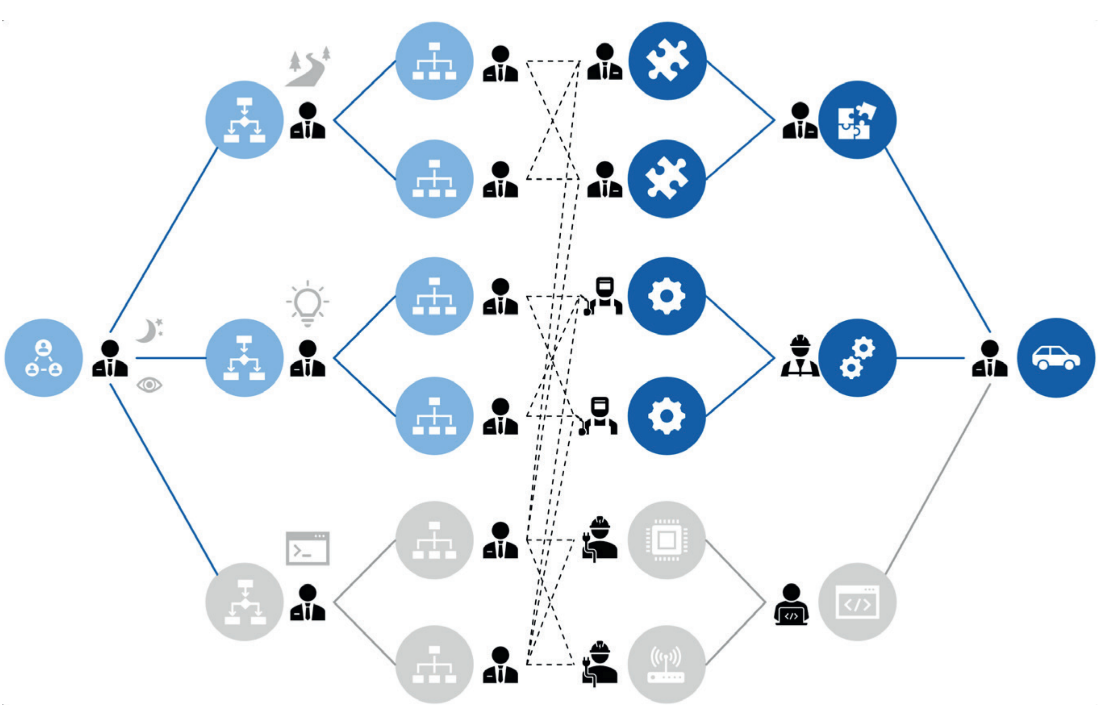
Louis Wachtmeister

An Industrial Method for the Model-Based Development of Automotive Systems



Aachener Informatik-Berichte,
Software Engineering

Band 65

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

An Industrial Method for the Model-Based Development of Automotive Systems

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

vorgelegt von

Louis Wachtmeister, M.Sc. RWTH
aus Köln

Berichter: Universitätsprofessor Dr. Bernhard Rumpe
Universitätsprofessor Dr. Andreas Vogelsang

Tag der mündlichen Prüfung: 24. Oktober, 2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 65

Louis Wachtmeister
RWTH Aachen University

An Industrial Method for the Model-Based Development of Automotive Systems

Shaker Verlag
Düren 2026

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Zugl.: D 82 (Diss. RWTH Aachen University, 2025)

Copyright Shaker Verlag 2026
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

Printed in Germany.

Print-ISBN 978-3-8191-0523-4
PDF-ISBN 978-3-8191-0477-0
ISSN 1869-9170
eISSN 2944-6910
https://doi.org/10.2370/9783819104770

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Phone: 0049/2421/99011-0 • Telefax: 0049/2421/99011-9
Internet: www.shaker.de • e-mail: info@shaker.de

Eidestattliche Erklärung

Louis Wachtmeister

erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden.

Hiermit erkläre ich an Eides statt

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand dieser Fakultät und Universität angefertigt;

2. Sofern irgendein Bestandteil dieser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt;

3. Wenn immer andere eigene- oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt;

4. Wenn aus anderen eigenen- oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben. Diese Dissertation ist vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate;

5. Alle wesentlichen Quellen von Unterstützung wurden benannt;

6. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde;

7. Ein Teil oder Teile dieser Arbeit wurden zuvor veröffentlicht und zwar in:

[GKM+25] Christian Granrath, Christopher Kugler, Judith Michael, Bernhard Rumpe, and Louis Wachtmeister. Generating Logical Architectures from SysML Behavior Models. INCOSE Systems Engineering, 28(6):762–778, November 2025.

[TMW+24] Temmen, Till and Meyer, Max-Arno and Wachtmeister, Louis and Zabihi, Mohammadsadegh and Kugler, Christopher and Christiaens, Sebastien and Rumpe, Bernhard and Andert, Jakob. Application of Model-Based Systems Engineering Methods in Virtual Homologation Procedures for Automated Driving Functions. In Heintzel, Alexander, editor, Automatisiertes Fahren 2024, pages 1–14, Frankfurt, March 2024. Springer Fachmedien Wiesbaden.

[GOR+23] Christian Granrath, Philipp Orth, Bernhard Rumpe, and Louis Wachtmeister. Optimierungspotentiale zur effizienten Systemmodellierung im Kontext der Automobilen Systementwicklung. In Tag des Systems Engineering 2023, pages 307–313. Gesellschaft für Systems Engineering (GfSE) e.V., November 2023.

[BBK+22a] Vincent Bertram, Miriam Boß, Evgeny Kusmenko, Imke Nachmann, Bernhard Rumpe, Danilo Trotta, and Louis Wachtmeister. Neural Language Models and Few-Shot Learning for Systematic Requirements Processing in MDSE. In International Conference on Software Language Engineering (SLE'22), pages 260–265. ACM, December 2022.

[BBK+22b] Vincent Bertram, Miriam Boß, Evgeny Kusmenko, Imke Helene Nachmann, Bernhard Rumpe, Danilo Trotta, and Louis Wachtmeister. Technical Report on Neural Language Models and Few-Shot Learning for Systematic Requirements Processing in MDSE, November 2022.

[DJR+22] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, Manuel Wimmer, and Andreas Wortmann. A cross-domain systematic mapping study on software engineering for Digital Twins. Journal of Systems and Software (JSS), 193, November 2022.

[KRW22] Oliver Kautz, Bernhard Rumpe, and Louis Wachtmeister. Semantic Differencing of Use Case Diagrams. Journal of Object Technology (JOT), 21:3:1–14, July 2022.

[MSG+22] Max-Arno Meyer, Sebastian Silberg, Christian Granrath, Christopher Kugler, Louis Wachtmeister, Bernhard Rumpe, Sebastien Christiaens, and Jakob Lukas Andert. Scenario- and Model-Based Systems Engineering Procedure for the SOTIF-Compliant Design of Automated Driving Functions. In 2022 IEEE Intelligent Vehicles Symposium (IV'22), pages 1599–1604. IEEE, June 2022.

[JGW+21] Nicolas Jäckel, Christian Granrath, Louis Wachtmeister, Abdulsamed Karaduman, Bernhard Rumpe, and Jakob Lukas Andert. Feature-Driven Specification of VTOL Air-Taxis with the Use of the Model-Based Systems Engineering (MBSE) Methodology CUBE. In 77th Annual Vertical Flight Society Forum and Technology Display (FORUM 77)), pages 2776–2784. Curran Associates, Inc., May 2021.

[GJK+21] Malin Gandor, Nicolas Jäckel, Lorenz Käser, Alexander Schlie, Ingo Stierand, Axel Terfloth, Steffen Toborg, Louis Wachtmeister, and Anna Wißdorf. Architectures for Dynamically Coupled Systems. In Model-Based Engineering of Collaborative Embedded Systems, pages 95–124. Springer, January 2021.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In Model-Based Engineering of Collaborative Embedded Systems, pages 315–331. Springer, January 2021.

[MGWJ20] Max Meyer, Christian Granrath, Louis Wachtmeister, and Nicolas Jäckel. Methods for the Development of Collaborative Embedded Systems in Automated Vehicles. ATZelectronics worldwide, 15(12):58–63, December 2020.

[DJK+19] Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, and Andreas Wortmann. Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext. In International Workshop on Domain-Specific Modeling (DSM'19), pages 40–49. ACM, October 2019.

[DJR+19] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, Louis Wachtmeister, and Andreas Wortmann. Model-Driven Systems Engineering for Virtual Product Design. In Proceedings of MODELS 2019. Workshop MPM4CPS, pages 430–435. IEEE, September 2019.

[NW17] Thomas Noll and Louis Wachtmeister. Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications. Mar 2017.

Aachen, 6. März 2025

Louis Wachtmeister

Abstract

Software and systems engineering offers a multitude of innovative methods for the design, realization, and management of systems that support software and systems engineers from the initial idea through design and specification to the operation and disposal of the system. This thesis therefore investigates how these methods can contribute to the efficient and effective specification and design of automotive systems and explores ways to utilize these methods for the development of such large and complicated systems. Although systems engineering achieved remarkable results in developing cyber-physical systems in selected examples, many previous works in the field of automotive systems engineering have yet to consider several real-world applications in a wider context of their utilization. Therefore, many aspects that appear reasonable in a single application of the methods lead to significant problems in their industrialization as soon as several development projects and different possibilities to decompose the system into system levels are considered. For example, while the execution of the same task from different perspectives may seem careful and suited to reveal further insights in a single application or a single system model with a few elements, the scaled application for systems with hundreds or thousands of model elements reveals that performing the same task multiple times using an inefficient modeling method leads to excessive efforts in system modeling that frustrates the system engineers.

Consequently, this thesis considers several industry projects and provides tailored methods for defining and applying system models. To this end, this thesis contributes:

- An adaptation of a specification method for requirements and design for the application of model-driven development in the automotive industry;
- A domain-specific language to support the formulation of natural language requirements in legacy projects;
- A framework for modeling stakeholder needs in use case diagrams;
- A specification method to specify reusable system behaviors as operating principles;
- A derivation technique and modeling principles to derive reusable logical architecture specifications from these operating principles;
- And experiences from applying these methods in the automotive industry.

With these contributions, this thesis aims to increase the general acceptance and application of system modeling in the automotive industry. All methods this thesis presents have already been applied by an engineering service provider in cooperation projects with several automotive companies, covering system development projects at various decomposition levels from the software control module of a powertrain component to the powertrain and the complete vehicle specification.

Kurzfassung

Die Software- und Systemtechnik bietet eine Vielzahl innovativer Methoden für den Entwurf, die Realisierung und das Management von Systemen, die Software- und Systemingenieure von der ersten Idee über den Entwurf und die Spezifikation bis hin zum Betrieb und der Entsorgung des Systems unterstützen. Diese Arbeit untersucht daher, wie diese Methoden zur effizienten und effektiven Spezifikation und Gestaltung von Fahrzeugsystemen beitragen können und erforscht Wege, diese Methoden zur Entwicklung solch großer und komplexer Systeme zu nutzen. Obwohl die Systemtechnik in der Entwicklung von cyber-physischen Systemen in einigen Fällen bemerkenswerte Ergebnisse erzielt hat, haben viele frühere Arbeiten auf dem Gebiet der Systemtechnik im Automobilbereich noch nicht mehrere reale Anwendungen in einem größeren Kontext ihrer Anwendung berücksichtigt. Daher führen viele Aspekte, die bei einer einzelnen Anwendung der Methoden sinnvoll erscheinen, zu erheblichen Problemen bei deren Industrialisierung, sobald mehreren Entwicklungsprojekte und unterschiedliche Möglichkeiten das System in Systemebenen zu dekomponieren betrachtet werden. Während beispielsweise die Ausführung derselben Aufgabe aus verschiedenen Blickwinkeln sorgfältig und angemessen erscheinen mag, um in einer einzelnen Anwendung oder einem Systemmodell mit wenigen Elementen weitere Erkenntnisse zu gewinnen, zeigt die skalierte Anwendung für Modelle mit Hunderte oder Tausende von Elementen, dass die mehrfache Ausführung derselben Aufgabe mit einer ineffizienten Modellierungsmethode zu einem übermäßigen Aufwand bei der Systemmodellierung führt, der die Systemingenieure frustriert.

Daher werden in dieser Arbeit mehrere Industrieprojekte betrachtet und maßgeschneiderte Methoden zur Unterstützung bei der Definition und Anwendung von Systemmodellen bereitgestellt. Zu diesem Zweck leistet diese Arbeit als Beitrag:

- Eine Adaption einer Spezifikationsmethode für Anforderungen und Design für die Anwendung von modellgetriebener Entwicklung in der Automobilbranche;

- Eine domänenspezifische Sprache zur Unterstützung der Formulierung von natürlichsprachlichen Anforderungen in Legacy-Projekten;

- Ein Rahmenwerk zur Modellierung von Stakeholderbedarfen in Anwendungsfalldiagrammen;

- Eine Spezifikationsmethode, um wiederverwendbare Systemverhaltensweisen als Wirkprinzipien zu spezifizieren;

- Eine Ableitungstechnik und Modellierungsprinzipien zur Ableitung von wiederverwendbaren logischen Architekturspezifikationen aus diesen Wirkprinzipien;

- Erfahrungen aus der Anwendung dieser Methoden in der Automobilindustrie.

Mit diesen Beiträgen zielt diese Arbeit darauf ab, die allgemeine Akzeptanz und Anwendung der Systemmodellierung in der Automobilbranche zu erhöhen. Alle in dieser Arbeit vorgestellten Methoden wurden bereits von einem Ingenieurdienstleister in Kooperationsprojekten mit mehreren Automobilunternehmen angewandt. Dabei wurden Systementwicklungsprojekte auf verschiedenen Dekompositionsebenen vom Software-Steuermodul einer Antriebsstrangkomponente über den Antriebsstrang bis hin zur Gesamtfahrzeugspezifikation abgedeckt.

Danksagung

Glücklicherweise wurde ich während meiner Promotion von vielen Menschen begleitet und unterstützt bei denen ich mich auf diesem Wege bedanken möchte.

Zuerst danke ich meinem Doktorvater Prof. Dr. Bernhard Rumpe für seine fachliche Anleitung als Betreuer meiner Promotion, die wertvollen Ratschläge und Diskussionen während meiner Forschungsarbeit und die Möglichkeit im spannenden industriellen Umfeld dieser Arbeit promovieren zu können. Weiterhin möchte ich Prof. Dr. Andreas Vogelsang dafür danken, dass er das Zweitgutachten dieser Arbeit übernommen hat, Prof. Dr. Stefan Kowalewski für die Leitung meines Promotionskomitees und Prof. Dr. Thomas Noll dafür, in diesem mitzuarbeiten.

Ein spezieller Dank geht außerdem an die acronio GmbH, die FEV Europe GmbH und die FEV.io GmbH, die mir nicht nur die erforderlichen Ressourcen zur Verfügung gestellt haben, sondern auch eine unterstützende Umgebung geschaffen haben, in der ich meine Forschung vorantreiben konnte. Besonders danken möchte ich in diesem Zusammenhang Dr. Philipp Orth, Dr. Christopher Kugler, Dr. Christian Granrath, Dr. Stefan Kriebel, Dr. Jan-Jöran Gehrt, Anuj Malvankar, Kevin Heinen und Michael Koch. Weiterhin danke ich Andre Bergmann, Alexander Bierig, Louis Cassel, Anett Engbrodt, Dr. Wolfgang Jansohn, Claire Kuchta, Shuang Li, Felix Pischinger, Michael Plate, Suresh Manipettakunnu, Christian Nawratil, Amin Sehati, Sebastian Scholz, Konstantin Schulz, Sebastian Silberg, Maximilian Weiß und Yi Zhang für die gemeinsame Zeit, die gute Zusammenarbeit in unseren gemeinsamen Projekten und die vielen fachlichen Diskussionen.

Außerdem möchte ich mich bei meinen Kollegen am Lehrstuhl für Software Engineering für die nicht nur erfolgreiche sondern auch angenehme Zusammenarbeit bedanken. Mein besonderer Dank geht dabei an Prof. Dr. Andreas Wortmann, Dr. Evgeny Kusmenko und Dr. Christian Kirchhof dafür, dass sie mich in Fragen des akademischen Schreibens und Arbeitens beraten und unterstützt habe. Außerdem danke ich Dr. Oliver Kautz für den intensiven Austausch über die Semantikdefinitionen der verwendeten SysML Diagramme. Weiterhin danke ich Manuela Dalibor, David Schmalzing und Vincent Bertram für die unvergessliche Zeit, die wir gemeinsam im Büro am Lehrstuhl verbracht haben. Des weiteren bedanke ich mich bei Kai Adam, Daoud Ali, Vassily Aliseyko, Miriam Boß, Jonas Böcker, Marita Breuer, Lennart Bucher, Joel Charles, Juan Sebastian Diaz, Niklas Dienstknecht, Florian Drux, Robert Eikermann, Christoph Engels, Lars Fischer, Stefanie Fischer, Arkadii Gerasimov, Dr. Timo Greifenberg, Sylvia Gunder, Malte Heithoff, Alexander Hellwig, Steffen Hillemacher, Dr. Katrin Hölldobler, Anna Huehnerbein, Nico Jansen, Hendrik Kausch, Dr. Marco Konersmann, Dr. Achim Lindt, Alexander Lüpges, Dr. Matthias Markthaler, Joshua Mingers, Sonja Müßigbrodt, Dr. Judith Michael, Dr. Imke Nachmann, Prof. Dr. Manfred Nagl, Prof. Dr. Pedram Mir Seyed Nazari, Bogdan Noskov, Lukas Netz, Haron Nqiri, Jerome Pfeiffer, Mathias Pfeiffer, Nina Pichler, Manuel Pützer, Deni Raco, Dr. Florian Rademacher, Dr. Mar-

tin Schindler, Marc Schmidt, Steffi Schrader, Dr. Christoph Schulze, Brian Sinkovec, Max Stachon, Felix Steinfurth, Sebastian Stüber, Samuel Thesing, Simon Varga, Galina Volkova, Dr. Michael von Wenckstern und Lucas Wollenhaupt.

Bei meinen Studenten Markus Blanke, Joel Choi, Christina Katzer, Ronja Köberlein und Mohammadsadegh Zabihi möchte ich mich außderdem dafür bedanken, dass sie mich bei der technischen Umsetzung einiger Teile dieser Arbeit unterstützt haben.

Neben meinen direkten Kollegen am Lehrstuhl für Software Engineering danke ich weiterhin Dr. Max-Arno Meyer, Till Temmen und Prof. Dr. Jakob Andert vom Lehr- und Forschungsgebiet Mechatronik in mobilen Antrieben, sowie Katrin Himmelseher vom Lehrstuhl für Thermodynamik mobiler Energiewandlungssysteme und Institut für Thermodynamik für die gemeinsame Zeit in Projekten und die vielen Diskussionen in unserer gemeinsamen Zeit als Forscher in der Automobilindustrie.

Insbesondere danken möchte ich außerdem Frank, Christian, Phillip, Felix, Vincent und Anuj für ihr Feedback zu einigen Abschnitten dieser Arbeit.

Ich bedanke mich weiterhin bei meinen Freunden und meiner Familie für ihre Unterstützung, die schon während meines Studiums begonnen hat und auch während des Schreibens dieser Arbeit und der Prüfungsvorbereitung einen wichtigen Beitrag geleistet hat. Ganz besonders bedanken möchte ich mich bei meinen Eltern Irmtraud und Frank, die mir mit ihrer liebevollen Unterstützung den Weg, der zu dieser Arbeit geführt hat, überhaupt erst ermöglicht haben. Abschließend möchte ich mich noch bei meiner Partnerin Miriam bedanken. Du hast mich motiviert, mich in jeder Phase dieser Arbeit unterstützt und mir durch deine Unterstützung sehr viele Freiräume geschaffen, die ich dieser Arbeit widmen konnte. Vielen Dank für diese Unterstützung!

Aachen, Oktober 2025
Louis Wachtmeister

Contents

1 Introduction **1**
1.1 Research Questions 2
1.2 Thesis Organization 4
1.3 Publications 5

2 Foundations of System Development in the Automotive Industry **9**
2.1 Foundations of Systems Engineering 9
2.1.1 System Definition 10
2.1.2 System Formalization 12
2.1.3 Systems Engineering Methodology 14
2.2 Model-Driven Systems Engineering 18
2.3 Functional Modeling Paradigm for Automotive Systems 21
2.4 Variability in the Automotive Industry 25
2.4.1 Mass Customization 26
2.4.2 Modularity and Platforms 27
2.4.3 Software Variability, Features and Functions 29
2.4.4 Consequences for Systems Engineering 31
2.5 Modeling Languages for Automotive Systems Engineering 32
2.6 Methodologies for Automotive System Development 41
2.6.1 V-Model 44
2.6.2 Software & Systems Engineering according to ISO/IEC/IEEE 15288 44
2.6.3 IBM Telelogic Harmony 45
2.6.4 IBM Rational Unified Process for Systems Engineering (RUP SE) 45
2.6.5 Vitech Model-Based Systems Engineering Methodology 45
2.6.6 JPL State Analysis (SA) 46
2.6.7 Object-Process Methodology (OPM) 46
2.6.8 INCOSE Object-Oriented Systems Engineering Method (OOSEM) 46
2.6.9 Function-Based Systems Engineering (FuSE) 47
2.6.10 Software Platform Embedded Systems (SPES) 47
2.6.11 Specification Method for Requirements, Design, and Test (SMArDT) 48
2.6.12 Compositional Unified system-Based Engineering (CUBE) 49
2.6.13 Commonalities and Research Gap in SE Methodologies 49
2.7 Interim Conclusion and Discussion 51

3 Model-Driven Systems Engineering using CUBE 53
3.1 The CUBE Methodology . . . 53
3.1.1 CUBE as Product Specification Procedure . . . 58
3.1.2 Interim Summary and Challenges . . . 61
3.2 A CUBE-Specific Architecture Framework . . . 61
3.2.1 System Specifications in the Automotive Industry . . . 65
3.2.2 Cross-Cutting Issue: Specifying Textual Requirements . . . 68
3.2.3 Specifying Customer Requirements as Stakeholder Needs . . . 70
3.2.4 Specifying Functionalities as Operating Principles . . . 72
3.2.5 Specifying Technical Solutions in System Architectures . . . 74
3.2.6 Specifying the Realization . . . 79
3.3 A Critical View on CUBE and its Research Gaps . . . 80

4 System Models in Industry Projects 85
4.1 Selected System Development Projects in the Automotive Industry . . . 85
4.1.1 Alpha: An Example for Document-Based Overall Vehicle Development . . . 86
4.1.2 Beta: An Example for Models next to a Textual Specification of a Powertrain . . . 88
4.1.3 Gamma: An Example for Feature-Driven and Model-Based Systems Engineering for a Driving Function . . . 89
4.1.4 Delta: An Example for Scenario-Based Systems Engineering for an Automated Driving Function . . . 91
4.1.5 Epsilon: Top-Down versus Bottom-Up Model-Creation for a Powertrain Component . . . 91
4.1.6 Zeta: An Example for Model-Driven Engineering of an Autonomous Vehicle . . . 93
4.2 An Automotive Example . . . 94
4.3 Interim Summary: How are Systems Engineering Projects Conducted in the Automotive Industry? . . . 95

5 Modeling Textual Requirements using a Requirement DSL 97
5.1 Requirement Diagrams for Natural Language Requirement Modeling . . . 98
5.2 Sentence Patterns for Requirement Formulation . . . 100
5.2.1 A Reduced Pattern for Functional Requirements . . . 101
5.2.2 A Reduced Pattern for Variability in Quality Requirements . . . 104
5.3 A Requirements DSL for the Automotive Industry . . . 105
5.4 Evaluation Requirements in the Automotive Industry . . . 111
5.4.1 Preprocessing . . . 112
5.4.2 Parsing . . . 113
5.4.3 CoCos, PrettyPrinter and Overall Comparison . . . 113

5.4.4 Comparison with Expert Analysis 115
5.4.5 Reflection on Research Questions 118
5.5 Interim Conclusion, Discussion and Lessons Learned 118
5.5.1 Threats to Validity 119
5.5.2 Lessons Learned from using the SPECTRE Requirement DSL in the Automotive Industry 120

6 Expressing Stakeholder Values using Use Case Diagrams 125
6.1 Use Case Diagrams for Stakeholder Value Specification 126
6.1.1 Diagram Elements to Model Stakeholder Values using SysML Use Case Diagrams 126
6.1.2 Stakeholder Needs for a Simple Vehicle Expressed as a SysML Use Case Diagram 130
6.2 Semantic Foundations of Use Case Diagrams based on [KRW22] 133
6.2.1 Basis Notation: Actors, Use Cases, Variables, Boolean Expressions 134
6.2.2 Abstract Use Case Diagram Syntax Definition 134
6.2.3 Mapping the Abstract Syntax to the Diagram Elements for Stakeholder Value Models in SysML Use Case Diagrams 136
6.2.4 Semantic Use Case Domain 137
6.2.5 Formalization of the Use Case Diagram Semantics 138
6.2.6 A Concrete Textual Syntax for Stakeholder Value Modeling 140
6.3 Feature-Driven Use Case Development 143
6.3.1 Formulating System Functionalities in Use Case Notation 145
6.3.2 Formulating other System Characteristics in Use Case Notation . . 146
6.3.3 Refining Associations Based on the Role of the Actor 146
6.3.4 A Refined Use Case Syntax 150
6.3.5 Feature Driven Use Case Development of a Simple Vehicle 153
6.4 Applying Use Case Diagrams and Generated Requirements in Automotive Systems Engineering Development Projects 157
6.4.1 Generating High-Level Requirements from Feature Use Cases . . . 157
6.4.2 An Evaluation Scheme for Generated Natural Language Requirements 161
6.4.3 Correctness Evaluation of Generated Stakeholder Value Requirements in Model-Based Projects 161
6.4.4 Completeness Evaluation of Generated Stakeholder Value Requirements in Model-Based Projects compared to a Document-Based Specification 167
6.4.5 An Experiment to Measure the Effort Required to Maintain Use Case Diagrams next to Natural Language Requirements Manually 170
6.5 Interim Conclusion and Discussion 172

7 Specifying System Behaviors and Functionalities Based on Operating Principles 175
7.1 Modeling Operating Principles as Activity Diagrams 176
7.1.1 Diagram Elements to Model Operating Principles as Activity Diagrams 176
7.1.2 Diagram Elements to Allocating Actions to their Executing Logical Element 181
7.1.3 An Example Operating Principle of a Transportation Task 182
7.2 Modeling Operating Principles as Scenario Specifications 185
7.2.1 A Scenario Specification Template for Operating Principle Modeling 185
7.2.2 An Example Operating Principle of a Transportation Task Represented and Decomposed using Template-Based Use Case Specification 187
7.2.3 Generating Operating Principles from the Scenario Specification Templates (and vice versa) 188
7.2.4 A Concrete Textual Syntax for Scenario Specification Templates using Activity Diagrams 190
7.3 Applying Operating Principle Models to Efficiently Design Automotive Systems 194
7.3.1 Generating Textual Functionality and Interface Requirements using Activity Diagrams 194
7.3.2 Correctness Evaluation of Generated Operating Principle Requirements in Model-Based Projects 197
7.3.3 Completeness Evaluation of Generated Operating Principle Requirements in Model-Based Projects compared to a Document-Based Specification 201
7.3.4 An Experience Report on Using Tabular Scenario Specification Templates and Activity Diagrams 204
7.4 Interim Conclusion and Discussion 205

8 Deriving Logical System Architectures from Operating Principles 207
8.1 A Logical Reference Architecture for Element Allocation 208
8.1.1 Structural Logical Reference Architecture 209
8.1.2 Modeling sLRA (structural Logical Reference Architecture) as SysML BDDs 216
8.1.3 Behavioral Logical Reference Architecture 219
8.1.4 Modeling the dynamic behavior in the LRA as State Machine Diagrams 222
8.2 Generating Logical Architectures from Operating Principles 226
8.2.1 Modeling Process for Semi-Automated Logical Architecture Generation Based on Operating Principles 226

8.2.2 Modeling Logical Architectures . . . 227
8.2.3 An Example Logical Architecture of a Transportation Task . . . 230
8.2.4 Transformation Rules from Operating Principles to Logical Architectures . . . 231
8.3 Applying Logical Architecture Models to Efficiently Design Automotive Systems . . . 235
8.3.1 Generating Textual System and Interface Requirements using Logical Architecture Diagrams . . . 236
8.3.2 Correctness Evaluation of Generated Logical Architecture Requirements in Model-Based Projects . . . 239
8.3.3 Completeness and Transferability Evaluation of Generated Physical Architecture and Interface Requirements in Model-Based Projects compared to a Document-Based Specification . . . 242
8.3.4 A Critical Reflection and Lessons Learned from Generating Logical Architectures in the Automotive Industry . . . 244
8.4 Interim Conclusion and Discussion . . . 255

9 Conclusion and Future Work **257**
9.1 Summary and Research Questions . . . 257
9.2 Combined View on Generating Textual Requirements from Models . . . 260
9.3 Lessons Learned from the Industrial Application . . . 263
9.3.1 Lesson 1: Systems Thinking . . . 263
9.3.2 Lesson 2: Analytical Modeling . . . 264
9.3.3 Lesson 3: Difference between Syntax and Semantics . . . 265
9.3.4 Lesson 4: Degree of Formalization . . . 265
9.3.5 Lesson 5: Reusable Models . . . 266
9.3.6 Lesson 6: Modular System Models . . . 266
9.3.7 Lesson 7: Correlation Between Organizational Structure and System Model . . . 267
9.4 Potential Future Research . . . 267

Bibliography **269**

Acronyms **301**

A Diagram and Listing Tags **305**

B SPECTRE Grammars **307**
B.1 Action . . . 307
B.2 Condition . . . 308
B.3 Constraint . . . 309

B.4 LogicalExpression . . . 310
B.5 SPECTREName . . . 311
B.6 Sentence . . . 312
B.7 Substantive . . . 313
B.8 Unit . . . 314
B.9 Variable . . . 314
B.10 WordBasis . . . 315

C Use Case Specification Template, Language and Examples 319
C.1 Use Case Specification Template . . . 319
C.2 Transformation Rules to Transform Use Case Specifications to Activity Diagrams and Back . . . 319
C.3 Use Case Specification Language . . . 329
C.3.1 ActivityDiagram . . . 329
C.3.2 UCST . . . 331
C.4 Use Case Specification Examples . . . 333

D Transformation Rules and Guidelines to Generate Natural Language Requirements from SysML Diagrams 341
D.1 Transformation Rules and Guidelines to Generate Natural Language Requirements from Use Case Diagrams . . . 341
D.2 Transformation Rules and Guidelines to Generate Natural Language Requirements from Activity Diagrams . . . 348
D.3 Transformation Rules and Guidelines to Generate Natural Language Requirements from Internal Block Diagrams . . . 362
D.4 Transformation Rules and Guidelines to Generate Natural Language Requirements from State Machine Diagrams . . . 362

List of Figures 367

Listings 373

List of Tables 375

List of Definitions 381

Chapter 1

Introduction

Contemporary vehicles are becoming more and more complicated [SZ16] as vehicle manufacturers increasingly interweave traditional mechanical parts with electrical and electronic components controlled by software units [SZ16, EF17] to offer their customers an ever-increasing number of functionalities [Bro06, DS17]. As a result, the vehicle is becoming a cyber-physical system [Lee08] in which software, physical sensors, and actuators work together to make the overall system interact with its environment [Wor21]. Developing these complicated automotive systems poses challenges that neither traditional development techniques from mechanical and electrical engineering nor development methods from software engineering alone can solve [LAB+11, Kus21]. Unsurprisingly, this results in the fact that conventional and established development methods that follow the V-model [VDI21] reach their limits, as [DW15] or [Dou16] observe.

