Susanne Stampf, and Ralf J. Kohal. Software-based evaluation of human attractiveness: A pilot study. *The Journal of Prosthetic Dentistry*, 112(5):1176 – 1181, 2014.

[PSV13] Vaclav Pech, Alex Shatalin, and Markus Voelter. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 165–168. ACM, 2013.

[Pto14] Claudius Ptolemaeus. *System design, modeling, and simulation: using Ptolemy II*, volume 1. Ptolemy. org Berkeley, 2014.

[PTV+10] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. das Chagas Mendonca. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software*, 27(5):82–89, Sept 2010.

[pur14] pure-systems GmbH. pure::variants - Connector für MATLAB/Simulink: Integriert Variantenmanagement in MATLAB/Simulink. `http://www.pure-systems.com/fileadmin/downloads/pure-variants/pv-matlab-connector-de.pdf`, 2014. Accessed: 2018-07-31.

[Pus98] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical report, TU Munich, 1998.

[RBL+08] Fred W. Rauskolb, Kai Berger, Christian Lipski, Marcus Magnor, Karsten Cornelsen, Jan Effertz, Thomas Form, Fabian Graefe, Sebastian Ohl, Walter Schumacher, Jörn-Marten Wille, Peter Hecker, Tobias Nothdurft, Michael Doering, Kai Homeier, Johannes Morgenroth, Lars Wolf, Christian Basarke, Christian Berger, Tim Gülke, Felix Klose, and Bernhard Rumpe. Caroline: An autonomously driving vehicle for urban environments. *Journal of Field Robotics*, 25(9):674–724, 2008.

[Rec08] Jörg Rech. *Model-Driven Software Development: Integrating Quality Assurance: Integrating Quality Assurance*. IGI Global, 2008.

[Red09] Roman R Redziejowski. Mouse: from parsing expressions to a practical parser. In *Concurrency Specification and Programming Workshop*. Citeseer, 2009.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[RFH+05] Sabine Rittmann, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, and Doris Wild. Integrating service specifications at different levels of abstraction. In *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, pages 63–70. IEEE, 2005.

[RH08] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, Dec 2008.

[Rie18] Udo Rieber. From PC-Experiment to ECU Code with ASCET-DEVELOPER. `https://drivingembeddedexcellence.com/2018/02/13/ascet-developer/`, 2018. Accessed: 2018-07-31.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RM06] Adnan Rawashdeh and Bassem Matalkah. A new software quality model for evaluating COTS components. *Journal of Computer Science*, 2(4):373–381, 2006.

[RM14] Per Runeson and Sten Minör. The 4+ 1 view model of industry–academia collaboration. In *Proceedings of the 2014 international workshop on Long-term industrial collaboration on software engineering*, pages 21–24. ACM, 2014.

[Rom85] G. C. Roman. A taxonomy of current issues in requirements engineering. *Computer*, 18(4):14–23, April 1985.

[Ron17] Jean-Marc Ronck. *Creation of a Multi-User Online IDE for Domain-Specific Languages*. Bachelor Thesis at RWTH Aachen University, 2017.

[Roq16] Pascal Roques. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In Holger Giese, Michaela Huhn, Jan Philipps, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, pages 30–43, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RRW16] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*, pages 1–12. fortiss, An-Institut TU München, Technical Report, March 2016.

[RSRB06] Elvinia Riccobene, Patrizia Scandurra, Alberto Rosti, and Sara Bocchio. A model-driven design environment for embedded systems. In *Proceedings of the 43rd annual Design Automation Conference*, pages 915–918. ACM, 2006.

[RSRS15] Bernhard Rumpe, Christoph Schulze, Johannes Richenhagen, and Axel Schloßer. Agile Synchronization between a Software Product Line and its Products. In *Informatik 2015)*, volume P-246 of *LNI*, pages 1687–1698. Bonner Köllen Verlag, 2015.

[RSvW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[RSvW16] Bernhard Rumpe, Christoph Schulze, and Michael von Wenckstern. MontiMatcher: Ähnlichkeitsanalyse-Framework zur Produktlinienextraktion und Evolutionsüberwachung. *Softwaretechnik-Trends*, 36(2), May 2016.

[RT14] Jan Reineke and Stavros Tripakis. Basic problems in multi-view modeling. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 217–232. Springer, 2014.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[RVR+10] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, Linus Tolke, and Michiel van der Wulp. ArgoUML User Manual. `http://argouml-stats.tigris.org/documentation/manual-0.32/`, 2010. Accessed: 2018-07-31.