To address these challenges, systems engineering [FMS14b, ISO15, She17] in general and model-based systems engineering [Wym93, WRF+15, HMR+19] in particular provide methods for the design, realization and management of systems that support software and systems engineers from the initial idea through design and specification to the operation and disposal of the system. There are various methods for applying systems engineering throughout the entire life cycle of a system. For example, feature-driven systems engineering methodologies [Gra22], methodologies tailored to embedded systems [PHAB12, PBDH16], for which multiple surveys on model-based systems engineering methodologies [Est07, NLF+15, HS19] provide an evaluation and comparison. Although several handbooks describe the application of systems engineering methodologies [WRF+15, She17], systems engineering is still only applied in a minority of the overall automotive system development projects [HS19, KFD23]. To explain this trend, a recent study analyzed the perceived value, metrics and evidence for the adoption of model-based system development in several industries [CTE+22]. It revealed that some of the companies seeking to adopt model-based systems engineering as a new approach to engineering projects fear that this paradigm shift is not improving budgets, schedules, efficiency, safety and morale as intended, but actually worsening them [CTE+22]. Moreover, [MP19] reveals that the engineering projects potentially gaining the most from using models in the engineering process are at high stakes and, therefore, experiments concerning the applied methodology are challenging and risky. As automotive system

development projects are typically at high stakes with constant pressure by a steady reduction in time-to-market and development costs while ensuring high product quality [SZ16], a systems engineering methodology that enables a practical but also efficient system development is crucial. Consequently, approaches that target an introduction in such an environment need to pay special attention to the effort required to create models for different perspectives and prevent an unrealistic time effort from the modeler *e.g.,* by providing a high degree of automation or reuse in the model creation.

These observations indicate that further research is needed to address the mentioned challenges. Thus, this thesis examines how the established and innovative methods from software and systems engineering can contribute to the efficient and effective specification and design of automotive systems and explores ways to utilize these methods to develop automotive systems. Therefore, this thesis considers several industry projects. It offers a method tailored to these industrial needs that supports projects in the definition and application of system models, taking into account a realistic application environment, with the aim of increasing the general acceptance and application of system modeling in the automotive industry.

1.1 Research Questions

Because several methods to address these challenges already exist in software and systems engineering, this thesis aims to adopt and refine these methods for the application in industrial automotive systems development and to understand their particular improvement. In particular, it primarily focuses on methods for the definition and application of system models in the system development processes of the automotive industry. Therefore, the main research question of this thesis is:

> *How can innovative software and systems engineering methods be utilized in the automotive industry to define and apply system models effectively and efficiently to specify automotive systems?*

Following the divide and conquer principle, this thesis aims to answer this question by subdividing it into several partial research questions that are easier to answer. Therefore, this thesis first focuses on the recent methods in software and systems engineering to set its foundations with the following research questions.

RQ-1 *Which software and systems engineering methods exist that are applicable to specify automotive systems?*

- **RQ-1.1** *What are the foundations of systems engineering?*
- **RQ-1.2** *What are the characteristics of a system model in the automotive industry?*
- **R-Q-1.3** *Which systems engineering methodologies exist, how is the system decomposed in this methodology, which viewpoints are created, and which paradigm*

is applied?

Then, this thesis focuses on concrete system development projects in the automotive industry to answer the following questions:

RQ-2 *Is there a common practice for creating, interpreting, analyzing, and using system specifications in the considered projects in the automotive industry?*

- **RQ-2.1** *Which system specification process is applied in these projects?*
- **RQ-2.2** *Which artifacts are created during this process?*
- **RQ-2.3** *Which elements are required to specify a system in the automotive industry?*
- **RQ-2.4** *How is the system decomposed in the project, which viewpoints are created, and which paradigm is applied?*

Knowing the required elements of systems modeling, the next challenge concerns expressing these elements in the system model. Accordingly, this thesis focuses on the question of which elements from the SysML [Obj19] are suited to support these methods.

RQ-3 *Which elements of the SysML support the creation of the elements developed under **RQ-2**?*

- **RQ-3.1** *How can requirements be expressed?*
- **RQ-3.2** *How can stakeholder values be formulated?*
- **RQ-3.3** *How to model operating principles?*
- **RQ-3.4** *How to model logical architectures?*

Finally, this thesis focuses on defining and applying these system models effectively and efficiently to specify automotive systems by tackling the following research questions.

RQ-4 *Is an efficient and effective specification of automotive systems possible using model-driven systems engineering?*

- **RQ-4.1** *Can operating principles be derived from use case scenario specifications?*
- **RQ-4.2** *Are logical architectures derivable from operating principles?*
- **RQ-4.3** *Is the efficiency increased by this automation?*
- **RQ-4.4** *Are the models that result from a model-driven specification comparable to document-based specifications?*

By answering these research questions, this work aims to provide the following contributions:

- An overview of the required artifacts to design automotive systems;

- A refined adaptation of a specification method for requirements, design, and testing for the application of model-driven development in the automotive industry based on the considered projects for an application in similar projects;
- A domain-specific language to support the formulation of natural language requirements in legacy projects;
- A framework for modeling stakeholder values in use case diagrams;
- A specification method to specify reusable system behaviors as operating principles;
- A derivation technique and modeling principles to derive reusable logical architecture specifications from these operating principles;
- And experiences from applying these methods in the automotive industry.

1.2 Thesis Organization

In the process of answering the research questions as mentioned earlier, providing contributions, and applying the methods for the definition and application of system models in the automotive industry, this thesis tackles these aspects as follows:

chapter 2 provides an introduction to the foundations of system modeling and examines the state-of-the-art methodologies of systems engineering for the development of automotive systems;

chapter 3 presents a model-driven systems engineering method based on CUBE and examines the processes and structures that systems engineers follow when applying this method;

chapter 4 summarizes the projects in which the methods developed in this thesis were applied and presents a running example;

chapter 5 examines a method for the specification of systems using structured natural language and investigates the status of requirements in two projects for the development of automotive systems;

chapter 6 introduces a method and the semantic foundations to model system stakeholders, their needs, and values using SysML use case diagrams and applies this method to replace natural language requirements as a means of expressing stakeholder requirements in document-based development projects;

chapter 7 establishes SysML activity diagrams and use case behavior modeling templates as a method for modeling the desired system behavior as the operating principle of a system and applies this method as a way to model system behavior in automotive development projects;

chapter 8 provides modeling methods for defining and decomposing logical automotive systems and applies transformation rules to efficiently generate logical architecture models from operating principle specifications of a system function;

chapter 9 concludes this thesis, provides an overview of potentials for future work, and gives an overview of the experience gained in the industrial application of the methods used in this thesis.

1.3 Publications

This work results from a long time of research. Therefore, some aspects and parts of the results have already been published in other contexts. The following chronological list provides an overview of these publications and briefly summarizes their contents concerning their contribution to this work.

- [GKM+25] provides the transformation rules and evaluations of the method to derive logical architectures from operating principle models. The results primarily contribute to the method chapter 8 presents. With the evaluation and some insights from the application in the automotive industry, this publication aims at answering how can stakeholder values be formulated and not only contributes to the running example in subsection 6.1.2, subsection 7.1.3, subsection 8.2.3, but also set the basis for the method and transformation chapter 8 presents.

- [TMW+24] describes an extension of the method this thesis presents to enable virtual homologation of automated driving functions. To this end, the approach relates the stakeholder values to chapter 6 models and the operating principles chapter 7 introduces with test cases.

- [GORW23] introduces the process structure model this thesis revisits in section 3.2 and contributes to the lessons learned in section 9.3, which this thesis backs up with additional evaluation results.

- [BBK+22b] and [BBK+22a] present evaluations and ideas on specifying legacy requirements as DSLs for which this thesis provides related methods in chapter 5 and conducts comparable expert elevations in subsection 6.4.3, subsection 7.3.2, and subsection 8.3.2 based on the requirement classification scheme in subsection 6.4.2.

- [KRW22] provides the foundations of the definition of the use case diagram semantics, this thesis extends in section 6.2 for the application for automotive systems modeling using CUBE.
- [MSG+22] contributes to the development of safe systems in the automotive industry, by applying a SOTIF compliant variant of the CUBE method. Although the presented scenario format is not considered in this thesis, the ideas for use case scenario modeling influenced the method chapter 7 presents, especially in the context of the scenario behavior specification in section 7.2. Furthermore, the general visualization understanding of the CUBE procedure influenced the results on the CUBE process in section 3.1.
- [JGW+21] developed an example model to which this thesis refers in subsection 6.4.3, subsection 7.3.2, and subsection 8.3.2 to additionally provide results for the application of the method to publicly available models. Moreover, the thesis presents several ideas in the context of operating principle modeling, as chapter 7 provides.
- [HJK+21] presents results from the CrESt project in the context of artifact-based analysis for the development of collaborative embedded systems, which methodically contributed to identifying the artifacts created using the CUBE procedure in section 3.2.
- [DJR+19] introduces several concepts related to model-driven systems engineering in the context of virtual product design. To make these ideas applicable in automotive system development, section 2.2 and chapter 3 transfer these concepts into automotive system development.

Moreover, several other publications resulted from the study related to this thesis, which only have a minor or insignificant influence on this thesis:

- [DJR+22] presents the results of a cross-domain systematic mapping study on software engineering of digital twins.
- [GJK+21] provides additional results from the CrESt project related to architectures for dynamically coupled systems, especially in software product lines.
- [MGWJ20] presents concepts on how the results from the CrESt project are transferable for the development of automated vehicles.
- [DJK+19] translates MontiCore grammars and XText grammars and provides lessons learned from these translations.
- [NW17] provides methods for analyzing cryptographically-masked information flows in MILS-AADL architectures using slicing techniques.

- [Wac17] provides the context and foundations for the techniques [NW17] presents.

Finally, several student theses contribute to the results this thesis presents. Therefore, the following list provides an overview of these theses the author supervised or co-supervised and highlights their contributions concerning this work:

- [Bla23] provided a prototype implementation of the scenario behavior modeling template in subsection 7.2.4 as well as further results on modeling systems and system architectures in chapter 6 and chapter 8.
- [Zab23] developed a prototype implementation for the generation of natural language requirements from system models as subsection 6.4.1, subsection 7.3.1, and subsection 8.3.1 presents based on the templates in Appendix D and laid the groundwork for the evaluation subsection 6.4.3, subsection 7.3.2, and subsection 8.3.2 present.
- [Cho21] developed a model-driven development method for component structures based on assembly line models expressed as activity diagrams, which influenced the concrete syntax of the activity diagrams in the scenario templates in subsection 7.2.4.
- [Köb21] provided translation ideas and prototype implementations for the logical architecture generation in section 8.2 for which [Mal22] additionally suggested extensions.
- [Kat21] provided the initial grammars (*cf.* Appendix B) and the prototype implementation for the DSL in section 5.3 and its application in section 5.5.
- [Wac19] presents general modeling guidelines for SysML modeling in model-driven systems engineering projects published in [DJR+19].

In addition to these theses above, the author supervised and co-supervised multiple other theses, which focused on other aspects of software and systems engineering that did not contribute to the method and applications this thesis presents.

Chapter 2

Foundations of System Development in the Automotive Industry

Since system models are a broad field of research, this chapter introduces a common framework and understanding of concurrent systems engineering concepts to lay a foundation for their application in automotive system design methods concerning **RQ-1** ("Which established methods software and systems engineering methods exist that are applicable to specify automotive systems?"). To this end, this chapter structures as follows: First, section 2.1 introduces the overall automotive independent foundations of systems engineering. Next, section 2.2 refines the general concept of Model-Based Systems Engineering (MBSE) to a model-driven approach, which focuses on models and their automated generation to address the needs of the time- and cost-driven framework set by the automotive system development environment. To further formalize the concepts required in the automotive industry, section 2.3 applies the functional modeling paradigm to automotive system development before section 2.4 further describes the unique challenges of variant handling and variability management in the context of the automotive industry. To set a framework for suited systems modeling methods, section 2.5 introduces systems modeling languages to define and apply system models in the automotive industry before section 2.6 investigates commonly used systems engineering methodologies to identify shared concepts and open topics.

2.1 Foundations of Systems Engineering

Systems engineering is an interdisciplinary approach [Wor21] and the research foundation for the method specified in this thesis. Therefore, this section aims to derive a common understanding of systems engineering as a foundation for the method presented in the following chapters. To this end, this thesis draws on the previous analysis from [Wac19] that led to the understanding of model-driven systems engineering [DJR+19] published. Moreover, this section refers to several handbooks [She17, NPR07, WRF+15], standards [ISO15, ISO17b, ISO11c], and related works such as [Sta73, Rum96, Web09, Sch12, FMS14b] to relate the definitions this thesis uses with the state-of-the-art in related works.

2.1.1 System Definition

The system definition is a fundamental part of systems engineering research since it defines the engineered artifact, which is the central element of this branch of science. According to the original ancient Greek word definition translated in [LSJM96], a system (written as σύστημα) refers to a whole consisting of several parts or members. Although this definition still seems to be generally applicable to most systems developed in systems engineering today, many authors in the systems engineering literature introduce their system definition to emphasize additional aspects of systems in systems engineering. Consequently, researchers in systems engineering have published various definitions over the past years:

> A system is "the combination of elements that function together to produce the capability required to meet a need. The elements include all hardware, software, equipment, facilities, personnel, processes, and procedures needed for this purpose" [She17, NPR07].
>
> "A system is a purposeful whole that consists of interacting parts" [WRF^{+}15].
>
> "A system consists of a set of elements that interact with one another, and can be viewed as a whole that interacts with its external environment to achieve an objective" [FMS14b].
>
> A system is a "combination of interacting elements organized to achieve one or more stated purposes" [ISO15].

As these quotes reflect, many authors add some purpose to the system definitions considered in systems engineering [She17, NPR07, WRF^{+}15, ISO15] or a goal [ISO15] of systems developed in systems engineering. Furthermore, [WRF^{+}15] explicitly emphasizes that a system's parts interact with themselves and their environment to fulfill this purpose. To underscore this understanding and its connection to the word's original meaning, anything that meets the following definition is considered a system for this thesis. As a result, the systems considered in systems engineering are not simply composed of several parts of members but also fulfill a goal or purpose and interact with their environment to achieve this purpose. Therefore, the following system definition serves as the basis of the systems considered in this thesis.

Definition 1 (System)**.** *A system is a whole consisting of several parts or members that interact with each other and their environment to fulfill a purpose.*

More specifically, automotive systems are typically mechatronic systems *i.e.,* they are systems that consist of parts from a mechanical, electrical or electronic, or software domain.

Definition 2 (Mechatronic System). *A mechatronic system is a system that consists of parts from a mechanical, electrical or electronic, or software domain.*

According to this more specific definition, systems may also belong to different engineering domains required to work together, which reflects, for example, also in the systems definition from [She17] and [NPR07] that integrates the similar domains of the elements into the system definition.

As the interaction between the system and environment indicates, it is crucial to differentiate between elements that are part of the system and elements that are not part of the system *i.e.,* part of the environment [WRF+15]. Thus, systems engineering literature as [WRF+15, She17] introduces the concept of a system boundary. In contrast to the system specification, the system boundary does not describe the system with its properties and capabilities but demarcates the system from its environment. In other words, the system boundary delimits which elements belong to the system and which do not—remembering that a system's interactions with its environment are frequent and may also be crucial to fulfilling the system's purpose as stated in the system definition Def. 1. In particular, in developing automotive systems, there are several direct or indirect interactions of systems with their environment. One of the main tasks of many automotive systems, for example, is the transport of people and goods from one location A to another location B and to protect the people within the system (*e.g.,* against rain or injury), while protecting the environment from unacceptable influences from system (*e.g.,* from injuries during an accident with the system, or from pollution caused by the operation of the system). The consideration of different concepts to realize this task in a new upcoming vehicle shows that the differentiation of the system from its environment is by no means as trivial as it may appear at first glance, as the following example illustrates: To transport persons from A to B, all vehicles licensed for public road traffic today are not able to transport the vehicle and everything in it fully autonomously to this destination. Instead, they depend on a driver who controls or monitors the vehicle for the transport task. Therefore, most vehicles produced today offer the driver the ability to drive the vehicle from A to B according to his or her wishes using clearly defined control options (steering wheel, accelerator pedal, and brake pedal). The vehicle system interacts with its environment in two ways. Firstly, it receives the driver's steering commands and converts these into feedback with the road via the tires. Indirectly, other interactions of the vehicle and its components with the environment can occur through the vehicle during the transport task. For example, internal combustion engine generates climate-reactive greenhouse gases, which thus affect the entire climate in the long term, or noise emissions, which may negatively affect the acceptance of mobility in general.

In addition to the difference between a system and its boundary, this example also illustrates the concept of system functionality. Two critical aspects arise from requiring a vehicle to transport people and goods from A to B. First, that the influence of a system on its environment typically describes the functionality of a system: The interaction

of a system with actors in its environment is an indicator of the functionalities of a system. Second, that a system user describes the connection between a system and its environment. In this context, [ISO11c, ISO15, WRF$^+$15, ISO17a] and [ISO17b] also defines the concept of a *system architecture* as fundamental concepts or properties of a system in its environment. The system architecture describes the system elements, relationships, and principles of design and evolution [ISO11c].

2.1.2 System Formalization

Not only engineering in general, but also software and systems engineering in particular, uses scientific terminologies and concepts to design, create and sustain systems. Tough the same standards [ISO15, ISO17b, ISO11c] lay the foundations to define the the terms and concepts for software and systems engineering, their understanding varies depending on their use in software or systems engineering. Thus, this section summarizes the basic definitions of software and systems engineering to define them in the context of this work. A common element of software and systems engineering is the attribute.

Definition 3 (Attribute)**.** *An attribute describes the observable characteristics and properties of a system.*

Attributes are, for example, the speed of a moving object, such as a vehicle, a measurements of a piece of hardware, or data stored in a piece of software. Note that also abstract characteristics with complicated observability such as an ASIL-Level [ISO11a] in the context of a security characteristic or active/passive safety measures such as a Euro NCAP-rating [vR17] or a cybersecurity assurance level [ISO21] are usually regarded as attributes [Web09].

Next, variables serve to regard attributes in the systems' context.

Definition 4 (Variable)**.** *Variables are symbolic representations of attributes with a domain from which values are assigned to the variable.*

For instance, the letter v often denotes the velocity of a moving object that represents the variable's symbol. Next, a value of $1\frac{m}{s}$ from the domain of all velocities between zero and the speed of light may define the current speed of this object. Thus, $v = 1\frac{m}{s}$ describes that a variable with symbol v has the value of $1\frac{m}{s}$. While attributes describe an observable characteristic of a system and variables can be used to symbolically represent them, both concepts alone are often insufficient to observe a system's characteristics. For this purpose, we require a measurement.

Definition 5 (Measurement based on [WRF$^+$15])**.** *A measurement is the outcome of a measurement process, which determines the value of a variable [WRF$^+$15].*

A measurement process describes a systematic way to observe the system under given conditions to retrieve a specific value as a result of a measurement. For example, determining the velocity of a moving object requires to determine the length that this

object travels in a defined period of time. Therefore, counting the seconds the element needs to travel a distance of *e.g.,*, one meter, leads to the measurement x in the unit of seconds per meter ($\frac{s}{m}$). The multiplicative inverse of this measurement results in the the standard measure meter per second ($\frac{m}{s}$).

Whereas in computer science, a variable is often connected to a storage location, in systems engineering, it could also represent a degree of freedom to be set later in the engineering process, or an unknown in an equation depending on who controls the variable (*e.g.,* the engineer, the user, the system, the environment). Thus, lengths or speeds are sometimes only achieved physically but not measured, or stored in the system. Also these values might be relevant and measured for testing purposes, the results are not directly interrogable for its user. For example, the vehicle may have a wheelbase or window lifter rotation speed, that is used during the engineering process of the system, but not stored or used as a measurement in the vehicle. Consequently, some attributes or measures are directly accessible in the system, whereas others have to be measured, calculated, or estimated *e.g.,*, whenever a control software needs the attribute values as an input for its calculation. Apart from attributes, systems may also be in a specific state, which might influence their behavior.

Definition 6 (State)**.** *A state of a system describes a system's condition in which its observed attributes fulfill a specific constraint.*

For example, in a vehicle's stopped state, the vehicle's velocity has a constant value of $0\frac{m}{s}$, whereas in the driving state, the vehicle's velocity has a value that is lower or greater than $0\frac{m}{s}$.

Systems typically provide functionalities for their stakeholders to create a benefit. These functionalities lead to the concept of a feature according to the following definition.

Definition 7 (Feature based on [GKM+25])**.** *A feature is an independent end-to-end functionality of a system that benefits at least one stakeholder.*

A feature of a vehicle is, for example, the transportation of people. The owner or user of the vehicle as a stakeholder can use the vehicle to transport them or other people with goods from the current position to another using the vehicle. Because features might have pre- and post-conditions for their execution, features may only be available in different states of the system or trigger a state change by their execution. The transport people feature could, for example, be modeled to only be available in an operational state, where all technically required attributes are in an available range, but not in a maintenance or error state, in which a safe operation of the vehicle is not guaranteed, *e.g.,* because the tires of the vehicle are currently changed. While attributes describe a characteristic of a vehicle, states and features describe a behavior.

The systems engineering domain defined in [WRF+15] refers to system theory, originating in biology and physics [vB50, Ber69], in which it is common to differentiate

between two behaviors, the *dynamic behavior* and the *emergent behavior*. While the first describes an evolution of the system state over time, the latter describes a behavior that results from the combined behavior that cannot be understood based on the description of the system elements only [WRF+15]. Dynamic behaviors of a vehicle are, for example, *braking* and *accelerating*. Both behaviors describe how the vehicle's speed increases (for acceleration) or decreases (for braking) over time. Emergent behaviors in the automotive domain are for example traffic jams, that [NP95] models and [BDG95] describes as a behavior emerging from the behavior of all vehicles that interact on the road and even moves backward [BDG95] while all individual behaviors move forward. As automotive systems and system specifications are typically required to be understood in all safe situations, this thesis does not differentiate between these concepts and defines the system behavior as the dynamic behavior.

Definition 8 (Behavior)**.** *A behavior describes the evolution of a system state over time.*

During system development, it is often differentiated between two system representations. First, a so-called *black box view* that describes the system operating with only the system attributes from an external viewpoint. Second, a *white box view* that regards the internal structure of the system and its components in addition to the system's attributes as an internal view. The connection between these two concepts is essential to use these views. Starting from the black box view, systems engineers typically derive the white box view from the black box view by considering the systems' internal elements and connections. By combining internal view and external view, the systems engineer obtains a description of the overall system.

2.1.3 Systems Engineering Methodology

While the previous section focused on the definition of the system, this section now moves to the engineering part of systems engineering. As empathized in [Rum96], traditional engineering disciplines usually apply norms, standards, and guidelines to support the engineering processes in all product development phases. This observation also manifests in the different views on systems engineering. As the definition of the system in systems engineering, also the the definition of systems engineering depends on the considered literature as the following examples reveal:

> "At NASAs, systems engineering is defined as a methodical, multi-disciplinary approach for the design, realization, technical management, operations, and retirement of a system" [She17].
>
> Systems engineering is an "interdisciplinary approach governing the total technical and managerial effort required to transform a set of stakeholder needs, expectations, and constraints into a solution and to support that solution throughout its life" [ISO15].

> "Systems engineering is a multidisciplinary approach to develop balanced system solutions in response to diverse stakeholder needs" [FMS14b].

Despite their differences, all definitions of systems engineering mentioned above have in common that they highlight an *interdisciplinary* aspect of systems engineering, which also manifests in the different systems and engineering disciplines involved in systems engineering. As [WRF+15] explains, systems engineering is typically applied in large and complicated system development projects in different disciplines, such as medical, infrastructure, and transportation. All these disciplines have in common that multiple disciplines are of interest and involved in the system development process. For example, biochemical engineering experts must understand the human body and its behavior in medical system development. In contrast, technical experts from electrical, mechanical, and software engineering are needed to develop systems that cure or analyze the human body in different application scenarios.

Additionally, in most transportation systems, many engineering disciplines must interact to realize transportation systems. For example, chemical, electronic, or other physical processes build the foundation to implement a propulsion system based on energy conversion. Moreover, boundary conditions from the environment, such as properties from fluid mechanics, affect the system's ability to move efficiently in its environment. Furthermore, broader environmental influences on or from the system must be understood to ensure sustainable and resource-efficient use and disposal of the transportation system. Because of the interdisciplinary nature of the systems developed with a systems engineering approach, systems engineering must also be an interdisciplinary approach to reflect the interdisciplinary nature of the analyzed and developed systems. Another essential aspect of systems engineering is stakeholders and their needs, as especially the definitions from [ISO15, FMS14b] indicate.

Definition 9 (Stakeholder according to [ISO15])**.** *According to [ISO15], stakeholders are individuals or organizations with a right, share, claim, or interest in a system or its possession of characteristics that meet their needs and expectations.*

These needs and expectations from a stakeholder manifest in the fixed term of *stakeholder needs*. The work from [GJ10] urges system developers to pay more attention to stakeholders that are neither individuals nor organizations, such as the environment or other technical systems. Therefore, this thesis also regards any other actors as stakeholders as long as they might withdraw their support if their wants or expectations are unmet or inflicting unacceptable levels of damage to the system or its environment [GJ10]. The following SE definition describes the definition used throughout this thesis and combines these views.

Definition 10 (Systems Engineering (SE))**.** *Systems engineering is a multidisciplinary approach that provides methods for a system's design, realization, technical management,*

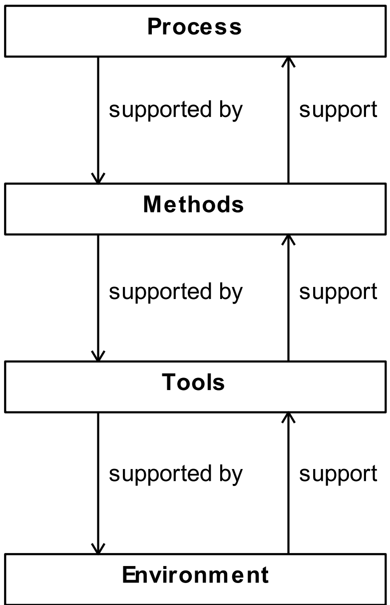


Figure 2.1: The Process, Methods, Tools and Environment (PMTE) paradigm based on [Mar94].

operations, and retirement. Systems engineering aims to provide system solutions based on diverse stakeholder needs for the system's complete life.

As this definition shows, systems engineering summarizes multiple methods in different phases during a system's life cycle in the context of processes, methods, and tools embedded in an environment.

As [Est07] describes, differentiating between the terms systems engineering *process* and *methodology* is crucial. This difference manifests in the underlying PMTE paradigm [Mar94] that defines the relationship between processes, methods, tools, and environment as follows:

- A *process* describes a logical sequence of required tasks to achieve an objective. In other words, it describes "what" has to be done without specifying "how" to perform this task.
- A *method* specifies the techniques needed to perform a task. Therefore, it describes "how" a task has to be done and concertizes the task description from a process.
- A *tool* is an instrument that aids in executing a task by supporting the applying

a method. It aims to enhance the efficiency of the task when applied by a user with the right skills and training. Therefore, it refines the “what” and the “how” dimension [Mar94].

- An *environment* consists of external objects, conditions, or factors that influence an object’s actions, person, or group [Mar94]. The conditions are often social, cultural, personal, physical, organizational, or functional. Typical environment categories are *e.g.,* computing environment, communication environment, personal environment, managerial environment, physical environment, and system life cycle environment. According to [Est07], purpose of a project environment should therefore be to integrate and support the use of the tools and methods used on that project.

According to the PMTE paradigm, a process is supported by methods, which are supported by tools integrated into and supported by an environment [Mar94]. As some company requirements might even be hostile to introducing new methods and tools [Mar94], it is vital to have a supportive environment for introducing new methods and tools. To better understand this concept, Figure 2.1 presents the relationships graphically. Combining the elements of the PMTE paradigm leads to the notion of a methodology.

Definition 11 (Methodology as in [Est07])**.** *A methodology is a collection of related processes, methods, and tools. It aims to provide a general approach that applies these related processes, methods, and tools to a class of related problems.*

This general definition of a methodology applies to a systems engineering methodology by restricting the application to processes, methods, and tools regarding related systems.

Definition 12 (Systems Engineering Methodology)**.** *A systems engineering methodology is a methodology that collects related processes, methods, and tools to apply them to a class of related systems [Est07].*

While methods and tools are typically company-specific, there exist several standards and related work on systems engineering processes such as [ISO15], [She17] or [WRF+15]. Since these processes aim at general applicability, the life cycle models and systems engineering processes often are only partially applicable in some system development projects. Therefore, the guideline from [WRF+15] suggests a project, industry or company-specific tailoring of these processes to adapt it to individual needs. However, the tailoring of methods is a balance between risk and costs. As processes that profoundly diverge from a standard are often not well explored and, therefore, more likely to fail [WRF+15]. On the other hand, untailored processes are often more time-consuming and costly to execute, since executing all defined process steps is mandatory, regardless of whether or not executing the process step is helpful in this level of detail in the respective company or system development project [WRF+15]. A detailed overview on the tailoring and the summarized standards is for instance provided in [ISO15] and [WRF+15].

2.2 Model-Driven Systems Engineering

There is a common differentiation between two contrasting approaches to software and systems engineering made in the systems engineering literature: the *document-based* and the *model-based* systems engineering methodology [WRF+15, FMS14b, Rum16]. For example, [WRF+15] states that the document-based approach to systems engineering is the "traditional" systems engineering methodology. Following this approach, the system specifications manifest in documents *e.g.,* interface control documents, system description documents, trade studies, analysis reports [WRF+15], or test reports containing verification and validation results. Since the document-based approach relies on numerous stand-alone documents, the different document-responsible engineers often edit these documents independently. As a result, inconsistencies between and within documents must be accepted in editing at first and recognized and corrected in a second step [Rum16], the main challenge of a document-based systems engineering approach is the maintenance and synchronization of information contained in the documents, which results in additional efforts for document handling, which is also the primary drawback of the approach.