[RW11] Bernhard Rumpe and Ingo Weisemöller. A domain specific transformation language. In *ME 2011 - Models and Evolution*, 2011.

[RWT18a] RWTH Aachen Campus. Center for Systems Engineering. `https://www.rwth-campus.com/center-for-systems-engineering/`, 2018. Accessed: 2018-07-31.

[RWT18b] RWTH Aachen University, IT Center. Nutzerzertifikat beantragen. `https://doc.itc.rwth-aachen.de/display/CERT/Nutzerzertifikat+beantragen`, 2018. Accessed: 2018-07-31.

[Ryn18] Alexander Ryndin. *Modelling of Component-and-Connector Architectures for Autonomous Vehicles*. Master Thesis at RWTH Aachen University, 2018.

[SAJ09] M. I. Soliman and A. F. Al-Junaid. SystemC implementation of mat-core: A matrix core extension for general-purpose processors. In *2009 4th International Conference on Design Technology of Integrated Systems in Nanoscal Era*, pages 9–14, April 2009.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SC16] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 2016.

[SCFG15] Rick Salay, Marsha Chechik, Michalis Famelis, and Jan Gorzny. A Methodology for Verifying Refinements of Partial Models. *Journal of Object Technology*, 14(3):3–1, 2015.

[Sch06] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, Feb 2006.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sch16] Jörg Schäuffele. E/E Architectural Design and Optimization using PREEvision. In *SAE 2016 World Congress and Exhibition*. SAE International, apr 2016.

[Sch17] Saschsa Schneiders. *Development of a C++ Generator for Embedded Modeling Languages*. Bachelor Thesis at RWTH Aachen University, 2017.

[Sch18] Manuel Schrick. *Visualisation of Textual Component and Connector Models*. Master Thesis at RWTH Aachen University, 2018.

[SCS11a] Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. UML-based modeling of non-functional requirements in telecommunication systems. In *6th Int. Conf. on Software Engineering Advances (ICSEA)*, 2011.

[SCS11b] Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. Uml-based modeling of non-functional requirements in telecommunication systems. In *The Sixth International Conference on Software Engineering Advances (ICSEA 2011)*, pages 213–220. Barcelona, Spain: The Institute of Electrical and Electronics Engineers, Inc., 2011.

[SE06] Mehrdad Sabetzadeh and Steve Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requirements Engineering*, 11(3):174–193, 2006.

[SE18] RWTH Aachen University Software Engineering. MontiCore - Language Workbench plus Tool Framework: JavaDSL. `https://github.com/monticore/javadsl`, 2018. Accessed: 2018-07-31.

[Sei03] Edwin Seidewitz. What models mean. *IEEE software*, 20(5):26–32, 2003.

[Sei12] Sabine Seifert. Die Freiwillige Selbstkontrolle der Film-wirtschaft (FSK. *Jugend Medien Schutz-Report*, 34(6):2–4, 2012.

[Sel96] Bran Selic. Real-Time Object-Oriented Modeling. *IFAC Proceedings Volumes*, 29(5):1 – 6, 1996. IFAC Workshop on real Time Programming WRTP 96, Gramado, Brazil, 4-6 November.

[Sel07] Bran Selic. A systematic approach to domain-specific language design using UML. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 2–9. IEEE, 2007.

[Sen12] Séverine Sentilles. *Managing extra-functional properties in component-based development of embedded systems*. PhD thesis, Pau, 2012.

[SFC12] Rick Salay, Michalis Famelis, and Marsha Chechik. Language independent refinement using partial modeling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 224–239. Springer, 2012.

[SG04] Bradley Schmerl and David Garlan. AcmeStudio: Supporting style-centered architecture development. In *Proceedings of the 26th International Conference on Software Engineering*, pages 704–705. IEEE Computer Society, 2004.

[SG07] Dietmar Schreiner and Karl M Goschka. A component model for the AUTOSAR Virtual Function Bus. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 2, pages 635–641. IEEE, 2007.

[SGCH01] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 99–108. ACM, 2001.

[SHC17] Daniel Schwencke, Hardi Hungar, and Mirko Caspar. Between Academics and Practice: Model-based Development of Generic Safety-Critical Systems. In *Modellbasierte Entwicklung eingebetteter Systeme*, 2017.