In contrast to this traditional document-based methodology, the model-based approach to systems engineering captures the system information in a unified system model [WRF+15]. According to [Sta73], a model must fulfill three characteristics:

1. It must reproduce or represent an original,
2. it must reduce the properties of this original to those that are known or relevant to the modeler and
3. it must fulfill a replacement function to achieve a specific purpose.

System models in systems engineering typically fulfill these characteristics, as:

1. A systems model represents the system under development in a systems engineering process.
2. It reduces its properties to those known or relevant in the current development state.
3. Serve different purposes throughout the system development process, such as system design, realization, technical management, operations, or system retirement.

Considering automotive system models, this understanding is extendable to the following definition of an automotive system model.

Definition 13 (A System Model in the Automotive Industry)**.** *A system model in the automotive industry is a system model according to [Sta73] that reproduces or represents the original automotive system.*

As a paradigm to create these models, the model-based engineering paradigm is applicable [Sch12]. This principle is often applied in mechanical, electrical, and software engineering [FMS14b]. In mechanical engineering, for example, traditional hand-drawn part drawings that specify the design of a mechanical component shifted to digital CAD models that support the manufacturing process after the design phase and enable simulation for cheap and efficient verification and validation. Moreover, mechanical system design methodologies like those presented in [PBFG07] or [BG21] further formalized the engineering design. Likewise, electrical engineering changed from document-based circuit design that draws on drawings to a computer-aided circuit design and analysis method. Consequently, software engineering [Rum96, Rum16, Rum17] and CASE [MNS96], also use computer-aided methods. Starting from the 1980s and particularly after the introduction of UML in the 1990s, graphical models slowly gained relevance by providing additional abstraction levels on top of the programming language.

A similar trend from document-based to model-based development approaches is also observable in systems engineering where a similar mathematical formalism for MBSE was introduced simultaneously in [Wym93]. Which, the INCOSE later identified as a goal to improve systems engineering in their "Systems Engineering Vision 2020" [INC07] by proposing a further integration of MBSE methods into the SE process.

In addition to this development goal, the INCOSE also provides a general MBSE definition, which is also used in many other relevant publications [FMS14b, WRF+15, Est07] and is also used in this thesis.

Definition 14 (Model-Based Systems Engineering (MBSE) Definition as in [INC07])**.** *Model-Based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.*

Focusing on automotive system development, models can also have other particular forms. Mathematical models aid in decision-making based on several criteria [AK94], or to assist in further developing the vehicle features and attributes that increase customer satisfaction and thus combine business and technical decisions [YG08]. Furthermore, models simulate vehicle properties [DRY01] or vehicle production [TW05, WMO+16] in different projects. Moreover, physical vehicle models and people models are also used for crash simulation, especially in the field of safety [HASH07]. As these examples show, models can have different forms, such as mathematical, textual, graphical, or physical. Additionally, system models can have different purposes. For example, system models in the systems engineering process not only enable systems engineers to specify the *structure* of system components based on their boundaries and connections but also to specify a components *behavior* in terms of the reaction of the system to its inputs *e.g.,* in terms of its physical movements, communication, decision making, or other observable actions. Moreover, requirement and parametric specification mechanisms

support specifying constraints these elements should fulfill. In a well-performed systems engineering process, the models for different system development steps should be reusable as inputs of the following development steps. In this case, the system model elements divide into connectable model elements of different domain-specific modeling languages. For example, the overall component design could be described in the SysML, all software components in a programmable version of the UML and all hardware components in CAD and CAE models. For example, the models or model elements of a unified model used for system design should be suited for a system analysis, which themselves should be suited for system verification and validation. From this model reuse for different development steps arises the challenge of making the created system model reusable for all these different purposes. While reusing descriptions in the context of system requirements and design needs an excellent interconnection within a logical domain, describing and executing test cases often requires various system model elements to be applicable in the involved engineering domains. During the testing of a system functionality, for example, co-simulations are a way to connect simulation model elements from different engineering domains into an overall system simulation. Manual or automatic exchange mechanisms are applicable to exchange all necessary information between the different system model elements. For example, CAD models from the development stage could be used to generate production instructions for the production stage [DJR+19]. Therefore, a system model may serve as a crucial driver in system engineering projects whenever high degree of automated artifact generation and tool connection is possible and provided.

However, as stated in [Lev09], engineers use the word model in such a vague way that a *model* would include any *description* in an engineering context. Thus, a vague MBSE definition as Def. 14 provides also includes a creation of multiple graphical models in the same form as a document-based SE approach. The primary reason for this is that creating a formal modeling alone does not imply that the created models generates any use for system development or are further used as artifacts of the system development process, as also [BR23] states as one of its key insights. For example, [Lev09] states that any system description that uses text accompanied by graphical models and other formal descriptions would meet an interpretation of the MBSE definition in [WRF+15], which in [Lev09] is also called a *description-based* engineering approach. deliberately this is done the better. Consequently, if a document with comprehensive model information becomes the primary artifact of the SE process, the difference to the document-based SE approach is almost negligible. Even worse, using models this way might only increase the effort for the model creation on top of the efforts required for the document-based process. Hence, all drawbacks of this approach still apply to this process without adding the benefits of a model-based systems engineering approach.

To differentiate between these potential application of models in software engineering, software engineers use a *model-driven* approach whenever the model should not only be an ordinary development artifact but a vital artifact of the development approach. As [Sch12] defines, model-driven software engineering is a model-based software engineering

approach in which formal models serve as a central and essential artifact. To this end, [Sch12] defines the automatic processing of models in different software development stages using methods such as analyses, generation of automated test cases, or generation of complete software systems. To apply this definition to systems engineering, the term Model-Driven Systems Engineering (MDSE) is used in this thesis to describe a model-based systems engineering approach, where system models are the key artifacts of the process based on [DJR+19].

Definition 15 (Model-Driven Systems Engineering (MDSE) based on [DJR+19])**.** *Model-Driven Systems Engineering (MDSE) is a MBSE approach in which system models are the key artifacts of the engineering processes. For that reason, system models created in a MDSE approach must be sufficient to generate (by model refinement or transformation) other descriptions that address system requirements, design, analysis, verification, and validation activities.*

Apart from the systems engineering methodology, other modeling paradigms and principles can be applied during a model-driven systems engineering methodology. Moreover, the term MDSE only defines the role of models in a systems engineering process supported by methods, but not the kind of processes, methods, and models suited to describe and develop systems in a model-driven approach. Consequently, the following sections further investigate different modeling paradigms, languages, and methodologies, as well as their relevance for the system development processes of the automotive industry.

2.3 Functional Modeling Paradigm for Automotive Systems

Especially in the automotive industry, innovations in system development are driven by new features, *i.e.,* functionalities the vehicle can provide to its users. As [BMM09] shows, the number of innovative features increased drastically from 1950 until today, which not only manifests in the change of a mechanical design but also the integration of electrics and electronics [CBK+13], and software components [EF17]. As defined in Def. 7, these new features consist of independent end-to-end functionalities that benefit at least one stakeholder. One of the main challenges in system development is the description of these functionalities and their integration into the product under development.

The functional development paradigm[HNZ+23] describes a system's functionality in terms of how it interacts with its inputs and outputs [HE88]. To precisely specify and model such functional interactions, [BS01] introduces the FOCUS theory, which formalizes exchanged processed information within and between several functions as streams, and the functions as stream processing functions as an exchange of discrete data messages [Bro10, Bro18]. In this context, several other works focus on different aspects of modeling such systems. For example, [Rum96] provides a general method to specify distributed object-oriented systems, [Wor16] extends the MontiArc modeling language

from [Hab16] for also modeling automata based on the foundations of the focus theory, and [vW20] provides another extension to express embedded systems.

All of these methods have in common that they mainly focus on the software side and information flows. Since cyber-physical systems such as modern vehicles process many different inputs and outputs such as fuel, electrical energy, luggage, persons, control inputs, media information, or radio signals, it is hard to differentiate between their input and output types in more general considerations that also include energies and materials. Thus, two issues arise: First not all of these signals are discrete, second a differentiation between the different occurring types is challenging. To address the first issue, the FOCUS theory was extended to dense and hybrid systems [Bro01b, Bro12]. To overcome the second issue, construction design literature such as [PBFG07] or [KK98] suggest differentiating between energy, material, and signals. The idea emerged from the philosophical theory of ur-alternatives, also called archetypal objects [Wei84], which identifies energy, matter, and information as elementary concepts for system description. Because the functional specifications considered in [KK98] and [PBFG07] only consider information that systems exchange, they only consider signals, *i.e.,* physically encoded information for the exchange between entities. As this concept conforms to the understanding of *data* in computer science, the remainder of this thesis refers to it as data. Because the development of a cyber-physical system is based on information exchange between the members of the system or between the system and its environment [BS01], only data is suited for an exchange in this model; therefore, unencoded information cannot appear for consideration in this framework.

As information encoded in data, also *energy* is typically conveyed in different forms between systems or between a system and its environment. For example, an electric motor converts electrical into mechanical and thermal energy [PBFG07], a combustion engine converts chemicals into mechanical and thermal energy [PBFG07], and a steam engine converts thermal into mechanical energy [PBFG07]. Thus, many forms of energy exist, such as chemical energy that results from chemical reactions, mechanical energy, such as kinetic energy of a moving object, potential energy stored by an object, or thermal energy. Since energy cannot be lost, which manifests in the law of energy conservation that states that the total energy in isolated systems stays constant [FLS65], energy must be transformed to achieve a different form of energy. Due to this law, no system without an external power supply can deliver an unlimited amount of energy [Pla03]. Thus, a perpetual motion machine, *e.g.,* a vehicle that drives forever without an external power supply, cannot exist. Since the notion of energy in the functional modeling paradigm relies on the abstraction of energies from its transmission encoding, material flows of chemical substances solely used for energy production are energy flows, not material flows. Thus, instead of modeling a fuel flow as a material flow, the functional modeling paradigm would require modeling an energy flow for chemical energy.

As this example shows, also *materials* are conveyable in different forms. According to their physical definition, everything with a volume and a mass is a material [SB91].

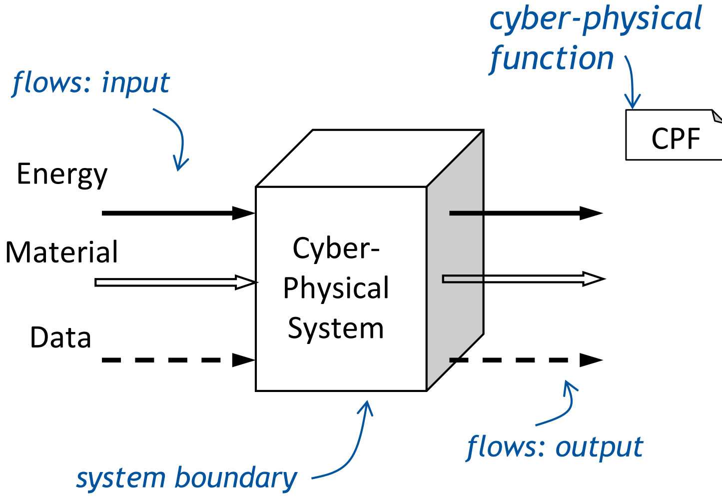


Figure 2.2: A Cyber-Physical System processes input and output flows of energy, material, and data.

Apart from this differentiation, materials can be solid or fluid, which results in different behaviors when used at the inputs and outputs of a system. While solid items form a flow of discrete items, fluid materials are conveyed continuously.

Moreover, as stated in [PBFG07], materials cannot only be separated, dyed, coated, packed, transported, or reshaped, but also have their state or shape changed entirely. For example, production plants often turn raw materials into part-finished and finished products [PBFG07]. In this process, the mechanical parts reach their final shapes and surface finish during production and also undergo rigorous tests that might ultimately destroy them. Throughout a product's life, the product itself or its material can also reach the end of life. It might be recycled, which in the reverse direction transforms finished products into raw materials for new products.

Without a semantically sound modeling mechanism, it is possible to use a natural language to describe a function. [PBFG07] suggests using verb and noun pairs such as "increase speed", "transfer torque", or "reduce speed" for a natural language specification followed by a specification of the physical quantities.

Time is essential in these concepts, especially when the systems exchange data, material, or energy at their inputs and outputs. The notion of time is implicit in the specification of cyber-physical functions in the channels that input energy, material, and data.

Since technical systems such as vehicles typically perform more than one function, the function of a technical system can be summarized as a cyber-physical function (*cf.* Figure 2.2) that the system performs based on the inputs and outputs it receives or sends at its boundary. These functions are not intended to represent a specific techni-

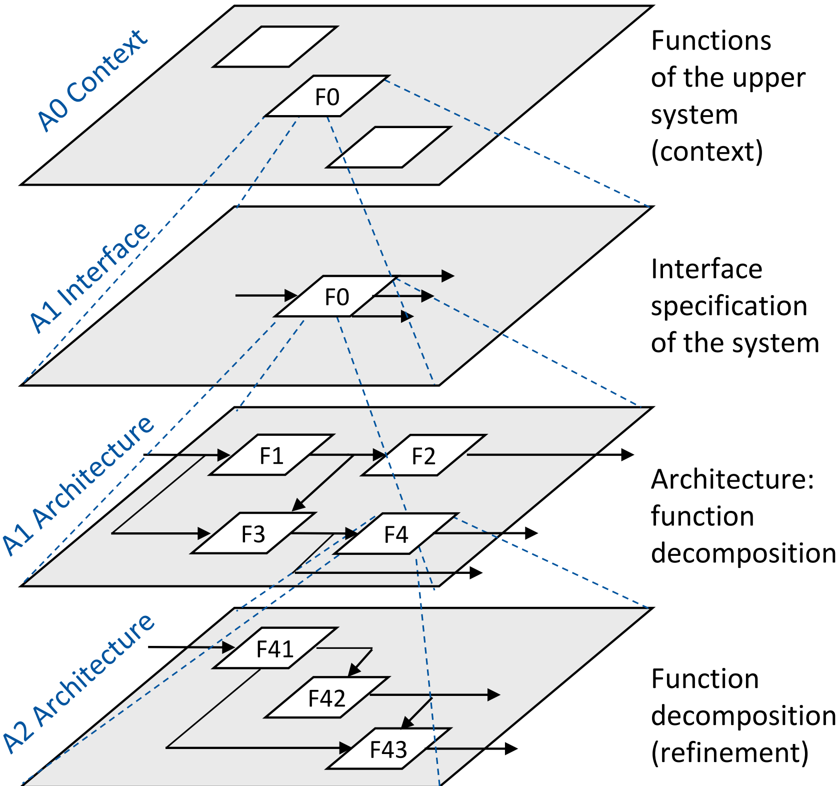


Figure 2.3: Functional decomposition of an overall function to interconnected subfunctions on decomposition levels based on [PBFG07].

cal solution but intentionally represent an abstract solution-neutral task. By this, the technical implementation is still open in later development phases, and better-suited solution alternatives are unrestricted as long as they are suited to perform the abstract task specified by the cyber-physical function. Because systems often fulfill multiple purposes, a system may also have multiple functions with other input and output behaviors. For that reason, it is good practice to create multiple views on the system *e.g.,* one for each function, to make a differentiation between the functions more accessible. By this, the black box view that only describes the inputs and outputs of a system emerges into a more transparent view, in which only the currently relevant input and outputs are visible.

Functional decomposition is an important principle that further specifies a system's function. Following the divide and conquer principle, functional decomposition subdivides complicated functions into multiple distinctive subfunctions that each fulfill subtasks of the overall function and together realize all aspects of the overall function. According to [PBFG07], the overall function must additionally govern the interaction

of the subfunctions by certain constraints or ensure that one subfunction finishes before another subfunction starts. The overall functionality, however, is still a composition of the individual subfunctions. Moreover, combining different subfunctions that realize the same or a slightly modified overall function is a good starting point for variability or related system architecture alternative evaluation as required in automotive standards such as ASPICE [SPI15]. To visualize these interacting functionalities and functions, [GHK+08a] suggest the introduction of *function nets*, *i.e.,* views of related functions, to visualize the interconnection of systems for features, variants, and modes. The same principle is also suggested in construction theory [PBFG07] under the term *function structure* to subdivide a black box view of a system into interconnected black box views of a system.

2.4 Variability in the Automotive Industry

System variability is one of the significant complication drivers in the automotive industry. Since manufacturers of automotive systems have to constantly adapt their system due to changing technology and customer requirements, those manufacturers who can produce and market their products particularly well under these changing conditions have been successful for some time [ZVC06]. Consider, for example, a car manufacturer that can adapt its infotainment system to different brands and vehicles compared to a car manufacturer that has individually developed this system for each vehicle. In that case, the manufacturer with the possibility of adapting its system, can massively reduce the development efforts because, instead of developing several different systems for each brand or vehicle, only one adaptable system must be developed. Although the development effort for this adaptable infotainment system is higher than the effort for an infotainment system for a single vehicle, the total effort for developing an adaptable system is lower than developing independent systems for different manufacturers of different vehicles. Therefore, the manufacturer with the possibility of adapting its system has an advantage over the manufacturer without these possibilities. However, in addition to developing a customizable system, it is also conceivable that instead of developing a customizable system, manufacturers could develop a series or line of several related systems where the development and production effort is reduced, for example, by designing the systems such that some parts are reusable. For example, many vehicle manufacturers produce series or lines of vehicles that share the same parts, such as doors or engines, to reduce each vehicle's cost and development time [Web09]. In order to offer these flexibly adaptable products and product lines, the automotive industry uses various techniques to handle different system variations, which are generally summarized in this thesis under the term system variability. These variability techniques aim at developing adaptable *i.e.,* variable systems that can cover several use cases instead of a specific system for a concrete use case. As the examples of the infotainment and drive system

show, there are two approaches: Either develop a customizable system that can cover multiple use cases or can be adapted to different use cases by simple customization, or develop a group of related systems using the same development artifacts that address these use cases collectively. Since most automotive systems are mechatronic systems, approaches are therefore used not only for hardware development but also for software development.

2.4.1 Mass Customization

Mass customization is one of the principles automotive manufacturers apply in systems development to adapt the system to changing customer requirements. The literature on mass customization in [TVRN01] and [TVRNRN04] defines *mass customization* as the ability to produce varieties of customized products quickly, on a large scale, and at a cost comparable to mass-production through technical and managerial innovations [TVRNRN04]. It aims to produce high-volume customized products at low costs [DBF01]. To achieve this aim, [JCA04] suggests postponing the differentiating of a product for a customer to the latest possible point in the supply network. For the production of a vehicle, this means, for example, that the adaption of the vehicle to the customer requirements is traditionally postponed to the production of the vehicle [ASN00] or even left to the customer in after-market sales, where the customer can buy and integrate additional functions at the dealer or in the in-vehicle infotainment system [BCSLS05].

To achieve product customization, [ASN00] suggests three strategies for different production volumes. For lower production volumes and customers with a low cost-sensitivity [ASN00] suggests a *core customization*, where a core product serves as a starting point for tailoring a bespoke product. Based on the customer's selection of predefined and individually tailored features as well as uniquely defined requirements, an individual product variant of the core product is developed. Therefore, core customization achieves a high degree of customization, but might result in high efforts and costs. As a consequence of core optimization, not only might the design and manufacturing of the resulting system might change, but also the manufacturing and distribution of the product. For high-volume system development projects with cost-sensitive customers, [ASN00] suggests an *optional customization* approach from a plethora of options that are integrated into the system design such that the system design does not change due to the selection of the available options. Finally, for high-volume systems with highly cost-sensitive customers [ASN00] suggests a *form customization*, where the form of the standard product is changed at the dealer and not at system production, *i.e.,* by later adding parts or configuring the product later at the dealership to tailor the system to the customer needs. Apart from these technical form changes, contract changes such as different warranty options, insurances, or service options are also part of this form of mass customization.

Since the automotive industry does not only distribute vehicles to end customers (OEMs) but also vehicle parts to vehicle manufacturers (Tier 1), non-vehicle parts to

vehicle part suppliers (Tier 2), or raw materials (Tier 3), all of these configuration options also apply to business-to-businesses contracts and mixed supply chains. To provide a core customized vehicle to an end-customer, an OEM could, for example, contract a Tier 1 supplier to produce an optional customized vehicle system, for which the Tier 1 supplier contracts a Tier 2 supplier to produce form customized products. Thus, these customization principles are often not applied individually but combined within the system and its decomposition.

2.4.2 Modularity and Platforms

To enable mass customization for manufactured automotive systems, automotive companies use modularity concepts [WLDM20]. According to the general definition of modularity-based manufacturing from [Ulr95], modularity is the application of standardization and substitution to create modular components and processes suited for configuration into a wide range of products to meet specific customer needs. As stated in [Sch00], system theory, also used as a fundamental of systems engineering, is very suited for defining modular systems. The main reason is that systems engineering strives to decompose the system of interest into system components that are again systems decomposed to components. Hence, this form of decomposition supports the substitution principle that the definition of modularity according to [Ulr95] requires. Especially in the automotive industry, modularity plays an important role. For a better distinction, [Mac13] identifies three subtypes of modularity.

First, there is the *modularity-as-property* definition according to [Ulr95]. Following this understanding, modularity is a system or product architecture property in which a precise and unique mapping exists from system functions to executing components. Consequently, according to the modularity-as-property understanding, an ideal system has a one-to-one mapping of functions to a realizing system or physical component. As a result, the interface design of the system primarily serves to decouple systems into reusable modules. Integrality is an opposing property of a system that realizes a one-to-many or many-to-one architecture *i.e.,* a system that fulfills multiple functions or functions realized by multiple systems. As stated in [Mac13], this understanding of modularity is rarely achieved or desirable, as it fits relatively few products. One of the main reasons for this is the resulting interfaces, which strive to achieve a functional decoupling of the realized systems to achieve a one-to-one mapping of functions to systems but neglect other characteristics, such as the utilization and load of the input and output channels. As a result, the interfaces tend to cause a high information, energy, or material exchange, which usually do not meet other system design requirements such as performance or specific quality requirements. Consequently, other works on modularity-as-property [Sch00, Fix05, Bal08] argue that modularity is only a range since all systems can have some form of modularity [Sch00], that a non-ideal but sufficiently good system has fewer inter-module dependencies than intra-module dependencies [Bal08], and that

the degree of modularization must follow from the system decomposition [Fix05] (and not vice versa, as an ideally modularized system would aim for).

Based on these findings, [Mac13] identifies the second understanding of modularity in the automotive industry, the *modularization-as-process.* As stated in the observations of [Sim91], a system decomposition is not solely an architectural property that leads to a single goal (such as modularity) as the previously described understandings emphasize, but is more an evolutionary process that strives to gradually advance one ore more design goals [Mac13]. Thus, the system architect, *i.e.,* the systems engineer that defines the system decomposition, system boundaries, and its interfaces, plays an essential role in this process, as the role is concerned with handling and evaluating different interests and objects, which may lead to different system modules and interfaces [BC00].

Finally, [Mac13] aims to combine both views for the automotive industry by introducing *modularity-as-frame*, which sees the modularity of a system as a mental frame for systems engineers, which derives a particular dynamic and directionality as a result of the interplay between modularity-as-property and modularization-as-process [Mac13]. In other words, modularity as a frame accepts that modularity is not only a particular property of a system but also an inevitable process in which system designers who strive to design a modular system must constantly adapt a modular system design to different goals or priorities. Based on this observation, [Mac13] argues that modularization-as-process might not always lead to a greater modularity-as-property but might also cause an unintentional integrality. As an initial frame for a modular system development, systems engineers equally rely on a modularity-as-property and modularization-as-process understanding, which then has to be adapted and optimized in the following steps using principles and concepts from both modularity-as-property and modularization-as-process. Thus, [Mac13] argues that seeing modularity as a cognitive frame might assist systems engineers in shaping the interrelationship and dynamics of change between these two aspects.

As a particular case of the general modularity concept, the automotive industry uses the *platform* concept [ML97, Muf99], in which modular components, modules, and technologies form a reusable basis for various products. A platform of modular components forms a building kit or toolbox for system designers, which provides a set of reusable and modifiable elements forming the backbone of the system development. Using these building blocks enables automotive system designers to design various products for different market segments and application scenarios by combining different components from a platform in various ways, configurations, or with additional non-platform elements. By this, a platform provides several benefits for automotive systems engineers: First, reusing already developed components reduces the development costs as no additional development efforts are required; second, a lower development effort for new components leads to faster times to market; and third, providing a different possible combination of the modular building kit enables automotive system developers with greater flexibility to changing market demands. As an additional organizational advantage, using a common

platform enables automotive manufacturers and suppliers to distribute the platform development costs over several development projects since the same platform serves as a basis for different systems [Muf99]. By this, the platform components, which for a single system might cost more in the development due to additional modularity development efforts [SRW98], realize savings as the reuse of the modular components saves more effort if significantly many systems use the elements from the platform [FS03]. Especially for mechanical and electrical components, companies that apply the platform concept in their product development and production report additional cost savings since mass production of standardized parts usually improves the product quality while reducing the production costs, as shown in [HU14].

Because of these advantages, many automotive manufacturers use platforms in their vehicle development projects. One application scenario is cross-brand platforms, where different brands of a group of vehicle manufacturers share a similar platform. For example, the Volkswagen Group uses the MQB (Modularer Querbaukasten) platform throughout several brands of the Volkswagen Group with transversely installed combustion engines for the MQB such as VW, Skoda, Audi, and Seat. Apart from these inter-group platforms, shared platforms between manufacturers of different groups are possible and commonly used in the automotive industry, as BMW's Cluster Architecture (CLAR) platform shows, which is used for a broad range of BMW models and shared with Toyota for the fifth generation of the Toyota Supra. Even though the platform concepts are often associated with vehicle platforms, system and component platforms are common. As OEM system platform, the Volkswagen Group has, for example, the MIB platform, which provides a platform for the infotainment systems used in the group brand. While predominantly OEMs also have system platforms for propulsion or electrical systems, suppliers often produce their system components following a platform concept. For Tier 1 suppliers, Continental has, for example, the Continental Automated Driving Platform for the development of autonomous driving systems, which provides a variety of adaptable components, including sensors, actuators, and control units for integration in different vehicle platforms and configurations. Likewise, Bosch's Automotive Connectivity Platform provides software and hardware components for connected vehicles, such as connectivity modules, cloud services, and software frameworks. It aims to be scalable and adaptable to different vehicle models and applications. Especially for software, the automotive industry relies on standardized platforms such as the AUTOSAR (Automotive Open System Architecture) platform. The platforms were developed as an open-source standard to define a common software architecture and development methodology for automotive electronic control units (ECUs).

2.4.3 Software Variability, Features and Functions

As for mechanical and electrical components, the automotive industry systematically reuses modular software units, aiming to achieve a higher software quality at reduced

costs [Sch19].

Software parameterization [Gog96] is a simple but essential technique to enable software variability in automotive software engineering. This approach uses programming concepts, such as function parameters or global variables, to adapt the software to meet the needs of a specific configuration in the context of a modular architecture. Following this approach, the software components' design and implementation support a configuration based on a set of parameters that serve to customize or modify the software to meet the specific requirements of different vehicle or vehicle system models or variants. These parameters configure the software not only at design and production time but sometimes also serve as a basis to reconfigure the software in maintenance later or to reconfigure functionalities in operation after the customer receives his vehicle. Since the software of modern vehicles often controls crucial vehicle functionalities such as the engine or transmission, re-configurable parameters aid automotive software engineers in defining performance characteristics of that function, such as power output, fuel efficiency, or acceleration. By this, the same engine control software is reusable for different engines in different vehicles, saving development efforts and reducing the certification and testing efforts [ON15].

Although software parameterization is a simple and effective way to develop variable software modules rapidly that aids in an easy integration into existing software development processes without significant changes, it bears several risks and disadvantages. First, the parameters might blow up and complicate the software if the application scenario requires many parameters. Second, its applicability is hard to manage in complicated variability scenarios, for instance, when the dependencies between vehicle variants do not affect one but many interconnected and variable interdependent software modules or when the configuration of the parameters correlates according to a complicated logical relationship. To avoid these connections and other drawbacks, software developers might refrain from introducing many software modules but strive to implement many functionalities in the same monolithic software modules for which the parameters seem more straightforward to manage. Consequently, a tendency to use integral instead of modular software components is the third disadvantage of this approach, which, applied on a large scale, could prevent its intended usage: The creation of modular and reusable software components.

To overcome these issues, Software Product Lines (SPLs) are a paradigm to develop a set of related software products that share a common platform [PBL05] that provides a set of features and is configurable to meet specific requirements of different vehicle models or variants [CN02]. A SPL enables automotive system developers to design and maintain a reusable core set of software components across multiple vehicle platforms, reducing development time and costs. To this end, SPLs captures commonalities and variability of a developed set of software systems to derive multiple concrete software products utilizing a variant deviation mechanism [GJK+21]. Different methods exist for the development of SPLs [GJK+21]. First, there is the proactive approach on SPL

development, which designs a SPL from scratch [PBL05]. Second, there are extractive SPL engineering methods, which aim at extracting SPLs from a given set of software products [PBL05].

One important concept of SPLs is the feature model [ARS+14]. In contrast to the feature definition in this thesis (*cf.* Def. 7), these features represent variation points that in software often correlate with system behavior as specified in Def. 7. Moreover, not everything that a car configuration or manufacturer sells as a feature is a feature according to Def. 7. For example, most vehicle manufacturers allow the customer to configure features such as the vehicle color, a chrome finish, or an advanced driver assistance function package, which are customer features from a variability and marketing perspective but not technical features from a systems engineering perspective since they are only observable characteristic of the vehicle or packages of system functionalities. Consequently, the marketing features that lead to the vehicle configuration for vehicle production are variation points and not technical features in this thesis.

Finally, software variability is present in automotive software engineering through software architecture patterns and design techniques that support the development of configurable software systems. These patterns and techniques enable the creation of software architectures that can be easily customized or extended to meet the needs of different vehicle models or variants while maintaining high consistency and maintainability.

2.4.4 Consequences for Systems Engineering

The previous introduction to variability in the automotive industry shows that variability is an elementary aspect and, thus, a crucial factor for automotive system development. Hence, these aspects are required to provide a suitable framework for the methods used to define and apply system models in the automotive industry. However, managing variant variability and product lines is not this thesis's primary focus. As mass customization is omnipresent in the automotive industry, the design of automotive systems must not only enable the mass production of customized products but also allow individual customization of the systems to be achievable despite the high production volumes. Methodologically, this means that methods for system definition must have a mechanism for handling variability.

Furthermore, as in automotive manufacturing, a high degree of modularity and reusability must be achieved in system development so that system models and system development artifacts are reusable across projects. Specifically for a system model, this means that many of the models created in the automotive industry could be used not only for a system development project with a concrete product as the project goal but also for a platform project with many products on a joint construction kit, potentially also in other contexts. Methodologically, therefore, the challenges are, on the one hand, to create and manage models in such a way that individual systems or system elements

are easy to extract and reuse in other models and, on the other hand, to allow the exact system model to serve as a specification for a large number of products.

In addition to the many benefits of this high degree of adaptability and reusability, there is also an immense risk in developing a flawed system. While unsatisfactory individual development is correctable by new development, reused systems find their way into many products, negatively affecting a company's long-term success. For example, a shared infotainment system used throughout many product lines in the Group can jeopardize the success of multiple vehicle projects and pose a risk to the entire Group, as it affects all of the Group's vehicle projects that integrate this system. Consequently, system models and methods used in the automotive industry must be of exceptionally high quality to counteract these risks and reservations about introducing system modeling.

In interdisciplinary research, the challenge often lies in the varying meanings assigned to the same terms across different disciplines. Therefore, for clarification, this thesis uses the following vocabulary about the feature as an elementary concept of variability: A feature is an end-to-end functionality according to Def. 7 and thus is to be understood as a technical feature. In contrast, a marketing feature is a variability item.

2.5 Modeling Languages for Automotive Systems Engineering

Tools play an important role in applying the processes and methods defined in the PMTE paradigm [Mar94]. A suitable system modeling language is a central means to express system models in system modeling. In MBSE, the modeling language is the elementary description tool for defining and describing a system and serves as the basis for further analysis, verification, and validation activities. Consequently, selecting a suitable language is essential because it must describe all desired aspects of a system model and is the primary means of expression for the system engineers who model the system. Moreover, since efficient systems engineering requires a form of automation, primarily to perform testing activities [NLF+15], system models expressed in a system modeling language must be machine readable. For that reason, toolchains, as defined in [MWJ+22], integrate modeling, simulation, and design tools to perform system development tasks or to create a system solution.

According to a recent systematic literature study [MWJ+22], vehicle and automotive system development is one of the main application areas of system engineering toolchains, followed by aerospace, machine, and aircraft systems. Furthermore, toolchains for system design as an elementary aspect of systems engineering are a predominant application, but also among other applications often used for software design, control system design, and mechanical system design [MWJ+22].

Accounting for over two-thirds of overall usage, the SysML (60%) [Obj19] and the UML (15%) [Obj17], as defined by the OMG, emerge as the predominant languages for systems modeling within concurrent model-based systems engineering toolchains [MWJ+22]. Be-

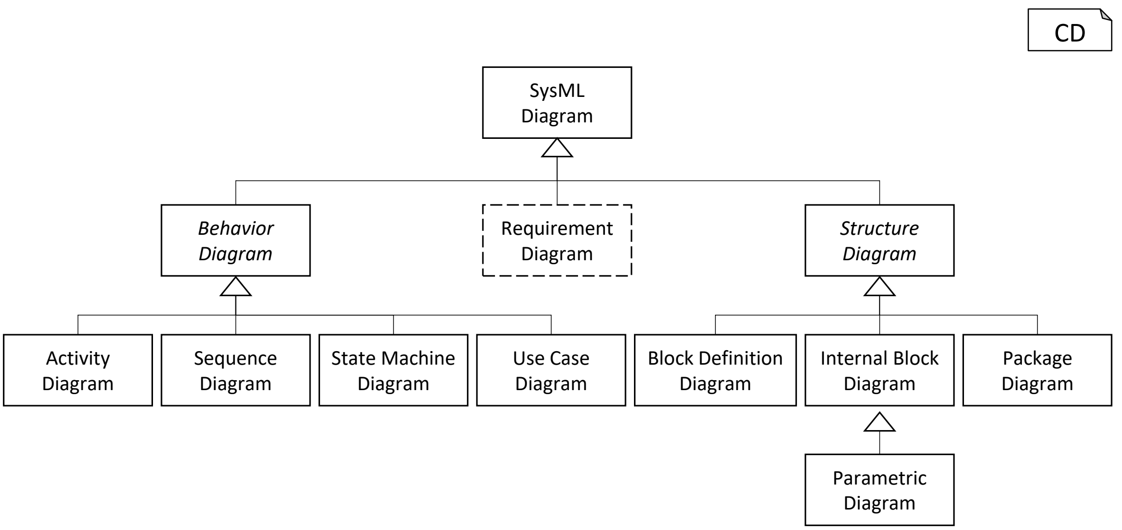


Figure 2.4: The SysML structure based on [Obj19].

cause of this dominance in concurrent industrial and academic use, this thesis focuses on the SysML as a modeling language in the presented methods. However, it does not restrict the applications to this language and encourages the use in other modeling environments and tools in future works or adaptions of the presented method.

The Systems Modeling Language (SysML) and the Unified Modeling Language (UML) are graphical general-purpose modeling languages that have emerged to model software and systems, especially their designs. While both languages have some commonalities, there are some significant differences. On the commonality side, the SysML reuses and tailors parts of the graphical notation of the UML. Moreover, both languages share similar diagrams, including use case, activity, and sequence diagrams. On the other hand, there is the intended application environment. While the UML focuses on software engineering and related fields, the SysML is a modeling language for systems [WRF+15]. It provides nine diagram types, which reuse and extend a subset of the diagrams the UML standardizes in [Obj17]. As a means for expressing textual requirements in the systems engineering model, the SysML additionally defines two new unique diagrams: The requirement diagram to express the relation between textual requirements and the parametric diagram to express the relationship between system components and their parameters, constraints, and values. Figure 2.4 describes the relationship between the different diagrams.

The SysML provides the *package diagram* to describe the organization of a SysML model concerning package elements, which contain model elements. Figure 2.6a shows an example of a package diagram containing package called “Four Pillars of SysML” and

specifies its breakdown into the four sub-packages for “Requirements”, “Parametrics”, “Structure” and “Behavior”. The SysML specification of the package diagram is analogous to the package diagram definition in the UML standard.

To capture text-based requirements and their relationship, the SysML *requirements diagram* provides a diagram surface to graphically represent these texts. The diagram form was not originally part of the UML and is therefore on of the extensions the SysML provides. As an example, Figure 2.5a presents a requirement diagram in which two sub-requirements with $id = 1.1$ and $id = 1.2$ refine a requirement with $id = 1$.

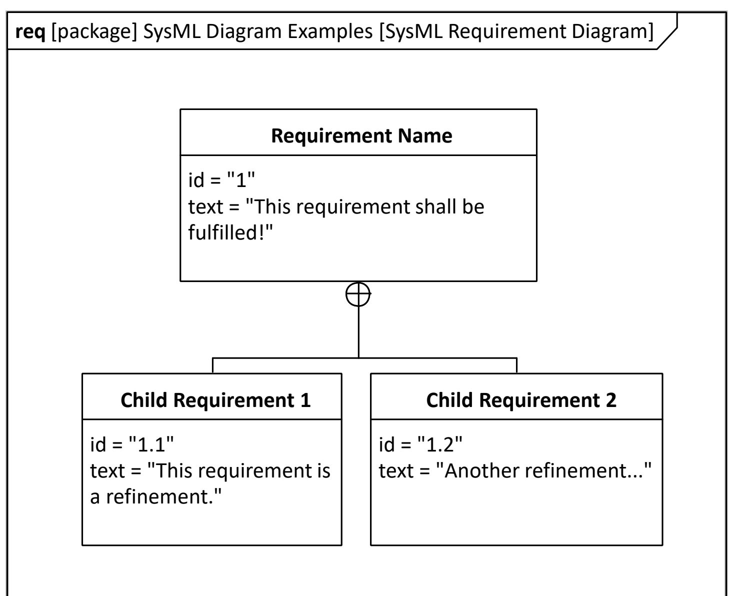
On the one hand, system *structure diagrams* represent systems and their internal and external structure as blocks. For this purpose, the SysML provides two types of diagrams. The *block definition diagram* defines and classifies structural elements as so-called blocks. As also indicated by the name of the diagram, the system presented in this diagram defines the system structure, which primarily consists of the definition of system elements and their hierarchy. Regarding the relation to the UML, the diagram type reuses elements of the UML but is not fully part of the UML standard [Obj17]. Figure 2.6b shows a block definition diagram with some essential concepts, such as generalization, associations, and block composition. As a way to describe the internal structure and interconnection of blocks and their parts, the SysML provides the *internal block diagram.* For this purpose, diagram provides elements to visualize system properties, such as ports, interfaces, and property connectors. As block definition diagrams, the internal block diagram uses elements of the UML as a basis. However, it is itself not part of the UML standard. As an example, Figure 2.6c presents an internal block diagram to visualize the inside view of the “Block Name” block from Figure 2.6b. By this, the internal block diagram defines an internal flow between the components A and B using ports and connections. To additionally constrain the parameters and properties of a block, a *parametric diagram* provides means to constrain property values of blocks *e.g.,*, to support an engineering analysis. As an example, Figure 2.5b expresses the correlation of Newton’s second and third law of motion by describing the gravitational force of an object as $F = m \cdot a$ and relating two forces. As the parametric diagram definition is unique for the SysML, no parts of the UML are reused in this diagram.

On the other hand, *system behavior diagrams* describe how the system behaves under specific conditions. To this aim, a *use case diagram* describes the system functionality by specifying the system usage concerning its interaction with external entities to accomplish a set of predefined goals [ZHCL18]. Figure 2.7a presents a small example of a use case diagram with a single actor, a generalization, an extension, and a use case inclusion. The UML standard fully covers the diagram definition. To depict an action-based transformation of inputs to outputs in such an interaction, an *activity diagram* provides the suitable diagram elements. In this diagram, the execution order of actions follows from the availability of inputs, outputs, and control sequences of actions, which specify the flow-based system behavior [FMS14b]. The diagram modifies the UML activity diagram, and adds continuous object-flows for CPSs interaction in the form of material or energy

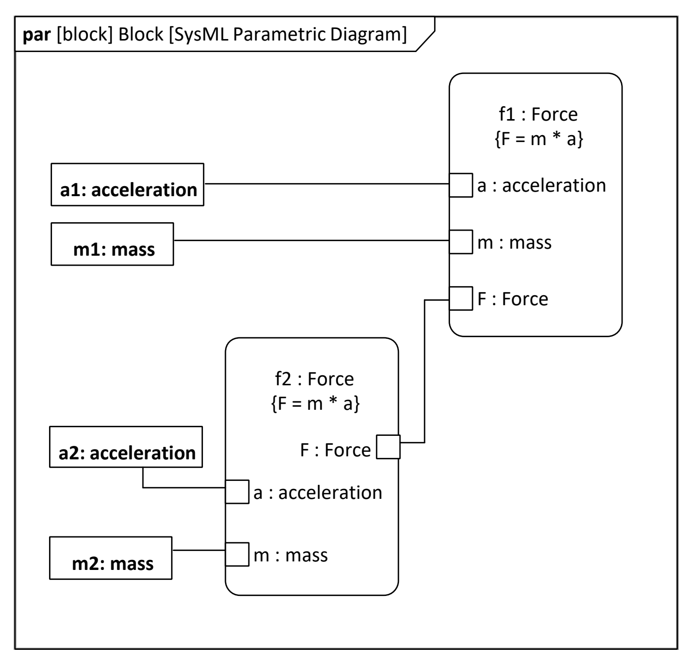
flows. Figure 2.7b presents a basic activity diagram to display the essential concepts. Especially for interface interaction, the *sequence diagram* enables systems engineers to specify how systems or their parts interact with each other by specifying the message sequences they exchange. The definition refers to UML sequence diagram and does not make any changes. As en example, Figure 2.8a contains some of the most common concepts of the sequence diagram. In another way, the *state machine diagram* describes the behavior of a system entity by identifying the states the system entity may be in. Next, the state machine provides transitions between these states triggered by events based on the fulfillment of specific logical guards. The SysML definition of the diagram fully takes over the UML state machine diagram. As an example, Figure 2.9a depicts simple state machine with two states: an initial state called “Initial State” and a final state called “Final State”. To move between these states the diagram provides a transition from the initial state to the final state. According to the SysML specification [Obj19], all behavior diagrams are usable for describing the element of another behavioral diagram, regardless of its reasonableness. Thus, a use-case diagram could, for example, specify the behavior actions of an activity diagram, and vice versa, an activity diagram could specify the behavior of a use case.

In addition to the standardized diagrams, the SysML provides relationship elements to describe the logical relationship between the elements of different diagrams, for instance, the allocation of functions to elements or logical to physical elements [WRF+15]. Since the SysML is a multi-purpose modeling language, it aims at supporting different systems engineering methods. Therefore, it is common practice in methodologies that use SysML as a modeling language to restrict the used elements to a subset of diagrams or diagram elements of the language [FMS14b]. Moreover, extending the SysML with domain-specific profiles can enhance tool support or description capacity. Both methods are *e.g.,* described in [FMS14b] and will be used later in this thesis to tailor the SysML to the needs of an automotive system specification. The OMG specification [Obj19] or at the official OMG SysML website [Obj25], also provide additional information about the SysML, for example, concerning a newly developed release of the SysML, which also provides a textual syntax for the graphical elements.

As a common concept, all SysML diagrams use a diagram boundary, which provides information about the package and the name of the diagram. As this information is not relevant for most parts of this thesis, the following diagrams do not show this diagram frame explicitly, but orients on the UML notion from [Rum16] using the diagram notes Appendix A summarizes.

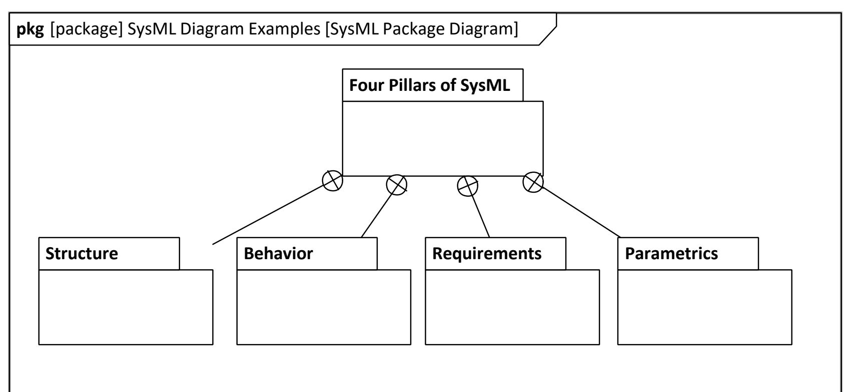


(a) The SysML requirement diagram.

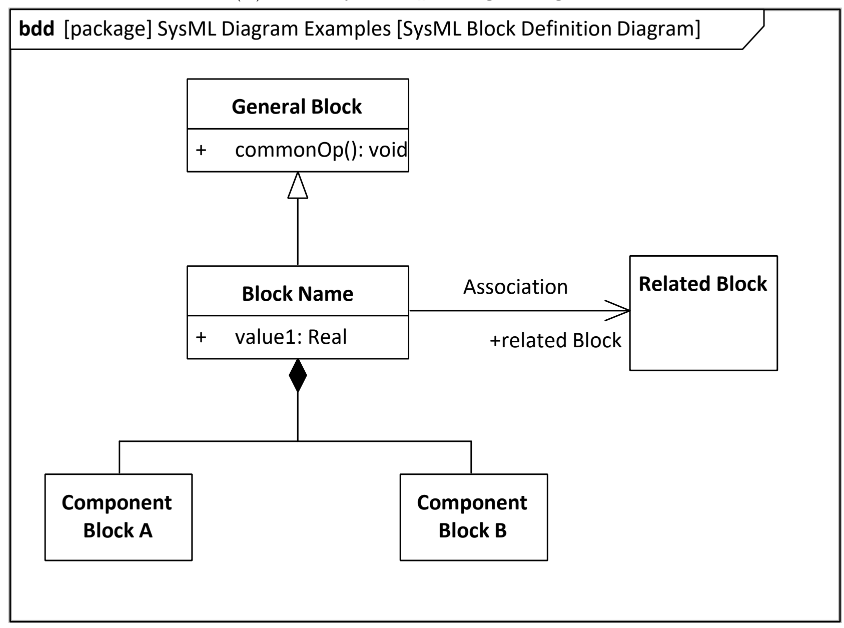


(b) The SysML parametric diagram.

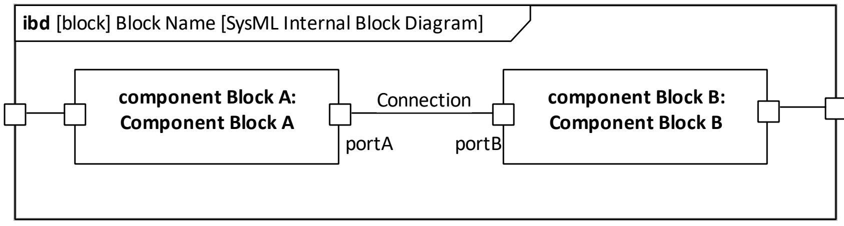
Figure 2.5: The unique SysML diagrams.



(a) The SysML package diagram.

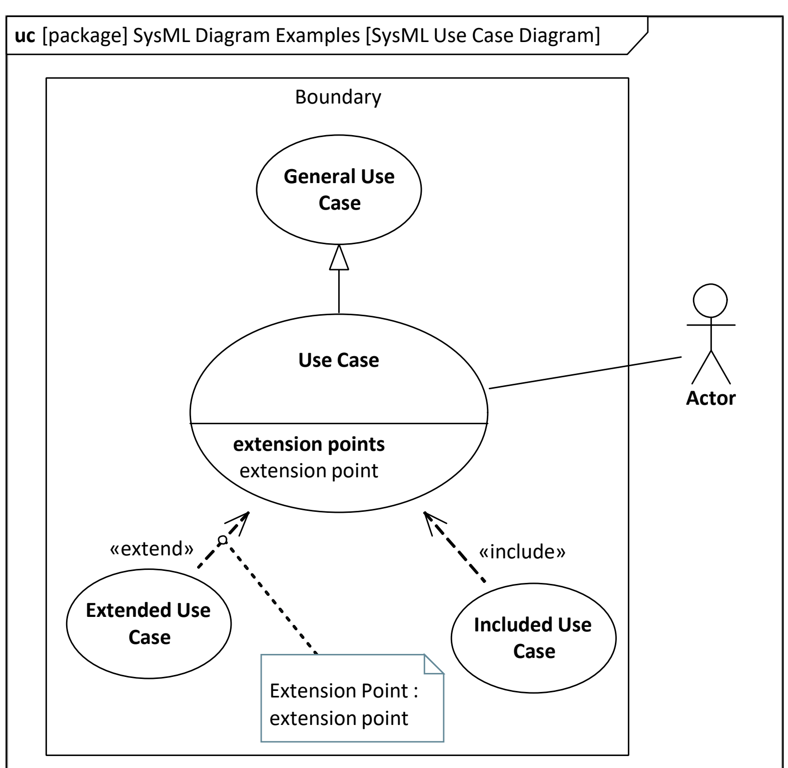


(b) The SysML block definition diagram.

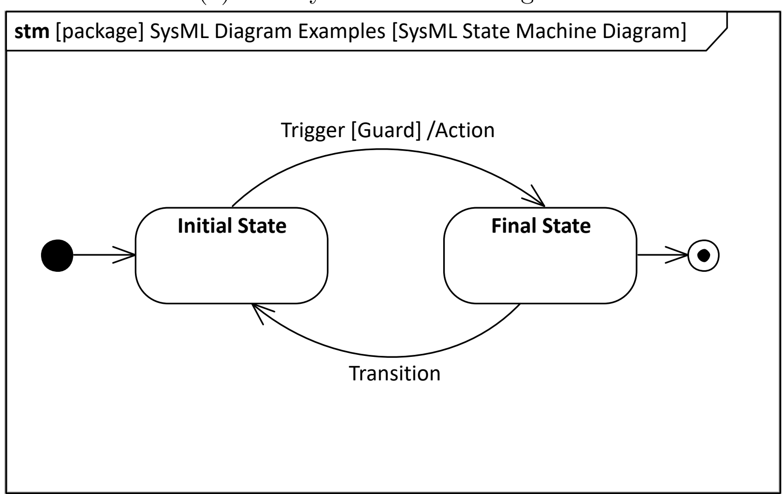


(c) The SysML internal block diagram.

Figure 2.6: The SysML structure diagrams.

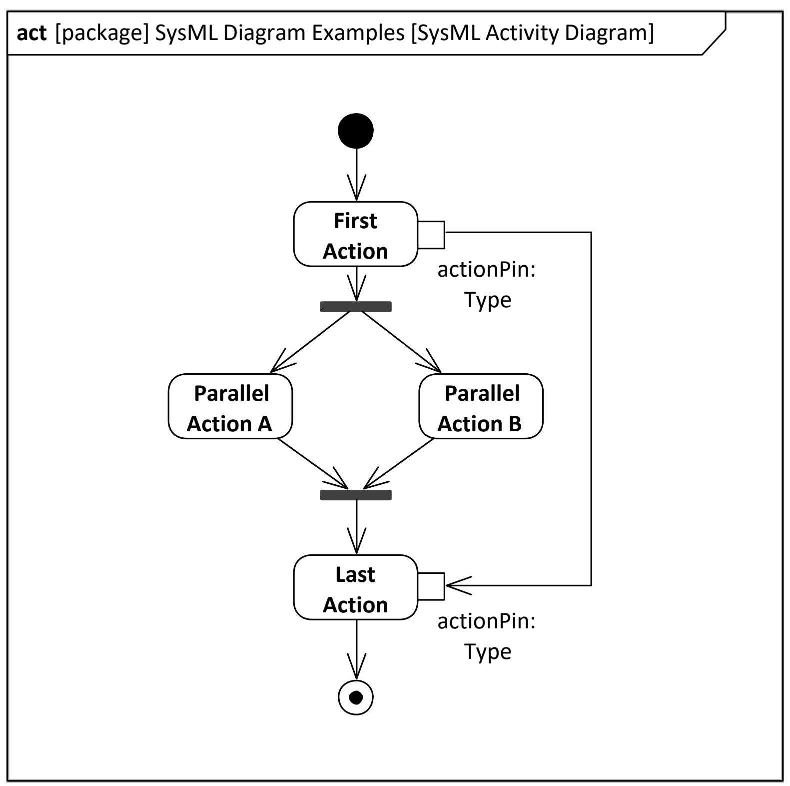


(a) The SysML use case diagram.



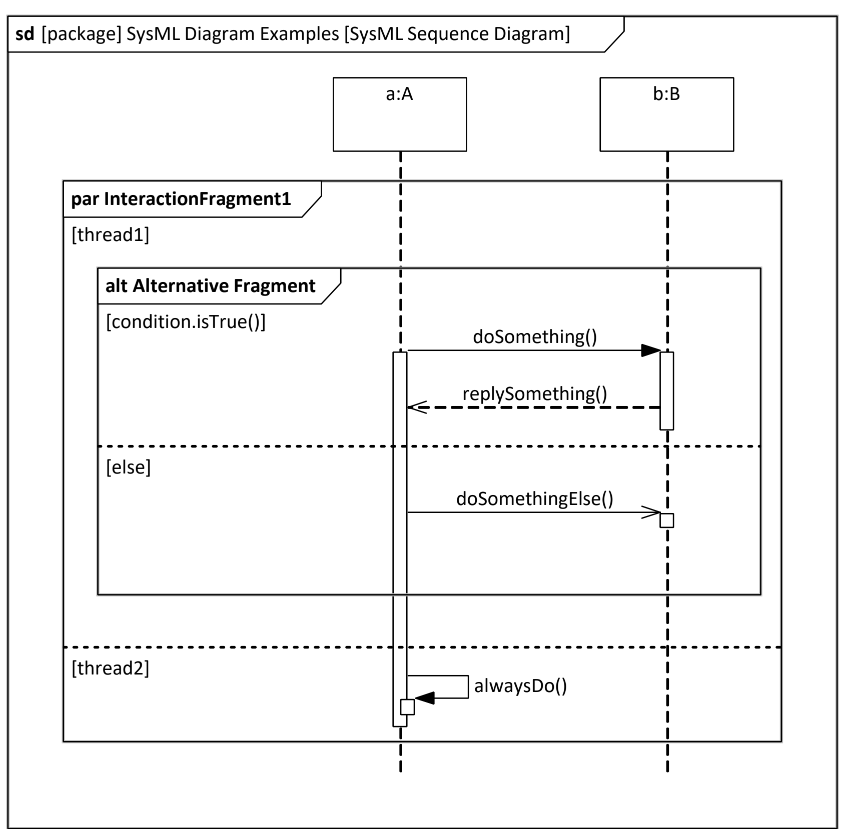
(b) The SysML state machine.

Figure 2.7: The SysML behavior diagrams I.



(a) The SysML activity diagram.

Figure 2.8: The SysML behavior diagrams II.



(a) The SysML sequence diagram.

Figure 2.9: The SysML behavior diagrams III.

2.6 Methodologies for Automotive System Development

As mentioned in section 2.2, the definition of systems engineering and its variants as defined in Def. 10, Def. 14 and Def. 15 only provides the terminology but not the methods for describing and developing systems following this model-based approach. Consequently, additional systems engineering processes, methods, and tools are required to implement a systems engineering approach in a project environment such as a vehicle development project in the automotive industry. Previous works have performed surveys to identify suitable and candidate methodologies for different contexts. For example, [Gra22] compares nine systems engineering approaches to classify them as activities, methods, or procedures to identify the research gap towards feature-driven systems engineering in system development. From another perspective, [RFB12] investigates the state of SE methodologies that use the MBE paradigm. Moreover, the INCOSE earlier conducted a general survey on candidate methodologies for MBSE in [Est07], which identified the following general applicable six methodologies: INCOSE Object-Oriented Systems Engineering Method (OOSEM) [FMS14b], IBM Rational Telelogic Harmony-SE [IBM10], IBM Rational Unified Process for Systems Engineering (RUP-SE) [Can03], Vitech Model-Based Systems Engineering Methodology [LS12], JPL State Analysis (SA) [IRBM05], and Object-Process Methodology (OPM) [Dor02, Dor16]. Throughout this survey, the presented methodologies were categorized according to the process, method, and tools dimension according to the PMTE-Paradigm [Mar94].

Drawing on these ideas and the methodologies already identified, this thesis revisits the candidate methodologies identified in these works. It reclassifies them concerning **RQ-1.3** (Which systems engineering methodologies exist, how is the system decomposed in this methodology, which viewpoints are created, and which paradigm is applied?) to identify potentials and drawbacks for their applicability in the automotive industry. To this end, Table 2.1 first classifies the methodologies according to the PMTE-paradigm [Mar94], to identify whether process, methods, and tools are available for applying the methodology in an automotive system development context. Finally, the design of some methodologies restricts their application to a reduced field or environment of application. Therefore, the environment category identifies the restriction of the methodology to its application domain.

Since systems usually decompose into subsystems, the next dimension of Table 2.1 investigates the system decomposition to identify how the systems engineering methodology aims to identify the subsystems of a system. To this end, the methodologies differentiate whether they decompose the system into one or more categories: Usage scenario, functional behavior, logical group, or physical (de-)composition. A system that decomposes according to its usage scenario has a subsystem for each application scenario. A vehicle decomposed to this understanding would, for example, consist of a transport people system, a transport goods system, and more. According to functional decomposition, the system comprises (sub-)systems responsible for a concrete task or functionality.

Table 2.1: A Systems Engineering Methodology Comparison. ● = fulfilled, ◐ = partly fulfilled, ○ = not fulfilled, Automotive-icon = Automotive, Avionics-icon = Avionics, Embedded-icon = Embedded, General-icon = General, Mechatronic-icon = Mechatronic.

	Name	V-Model (VDI 2206) [VDI21]	ISO/IEC/IEEE 15288 [ISO15]	Telelogic Harmony-SE [IBM20]	IBM RUP SE [IBM20, IBM20]	Vitech MBSE [Est07, LS12]	JPL SA [IRBM05]	OPM [Dor02, Dor16]	OOSEM [FMS14b]	FuSE [HMST07]	SPES [PHAB12]	SPES XT [PBDH16]	SMArDT [Mar22]	CUBE [GKS+21]	State of Research
PMTE	Process	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Method	○	○	●	●	●	●	●	●	●	●	●	●	●	●
	Tool	○	○	◐	●	◐	●	●	◐	○	○	●	◐	○	○
	Environment	Mechatronic	General	General	General	General	Avionics	General	General	General	Embedded	Embedded	Automotive	General	General
Decom-position	Scenario	○	○	●	○	○	○	○	○	○	○	○	○	◐	○
	Functional	●	◐	○	◐	◐	○	●	○	●	◐	◐	●	◐	◐
	Logical	○	●	○	●	○	●	○	●	○	●	●	◐	●	●
	Physical	●	●	●	○	●	◐	●	◐	●	◐	◐	◐	◐	●
Viewpoints	Requirements	●	●	●	●	●	●	●	●	○	●	●	●	●	●
	Use Cases	◐	○	●	●	○	●	●	●	○	◐	○	●	●	●
	Functional Architecture	◐	◐	●	○	●	●	●	○	●	●	●	◐	◐	●
	Logical Architecture	◐	◐	○	●	○	○	○	●	○	●	●	●	●	●
	Physical Architecture	●	●	●	●	●	●	●	●	○	●	●	●	●	●
	Realization	○	○	●	●	●	○	○	●	○	○	○	●	●	○
	Others	●	●	●	●	●	○	●	●	○	◐	◐	○	◐	●
Paradigm	Functional Modeling (FM)	○	○	○	○	○	○	○	○	●	○	○	○	○	○
	Feature-Driven Development (FDD)	○	○	○	○	○	○	○	○	○	○	○	◐	●	○
	Model-Based Engineering (MBE)	◐	◐	●	●	●	○	●	●	●	●	●	●	◐	●
	Model-Driven Development (MDD)	○	○	●	●	○	●	○	○	○	○	◐	◐	○	○
	Model-Driven Architecture (MDA)	○	○	○	○	○	○	○	○	○	○	○	○	○	○

According to this understanding, a vehicle could decompose into a propulsion function, a drive function, a driver assistance function, a passenger information function, a passenger entertainment function, and more. Likewise, logical decomposition also decomposes the system into abstract functional groups. In contrast to functional decomposition, logical decomposition also allows logical systems to perform more than one function. Consequently, a vehicle that decomposes to logical subsystems could still have propulsion, drive, and driver assistance systems that are rather identical to their functional counterparts but also have an infotainment system, which combines the passenger information and passenger entertainment functions of the previous examples. Finally, a system could decompose physically. A physical decomposition decomposes a system according to the physical elements of the system. As a result, each element in the system architecture has an exact physical counterpart in the final product. Thus, a vehicle decomposed physically would have a propulsion system, a chassis, an electrical system *etc.*

Many systems engineering methodologies specify the system under development systems according to different viewpoints. Therefore, Table 2.1 categorizes the systems engineering domains according to the viewpoints they support, provide, or require. As the most commonly mentioned viewpoints, this thesis considers the following: Requirements, use cases, functional architecture, logical architecture, physical architecture, and realization. In addition, the other category indicates that the methodology also provides other categories, *e.g.,* for validation and verification.