[She10] LQQXF Shengdi. Simulating bioreaction processes based on SimBiology. *Computer Applications and Software*, 8:065, 2010.

[Sko03] Sigurd Skogestad. Simple analytic rules for model reduction and PID controller tuning. *Journal of process control*, 13(4):291–309, 2003.

[SKS92] Tor Stålhane, Even-André Karlsson, and Guttorm Sindre. Software Reuse in an Educational Perspective. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 99–114, London, UK, UK, 1992. Springer-Verlag.

[Soc06] Society of Automobile Engineers. SAE Architecture Analysis and Design Language (AADL), SAE Standard AS5506/1, 2006.

[Sof16] Software Engineering, RWTH Aachen University. How-To: Testing MontiArc Components. `http://www.monticore.de/languages/montiarc/howtotest/`, 2016. Accessed: 2018-07-31.

[Spa17] Sparx Systems. Enterprise Archtect: User Guide Series: Tutorial. `http://www.sparxsystems.com.au/resources/user-guides/basics/tutorial.pdf`, 2017. Accessed: 2018-07-31.

[SPCH08] S. Sentilles, P. Pettersson, I. Crnkovic, and J. Hakansson. Save-IDE: An Integrated Development Environment for Building Predictable Component-Based Embedded Systems. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 493–494, Washington, DC, USA, 2008. IEEE Computer Society.

[SPE11] SPECTRA. Three Key Requirements of a Sound Disaster Recovery Strategy. `https://edge.spectralogic.com/index.cfm?&fuseaction=home.displayFile&DocID=3664`, 2011. Accessed: 2018-07-31.

[SPSG14] Narayanamurthy Srinivas, Narendrakumar Panditi, Stefan Schmidt, and Ralf Garrelfs. MIL/SIL/PIL Approach: A new paradigm in Model Based Development. `https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/solutions/automotive/files/in-expo-2014/mil-sil-pil-a-new-paradigm-in-model-based-development.pdf`, 2014. Accessed: 2018-07-31.

[SRK+08] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake. Measuring Non-Functional Properties in Software Product Line for Product Derivation. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 187–194, Dec 2008.

[SRK+12] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3):487–517, Sep 2012.

[SS76] D Scott and C Strachey. The Denotational Semantics of Programming Languages. *Communications*, 1976.

[SSC96a] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring Compliance of a Software System with Its High-level Design Models. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society.

[SSC96b] Mohlalefi Sefika, Aamod Sane, and Roy H Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th international conference on Software engineering*, pages 387–396. IEEE Computer Society, 1996.

[SSCC09] Séverine Sentilles, Petr Stepan, Jan Carlson, and Ivica Crnkovic. Integration of Extra-Functional Properties in Component Models. In *CBSE*, 2009.

[SSCS16] Gaetana Sapienza, Séverine Sentilles, Ivica Crnkovic, and Tiberiu Seceleanu. Extra-Functional Properties Composability for Embedded Systems Partitioning. In *CBSE*, 2016.

[SSS+13] Jagadish Suryadevara, Gaetana Sapienza, Cristina Cerschi Seceleanu, Tiberiu Seceleanu, Stein Erik Ellevseth, and Paul Pettersson. Wind Turbine System: An Industrial Case Study in Formal Modeling and Verification. In *FTSCS*, pages 229–245, 2013.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[Str17] Nicolai Strodthoff. *Strukturelle Analysen von MontiArc-Modellen mittels Z3-Solver*. Bachelor Thesis at RWTH Aachen University, 2017.

[Str18a] Ievgen Strepkov. *Development of Web Playground for Component and Connector Models*. Master Thesis at RWTH Aachen University, 2018.

[Str18b] Ievgen Strepkov. Modeling a Parking Controller. `https://youtu.be/rMmbDGlJ0Gk`, 2018. Accessed: 2018-07-31.

[Str18c] Ievgen Strepkov. Modeling an Elk Test Controller. `https://youtu.be/PQwU-FhZo-s`, 2018. Accessed: 2018-07-31.

[SV18] Andreas Spillner and Karin Vosseberg. Cross-funktionale Kooperation: Wie Programmierung und Test ineinander greifen. *OBJEKTspektrum 02/2018 (Modernes Testen)*, 2018(2), 2018.

[SVB+08] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In Michel R. V. Chaudron, Clemens Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering*, pages 310–317, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[SVL15] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. T-Core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, 14(3):1215–1243, 2015.