Since some methodologies require or support further development and modeling paradigms, the development and modeling paradigm category aims to identify whether the methodology requires or supports a specific modeling paradigm. As characteristics of this category, this thesis considers the FDD paradigm, the MBE paradigm, the MDD paradigm, and the MDA paradigm. The feature-driven development paradigm (FDD) describes an agile development paradigm in which iterative development cycles with the help of small, independent development artifacts (features) [CH11] lead to agile and highly adaptable development [Gra22]. It is originally based on the idea of breaking down the entire problem domain into small problems, each of which can be solved independently by one person in a short period of time Challenges of automotive system and simulation model development [CH11]. Moreover, the model-based engineering (MBE) paradigm summarizes approaches that use models not only for documentation purposes, but also for software development itself [Sch12]. As [Sch12] explains, the simplest forms of these models can be informal models for the collection of requirements or the system design. As a stronger variant of this paradigm, [Sch12] defines the model-driven engineering (MDE) paradigm, as a specialization of model-based engineering in which formal models are used as a central and essential artefact in software development. Thus, the specification enables the users to automatically process and analyze the models and ensure their consistency with the software system [Sch12]. Finally, model-driven architecture (MDA) describes a paradigm with the core concept to model at different levels of abstraction by concretizing abstract models close to the problem domain through

transformations into models that relate to a specific target platform [Sch12]. According to the MDA paradigm, it is not specified whether the refinements are made manually, semi-automatically or automatically [Sch12].

Because a methodology does not always uniquely fulfill or not fulfill all categories, Table 2.1 additionally uses a third option to indicate that a methodology names one of these categories as possible or extendable but provides no details on how to perform this aspect or how to extend the methodology for this purpose. To further investigate these methodologies and to explain the marked categories in Table 2.1, the following sections give a more detailed overview of the different methodologies.

2.6.1 V-Model

One of the standard software and systems engineering models is the V-Model [Boe84], which defines a set of processes for software development. It draws on the older waterfall model [Ben83], which applied principles from civil engineering to the software engineering domain. Since the first publication in [Boe84], several extensions have been implemented. One of these extensions is the VDI/VDE 2206 standard for the development of mechatronic and cyber-physical systems [VDI21], which is tailored for engineering domains such as the automotive industry and referred to in this thesis in Table 2.1. The development model presented in [VDI21] only consists of a process without additional methods or tools. During this process, the system is decomposed either functionally or physically. The viewpoints for use cases are only touched upon and not specified further. In the system architecture, a physical system architecture is assumed, but a functional and logical one is mentioned as a logical option in the definition of the system architecture. The specification in [VDI21] mentions that a model-based engineering paradigm based on modeling languages such as the SysML can shift from document-based to model-based systems engineering but provides no own application.

2.6.2 Software & Systems Engineering according to ISO/IEC/IEEE 15288

Another collection of processes for system development is provided in the ISO/IEC/IEEE 15288 [ISO15]. The standard does not explicitly define a suggested way of decomposing the system but mentions that logical decomposition (to which it also counts functional decomposition) and physical decomposition are possible decomposition types. In the same way, it only vaguely describes functional and logical architectures as possible viewpoints. Viewpoints for requirements and physical architectures, however, are explicitly required. Model-based development is a possible paradigm but has yet to be described as mandatory or required.

2.6.3 IBM Telelogic Harmony

The IBM Telelogic Harmony [IBM10] was part of the INCOSE Systems Engineering methodology survey [Est07] and presents processes based on the V-model and methods for a systems and software engineering methodology. Even though the methods and tools are designed to be tool-independent, the vendor proclaims it to be supported by the IBM Rhapsody toolchain [IBM10, IBM20]. During the system development process, the system is decomposed following the physical decomposition of the product. The methodology relies on SysML as a modeling language and provides a view of system requirements, use cases, functional architecture, physical architecture, and others. It draws on the model-based and model-driven engineering paradigms, which should support code generation based on the modeling artifacts [IBM10].

2.6.4 IBM Rational Unified Process for Systems Engineering (RUP SE)

The other IBM methodology is the IBM Rational Unified Process for Systems Engineering (RUP SE) [Can03], which was also part of the INCOSE Survey [Est07]. It provides a process framework with methods for the IBM modeling tool Rhapsody [IBM20]. As IBM Telelogic Harmony, it is consistent with the V-model. The methodology is embedded into the RUP [Kru04] and addresses the needs of systems engineering projects. During the process, the system is decomposed functionally to adequate functionalities to realize the system use cases and logically to subsystems realizing the functionalities. Since the logical system decomposition is the driver of the system decomposition, it is marked as decomposition type while functional decomposition is only listed as an option in Table 2.1. During the systems engineering process, the RUP-SE methodology introduces views for requirements, use cases, logical and physical architecture, realization/implementation, and others mostly related to the verification and validation of the system under development. It proclaims model-driven system development paradigm as engineering paradigm [BBCM06] using models based on the diagrams of the UML or SysML.

2.6.5 Vitech Model-Based Systems Engineering Methodology

The Vitech Model-Based Systems Engineering Methodology is another methodology investigated in [Est07], which was only later officially published in [LS12]. For this purpose, the methodology provides processes and methods. It strongly connects them to the Vitech MBSE tool CORE without detailing this support or connection to keep the methodology tool independent. Following the methodology, the system levels form according to an onion model. First, the system of one level is modeled. After completing this level, this onion layer is "peeled off" to explore the next level. The decomposition is carried out using the system architecture of the level under consideration, which primarily follows the physical decomposition of the product, which is allocated to an orthogonal

functional behavior decomposition. The orientation is towards the system evolution but reflects the final physical decomposition of the realized system. Executing the MBSE process following the provided methods results in several viewpoints of the system: A system requirements viewpoint, a functional behavior viewpoint, a physical architecture viewpoint, a design realization viewpoint, and others viewpoints for verification and validation. In the methodology, models are firmly anchored in the system development process, which is why a model-based development paradigm exists.

2.6.6 JPL State Analysis (SA)

The next methodology presented in [Est07] is SA [IRBM05]. It is a methodology by the JPL for the application in aerospace systems engineering projects and provides processes, methods, and tools for this domain. The system decomposition in SA follows a functional decomposition scheme resulting from functional behavior analysis. The methodology relies on creating viewpoints for system requirements, operational (use-case) scenarios, functions, and (physical) components. According to SA, the system is developed following a model-based development paradigm, combining target and state modeling.

2.6.7 Object-Process Methodology (OPM)

The OPM [Dor02] relies on a formal paradigm for which it provides methods and tools. It follows a model-based engineering paradigm for which it provides its own systems modeling language: the OPL [Dor02]. The method also provides tools [DRBS03, DLM10]. The OPM relies on the principle that a system's structure and behavior are highly intertwined and that separating these two aspects might harm the system development process [Dor02]. To overcome this challenge, the OPM proposes an orthogonal decomposition in a behavioral or functional and a structural or physical decomposition simultaneously. Instead of multiple diagrams with multiple views, such as proposed by the OOSEM using the SysML [FMS14b], the OPL considers only one language/diagram for system specification. However, as described in [Dor16], both use cases, as in a use case diagram, *i.e.,* functions and processes described as activities, and the physical decomposition as in a SysML BDD or IBD are decomposed. Consequently, the methodology provides views for requirements, use cases, functional architecture, physical architecture, and others, but the individual views are more separated than SysML-based methods.

2.6.8 INCOSE Object-Oriented Systems Engineering Method (OOSEM)

The OOSEM [FMS14b, LFM00] is an object-oriented systems engineering methodology that supports a top-down model-based approach. It is consistent with the V-Model [RFB12] and uses SysML for specification, analysis, design, validation, and verification of the system and provides processes and methods for the application [FMS14b]. The

OOSEM was developed together with the SysML [LFM00] as a profile of the UML [Obj17] [FMS14b]. Because no specific SysML unique modeling tool is named and provided, the tools aspect is marked accordingly in Table 2.1. The OOSEM decomposes the system logically as well as physically [FMS14b]. Since the logical decomposition defines the next decomposition layer, only this decomposition aspect is marked as fulfilled and the physical decomposition as possible or extendable in Table 2.1. According to [FMS14b], a system model consists of elements that represent requirements, design, test cases, design rationales, and their interrelationship. In this context, the OOSEM defines views for requirements, system use cases/scenarios, logical decomposition and logical subsystems, physical/technical hardware and software components, and their deployment/realization. In addition, other views *e.g.,* for test cases, are created.

2.6.9 Function-Based Systems Engineering (FuSE)

Function-based systems engineering, as presented in [HMST07], draws on the functional modeling paradigm [PBFG07] and applies it to a system development process. The system models play an essential role in the system development process and are successively expanded through the different refining views of the system engineering methodology. In this process, first, a black box functional model is created. Next, a conceptional functional model that decomposes the black box functional model into sub-functions is determined as a functional structure. Then, the sub-functions are assigned to their executing systems to identify the system boundaries. Finally, the requirements are decomposed to the sub-function level, and a behavioral model for the sub-functions is created. These models allow potential system solutions to be iteratively identified and iterated until a suited system realization is found.

2.6.10 Software Platform Embedded Systems (SPES)

According to the developers of the SPES methodology, the model-based engineering of embedded systems is a relevant topic in the automotive industry [PHAB12]. Thus, this thesis also considers this methodology in Table 2.1. The SPES methodology provides processes and methods for developing embedded systems based on a model-based engineering paradigm. According to SPES, a systems engineering design space consists of (but is not limited to) requirements and functional, logical, and technical viewpoints, each of which forms an abstraction layer. Since use cases are a sub-layer of the requirements abstraction layer, this option is marked accordingly in Table 2.1. On each abstraction layer, a decomposition is possible and supported. Thus, a function may have a sub-function, a logical component, a logical sub-component, and a technical component, a technical sub-component. A structural decomposition, which divides a system into subsystems, however, is in [PHAB12] intended possible for logical and physical systems. Moreover, a system decomposition is only explicitly described for the logical

abstraction layer and therefore marked as the intended decomposition in Table 2.1.

To further develop SPES, SPESXT extends the original methodology. For this aim, SPESXT additionally provides tool support for the SPES methodology and an extended evaluation of the practical applicability. The other categories from the SPES methodology remain unchanged, apart from an additionally sketched application scenario, where a model-driven development paradigm could be applied using the newly provided platform and tool support.

2.6.11 Specification Method for Requirements, Design, and Test (SMArDT)

SMArDT was initially introduced in the BMW Group and jointly developed in cooperation with the exida GmbH, the FEV Europe GmbH, and the Chair of Software Engineering at RWTH Aachen University [Mar22] as a descendant of the SPES/SPESXT methodology.

When SMArDT was first mentioned in [KMS+17], SMArDT was a framework that supports traceable requirements as demanded by standards such as CMMI [CKS11], A-SPICE [SPI15], or ISO 22626 [ISO11a] for semi-automated test-case generation used in electric engine development projects at BMW. SMArDT is based on the V-model [VDI21] and provides processes and methods for systems engineering methods that are usable for automated test case generation. To this aim [KMS+18], evaluates the automated test case derivation procedure based on the modeling artifacts in a survey conducted at BMW. As a result, the survey shows that the participants expect to benefit from automated test case creation independent of their backgrounds and expect the test quality to increase [KMS+18]. Moreover, model-based test case creation is most useful for system, subsystem, and component tests [KMS+18]. Apart from the testing aspects, [KKRvW18] extends the SMArDT process to a methodology to identifying wrong or contradicting requirements at early development stages using component & connection views and activity diagrams. In the next evolution step of SMArDT, [HKK+18] presents a concrete application outside the electrical powertrain development domain. Moreover, [DGH+18] presents a concrete implementation of SMArDT's automated test case generation based on UML/P activity diagrams [Rum16] models modeled in MontiCore [Kra10, HLMSN+15]. Therefore, Table 2.1 marks the tooling category as partly fulfilled. Additionally, [KKRvW18] focuses again on the requirement consistency checking facet of SMArDT. Finally, [Mar22] and [DGH+19] summarize the contents and contributions of SMArDT and refine them with additional content. To decompose the system, SMArDT follows a functional system decomposition. During the system development, SMArDT creates a view for requirements, use cases, functional architectures, logical architectures, physical architectures, and system realization. SMArDT proposes a MBE paradigm and also sketches how the feature-driven development paradigm could be integrated into SMArDT, which is fully refined in the derived CUBE methodology.

2.6.12 Compositional Unified system-Based Engineering (CUBE)

Compositional Unified system-Based Engineering (CUBE) [GKS+21] is an extension of SMArDT [KMS+17] and SPES [PHAB12]/SPES XT [PBDH16] to introduce a feature-driven development paradigm (FDD) into these methodologies. To remain tool-independent CUBE only provides processes and methods for systems development. The system views follow similar abstraction layers as SMArDT and uses CUBE decomposition on all abstraction layers as suggested in SPES. Even though CUBE does not restrict a system decomposition on logical or physical system decomposition, a logical system decomposition is preferred. During the system development process, CUBE suggests a combination of textual and model-based requirements specification formulated in the SysML for system specification, without enforcing one or the other. Thus, following a model-based development paradigm is possible and recommended but not firmly anchored in the methodology.

2.6.13 Commonalities and Research Gap in SE Methodologies

Table 2.1 reveals that most of the considered methodologies investigate common understandings and differences, which shows potential for additional research.

While all methodologies in the table provide processes and most methodologies additional methods for systems development, tools that support processes and methods often need to be provided as part of the methodology. The authors commonly name two reasons for this. First, some authors claim that they intentionally provide no tools for their methodology. For example, CUBE [GKS+21], the OOSEM [FMS14b], and IBM Telelogic Harmony [IBM10] name the tool independence as goal or advantage of their methodology, to allow better reuse in different project environments. Second, some authors name tool development explicitly as their research goal but only refer to preliminary results in their publications about the methodology. For example, when SPES [PHAB12] was initially released, the tool development was only in a preliminary stage and only later published in the context of SPESXT [PBDH16]. Moreover, tools are considered helpful in the context of FuSE [HMST07], but still need to be developed for this methodology. Consequently, systems engineering tools are still a relevant research topic in the context of systems engineering methodologies.

There are different views concerning system decomposition in the relevant systems engineering literature. Especially the standards, such as the V-model [VDI21] and the ISO/IEC/IEEE 15288 [ISO15], proclaim a system decomposition based on the physical structure of the final product. However, almost all methodologies proclaiming a physical structure decomposition of the system also proclaim or at least sketch an orthogonal functional decomposition. This understanding also matches the ideas from design theory presented in [PBFG07] that lead to the functional modeling paradigm section 2.3 presented in the context of this thesis. According to this understanding, the structure

of a physical system under design follows from the functions it performs (*i.e.,* form follows function), and thus, the functional and physical decomposition form orthogonal decomposition dimensions of the system, where the functional decomposition influences the physical decomposition. The only exception to this functional and physical decomposition combination is the IBM Telelogic Harmony [IBM10], which follows a physical structure decomposition influenced by the usage scenarios and the system's behavior. On the other hand, a pure functional decomposition scheme is only followed in SMArDT, where a logical and physical system decomposition is created on the respective abstraction layer. However, only the functional system is the basis for determining the elements of the following system decomposition layer. With physical decomposition, logical system decomposition is one of the most mentioned forms of system decomposition. One of the rarest system decomposition types is the usage scenario system decomposition. Even though a scenario-based systems engineering approach is, for example, mentioned for CUBE in [MSG+22] and used for a physical system decomposition in [IBM10], no systems engineering methodology investigated used this approach to determine system elements.

Although all methodologies agree on a viewpoint for system requirements, it is not commonly described how these requirements should be formulated, which artifacts need to be created, and for which elements these requirements are formulated. One of the main differences is, for example, the form of requirements formulation. Whereas SPES [PHAB12] and SPESXT [PBDH16] suggest using SysML or other modeling languages to model the contents of the requirement, CUBE [GKS+21] and SMArDT differentiate between textually represented (*e.g.,* as a free-text in a requirements diagram) and model-based requirements. For example, in SysML models and textual requirements, which are typically formulated in natural language. Moreover, the OOSEM according to [FMS14b] introduces a combination of both views by using SysML requirement diagrams that contain natural language requirements in the linked diagram elements. A research gap is therefore given in the way of formulating the requirements displayed in this view.

Another general approach to requirements modeling is the use-case viewpoint, which bridges the gap between the requirement and behavioral formulation. Following a similar procedure approaches such as the OOSEM [FMS14b] and IBM Telelogic Harmony [IBM10] utilize the use cases to bridge the gap between requirements and functional specification by defining sub-views such as system scenario descriptions for each use case. Therefore, they utilize the use case description layer to bridge the gap between the requirements and the functional behavior and architecture definition. In comparison, use cases and their refined use case scenarios are, for example, a means for expressing requirements in SPES [PHAB12]. Other methodologies as CUBE [GKS+21] or SMArDT [Mar22] utilize the use case view to identify the high-level requirements a system needs to fulfill for a system feature identification.

This definition of functional architecture is also controversial in the investigated systems engineering literature. Whereas most of the definitions specify a detailed or

sketched version of this view, as Table 2.1 shows, the exact definition of this view reveals further differences in the definition and understanding of this view in the detailed examination. For example, some methodologies such as SPES [PHAB12], SPESXT [PBDH16], Vitech MBSE Methodology [LS12], and FuSE [HMST07] describe the functional architecture view as some structural architecture in which function blocks interact at their system boundaries each other similar to a function net [GHK+08a, MBRB10]. In contrast, other methodologies such as CUBE [GKS+21] or the OOSEM [FMS14b] their functional architecture views as some behavior description, in which the actual flow within a component and not their input and output behavior are described.

Another related architecture view is logical architecture, which defines logical components to execute system functionalities. As stated in [PBDH16], the technological hierarchy of a system often guides the structure of the logical architecture. Thus, the structure and decomposition of the components typically follow the different engineering and design areas, which remain unchanged even if another technical realization of a product is under development. For example, the distribution between mechanical, electrical, and software engineering is often unchanged between different systems. It could, therefore, serve as an identification of the logical architecture as suggested in [PBDH16]. On the other hand, this strict distinction between functional and logical architecture might be artificial since functional architecture is a particular case of logical architecture, where each system function maps to a logical component. This aspect is not only an intriguing aspect for future research but also an interesting observation in the investigated methodologies since most other methodologies apart from SPES/SPESXT only sketch a functional architecture as a viewpoint but do not detail them in their methodology. Therefore, the differences and commonalities between those two views are a current research gap that can be investigated in future research.

Regarding the modeling paradigms, Table 2.1 reveals that most methodologies explicitly mention or sketch an application of the model-based engineering paradigm. However, model-driven engineering and architecture are barely mentioned and rarely integrated into the development paradigms.

2.7 Interim Conclusion and Discussion

With these foundations, this chapter contributes to answering **RQ-1** ("Which established methods software and systems engineering methods exist that are applicable to specify automotive systems?") and provides the foundations for the remainder of this thesis. Although this chapter could not provide an exhaustive overview of all established methods from software and systems engineering and their applicability in the automotive industry for automotive system development, the presented overview is sufficient to address the derived research questions from section 1.1.

Regarding **RQ-1.1** ("What are the foundations of systems engineering?"), section 2.1

summarizes the contributions of several works on systems engineering [She17, NPR07, WRF$^+$15, FMS14b, ISO15, ISO17b, ISO11c, ISO11a, ISO21, Web09] and not only condenses a systems engineering understanding for this thesis in Def. 10, but also introduces the concepts for a system definition (*cf.* Def. 1 in subsection 2.1.1) and a rudimentary formalization of the primary concepts (subsection 2.1.2). Moreover, section 2.2 addresses **RQ-1.2** ("What are the characteristics of a system model in the automotive industry") and focuses on extending the characteristics of a model according to [Sta73] to the system model in the automotive industry in Def. 13.

Furthermore, section 2.2 introduces the groundwork for a model-driven systems engineering paradigm in Def. 15, which is the foundation for answering **RQ-4** ("Is an efficient and effective specification of automotive systems possible using model-driven systems engineering?"). As an additional framework to answer this question in the later chapters of this thesis, section 2.3 investigates the functional modeling paradigm from software and systems engineering, which supports the modeler during the creation of automotive system models. Furthermore, section 2.4 summarizes additional constraints, methods, and challenges from the automotive industry to manage variability in automotive systems that are required to design automotive systems efficiently. Regarding **RQ-3** ("Which elements of the SysML support the creation of the elements developed under **RQ-2**?"), section 2.5 investigates modeling languages to model automotive systems.

Finally, section 2.6 investigates system decomposition, viewpoints, and paradigms in a selection of systems engineering methodologies (*cf.* Table 2.1) to answer **RQ-1.3** ("Which systems engineering methodologies exists, how is the system decomposed in this methodology, which viewpoints are created, and which paradigm is applied?").

Chapter 3

Model-Driven Systems Engineering using CUBE

Concerning **RQ-2** (Is there a common practice for creating, interpreting, analyzing, and using system specifications in the considered projects in the automotive industry?), this chapter aims to shed light on the methodological perspective on automotive system specification. Therefore, this section investigates how model-based development manifests itself in the systems development projects in the automotive industry using CUBE as reference methodology.

For this purpose, the model-based systems engineering methodology CUBE, which is the current state of the methodological foundation of the systems development projects analyzed in chapter 4, is considered in more detail to answer **RQ-2.1** ("Which system specification process is applied in these projects?"). After an introduction to this general methodology in section 3.1, section 3.2 focuses on answering **RQ-2.2** ("Which artifacts are created during this process?") and **RQ-2.3** ("Which elements are required to specify a system in the automotive industry?") refining the methodology to a systematic approach with clearly defined artifacts. Moreover, section 3.2 also focuses on **RQ-3** ("Which elements of the SysML support the creation of the elements developed under **RQ-2**?") by identifying SysML models that are suited to support the methodology during the systems design to lay the foundation for further extensions in the upcoming sections. Because neither CUBE nor the other related approaches section 2.6 analyzes, address a model-driven system development as [DJR+19] suggest, section 3.3 extracts and identifies the research gaps to lift this initial state of CUBE from a model-based methodology to a model-driven systems development paradigm and derives methodology requirements to apply this approach in the automotive industry in chapter 5, chapter 6, chapter 7, and chapter 8.

3.1 The CUBE Methodology

The systems engineering methodology referred to in this thesis is the Compositional Unified system-Based Engineering (CUBE) [GKS+21], which is an extension of SMArDT [KMS+17] and SPES [PHAB12]/SPES XT [PBDH16]. Drawing on the ideas and prin-

ciples of those two methodologies, CUBE aims to compensate for the disadvantages of these two methodologies and provides an optimized foundation for the specification of automotive systems.

One of SMArDT's drawbacks is that it predominantly focuses on test case generation and the safety-critical aspects of systems engineering, neglecting the additional efforts in the requirement specification of system design or requirements engineering. In contrast to SMArDT, CUBE aims at establishing methods from systems and software engineering, such as feature-driven development [GKS+21] without requiring a specific modeling language or the existence of textual requirements. To achieve these aims, CUBE suggests a combination of textual and model-based requirements specification for system specification, without being limited to one of the specification methods or a specific modeling tool [GKS+21]. Moreover, as SPES and SPESXT by name and methodologically only refers to the specification of embedded systems, other elements of automotive system specifications such as a mechanical design are not in the focus of the methodology.

With the definition of a systems engineering methodology, CUBE aims at achieving the following objectives that [GKS+21] describes as follows:

Objective I "Specification of the product or product line with focus on stakeholder needs, respectively their expectations, as well as the system context" [GKS+21]

Objective II "Continuous extension of specification artifacts to ensure fulfillment of stakeholder expectations by the definition of various views on system elements" [GKS+21]

Objective III "Layered decomposition of the system, considering clear definitions of system boundaries for all decomposition elements, to reduce the complexity of the specification elements" [GKS+21]

Objective IV "Feature-based specification of the product to enable solution agnostic top-down development" [GKS+21]

Objective V "Consideration of system variants of a product or a product line, including variation points on different decomposition levels, to improve reusability of specification artifacts" [GKS+21]

Objective VI "Procedure as a base for the definition of conventional and flexible, agile specification processes, as well as for consideration of functional product versions" [GKS+21]

To achieve these objectives, CUBE addresses four dimensions as Figure 3.1 displays. The first central dimension is the system decomposition dimension which CUBE introduces to address Objective III. Since CUBE evolved from SMArDT and SPESXT, CUBE adopts the composition of systems as a base of the development from these methodologies. As explained in subsection 2.1.1, the analysis of the system and its boundary in

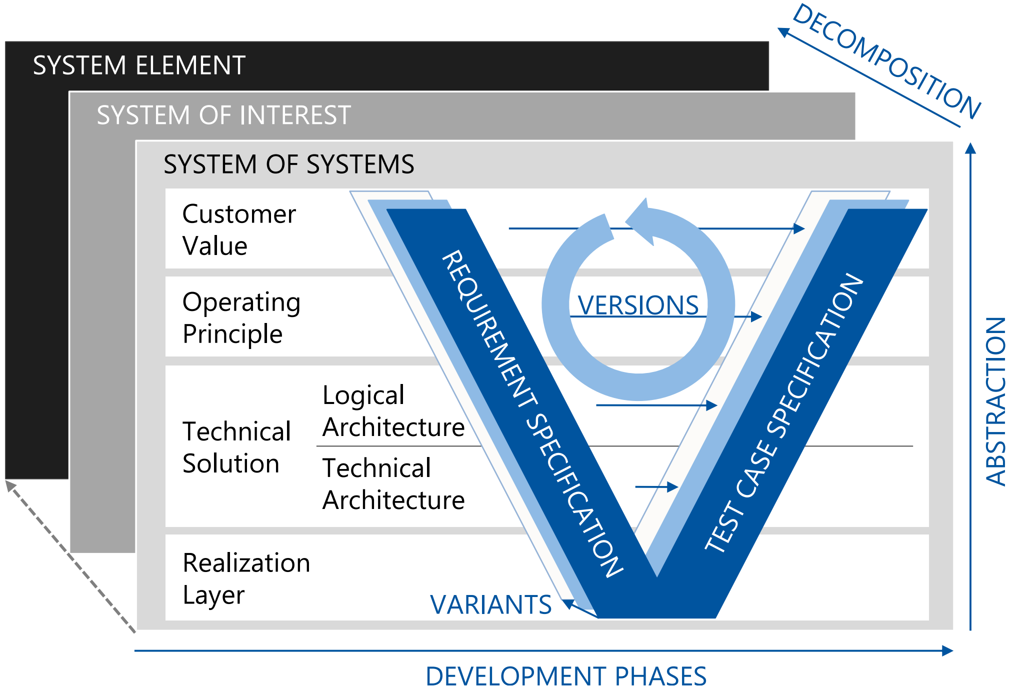


Figure 3.1: The Compositional Unified system-Based Engineering (CUBE) model based on [GKS+21] and [MSG+22].

the context of the system under development (also called System of Interest (SoI)) concerning their influence and use for the related stakeholders and interacting systems is a crucial part of the system development process. For analyzing how the system interacts with its boundary, the first decomposition level of CUBE defines a network of interacting systems, sometimes referred to as System of Systems (SoS) [FMS14b]. To specify how a system interacts with its environment, the SoS embeds the SoI to determine the interfaces and stakeholders of a system. The second decomposition level focuses on the SoI, the third on its subsystems, and the following on their elements and decomposition. As described in [WRF+15] and [SZ16], the decomposition of systems is not limited to a fixed number of decomposition levels. Therefore, CUBE allows the decomposition to an arbitrary number of elements [GKS+21].

The second central dimension of CUBE, called the abstraction dimension, addresses Objective II to continuously extend the stakeholder expectations by defining various views on system elements in an abstraction dimension. This dimension concertizes the stakeholder expectations from a high abstraction level to a concrete realization. For this, the abstraction dimension starts with identifying so-called customer values, which express the high-level system requirements as use cases and requirements. Because the

customer is only one stakeholder, but not the only one, this thesis refers to the customer value specification from [GKS+21] as stakeholder value specification and customer values as stakeholder values. All requirements formulated on this level are black-box requirements that describe the black-box view [WRF+15] of the decomposition layer instance. The second abstraction layer describes the operating principles of the decomposition layer instance. For this, the decomposition layer specifies a grey-box or white-box view [WRF+15] of the decomposition layer instance to describe the functionalities the system provides as an activity in a solution-neutral way. The third abstraction layer, the technical solution, splits into two sub-layers: The logical and technical architecture layers. In the technical solution, the logical architecture describes a realization of the operating principles in the form of a logical system view and architecture, whereas the product or technical architecture describes concrete hardware and software realizations together with the decisions leading to this architecture [JGW+21].

To this end, the logical architecture groups the activities from the operating principles to structural elements based on their functionality. Consequentially, the logical architecture forms functional groups to define a logical context for realizing the decomposition layer instance without considering technical decisions. By using functional groups instead of technical decisions, CUBE aims at enabling variant management as a separate dimension and reuse to address Objective V and the variability concepts the automotive industry applies (*cf.* section 2.4). As later discussed again in chapter 8, the specification artifacts from the logical architecture represent reusable elements of the product or product line without product-specific characteristics and aim to be valid for all possible product line product variants [GKS+21]. By this, the logical architecture aims at defining the core of a product line, *i.e.,* a static core that remains unchanged in all product derivatives, in the form of functional logical, multi-product architectural elements, as well as the technical variation points of the product line [GKS+21]. In contrast to this logical viewpoint, the technical or also called product architecture extends the logical understanding with technical implementation decisions. Hence, the technical architecture must document all decisions to implement the logical functionality. Based on the reusable, logical architectural elements, the technical architecture implements this view by adding product-specific information or determining a final product configuration [GKS+21]. Finally, the fourth abstraction layer specifies the realization *i.e.,* all implementation and production-specific aspects of the realized technical architecture [GKS+21]. Based on the results gathered in this abstraction layer and all previous abstraction layers, the specification of the considered decomposition layer instance is completed, and the implementation of the product specification process can start in the next phase.

The third dimension covers the implementation phases of the product specification process to address Objective II. To achieve this aim, it addresses the aspects of requirement specification and the definition of means for requirements verification and validation. By this, CUBE aims at ensuring that already during the system design of

the decomposition layer instance, a correct functionality of the system can be guaranteed by suited measures for a successful implementation of the functionality.

The fourth dimension of CUBE covers the specification of functional product variants to address Objective V. In contrast to the technical variants that the abstraction layer addresses, the functional product variants have a different implementation and provide different functionalities. For example, different operating principles could realize the same stakeholder value. The customer's need for mobility, for example, could be realized by a road vehicle, train, airplane, helicopter, or even spacecraft, depending on the functional characteristic of the system. Apart from this shared customer need, all systems provide different functionalities and use fundamentally different operating principles to fulfill mobility needs.

In contrast to this aspect of functional variability leading to entirely different products, the choice of product design, such as, for example, the product color, does not influence the functionality of the system and could, therefore, be assigned to the area of technical variability [GKS+21]. Note that both kinds of variability may appear on all decomposition levels and influence each other by formulating constraints on the realization of the next decomposition layer, which may drastically increase the number of product variants and make system development more complicated. One of the pitfalls is, for example, that the technical variability on a higher decomposition level may result in constraints leading to functional variability on a lower system level. While the choice of drive type at the vehicle level is merely a technical variability, since the type of drive for propulsion does not influence the overall mobility requirements, it is a decisive influence on functional variability at the propulsion system level. Therefore, the choice of propulsion (e.g., electric or electric motor) significantly influences the drive system's operating principle and interfaces. However, a technical decision still needs to be made on this decomposition layer, which is only possible if the decision was not already placed upstream on one of the decomposition layers above.