[Ten76] Robert D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, 1976.

[TH77] D. Teichroew and E. A. Hershey. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, SE-3(1):41–48, Jan 1977.

[The18a] The MathWorks, Inc. Documenation: colon, : (Vector creation, array subscripting, and for-loop iteration). `https://de.mathworks.com/help/matlab/ref/colon.html`, 2018. Accessed: 2018-07-31.

[The18b] The MathWorks, Inc. Documenation: diag (Create diagonal matrix or get diagonal elements of matrix). https://de.mathworks.com/help/matlab/ref/diag.html, 2018. Accessed: 2018-07-31.

[The18c] The MathWorks, Inc. Documenation: elist (List simulation methods in order in which they are executed during simulation). https://de.mathworks.com/help/simulink/slref/elist.html, 2018. Accessed: 2018-07-31.

[The18d] The MathWorks, Inc. Documenation: Logical Operator. https://www.mathworks.com/help/simulink/slref/logicaloperator.html, 2018. Accessed: 2018-07-31.

[The18e] The MathWorks, Inc. Documenation: Matrix Indexing. https://de.mathworks.com/help/matlab/math/matrix-indexing.html, 2018. Accessed: 2018-07-31.

[The18f] The MathWorks, Inc. Documenation: reshape (Reshape array). https://de.mathworks.com/help/matlab/math/matrix-indexing.html, 2018. Accessed: 2018-07-31.

[The18g] The MathWorks, Inc. Documenation: Setting Up Solvers for Physical Models. https://de.mathworks.com/help/physmod/simscape/ug/setting-up-solvers-for-physical-models.html, 2018. Accessed: 2018-07-31.

[The18h] The MathWorks, Inc. Documenation: Signal Routing with the From, Goto, and Goto Tag Visibility Blocks. https://de.mathworks.com/help/simulink/slref/_mw_13a98f23-c4cc-4133-a1d4-4af32494d727.html, 2018. Accessed: 2018-07-31.

[The18i] The MathWorks, Inc. Documenation: slist (Sorted list of model blocks). https://de.mathworks.com/help/simulink/slref/slist.html, 2018. Accessed: 2018-07-31.

[The18j] The MathWorks, Inc. Simulink Examples: Automotive Applications. https://de.mathworks.com/help/simulink/examples.html#d2e486, 2018. Accessed: 2018-07-31.

[The18k] The MathWorks, Inc. Simulink (R2018a): User's Guide. Technical Report R2018a, MATLAB & SIMULINK, 2018.

[The18l] The MathWorks, Inc. Simulink Release Notes, Release Range: R2015b to R2018a. https://de.mathworks.com/help/simulink/release-notes.html, 2018. Accessed: 2018-07-31.

[The18m] The MathWorks, Inc. Simulink Requirements. `https://de.mathworks.com/products/simulink-requirements.html`, 2018. Accessed: 2018-07-31.

[The18n] The MathWorks, Inc. Simulink User's Guide. Technical Report R2018b, MATLAB & SIMULINK, 2018.

[The18o] The MathWorks, Inc. Symbolic Math Toolbox (R2018a): User's Guide. Technical Report R2018a, MATLAB, 2018.

[TJ11] Anil Kumar Thurimella and Dirk Janzen. metadoc feature modeler: a plug-in for IBM rational DOORS. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 313–322. IEEE, 2011.

[TK06] Michael M Tiller and Burit Kittirungsi. UnitTesting: A Library for Modelica Unit Testing. In *Proceedings of the 5th International Modelica Conference, Vienna*, 2006.

[TMD10] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. slides to book: Software Architecture: Foundations, Theory, and Practice (John Wiley & Sons, Inc.). `http://sunset.usc.edu/classes/cs578_2018s/slides/ModelingNotations.ppt`, 2010. Accessed: 2018-07-31.

[Tol16] Severin Tolksdorf. *Kontrollflussgraphenanalyse für das Verifikationstool*. Bachelor Thesis at RWTH Aachen University, 2016.

[TP97] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 280–291. IEEE, 1997.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003.

[TX00] Jeffrey J.P. Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 10(1):207–223, Nov 2000.

[Uni18a] University of California. Concepts and Info: The xADL Way. `http://isr.uci.edu/projects/xarchuci/way.html`, 2018. Accessed: 2018-07-31.

[Uni18b] University of Tennessee. CLAPACK (f2c'ed version of LAPACK). `http://www.netlib.org/clapack/`, 2018.

[Urb15] Paul Urban. Work with Requirements in Simulink Using Requirements Perspective. https://de.mathworks.com/videos/work-with-requirements-in-simulink-using-requirements-perspective-1504216505830.html, 2015. Accessed: 2018-07-31.

[V-M06] V-Modell XT. Part 1: Fundamentals of the V-Modell. Technical report, Federal Government of Germany, 2006.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook.org, 2013.

[vdHMRRM01] André van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel, and Nenad Medvidovic. Taming Architectural Evolution. *SIGSOFT Softw. Eng. Notes*, 26(5):1–10, September 2001.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[vdWS10] Rüdiger von der Weth and Ulrike Starker. Integrating motivational and emotional factors in implementation strategies for new enterprise planning software. *Production Planning and Control*, 21(4):375–385, 2010.

[Vec18] Vector. VectorCAST/Ada - Automation for Ada Unit Testing. https://www.vectorcast.com/software-testing-products/ada-unit-testing, 2018. Accessed: 2018-07-31.

[VM93] Jeffrey M Voas and Keith W Miller. Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216, 1993.

[Voe11] Markus Voelter. Xtext/TS?a type system framework for Xtext. http://www.infoq.com/articles/xtext_ts, 2011. Accessed: 2018-07-31.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[VOVDLKM00] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[VS10] Markus Voelter and Konstantin Solomatov. Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In *Software Language Engineering (SLE'10)*. LNCS. Springer, 2010.

[vS13] Sabrina von Styp. Testing for AADL. https://moves.rwth-aachen.de/research/projects/testing-for-aadl/, 2013. Accessed: 2018-07-31.

[VST18] Frédéric Verdier, Abdelhak-Djamel Seriai, and Raoul Taffo Tiam. Reusing Platform-specific Models in Model-Driven Architecture for Software Product Lines. In *MODELSWARD*, pages 106–116, 2018.

[vW17a] Michael von Wenckstern. Statistics of C&C Views and Satisfaction Witnesses. `http://www.se-rwth.de/materials/cncviewscasestudy/ViewAndWitnessSizesStatistics.htm`, 2017. Accessed: 2018-07-31.

[vW17b] Michael von Wenckstern. Statistics of Simulink Models. `http://www.se-rwth.de/materials/cncviewscasestudy/SimulinkStatistics.htm`, 2017. Accessed: 2018-07-31.

[vW18] Michael von Wenckstern. EmbeddedMontiArcStudio: Overview Video. `https://youtu.be/VTKSWwWp-kg`, 2018. Accessed: 2018-07-31.

[W3C11] W3C. Scalable Vector Graphics (SVG) 1.1 (Second Edition). `https://www.w3.org/TR/SVG11/`, 2011. Accessed: 2018-07-31.

[W3C14] W3C. Mathematical Markup Language (MathML) Version 3.0 2nd Edition. `https://www.w3.org/TR/MathML3/`, 2014. Accessed: 2018-07-31.

[W3C17a] W3C. CSS Snapshot 2017. `https://www.w3.org/TR/css-2017/`, 2017. Accessed: 2018-07-31.

[W3C17b] W3C. HTML 5.2. `https://www.w3.org/TR/2017/REC-html52-20171214/`, 2017. Accessed: 2018-07-31.

[W3C18a] W3C. HTML Media Capture. `https://www.w3.org/TR/2018/REC-html-media-capture-20180201/`, 2018. Accessed: 2018-07-31.

[W3C18b] W3C. Payment Request API. `https://www.w3.org/TR/2018/CR-payment-request-20180718/`, 2018. Accessed: 2018-07-31.

[Wat18] Waterloo Maple Inc. Modelica.Math.Matrices: Library of functions operating on matrices. `https://www.maplesoft.com/documentation_center/online_manuals/modelica/Modelica_Math_Matrices.html`, 2018. Accessed: 2018-07-31.

[WC10] Byron J Williams and Jeffrey C Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, 2010.

[WCC95] Charles Weber, Bill Curtis, and Mary B Chrissis. *The capability maturity model: guidelines for improving the software process*. Addison-wesley, 1995.