Finally, the fifth and last dimension of CUBE aims to fulfill Objective VI: The definition of specification processes and the consideration of functional product versions. For this purpose, CUBE as presented in [GKS+21] considers two perspectives of product versions: From the first perspective, the product may have different versions in terms of time and function due to further development and changing customer requirements. From the second perspective, each product specification is subject to being written and valid for a time, so that during the specification phase, in particular up to the complete conclusion of the specification, versioning of the artifacts is necessary [GKS+21]. These depend on the project context, the enterprise, and the other working processes influencing the systems development process that require additional means for version management of the system specification artifacts in the fifth dimension of CUBE.

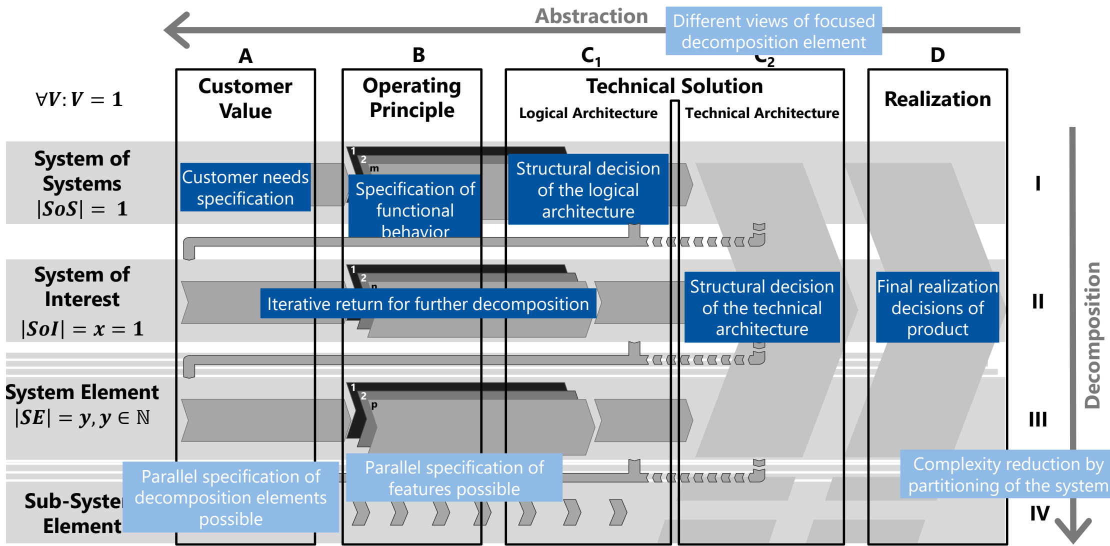


Figure 3.2: The CUBE model without variability based on [GKS+21] and [MSG+22].

3.1.1 CUBE as Product Specification Procedure

Since Figure 3.1 displays only the required dimensions for CUBE but not the order in which these dimensions need to be traversed, Figure 3.2 presents the product specification method [GKS+21] suggest for CUBE. One of the main challenges when formulating this order is that systems engineering projects are subject to project-, company-, and product-specific constraints [GKS+21]. To overcome this issue, general handbooks on systems engineering methodologies such as [WRF+15] or [She17] suggest the introduction of a base process based on theoretical considerations that can be tailored to these specific constraints that follow from the project, company, or developed project environment.

Figure 3.2 presents CUBE's general specification process as presented in [GKS+21]. In this general process, the system development starts on the SoS level and considers exactly one element on this level. For the system of interest, as for all other system decomposition levels, the standard process elicits stakeholder needs as a consideration of the stakeholder value. Based on these stakeholder needs, the systems engineers identify system features for which they define operating principles for the operating principle phase. Following these results, the systems engineers involved in this process derive operating principles for the next abstraction level. For each of these operating principles, the systems engineers then specify the actions that are required to perform this operating principle in the context of the system of system layer. Based on the specification of the operating principle, the systems engineers then specify a logical architecture specification for each feature. Combining all feature-specific logical architectures forms the overall logical architecture of the system decomposition layer in which each element

represents an element on the next decomposition layer. The only special case in this process is the transition from the SoS to the SoI level, where the next decomposition layer only considers the SoI in the CUBE process. From the system element layer on, each logical component from the logical architecture describes a new decomposition layer on which all steps, as suggested above, are repeated. To obtain the technical architecture from the logical architecture, the systems engineers describe the mapping from logical components and interfaces to their technical implementation to specify the technical architecture of the respective decomposition element. Since the decomposition follows the decomposition of the logical and not on the technical components, an additional mapping from the technical elements in a lower level to their top level and vice versa is required. Finally, the realization abstraction layer realizes the technical architecture. For this, the systems engineers must allocate the technical architecture specification to the realization specification.

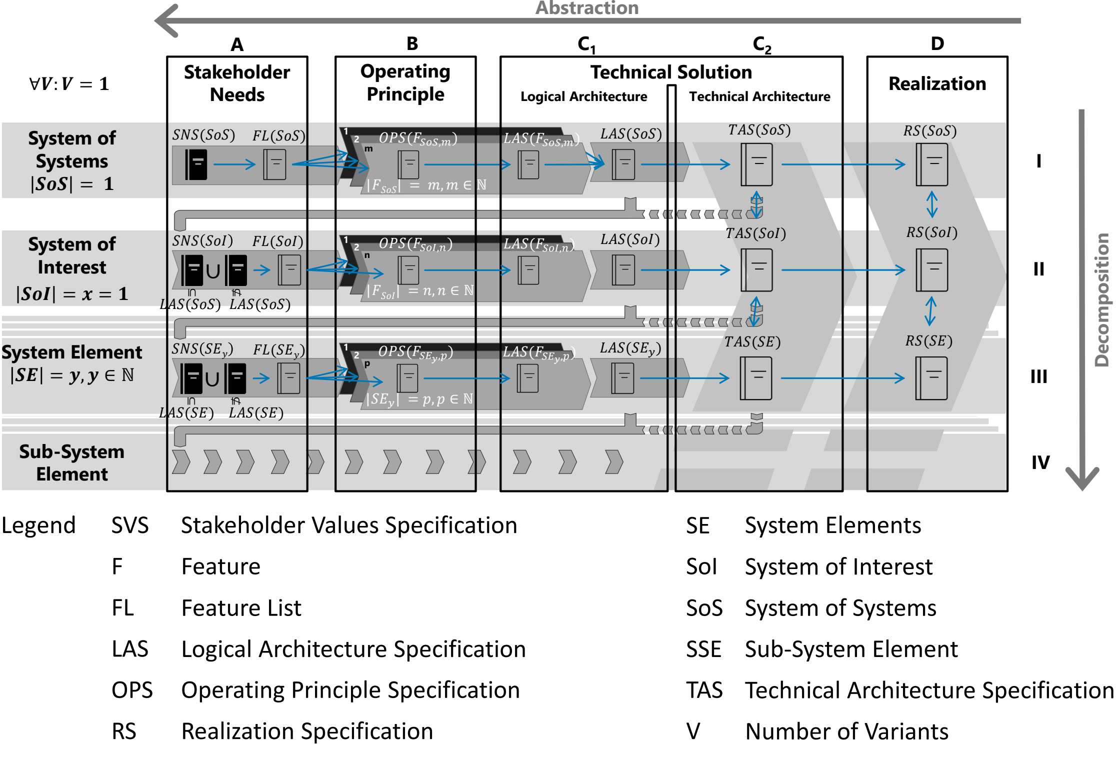


Figure 3.3: Artifacts created during the CUBE process based on [GKS+21] and [MSG+22].

As an additional reference for the artifacts created during this process, Figure 3.3 specifies artifacts for each level. Consequently, the $SVS(S)$ of a system S describes the stakeholder value specification artifact from which the systems engineers derive the

feature list artifact written as $FL(S)$ of the system S. For each feature F in this feature list, the systems engineers then define the operating principles in the operating principle specification of the feature written as $OPS(F)$. According to CUBE [Gra22, GKS+21], each feature then has its individual logical architecture, the systems engineer specify in the feature specific logical architecture specification artifact $LAS(F)$. The union of all feature-specific artifacts of the system S then results in the logical architecture specification of the system $LAS(S)$ for which the systems engineers then derive the technical architecture specification in the artifact $TAS(S)$. The realization specification of the system S is then in the $RS(S)$. On the next decomposition level, the stakeholder value specification is then the union of the stakeholder value specification of the previous decomposition layer P written as $SVS(P)$ and the stakeholder value specification of the current decomposition layer S written as $SVS(S)$.

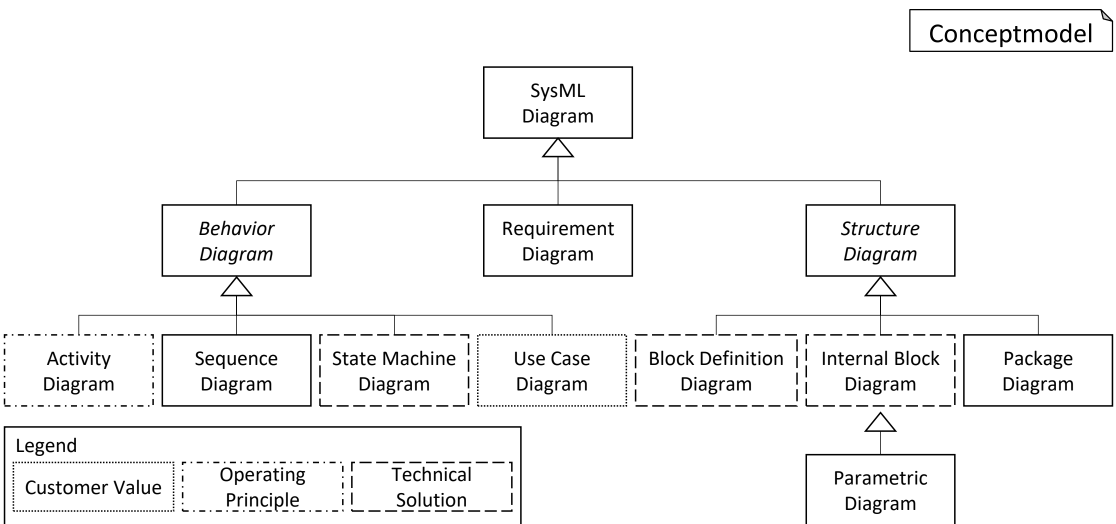


Figure 3.4: Mapping CUBE abstraction layers to the most suited diagram for system specification based on the SysML diagram overview presented in [Obj19].

Although CUBE defines these artifacts in [GKS+21], no modeling language is specified for modeling these artifacts, to allow a model-based documentation with different modeling languages and a document-based application with natural language descriptions. As a suggestion for a model-based application, Figure 3.4 maps the SysML diagrams to their counterparts in CUBE. Because the SysML is the de-facto standard modeling language for systems modeling in the automotive industry, as it is predominately used for systems modeling [MWJ+22], the SysML is used in this thesis for the following mapping: Since the SysML introduces a dedicated diagram type for requirements diagrams [Obj19], SysML requirement diagrams are very suited to integrate textual requirements

into the CUBE process into a modeling environment. As done in [JGW+21] or [MSG+22], CUBE applies a combination of different diagrams. To model stakeholder requirements and stakehoder needs, [JGW+21] and [MSG+22] apply a combination of use case and requirement diagrams. Especially for the SOTIF-compliant variant of CUBE, sequence diagrams also model scenario specifications on this level [MSG+22]. Feature lists are not modeled in the publications applying CUBE, whereas activity diagrams can serve to model operating principles. The architectures are then modeled in a combination of BDDs and IBDs, where BDDs model the structure and applicable elements as a black-box view of the system and the BDDs a gray- or with-box view.

3.1.2 Interim Summary and Challenges

For **RQ-3** ("Which elements of the SysML support the creation of the elements developed under **RQ-2**?"), the following summary is already possible. Generally, Requirement Diagrams, Activity Diagrams, State Machine Diagrams, Use Case Diagrams, Block Definition Diagrams, and Internal Block Diagrams support the basic CUBE methodology without further extensions. Regarding **RQ-3-1** ("How can requirements be expressed?") SysML requirement diagrams are an excellent candidate to express textual requirements in a model-based environment. For stakeholder values, as asked in **RQ-3-2** ("How can stakeholder values be formulated?"), the UML already introduced the UCD, which the SysML takes over from the UML without any extensions [Obj19]. For operating principles as asked in **RQ-3-3** ("How to model operating principles?") activity diagrams serve as a means to model these elements. Since the technical solution, the structural diagrams from the SysML are the only ones suited. Since no packages are used in the methodology, only the IBD and the BDD are left in the toolbox, and therefore considered as a suitable means for architecture modeling concerning **RQ-3.4** ("How to model logical architectures?") in the following.

Concerning **RQ-2.1** ("Which system specification process is applied in these projects?"), this thesis considers CUBE as a generalization of several processes. For **RQ-2.2** ("Which artifacts are created during this process?"), however, only the abstract specification artifacts SVS, FL, OPS, LAS, TAS, and RS from Figure 3.3 are identified as specification artifacts so far, which is not sufficient for the creation of a system specification in the automotive domain. Therefore, as requested by **RQ-2.3** ("Which elements are required to specify a system in the automotive industry?"), a further fine specification is required.

3.2 A CUBE-Specific Architecture Framework

Because CUBE deliberately restricts its process outcome specification on the abstract artifacts created during the systems engineering process (*cf.* Figure 3.5) and not on the information they contain to have a more flexible application framework for different

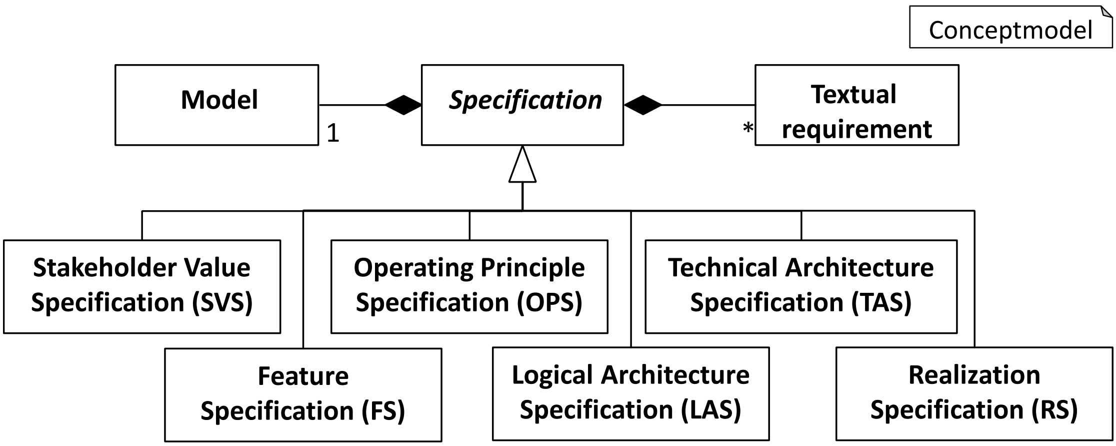


Figure 3.5: Specifications are an abstract concept consisting of a model and a set of requirements that is implemented by the specification artifacts defined in Figure 3.3.

system development projects at different manufacturers [GKS+21], authoring these specification artifacts might be troublesome and time-consuming. The main reason is that an unclear artifact and content specification requires the author to decide on the contents of these artifacts (if not further specified by a system development project-specific process), which might cause inconsistent specifications if different authors make different decisions in their specification artifacts. Moreover, as Figure 3.5 depicts, a model can occur in combination with a set of textual requirements. Thus, several combinations are possible. First, the textual requirements could accompany the model and contain additional specifications that are not contained in the model. Second, the textual requirements could contain exactly the same requirements than the model but in another representation. To cover the first and the third case, chapter 5 provides a DSL to specify textual requirements in a structured manner. Although it is generally a good engineering practice to reduce redundancy in the specification artifacts, chapter 6, chapter 7, and chapter 8 use transformation techniques to transform the model in a textual description as requirements to investigate the analogy in different representations. And third, that both of the previous cases are combined and some requirements are redundant to the model, whereas other requirements are uniquely formulated textually. Another challenge is that not only a model-based but also a document-based approach might be possible, which is not considered an application area and makes the projects that apply CUBE less comparable. Therefore, for **RQ-2.1** and **RQ-2.2**, the more specialized question arises: which elements are required to specify a system in the automotive industry as stated in **RQ-2.3**?

As an answer to this question, previous works already focused on developing an automotive architecture framework. Since CUBE, apart from the stakeholder value level, mainly describes different functional, logical, and technical architecture views, these architecture frameworks are highly related. Generally, there are standards on industry-independent architecture frameworks such as the ISO/IEC/IEEE 42010 [ISO11b]. According to this standard, an architecture framework within systems engineering and software development establishes a common practice for creating, interpreting, analyzing, and using architecture descriptions within a particular domain of application or stakeholder community [ISO11b]. Typical frameworks of other domains are, for example, the Department of Defense Architecture Framework (DoDAF) [Dep07], an enterprise architecture for defense planning and change management, the 4+1 model [Kru95], which describes the architecture of software-intensive systems, based on multiple concurrent views [Kru95]. Because this framework only describes the viewpoints rather than for which system or how they are created, industry-specific tailoring is required.

The automotive architecture framework [BGK+09, BGM+09] is one of the first approaches towards a standardized architecture framework in the automotive industry and is designed to describe vehicle systems across all functional and engineering domains [BGM+09]. It consists of two sets of elements: The mandatory viewpoints and the optional viewpoints. The mandatory viewpoints and their views include the functional viewpoint, which describes the functional decomposition and functional architecture; the logical viewpoint, which describes the logical decomposition and logical architecture; and the technical architecture, which describes the physical decomposition and technical architecture of the system [BGM+09]. As optional viewpoints, the automotive architecture framework as [BGM+09] consists of different optional views that aim at supporting vehicle manufacturers in achieving their priorities or quality attributes such as safety, security, reliability, availability, serviceability, energy, performance, cost, noise, vibration and harshness, and others as stated in [Web09]. With this structure, the automotive architecture framework aligns with CUBE, which requires a mandatory view for feature operating principles, a logical architecture, and a technical architecture. In this context, the view for feature operating principles can also be described as function-nets which are analogous to the functional architecture presented in [BGM+09]. Moreover, since the framework defines the required views and viewpoints, the framework conforms to the ISO/IEC/IEEE 42010 standard [ISO11b].

The architecture design framework [GGTP13] describes a related framework developed by Renault. It solely supports and focuses on the system design process defined in the ISO/IEC 15288 standard [ISO15] and is a derivation of the SAGACE method [Pen97, GGTP13]. It describes four architecture viewpoints based on artifacts created in a model-based systems engineering process and modeled in the SysML [Pen97, GGTP13]. The framework consists of an operational viewpoint, which describes the system and its behavior observed from a black box and user perspective. A functional viewpoint groups the performed activities (actions) from the operational viewpoint to system functions, a

constructional viewpoint, which describes the physical components of the organic architecture [GGTP13] to execute these functions, and the requirements viewpoint, which is orthogonal to the other viewpoints and describes requirements to the elements defined in the other views.

The architecture framework for automotive systems [DGS+14] describes another related architecture framework that bases on the previously described automotive architecture framework, architecture design framework, and additional concepts taken from architecture description languages [MT00, MLM+12] tailored to automotive domain, such as EAST-ADL [BLH+13], AADL [FGH06], and AML [BR01]. It consists of unifying the viewpoints presented in the other two frameworks. It adds specific viewpoints, such as the feature viewpoint for product line engineering, and an additional timing view that should specify the timing behavior of the components to enable timing analyses, such as bus schedulability analysis and CPU response time analysis views for the system [DGS+14].

Apart from these approaches, some other recent works describe an automotive architecture framework developed at Volvo and the experience with the application at Volvo [PKH+16, PKH+17]. To this aim, [PKH+16] presents an architecture framework based on the lessons learned at Volvo cars [EHPL15], which serves as a starting point for a framework proposal presented in [PKH+16]. It suggests seven viewpoints. (1) The continuous integration and deployment view aims to reduce the development time by enabling developers to add new functionalities incrementally, even after production. (2) The connected cars and safety view, which addresses future scenarios for connected vehicles. (3) The required safety measures in the vehicle architecture to ensure a safe operation. (4) The security and privacy of connected cars view, which displays the security and privacy aspects. (5) The ecosystem and transparency network equivalent to the value net viewpoint in [BGK+09, BGM+09]. (6) The autonomous cars view that addresses special architecture solutions for autonomous cars and self-adaptive systems [ST09]. (7) The modes management view, which describes the modes of operations of the vehicle and their transitions, and finally special viewpoints, which might be conceived to enable dissemination and communication of the architecture to developers and other stakeholders [PKH+16].

Unfortunately, none of these works led to the definition of a standard applied to the automotive industry. Since the approach presented in [BGK+09] and [BGM+09] does not focus its view on a single OEM, rely on a compatible understanding of optional and mandatory viewpoints as CUBE and also leave space for domain-specific additions [BGK+09], the following sections focus on this work as a reference and add additional aspects on the elements that the framework does not address.

Because the CUBE process as presented in [GKS+21] does not formulate any further requirements on when to use which textual or model-based specification, the following sections deal with this question in the context of **RQ-2.2**. Starting from textual requirements as a means of specification subsection 3.2.2 presents additional constraints and

artifacts to model the textual requirements as a cross-cutting concept in the context of CUBE. Then, subsection 3.2.3 discusses how use-case diagrams support the modeling of stakeholders, their needs and stakeholder values in the considered system development methodology. subsection 3.2.4 presents how operating principles manifest in the CUBE methodology and how a functional model aids in the development of system functionalities. Finally, subsection 3.2.5 presents the architecture modeling efforts and the required artifacts in the context of a logical and technical architecture specification based on [GKM+25].

3.2.1 System Specifications in the Automotive Industry

As a basis for the artifact model, this thesis draws on the automotive architecture framework in [BGK+09]. The framework relates to a meta-architecture framework that structures the system in views from different viewpoints [BGK+09] as Figure 3.6 presents. As basic concepts, the architecture framework presented in [BGK+09] draws on architecture views, viewpoints, and concerns, forming the overall architecture framework.

An architecture view in the meta-architecture framework (in Figure 3.6 denoted as "View") consists of one or more descriptions that are either formal, which means that the content of the view is described utilizing logical and mathematical ways of expression based on a precise and unambiguously defined syntax and semantics [HR04], or informal, which means that the contents of the view are not expressed based on a formally defined syntax and semantics *e.g.,* employing natural language in plain English. Note that both visual as well as textual representations might be formal or informal, as the definition in this context only refers to a defined semantics. Orthogonal to this dimension, the representation describes the notation of the description, which can be visual *i.e.,* represented in a textual tabular, graphical, or another related form of representation, or, opposed to that, non-visual *i.e.,* represented in a non-visual format like sound or electronically encoded data [BGK+09]. Because the automotive architecture framework only considers visual representations, Figure 3.6 only allows visual representations of a description in the framework. A view may also decompose into subviews. When examining a particular interest area, the meta-architecture framework introduces viewpoints that precisely one architecture view satisfies. It enables a user to examine a portion of a particular interest area [BGK+09]. In this model, viewpoint groups consider the concerns of involved stakeholders. As a result, at least one stakeholder and concern must be considered to form a valid architectural view. Finally, the stakeholder concerns the meta-architecture framework from [BGK+09] draws on, are a way of expressing crucial system requirements *i.e.,* requirements that address the system as a whole rather than a specific sub-set of its service or functionality [BGK+09, SSV98]. As a result, concerns are high-level requirements that reflect vague objectives or risks and usually refer to specific quality attributes like safety, maintainability, reliability, or cost [BGK+09].

In the meta-architecture framework, the relevant views are further defined by the scope

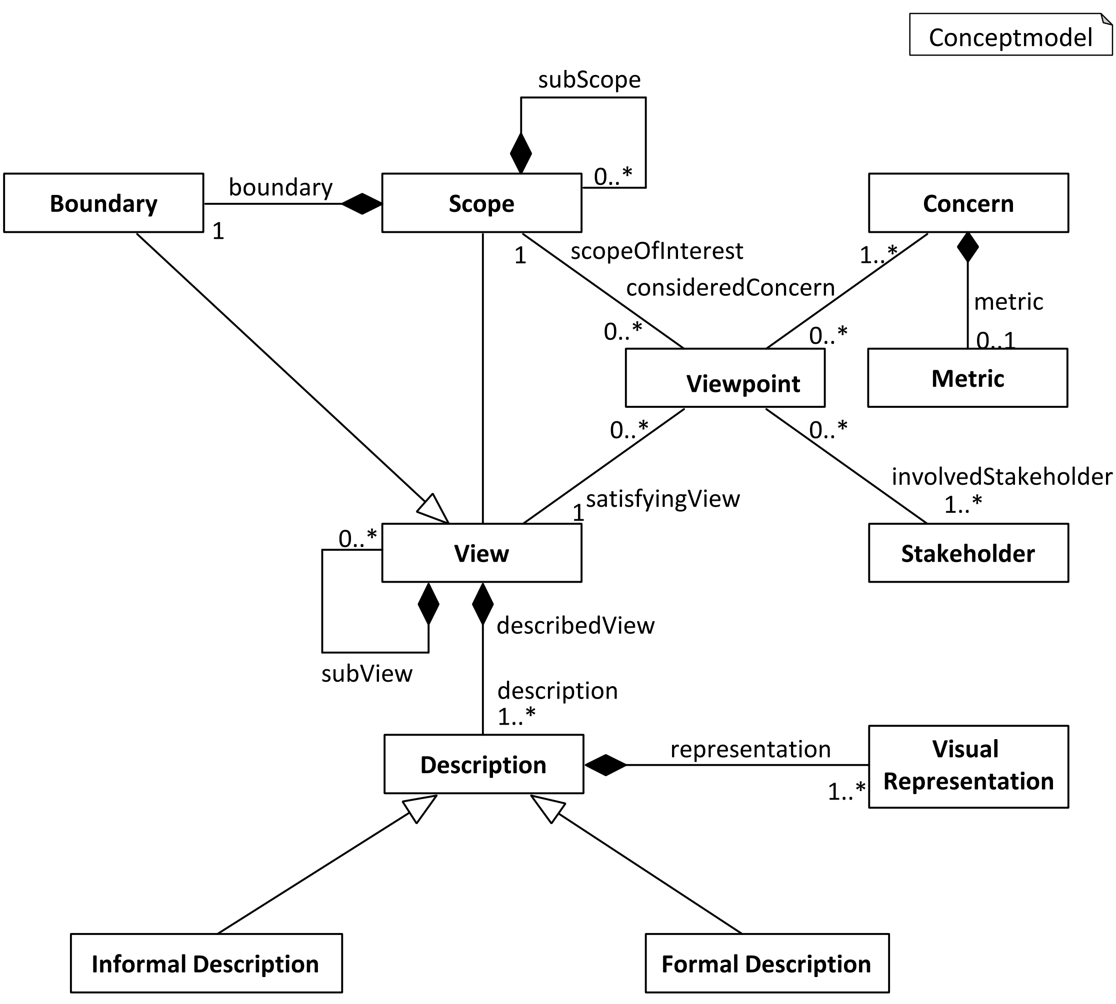


Figure 3.6: The meta-architecture framework as presented in [BGK+09].

of a system, which the boundary as a special form of view defines. The (system) boundary precisely distinguishes between the object of analysis and its context [BGK+09]. The meta-architecture framework refers to a conceptual or technical boundary of a system. It analyzes a set of related (architecture) elements *i.e.,* the inside view of the system, or the adjacent elements *i.e.,* the context or environment to this boundary [BGK+09]. Based on these concepts, [BGK+09] introduces the common architecture framework to which Figure 3.7 applies changes in the following areas: User functions are considered as features in this thesis, hardware typologies consist of general physical units, which might not only be electrical, but also mechanical or any other physical part, and finally, only

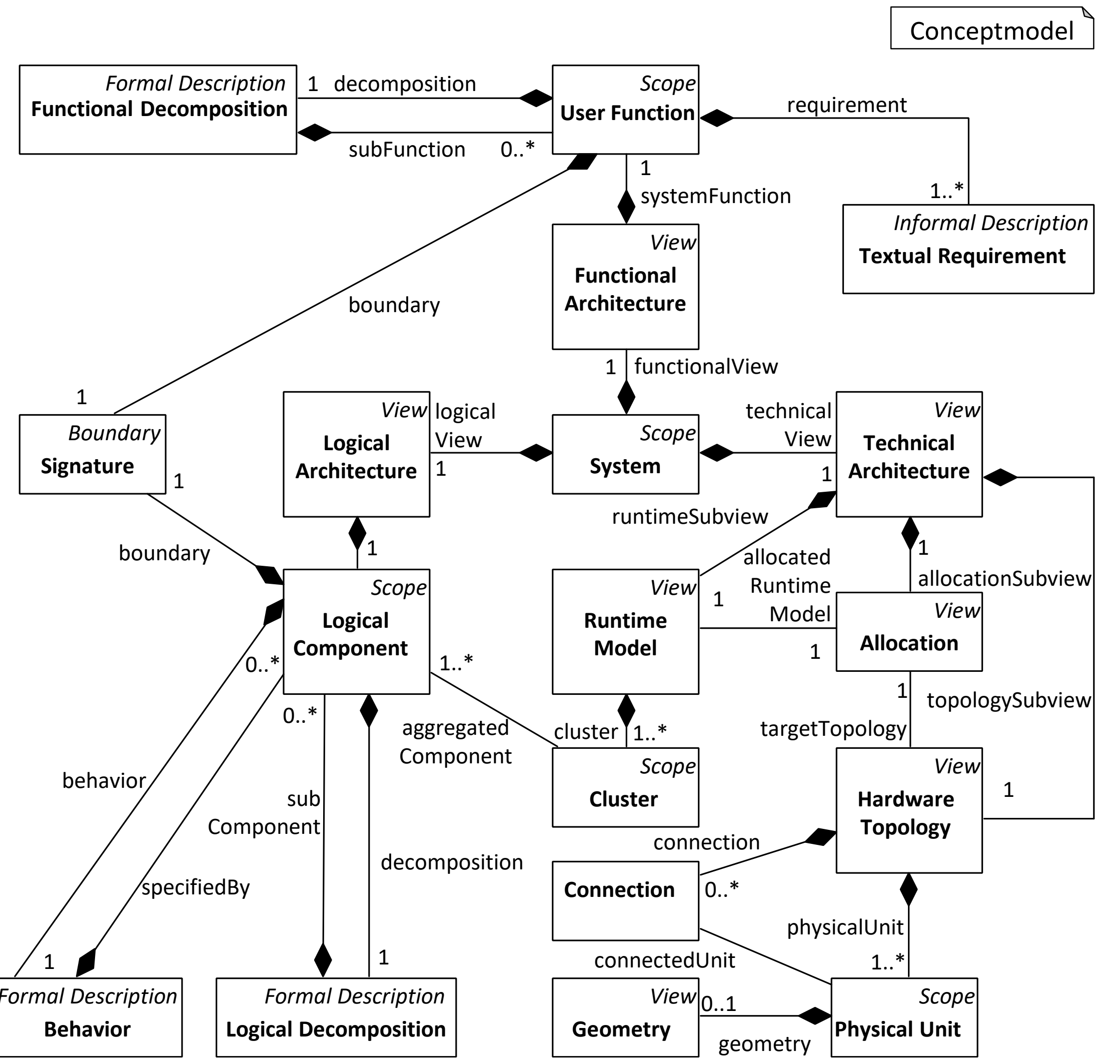


Figure 3.7: An adaption of the common architecture framework based on [BGK+09].

the system is considered as a scope, since the term total system contradicts the idea of a system as defined in Def. 1 as the central concept of this thesis.

To further describe and define these general frameworks in the context of CUBE concerning **RQ-2.2** and **RQ-2.3**, this section is structured as follows: first it discusses textual requirements as a cross-cutting issue of CUBE in subsection 3.2.2 and then follows the abstraction levels from CUBE to achieve an overall image from subsection 3.2.3 to subsection 3.2.6.

3.2.2 Cross-Cutting Issue: Specifying Textual Requirements

Since CUBE as presented in [GKS+21] does not restrict the use of textual requirements or models for system specification, the specification artifacts as presented in Figure 3.3 can either be models, textual requirements, or both. Because models are not necessarily diagrams in a graphical syntax but can also be textual representations, this thesis considers natural language requirements in the following as textual requirements. The specifications, as considered in this thesis, are an abstract concept that must address a concrete specification domain given by the specification process (in this context, the CUBE specification process as Figure 3.3 depicts). Thus, Figure 3.5 introduces the specification as an abstract class consisting of a model and an arbitrary number of textual requirements optionally. Because this thesis investigates system models for the development of automotive systems, the model as a means of specification is no longer optional, as presented in [GKS+21], but required as the cardinality in Figure 3.3 indicates.

As a way to combine the best of SysML models and natural language requirements, some approaches aim at introducing annotations [EDG+06], free-text attributes in the model elements such as in the SysML requirements [Obj19], or use different languages in the same context. By all of these methods, the knowledge of the system shall become explicit and unite stakeholders, architects, and stakeholders in a common language. Despite the advantages and disadvantages, different types of natural language requirements as a means of textual specification are known in the literature, and they are classified to give the reader the requirement. Typical requirement types are, for example, presented in [IEE94, KS98, vL01]. A typical classification is the differentiation between functional and non-functional requirements. Although differentiation between functional and non-functional requirements is typical in the automotive industry [GHZM17], it is a controversial topic [Gli07, LHM+14, EVF16]. While some authors regard the differentiation between functional and non-functional requirements as artificial because calling every requirement that does not specify a functionality as a non-functional requirement is overly simplistic, somewhat misleading, and therefore unhelpful [Bro15], others criticize many non-functional requirements also contain functional aspects [EVF16].

To overcome these problems, the approach presented in [Gli07] does not simply classify requirements as functional or non-functional but introduces a more sophisticated differentiation and classification scheme to classify textual requirements. As Figure 3.8 shows, the taxonomy proposed in [Gli07] distinguishes between system requirements, which set requirements for the system or product to be developed, project requirements, which a system development project must meet for successful project execution, and process requirements, which formulate requirements that the organization must follow for successful development. Because project and process requirements are not related to the system but lay an organizational context for the system development, these requirements are usually not further considered in the system development [Gli07]. According to [Gli07], the system requirements subdivide into four sub-types of requirements: The

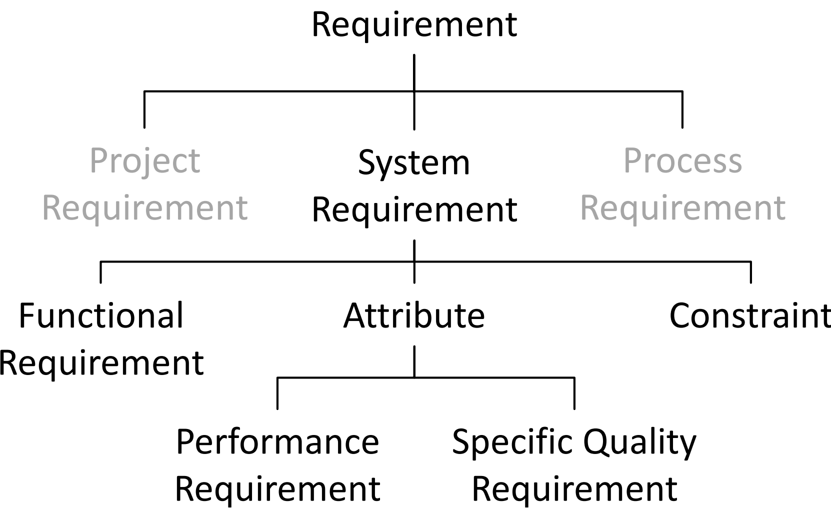


Figure 3.8: Different requirement types based on a taxonomy for concern-based requirements according to [Gli07].

functional requirements that describe functional concerns, the performance requirements that specify performance concerns, the specific quality requirements that formulate quality requirements not covered by meeting the providing the expected functionality, and constraints to the solution space. The specific quality requirements serve the quality concerns apart from meeting the functional requirements. Finally, some constraints constrain the solution space beyond the necessity to meet the requirements mentioned earlier.

Because the requirements may address multiple topics identified, for example, in [EVF16], classification rules weight the concerns within the taxonomy presented in [Gli07]. According to these rules, functional requirements weigh higher than performance requirements, which are higher weighted than specific quality requirements. Finally, all other constraints on the system are classified as constraints.

Based on these observations in the literature, the methods presented in this thesis focus on modeling system requirements categorized according to the taxonomy presented in [Gli07]. Moreover, as presented in [Nus01], the approach this thesis presents requires that requirements are given on all levels. Because of the controversy on functional vs. non-functional requirements as presented in [IEE94, KS98, vL01, Bro15], requirements formulated in this thesis are not distinguished based on their description of functional or non-functional behavior, but on their relevance for the system development process and reusability for other system development projects. Consequently, functional requirements form a reusable subset of functionalities that a system must provide regardless of how it realizes these functionalities and their performance. Attributes describe the system's quality by constraining the system functionalities or other qualities to achieve a

specified performance or special quality to address the latter topic. Finally, constraints constrain how the system shall realize these functionalities and attributes.

This clear distinction between the functionalities, attributes, and constraints aims to support the typical structure of the automotive domain and the automotive industry in general. Because automotive manufacturers aim to reuse their vehicle or part specifications between different vehicle development projects, separating functionalities aids in adopting the abstract characteristics that the automotive system under development shall have a specific functionality. Following the strict separation between functionality, performance and specific quality, it is possible to change these requirements depending on the system development project. Finally, the constraints enable the manufacturers to constrain the solution space during the development to meet the strict cost and time constraints typical for the automotive industry.

For a vehicle manufacturer, this implies that the expected base functionalities within its product are to be specified to maximize its reusability. Because almost all vehicles primarily serve to transport people and goods, it would be possible to specify a requirement that a manufactured vehicle provide the owner with the ability to transport people and goods. Because of its high level of abstraction, this requirement is reusable in most concurrent vehicle development projects. Because this requirement is not specific enough to define the vehicle type or its design, performance and specific quality requirements serve to restrict the solution space in the respective development process, *e.g.,* by defining the number of people to be transported at the same time or the weight of the goods to be transported. Finally, because most automotive manufacturers already know specific constraints to the development or solution, additional constraints could be used to exclude specific solutions *e.g.,* by specifying that a road vehicle with a specific number of tires or a specific propulsion solution shall be developed based on strategic decisions.

3.2.3 Specifying Customer Requirements as Stakeholder Needs

According to [GKS+21], customer values, or stakeholder values in this thesis, describe the stakeholder needs and expectations of other involved systems or actors. Because multiple stakeholders are typically affected by an automotive system, the systems engineers must prioritize and identify stakeholder values during the most abstract step of the CUBE specification procedure (*cf.* Figure 3.14). As vehicles are used in different contexts by different people, their values as vehicle users are as diverse as the users themselves. Therefore, people who own or use a vehicle become essential stakeholders in vehicle development, which diverse interests and values. A sports car owner, for example, would rate the fun and the race-track capabilities higher than the economic and fuel efficiency operation of the vehicle compared to a commuter who wants to use a vehicle as a comfortable and economic means of transportation to get to his workplace and back. Because road vehicles are omnipresent in many countries, the people affected by the usage of the vehicle are also crucial for the acceptance of the individual transportation.

Because noise, toxic emissions, and traffic accidents caused by many vehicle types or drivers are significant health issues [KWV+16], vehicle manufacturers must regard the stakeholder needs of the people affected by the vehicle and the customers. To protect people and the environment from these effects of mobility and their political interests, many countries have legal requirements that newly sold series vehicles must meet to be sold and used in that country. Other standardization organizations also set vehicle development standards besides the national influences. Prominent examples of these legal requirements or national and international standards are, for example, the UNECE vehicle regulations for Europe and the United Nations, the US FMVSS regulations, or the Chinese GB standards.

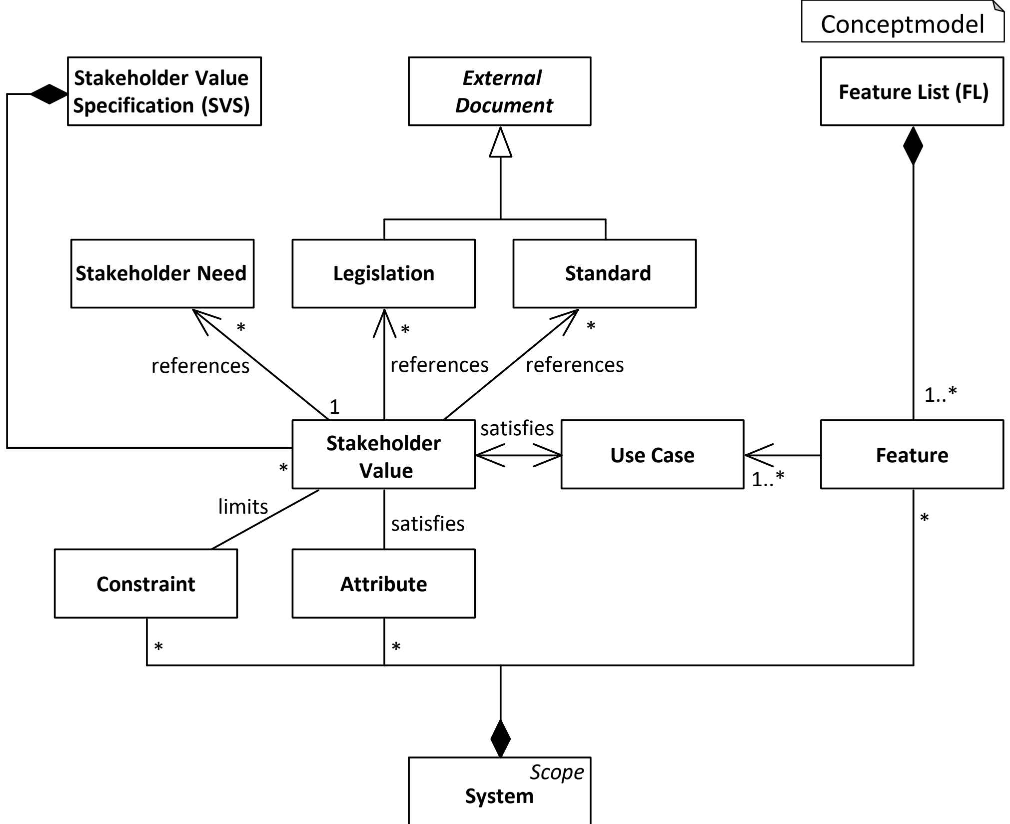


Figure 3.9: The stakeholder values denote the stakeholder value specification by referring to stakeholder needs or external documents as specified in the specification artifacts from Figure 3.3.

Based on these observations, a stakeholder value specification of a system S that Figure 3.3 presents as $SVS(S)$ consists of stakeholder values. These stakeholder values

then refer to a stakeholder need, legislation, standard, or any non-empty combination of them. According to the feature-driven interpretation of CUBE as presented in [GKS+21] and [Gra22], a system has to fulfill a use case to satisfy these needs. For example, a vehicle may provide a transport goods use case that satisfies the vehicle users need to transport goods to another location. Because many of these use cases might be related, for example, the transportation of people and goods after loading the vehicle the same, CUBE introduces a feature that addresses one or more use cases. As a way to give a better overview of the capabilities of a system, a feature list summarizes all features of a system. To derive a joint artifact structure for model-based system development projects as investigated under **RQ-2.3**, Figure 3.9 summarizes the artifacts and their relationships as presented before.

As stakeholder values primarily formulate system requirements, this thesis categorizes them as attribute, constraint, or functionality, as discussed in subsection 3.2.2. Therefore, a stakeholder value is either an attribute, if it classifies as performance or specific quality requirement according to the classification scheme from [Gli07]. Because features describe the functional characteristic of the system, all system features are consequently a functional requirement of the system and present a functionality or feature of the system. Since the performance and specific quality requirements can be understood as features, these are the system's attributes. Finally, the constraints that follow from the stakeholder value requirements can be formulated and understood as constraints in the system context.

As shown in Figure 3.5, the stakeholder value specification differs on SoS on SoI and SE levels, which is caused by the special nature of SoI and SE levels, where the requirements from the logical architecture of the previous levels must be considered. Therefore, the stakeholder value specification gets an additional part specifying these parts. While this will become important in the context of the specification methods described in chapter 6 and chapter 8, they, by design, must not affect the artifact structure to make a union possible as Figure 3.5 depicts.

3.2.4 Specifying Functionalities as Operating Principles

To define the implementation of a feature in a solution-neutral way, CUBE introduces the operating principle for each feature as a solution-neutral description of the system behavior when realizing this feature. An operating principle consists of actions that are required to implement the feature. For example, a high-level operating principle for a vehicle feature 'transport people' could consist of the following steps:

1. Load people,
2. Drive to destination,
3. Unload people.

Although, at first glance, this example seems sufficient for the specification of most trips, the benefit for system specification and trip execution is only sufficient with further refinements of what happens in each step in more detail and required inputs and outputs to other systems. As a solution to the refinement problem, it is helpful to decompose this level's abstract steps to decompose features further. These so-called sub-features further decompose the features of the higher level. By this, for example, the load vehicle step could be refined to the following steps:

1. Open door,
2. Load passenger,
3. Secure passenger,
4. Close door.

When looking at this refinement, another problem becomes evident: The execution order is not solely specified of the actions, and an altering is possible. For example, securing passengers *e.g.,* with a safety belt, is usually possible before and after closing the vehicle door. Moreover, the door's existence as a means to enter and exit the vehicle must be known at the feature decomposition level from the vehicle decomposition, and the term doors itself restricts the vehicle developers to the development of vehicles with doors which might seem not solution neutral. Therefore, the feature and system decomposition must align for the correct specification. It is also helpful to define these elements as an interface to solve the second problem of the required in- and outputs. For example, we could specify that the feature transport people takes humans as an input, drive to the destination requires energy, and unloading people transfers humans to the system environment. In a good specification, only the immediately required elements should be specified to allow high reuse and prevent potential conflicts with other interfaces in another decomposition. Combining all interfaces provided or required by the action leads to the feature's signature, which describes the inputs and outputs (*e.g.,* signals, events, or messages) [BGK+09].

To reflect these observations in an artifact model, Figure 3.10 presents the elements contained or used in the operating principle specification. The central element of an operating principle specification is the feature, which consists of one or more actions that must be executed to perform this feature. This way, the feature addresses one or more stakeholder value specification use cases. As a viewpoint, a feature list, as suggested in Figure 3.3, is introduced, containing all the system decomposition-level features. Because features may decompose into sub-features, a feature might have one or more sub-features, which have one or more actions and address one or more use cases. Consequently, the action might require or produce elements required by other actions. These actions have interfaces, which form the signature of the feature.

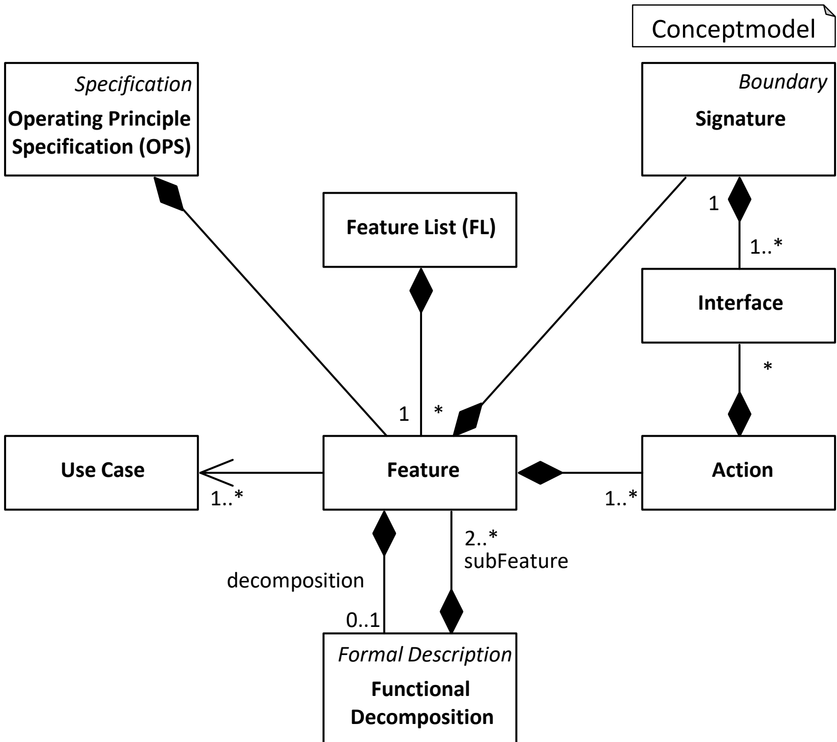


Figure 3.10: The features denote the operating principle specification by relating the addressed use cases with the action sequence and their interfaces that define the signature of the feature to implement the specification artifacts from Figure 3.3.

The actions serve as the functional requirements of the operating principle, and the signature and interfaces define the interface requirements. For example, to transport persons and goods with a vehicle, a driver needs to enter the vehicle (optionally with goods), drive the vehicle to the destination, and exit the vehicle (optionally with goods) after reaching the destination. Because the actions do not allow for an execution order of the actions within an operating principle, an additional functional execution requirement set is required, *e.g.,* in the form of an activity diagram.

3.2.5 Specifying Technical Solutions in System Architectures

Technical solutions in CUBE address two different architectural views—first, the logical architecture, which defines the realization of an operating principle from a logical perspective. Second, the technical architecture that implements the logical architecture using the concrete technical realization of this logical principle.

Specifying Technical Solutions as Logical Architectures

A logical architecture consists of (sub-)components executing the actions defined in a feature. For this purpose, the logical elements need to process the inputs of the actions they execute to produce the outputs of the actions. As a result, the logical component must have a signature that consists of the inputs and outputs of all actions they execute. The logical components of the logical architecture are connected to provide the outputs of an executed action as input for a logical component that executes an action that requires this output as input.

Since this principle also applies to the technical architecture [GKS+21], it is vital to note that the logical components that form the logical architecture reflect an abstract component capable of performing the functionalities of the actions it executes, independent of its later physical realization. Consequently, the logical architecture is based on the functional specifications of the operating principle and independent of the technical realization [JGS+20, JGW+21]. In addition to the logical components, the connectors within the logical architecture are independent of the actual communication form *i.e.,* they do not address aspects regarding the communication medium, encoding, or security. Consequently, the data, energy, or matter [Wei84] exchanged between the logical components solely depends on the inputs and outputs of the actions it executes, which are defined in the operating principles of the features the system shall realize. As features, logical components may be decomposed into sub-components. Therefore, logical components are similarly decomposable to a logical decomposition consisting of logical components.

Since logical architectures perform actions, they also share a signature defined by the interfaces (*i.e.,* the inputs and outputs) of all actions they execute. Apart from this manly static structure definition, the logical components have a (mainly formal) behavior description. In this context, this thesis follows a logical behavior definition as presented in [BGK+09]. Thus, the behavior description follows a state chart defined in the UML/P [Rum06], consisting of states and transitions. Transitions are triggered by triggers coming from an actor and are only used to change a state if the trigger is triggered and a guard condition is fulfilled. As an effect, a feature may be executed when a transition is taken or while a state is executed. To get a well-formed specification of a logical component, all actions it executes must be executed in at least one state or as an effect of a transition.

Since a logical architecture abstracts from the technical realization, abstract domains or patterns may serve as logical executing elements. A fictive vehicle could, for example, logically decompose to a propulsion system that uses the energy provided by an energy system to generate the motion and power or torque to propel the vehicle. The chassis then uses this force to transmit it to the environment to accelerate the vehicle.

Figure 3.11 specifies the artifact model of the elements specified in the logical architecture described in Figure 3.3. In contrast to the previously presented artifact models,

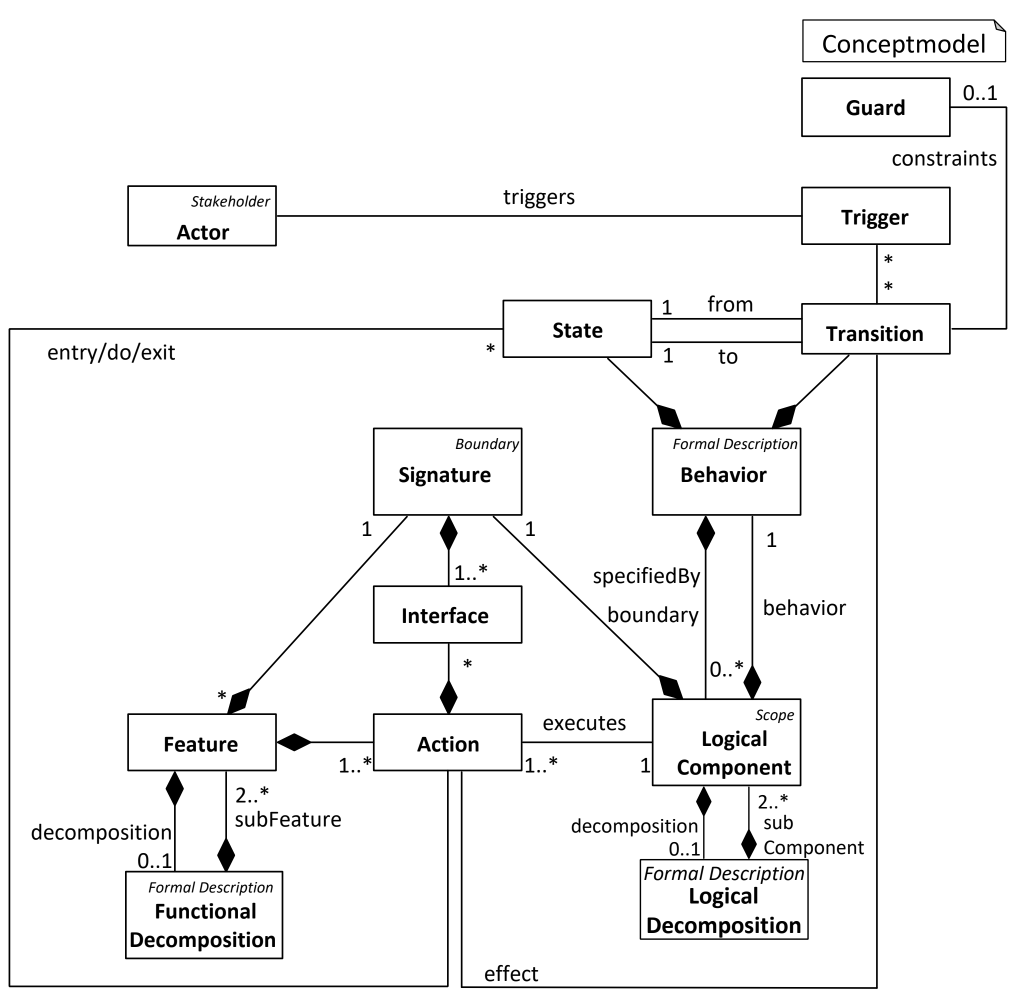


Figure 3.11: The logical components execute the the actions the operating principle specifies to implement the specification artifacts from Figure 3.3.

this view on the logical architecture specification considers two logical architecture specifications as described in Figure 3.3: The logical architecture specification of a feature ($LAS(F)$) and the logical architecture specification of a system $LAS(S)$. The logical architecture specification of a feature F denoted as $LAS(F)$ consists of all logical components that execute actions defined by the feature and the states and transitions using these actions. In contrast, the logical architecture of a system S denoted as $LAS(S)$ consists of all logical components a system consists of, their behavior description, and the actions they execute. As a result, the logical architecture specification of a feature ($LAS(F)$) describes a view of the logical architecture that only consists of the logical

components that execute actions of the feature F, which leads to the challenge that the corresponding behavior description might need to be completed. Consequently, two views on the behavior of the logical architecture are possible. First, a view containing all states and transitions as described in the behavior of the logical components contained in the considered logical architecture, without the actions that the other features than the current feature consists of. Second, a logical architecture view only consists of the states and transitions connected with actions that the considered feature consists of.

Specifying Technical Solutions as Technical Architectures

In contrast to the logical architecture, the technical architecture describes the technical implementation and the actual connection, not a logical principle for the implementation [GKS+21]. Therefore, a technical architecture must specify the physical realization of the technical components, considering the aspects related to their technical implementation, such as the communication medium, encryption, or security. For this reason, the technical architecture of a system must not only address the system's runtime behavior but also take the hardware implementation of the system into account. Since hardware works together to accomplish a specific purpose, hardware topology is an essential concept for describing the various parts (which can be mechanical, electrical, or electronic) since it describes the structure of the interconnected parts to achieve that purpose. To achieve this aim, the hardware topology, according to [BGK+09], consists of physical units and their connections. A physical unit further decomposes into a geometry, which describes the geometrical characteristics of the system [BGK+09]. According to [BGK+09], the physical unit represents all physical components that either interact with their environment, such as sensors, actuators, or mechanical devices, process software such as microcontrollers or provide other functionalities to the systems, such as hardwired controllers or mechanical components.

Whereas it is sufficient and desirable for the logical architecture to restrict itself to components on a high abstraction level, which subsumes several technical solutions, the technical architecture must consider these in detail. Therefore, a logical component for a propulsion system deliberately combines several energy converters that convert different forms of energy, such as thermal, chemical, hydraulic, pneumatic, or electrical energy into kinetic energy, whereas it is the task of the technical architecture to close this gap and to specify the technical realization of the underspecified solution given in the logical architecture.

Therefore, the technical architecture of a propulsion system must describe the technical realization of an energy conversion and thus specify, for example, an electric motor that converts electrical energy into kinetic energy. This internal combustion engine converts chemical energy into thermal energy or similar solutions. In addition, logical signals from the signature of the logical architectures must be refined to achieve an implementation-specific interface specification that also considers technical realization. Thus, the energy

input for an internal combustion engine must refine the energy input of the logical propulsion system to the specific fuel that the engine burns, and the control signals and communication paths must be not only mapped to the technical communication paths but also their content adapted to the combustion process.

As this example shows, at the level of the technical architecture, particular attention must be paid to the compatibility of the various systems and the system's alternative solutions and configuration variants. For example, an internal combustion engine cannot be operated with electric power, and an electric motor cannot be operated with fuel, which must also be considered at this level *e.g.,* in the technical architecture of the energy system.

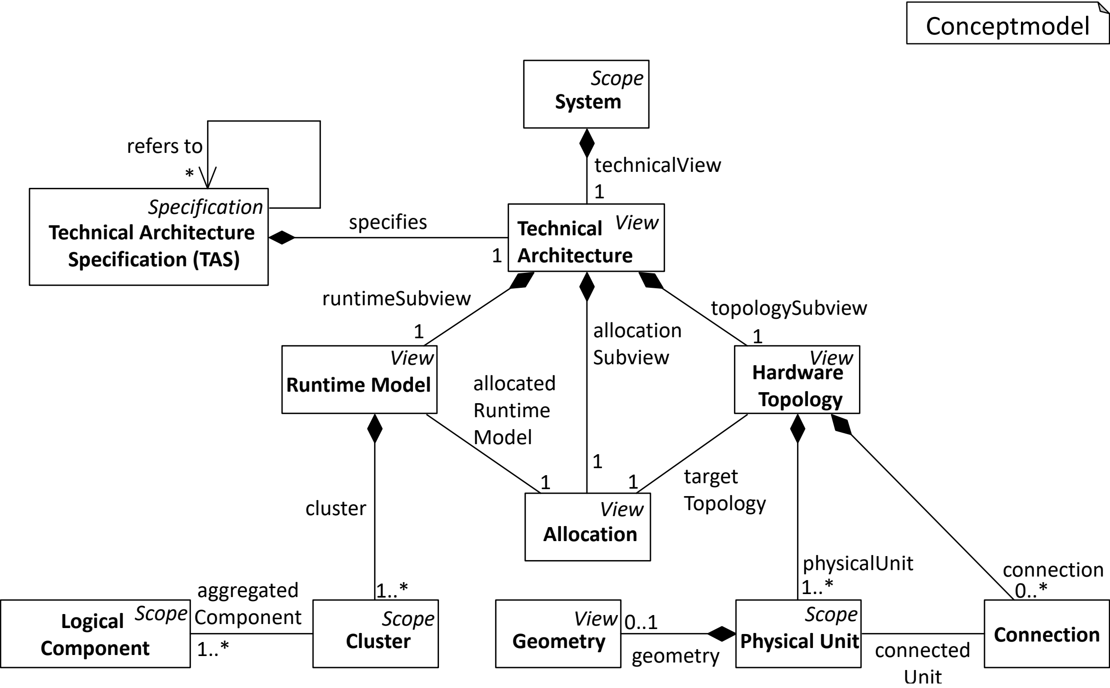


Figure 3.12: The technical architecture denotes the technical architecture specification by allocating the runtime model to the hardware topology implementing the specification artifacts defined in Figure 3.3 based on the automotive architecture framework described in [BGK+09].

Figure 3.12 depicts the technical architecture specification based on the [BGK+09]. According to the class diagram described in Figure 3.12, the technical architecture specification of a system describes a system from a technical viewpoint. This view consists of three subviews: the runtime model, the allocation, and the hardware topology [BGK+09].