[WCPL07] G. Willmann, D. F. Coutinho, L. F. A. Pereira, and F. B. Libano. Multiple-Loop H-Infinity Control Design for Uninterruptible Power Supplies. *IEEE Transactions on Industrial Electronics*, 54(3):1591–1602, June 2007.

[Wei12a] Tim Weilkiens. Variant Modeling With SysML. `https://model-based-systems-engineering.com/2012/05/07/variant-modeling-with-sysml`, 2012. Accessed: 2018-07-31.

[Wei12b] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wes18] Bernhard Westfechtel. Case-based exploration of bidirectional transformations in QVT Relations. *Software & Systems Modeling*, 17(3):989–1029, Jul 2018.

[WFH+06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, and Sabine Rittmann. An architecture-centric approach towards the construction of dependable automotive software. Technical report, SAE Technical Paper, 2006.

[WICE03] Brian C Williams, Michel D Ingham, Seung H Chung, and Paul H Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[Wik18] Wikipedia. SystemC. `https://en.wikipedia.org/wiki/SystemC`, 2018. Accessed: 2018-07-31.

[Wil15] Edward D Willink. Safe Navigation in OCL. In *OCL@ MoDELS*, pages 81–88, 2015.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[WYW+10] X. Wang, S. Yang, S. Wang, X. Niu, and J. Xu. An Application-Based Adaptive Replica Consistency for Cloud Storage. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 13–17, Nov 2010.

[WZX08] Ju An Wang, Fengwei Zhang, and Min Xia. Temporal metrics for software vulnerabilities. In *Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead*, page 44. ACM, 2008.

[Yam15] Shuichiro Yamamoto. An Approach to Assure Dependability Through ArchiMate. In Floor Koornneef and Coen van Gulijk, editors, *Computer Safety, Reliability, and Security*, pages 50–61, Cham, 2015. Springer International Publishing.

[YGJK16] Muhammad Younas, Imran Ghani, Dayang NA Jawawi, and Muhammad Murad Khan. A Framework for Agile Development in Cloud Computing Environment. *Journal of Internet Computing and Services(JICS)*, 17(5):67–74, 2016.

[Zel94] Gregory Zelesnik. UniCon Reference Manual 94: Variant. `https://www.cs.cmu.edu/~Vit/unicon/reference-manual/Reference_Manual_94.html`, 1994. Accessed: 2018-07-31.

[Zio17] Zion Market Research. Global Embedded Systems Market Will Reach USD 225.34 billion by 2021. `https://globenewswire.com/news-release/2017/06/08/1010414/0/en/Global-Embedded-Systems-Market-Will-Reach-USD-225-34-billion-by-2021.html`, 2017. Accessed: 2018-07-31.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.

[ZSM11] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. A taxonomy of model-based testing for embedded systems from multiple industry domains. *Model-based testing for embedded systems*, pages 3–22, 2011.

[ZTB08] Mohamed H Zaki, Sofiène Tahar, and Guy Bois. Formal verification of analog and mixed signal designs: A survey. *Microelectronics journal*, 39(12):1395–1404, 2008.

[Zve08] Sergey Zverlov. *Comparison of two level-based Approaches for the Development of Embedded Systems*. Bachelor Thesis at TU Munich, 2008.

PDF exports of websites referenced by this thesis are available from: `https://doi.org/10.5281/zenodo.3588296`. This ensures that the reader can inspect the websites in the same way as the author inspected them when writing this thesis in 2018 and 2019.

List of Figures

List of Tables

Curriculum Vitae

Name	Michael von Wenckstern
Place of Birth	Karl-Marx-Stadt (nowadays: Chemnitz), Germany
Birthday	10.01.1988
Nationality	German
2019 -	Security and Privacy Manager at Continental Engineering Services GmbH, Germany
2013 - 2019	PhD in Software Engineering at RWTH Aachen University, Germany Research Area: SysML, C&C modeling, V-model, test-driven and agile software engineering with focus on automotive industry
2013 - 2018	Research Assistant (wissenschaftlicher Mitarbeiter) at Software Engineering Chair at RWTH Aachen University, Germany
2008 - 2013	Mathematics and computer science studies at Technical University Bergakademie Freiberg, Germany; diploma in Applied Mathematics with distinction
2007 - 2008	National service in Frankenberg, Germany
2005 - 2007	Abitur at Carl-von-Bach-Gymnasium Stollberg, Germany
2004 - 2005	Exchange year at Cannon-County High School, Tennessee, U.S.A.

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.