The runtime model describes the system's behavior at the technical abstraction level. It consists of one or more clusters aggregating at least one part of the logical architecture [BGK+09], which describes the functionalities a physical unit or part of it has to implement. As a result, scheduled and deployable software and mechanical parts are required for the technical architecture to implement or execute this functionality. On the other hand, the hardware topology describes a view of the physical units and their connections to the element. In contrast to the original diagram from [BGK+09], the hardware topology does not only consist of electronic components but also other physical components such as electrical or mechanical components. Finally, the allocation connects the runtime models to their executing hardware topologies.

3.2.6 Specifying the Realization

Finally, the realization describes and specifies the components' realization in accordance with their specification in the technical architecture. Therefore, this view must not add any components or connections to the technical architecture but only specify the atomic components in detail to achieve the fine specification for realizing the system, which is the only step added here because the automotive architecture framework [BGK+09] does not differ between software and hardware. Moreover, since physical components are typically subdivided into (atomic parts) and assemblies, primarily for [BG21], this additional layer of abstraction is also added to the architecture framework from [BGK+09].

Figure 3.13 describes the realization specification. In this view, the specifications contain the requirements for realizing the components specified in the technical architecture as presented in Figure 3.13. It consists of a hardware specification and a software specification. The hardware specification specifies the physical unit for realization. The software specification specifies the software unit executed on a physical unit. Because not all physical units can execute software *e.g.,* a spring has no control unit and cannot execute any software, software units may only be specified for physical units with this capability.

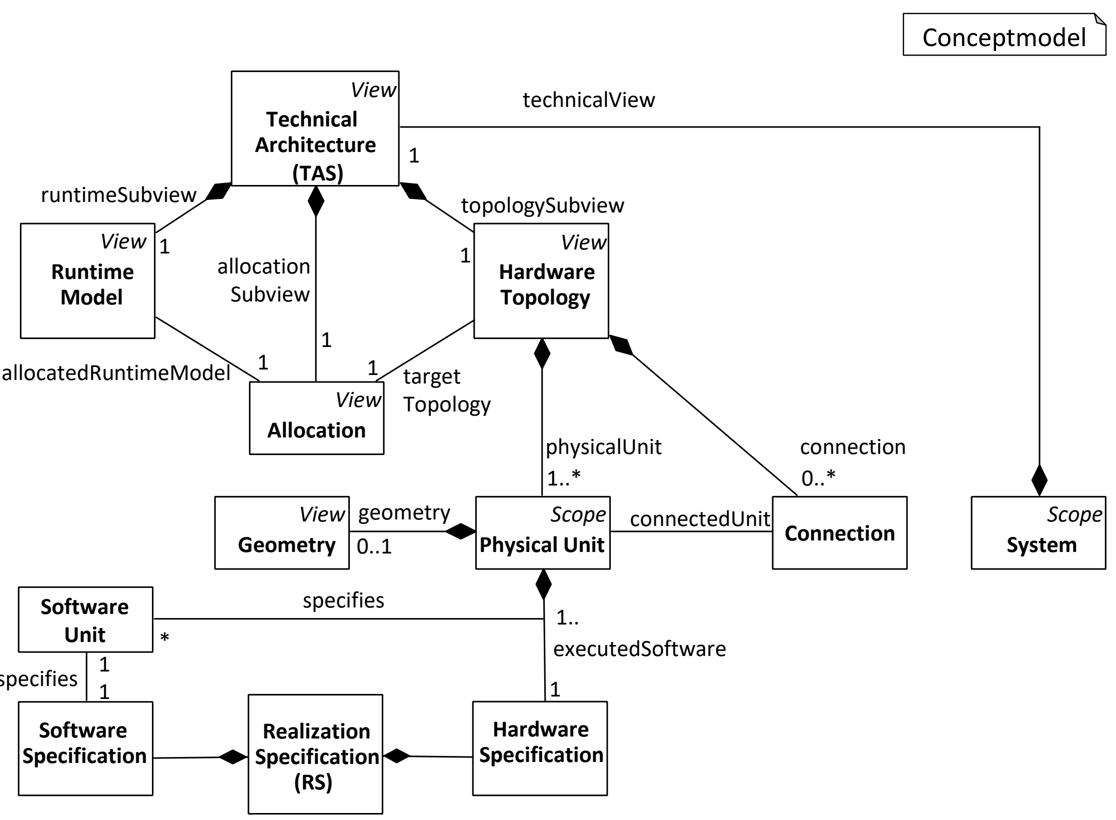


Figure 3.13: Realization to implement the specification artifacts defined in Figure 3.3 extending the automotive architecture framework described in [BGK+09].

3.3 A Critical View on CUBE and its Research Gaps

As the previous findings from section 2.6 show, the literature provides various systems engineering methodologies with a different focus on system development and its process. The CUBE methodology creates several views of the system, represented in the variant of the automotive architecture framework [BGM+09, BGK+09] developed in section 3.2 and its process in Figure 3.2 the mutual basis for all projects chapter 4 considers. Consequently, Figure 3.2 answers **RQ-2.1** ("Which system specification process is applied in these projects?").

Knowing this process, Figure 3.3 highlights the artifacts the systems engineers create using CUBE and therefore answers **RQ-2.2** ("Which artifacts are created during this process?"). As this overview describes the views required for system specification, the challenge no longer lies in identifying the artifacts to be created as part of system spec-

ification, which is particularly challenging in systems engineering projects according to [BGRW18] and [HJK+21], but in detailing the content of the artifacts in a way that they support system development according to the CUBE specification process. Since the artifacts from section 3.2 create multiple interrelated views of the system, managing these views to achieve a consistent system specification becomes a new challenge. Therefore, section 3.2 focuses on the derivation of the required system elements for a system architecture specification based on [BGK+09] and [BGM+09], leading over several individual views to the combined overview in Figure 3.14 as an answer to **RQ-2.3** (“Which elements are required to specify a system in the automotive industry?”).

With these outcomes, the next challenge lies in creating these elements. For example, the stakeholder requirements in a stakeholder value view significantly influence the realization and its elements described in the technical architecture, although these two views are not directly linked. Stakeholder requirements from environmental protection, for example, which are raised in the context of automotive development, have had a significant influence on powertrain development for years and lead not only to changes in conventional powertrains but also to utterly new powertrain concepts such as electric powertrains or hydrogen powertrains. The problem, however, is that these interrelationships are often only known to the developers and technical experts of the individual systems and need to be made explicit to other system developers or viewers of different abstraction levels. This problem is also exacerbated by the fact that the creators of the individual views are often different people and must coordinate their specifications with each other in order to create a self-contained and consistent system specification. In order to make these dependencies, which are initially only implicitly known to the respective developers, evident in the development project, the relationships between the views must be made explicit and integrated into the system model. For example, the creators of the stakeholder value view and all creators of other views must link their work and keep it consistent so that if a change in one view is required, the resulting change in another view also remains intelligible. The systems engineering methodologies presented in section 2.6 and CUBE as presented in [GKS+21] either do not further address these challenges or presuppose a correction of the individual views by the creator. However, this neglects the fact that, on the one hand, individual aspects are easy to overlook due to a lack of understanding of the relationship between the views, and, on the other hand, the creators of the individual views must spend a considerable amount of effort to keep the views consistent. As a result, the creators of these views might consider the maintenance of their system view as frustrating and their creation as time-consuming, which might not only negatively influence the introduction of MBSE in the company but also affect the acceptance of MBSE in projects where MBSE is already an established methodology, since these two aspects are mentioned as top three drawbacks of MBSE in [CTE+22]. Therefore, in the following chapters this thesis deals with the question of how model-driven system development is applicable in the automotive industry (**RQ-4**). Since in model-driven development, the models created in the development process are

not only created but also taken as the central drivers of the process [Sch12], the purpose of utilizing MDD is that the views or needed parts can also be the views or needed parts by utilizing model refinement or model transformation as defined in Def. 15.

As a way to develop the required transformations in an MDSE approach, the following chapters identify for each view a subset of the SysML as described in **RQ-3** that enables such a view and then elaborate a transformation into other related views. The work focuses on the following aspects and their application in the context of a selection of automotive system development projects at FEV GmbH in cooperation with the acronio GmbH. chapter 4 first describes the framework considered for the evaluation. Then, in the following chapters, the individual CUBE views' are explained in more detail. Since textual requirements are a cross-cutting issue on all levels of CUBE, chapter 5 first deals with how requirements are expressible as models in a model-driven systems engineering process. Both the possibility is considered whether and how textual requirements are usable for model generation, and the possibility to generate textual requirements from the system models as an additional view concerning **RQ-4.4** ("Are the models that result from a model-driven specification comparable to document-based specifications?"). Subsequently, chapter 6 deals with how stakeholder values can be a starting point for system development in a stakeholder-based view. For this purpose, chapter 6 elaborates a syntactic and semantic framework for this description, and presents an approach for the automatic derivation of operating principles on the next abstraction level (**RQ-4.1**). Based on these findings, chapter 7 deals with operating principles and the possible automated derivation of logical architectures. For this purpose, the models of the architecture and partial models of a logical reference architecture are linked and treated (**RQ-4.2**). chapter 8 then describes the collaborative creation of the architecture. Since technical architectures are either described top-down using a similar approach but in practice are more likely to be created bottom-up, this aspect is not considered further in this work in the context of creating transformation rules. Therefore, subsubsection 3.2.5 only presents a brief discussion of the findings as a starting point for future work.

To make the approach applicable in the industry, balancing the necessary efforts for the application of the method with the benefits for the applying systems engineers is required [GKM+25]. Moreover, fulfilling several process requirements as, for example, specified in [SPI15] becomes obligatory during an industrial application. Based on these requirements, this thesis sticks to the following methodical requirements that the methodology and the transformations must fulfill based on [GKM+25]:

MRQ *Which requirements must be fulfilled to make a systems engineering methodology applicable in the automotive industry?*

- **MR-1** *Specification models created using the method must be reusable to realize reduced time-to-market constraints.*
- **MR-2** *Specifications must be adaptable to specific markets or vehicle projects.*

- **MR-3** *Specification models must be consistent.*
- **MR-4** *Specification models must be bidirectionally traceable (e.g., from a requirement to a component and vice versa).*
- **MR-5** *Creating specification models must be possible without redundant manual modeling tasks.*

The following chapters refer to these qualities to discuss and evaluate their fulfillment in the context of the developed transformations and the model-driven systems engineering methodology based on CUBE, discussing the quality and applicability of the developed methodology.

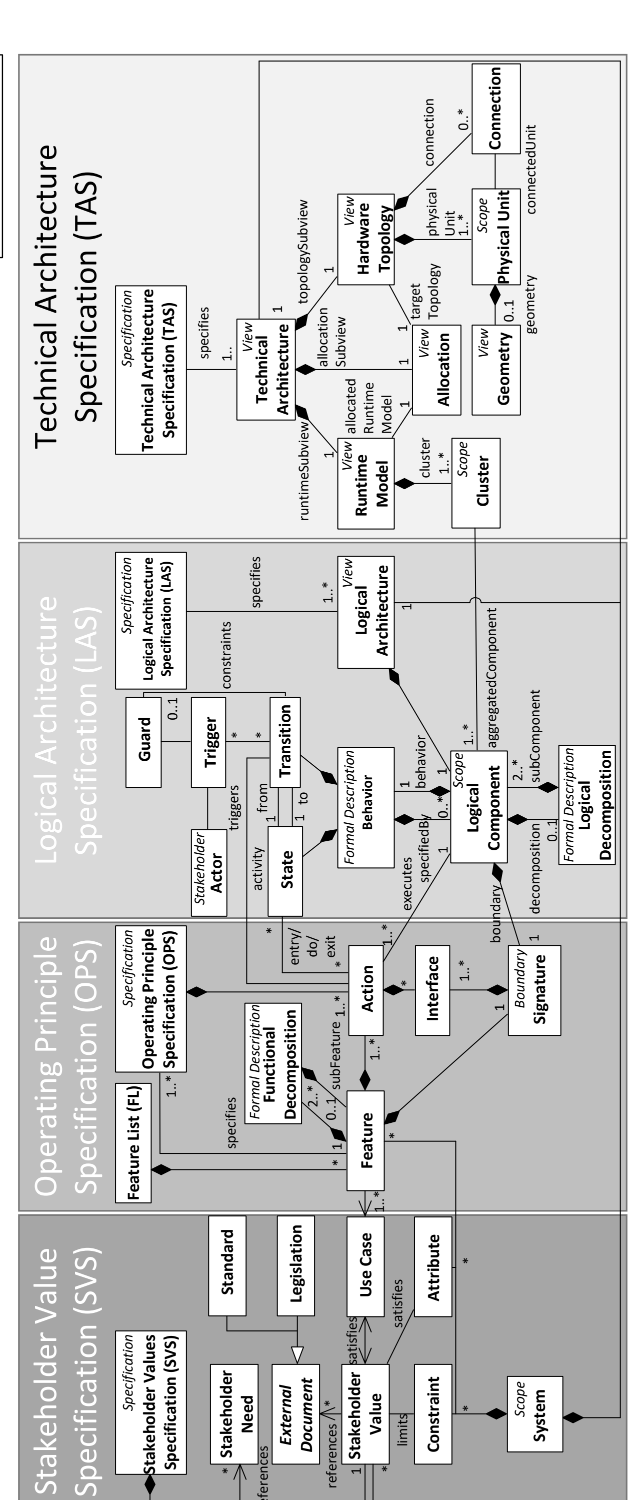


Figure 3.14: A combined view of the specification artifacts and their elements from Figure 3.9, Figure 3.10, Figure 3.11, Figure 3.12, based on [GORW23].

Chapter 4

System Models in Industry Projects

Since the focus of this thesis is on methods for the definition and application of system models in the automotive industry, this chapter answers the application perspective of how model-based systems engineering is utilized in system development projects in the automotive industry using the example of FEV Group. The offered services include engineering consulting, development and testing of powertrain and vehicle systems, design and prototyping, calibration and optimization, and simulation and analysis. It is a globally renowned engineering company specializing in the development and testing of powertrain and vehicle systems, which during the execution times of the considered projects ranked in the top ten engineering service providers in the automotive industry according to the annual sales worldwide measured in million euros [ATZ19, Imh20, Imh21, BW22].

Because FEV focuses on conducting projects in powertrain development, vehicle integration, calibration, controls, and overall vehicle development and cooperates with various international automotive OEMs and suppliers, it offers a broad insight into the working world of the automotive industry, even though a single company cannot provide a complete cross-section of the entire industry. Because of this wide range of services, it is a good starting point for research in this area. As the contents of almost all projects are confidential, this thesis only refers to anonymized results. Nevertheless, to establish examples to explain the methods developed in this thesis, a separate example is introduced in section 4.2 in the context of the domains relevant to the projects, which will serve as a reference in the following chapters. Moreover, chapter 6, chapter 7, and chapter 8 refer to the publicly available system model of a VTOL system from [JGS+20] and [JGW+21] that is not explicitly summarized in this chapter.

4.1 Selected System Development Projects in the Automotive Industry

Based on the overall evaluation of systems engineering methodologies in Table 2.1, this section introduces six reference system development projects FEV carried out or was significantly involved in the execution to evaluate the application and applicability of the methods for application and definition of system models in the automotive indus-

try. All six projects applied the CUBE product specification as presented in Figure 3.2 but performed different variations concerning system decomposition, created views, and modeling paradigm. Therefore, this section analyzes the projects concerning the following three general methodological questions to answer **RQ-2.4** ("How is the system decomposed in the project, which viewpoints are created, and which paradigm is applied?"):

RQ-2.4.1 *How does the project decompose its system under development?*

RQ-2.4.2 *Which views of the system were created in the project?*

RQ-2.4.3 *What development and modeling paradigms were applied in the projects?*

As decomposition, viewpoint, and paradigm classification dimension, this section reuses the categories and classification from Table 2.1 as specified in subsection 2.6.13. Since all projects applied the SysML as modeling language, the table does not include this dimension. For a better distinction between the projects, this section also provides additional project information regarding the system development scope and the customer without providing exact details to maintain confidentiality.

Table 4.1 presents the results of this evaluation. Two projects consider systems in the vehicle, powertrain, and ADAS contexts and address system development projects at Asian and German OEMs. As an exception, the Delta is a publicly funded project conducted with several partners from the automotive and non-automotive industries. Thus, there is a slight overhang to Europe, especially Germany, in the selected development areas. Moreover, the selected projects do not consider America, an essential market and vehicle development area. Alpha and Beta represent traditional document-based development projects, whereas the other four projects represent model-based systems engineering projects. Note that this proportion is rare in the current state of MBSE in the automotive industry, where document-based approaches are still predominantly used in systems development according to the CUBE process as [GKS+21] shows. However, since this thesis focuses on models and model-based development, this selection was deliberately made to compare different aspects of systems engineering in the automotive industry. Since all projects deal with different aspects of system design, the following sections highlight the different project contexts and methodological characteristics.

4.1.1 Alpha: An Example for Document-Based Overall Vehicle Development

In the context of the Alpha project, the project team followed a traditional document-based development process to develop a vehicle with different variants and options based on a common platform following the principles from section 2.4. Even though the project management considered a model-based development approach, the project decided on a document-based approach with a system decomposition based on the physical structure

Table 4.1: Selected industry projects and their decomposition method, viewpoints, and modeling paradigm. ● = followed/created, ◐ = partly followed/created, ○ = not followed/created, ⏲ = ADAS, 🚗 = Vehicle, ⚙ =Powertrain, AS = Asian OEM, EU = European Union, GE = German OEM.

Project Name		Alpha	Beta	Gamma	Delta	Epsilon	Zeta
Project	Scope	🚗	⚙	⏲	⏲	⚙	🚗
	Customer	AS	AS	GE	EU	GE	GE
Decom-position (**RQ-2.4.1**)	UC-Scenario	○	○	◐	○	○	◐
	Functional	○	○	◐	○	○	○
	Logical	◐	●	●	●	●	●
	Physical	●	◐	○	○	◐	○
Viewpoints (**RQ-2.4.2**)	Requirements	●	●	●	●	●	●
	Use Cases	◐	◐	●	●	○	●
	Functional Architecture	○	○	○	○	●	●
	Logical Achitecture	○	○	●	●	○	●
	Physical Architecture	●	●	●	●	●	○
	Realization	○	○	●	●	●	○
	Others	●	●	●	●	●	○
Paradigm (**RQ-2.4.3**)	Functional Modeling (FM)	○	○	○	○	●	●
	Feature-Driven Development (FDD)	○	○	●	○	●	●
	Model-Based Engineering (MBE)	○	○	●	●	●	●
	Model-Driven Development (MDD)	○	○	◐	●	●	●

and knowledge from previous projects. This decision based on the idea, to get the project started on the traditional approach, which most of the project team already knew from previous successful projects, hoping to save the resources required for systems modeling.

Consequently, the engineers only wrote textual requirements in a natural language for systems specification. An additional use-case-based requirement elicitation using the methods presented in [Coc01], and [Fow04], was evaluated for a small excerpt of the system but not rolled out to the overall project. Thus, the vehicle's features relied on the features known from previous development projects and on market analyses for new features. Moreover, the system design does not include the development of a functional or logical architecture. As a result, only a physical architecture based on the known components was developed and used for the system decomposition with three predefined layers for the vehicle. These predefined layers reflected the involved engineering domains and the (physical) components and were directly encoded in a generic BOM as suggested, for example, in [HW91]. As an additional view for verification and validation, the project relied on DVP&Rs for the elements of each decomposition layer to provide means for the verification and validation of the requirements and the evaluation results. Concerning the development paradigms Table 4.1 distinguishes, the project applied none. First, the project did not apply functional modeling. Thus, no functional models that apply this paradigm are considerable. Second, the project did not define features according to Def. 7 and applied as required for FDD. Thus, even though the project defined marketing features such as chrome finishes or an abstract lane assist, these features are not comparable to the features defined in the projects that apply FDD according to this paradigm. Finally, no system models resulted from the specification and design process. Hence, the project neither applied MBE nor MDE during the system development.

Consequently, this project solely provides textual requirements as specification artifacts. Therefore, this project is used for the evaluation of the methods to process textual requirements in the context of the methods provided in chapter 5 and serves as a representative of a legacy project, from which a company might want to carry over results to model-based development projects.

4.1.2 Beta: An Example for Models next to a Textual Specification of a Powertrain

The Beta project focused on developing and optimizing a new powertrain system to achieve best-in-class performance. To this purpose, the project focused on four tasks, which each contribute to the overall project goal:

1. Product Targets: In the first task, the project team elicitated the target characteristics the final powertrain must fulfill to achieve best-in-class performance.
2. Product Specification: In the second task, the project team specified the powertrain based on the elicitated product targets. To this end, the project team

tailored the CUBE process to define the powertrain system's requirements, interfaces, and functions. Additionally, the task involved defining and evaluating optimal powertrain architecture to meet the desired performance targets. In this process, the involved systems engineers spent an additional effort to comply with functional safety (FuSy), which included Function Failure Mode and Effects Analysis (FMEA) [Sta03] and Fault Tree Analysis (FTA) [LGTL85].

3. Simulation, Benchmarking, and Optimization: The project team evaluated different powertrain architectures through simulations. By this, the team aimed to analyze the powertrain performance and compare it against the previously defined product targets. Consequentially, the simulation techniques served to gain insights into the powertrain's behavior, efficiency, and overall performance. Based on the results gained in this step, the project team made incremental adjustments to the specification and architecture.

4. Verification & Validation: Finally, the fourth task involved defining a high-level test concept for the powertrain system. As in the Alpha project, the project team used a DVP&R to outline the necessary tests and procedures to verify that the powertrain meets the defined specifications and targets. By this, the DVP&R ensures that the powertrain underwent exhaustive testing and validation before its implementation in the final product.

To this end, the project applied a tailored variant of CUBE in which a textual system requirement in natural language serves as a system specification, and SysML models serve as architecture description. For decomposition, both the vehicle's logical architecture and the powertrain's physical architecture served as an orientation for the systems engineers. To design the system, the project team created viewpoints for requirements, use cases, and functional, logical, and physical architecture. As in the Alpha project, a DVP&R forms another view for verification and validation not further considered in this thesis.

Even though the project team created architecture models for the simulation and execution of models, this model merely served for testing purposes without interfaces and was not used in the system design and specification. Consequently, the entire specification of the system relies on textual requirements that are enriched with images of partial SysML models without legal obligation. Thus, this project also contributes to evaluating the methods to process textual requirements in the context of the methods provided in chapter 5.

4.1.3 Gamma: An Example for Feature-Driven and Model-Based Systems Engineering for a Driving Function

In the context of the Gamma project, the project team developed a system design and test specification for a partially automated driving function in which the vehicle carries

out all driving tasks in delimited driving scenarios, and the driver only has to take over control of the vehicle in emergency cases. As a function realization, the project developed a system design for the required software functions, sensors, and actuators, a safety concept, and a scenario-based test specification to facilitate system testing.

The project considered different decomposition options, including a use case-based decomposition based on the driving scenarios, a functional decomposition based on the required end-to-end functionalities the vehicle has to perform in these driving scenarios, and a logical decomposition, in which logical components represent combinations of the previous examples. Ultimately, the project decided to follow the logical decomposition, as the additional grouping of use cases to functions to abstract logical groups allowed previous system knowledge and organizational aspects to be considered. In this case, for example, the safety concept included that different teams had to develop different redundant components, which could also mirror the logical architecture, keeping the individual functions as black boxes for the execution identical. This safety concept of handling redundancy brought advantages and drawbacks, as chapter 8 further discusses. As viewpoints, the project created views for requirements, functional architecture, logical architecture, and physical architecture. To this end, another project team's initial set of stakeholder requirements served the project team as input for use case derivation. Based on the elicitated use cases, the project team identified features for which it developed functional architectures in operating principles. The actions performed in the operating principle were assigned to the logical components for the execution and then allocated to the executing element in the physical architecture. In addition, the project team created a document-based safety concept, in which the experts referred to the model elements in the system design model, and the testing team extended the design model with scenario models, in which they combined sequence and block diagrams for the definition and automated derivation of scenario-based test cases. Regarding the modeling paradigms used, the project did applied the functional modeling paradigm, and developed the feature signature and all interfaces from scratch. Regarding the FDD paradigm, the project identified the system features based on the use cases and assigned them to feature teams applying the FDD paradigm. Moreover, the project used MBE as a paradigm during the creation of the system models. Even though the project aimed at making the created models the primary artifacts according to Def. 15, the project only manually used and created all parts of the resulting model. Thus, the project only established parts of the MDE paradigm.

Because of the various ways of decomposing the systems, Table 4.1 compares these possibilities with the remaining projects, because of its limited scope, exploratory nature, and non-application of the model-driven development paradigm, the remaining chapters do not refer to the models created in this project.

4.1.4 Delta: An Example for Scenario-Based Systems Engineering for an Automated Driving Function

The Delta project focused on developing an automated driving function in which the vehicle carries out all driving tasks in delimited driving scenarios. In this project's scope, the potentially hazardous scenario for an autonomous vehicle is in focus, as scenario-based system testing is the primary focus next to the model-based and feature-driven specification of the vehicle and its architecture. As a special case of this project, the software architecture is designed based on these scenarios first to obtain a software-defined vehicle architecture [BMA+23].

In the Delta project, the project team defined a vehicle system and system of systems specification model based on CUBE. As a particular case compared to the previous projects, the objective of the project does not follow the standard feature-based development of the system as presented in [GKS+21], but implemented a scenario-based extension of CUBE, where identifying potentially hazardous scenarios to assess the safety of the intended functionality of the system is in the focus of the project [MSG+22]. In this process, the system modelers decomposed the vehicle according to a logical architecture. To specify the system, the project team created views for textual requirements, use cases, operating principles, functional architecture, logical architecture, and physical architecture. Moreover, for the scenario-based testing, according to [MSG+22], the project team created additional models for the scenario specification. From a modeling paradigm perspective, the project team applied a feature-driven and scenario-based development variation, Table 4.1 considers feature-driven development. Moreover, the project team applied the model-based development paradigm, as they based their product specification on system models. Since the project used the system models as key artifacts of the system specification and further used them in later development steps, using model generation and transformations, the project followed the MDD paradigm.

Because on these various model views, chapter 6, chapter 7, and chapter 8 refer to this project in the evaluation of the applied methods.

4.1.5 Epsilon: Top-Down versus Bottom-Up Model-Creation for a Powertrain Component

In Epsilon, the project team specified an already existing and operational component of the powertrain, developed in a document-based methodology, to fulfill new company-specific specification guidelines. To this end, the project team received the component's document-based specification and product documentation as input and reworked these input documents as a model-based system specification. To this end, one part of the project team followed a top-down approach based on the initial document-based requirement specification, and another pursued a bottom-up approach based on the technical specification. Ultimately, they compared the results of both approaches, leading to the

result that some interfaces and functional inter-dependencies between different components of the realization were not part of the original specification, which tracing the elements from the technical architecture to the stakeholder requirements revealed.

To create this system model, the project team decomposed the system logically based on a solution-neutral functional component description and several technical constraints to create a logical reference architecture for future systems developed in this area. By this, the system decomposition deviates from the previously developed system decomposition based on the physical architecture, which was considered a possible take-over for the decomposition of the system architecture but was quickly discarded to realize solution-independent architecture for future projects. As system views the project provided views for requirements, use cases, logical architecture, physical architecture, and realization of the system. The project team integrated the initial stakeholder requirements as natural language requirements into the model to create a requirements view. It derived additional natural language requirements from the system models where the enterprise policy required it. The use cases, functional architecture, and logical architecture were then modeled top-down according to the CUBE process as presented in Figure 3.2. To recreate the current technical architecture, the project team also created a bottom-up architecture based on the realization described in the technical documentation. The original product documentation referred to a model-based software development project in which the production code was generated from the links to the elements of this model and integrated into the system specification model. Finally, the project team mapped the elements of the top-down created architecture to the elements of the bottom-up created technical architecture, revealing that the technical architecture and realization models contained elements that could not be mapped or even explained by the model derived in the top-down by the original specification. As a result of this project, a follow-up project addressed these issues by finding out which realization and specification had to be adapted. Regarding the modeling paradigms used, the projects applied the functional modeling paradigm to identify the system features and the function and logical component interfaces. Moreover, the project applied the feature-driven development paradigm described in subsection 2.6.13 to develop the system model as a set of reusable features. As the model-based specification was the project's primary goal, the project applied the MBE paradigm. Moreover, the project aimed at making the created models the primary artifacts according to Def. 15 and applied the MDE paradigm.

Since this project uniquely combines a top-down development approach with a bottom-up approach based on reverse engineering, it is the representative project to evaluate the comparability between these approaches. Moreover, it is well suited and applied to determine whether the results from a model-driven specification comparable to document-based specifications (**RQ-4.4**) in the evaluations of chapter 6, chapter 7, and chapter 8.

4.1.6 Zeta: An Example for Model-Driven Engineering of an Autonomous Vehicle

In the Zeta project, the project team developed a system design for an autonomous vehicle, including textual requirements, system use cases, operating principles, and logical architecture. Instead of considering all the vehicle's features, the considered project phase only focused on the 16 features, which an evaluation performed before the project conducted as the most critical features for the vehicle's success. As an initial phase of the overall vehicle development process, the project only focused on the creation of a logical architecture for the following project phases without specifying a physical architecture. Hence, this project does not cover the creation and consideration of the physical architecture. Because the project followed a demanding schedule, the team decided to automate many modeling steps to reduce the effort required for system modeling.

The project team consisted of a project manager, who, as a domain expert, also modeled 3 features, and 4 systems engineers with different experience levels responsible for the modeling tasks leading to 5. Additionally, the author of this thesis led a team of 2 responsible for implementing the tooling required for the model-driven development of the vehicle features and architecture. Finally, the team was supported by a CUBE method expert to discuss modeling and method questions. In parallel, another team developed the LRA consisting of 6 vehicle systems and 6 subsystems of these systems. To perform the necessary tasks, another project team already extracted the legal requirements for the system features from the relevant legislation in a previous project phase. Moreover, the customer provided initial usage scenarios derived from a mission analysis, which served as a starting point for the modeling team.

During the system design process, the project team decomposed the system logically and created viewpoints for textual requirements, system use cases, operating principles, and logical architecture. Moreover, to hold a tight schedule, the project team followed the functional modeling paradigm and a feature-driven development, model-based engineering, and model-driven engineering paradigm. To this end, the project applied most of the methods this thesis presents in the following chapters. Instead of writing natural language requirements as investigated in chapter 5, the project decided to model stakeholder values as use cases as described in chapter 6 and used them to automatically derive operating principles from use case scenario templates as described in chapter 7. Finally, these operating principles generated the chapter 8 by using the interfaces in the operating principles defined according to the functional modeling paradigm. To explain the model to inexperienced stakeholders as natural language requirements as described in chapter 5, the project team developed a method to specify the named elements of the model so that they are usable to generate natural language requirements from the model.

4.2 An Automotive Example

Because the contents of most considered projects are confidential, this thesis uses a project-independent running example inspired by the context and use case diagram from [FMS14a] and provides new ideas on the vehicle decomposition and the specification of the drive vehicle use case.

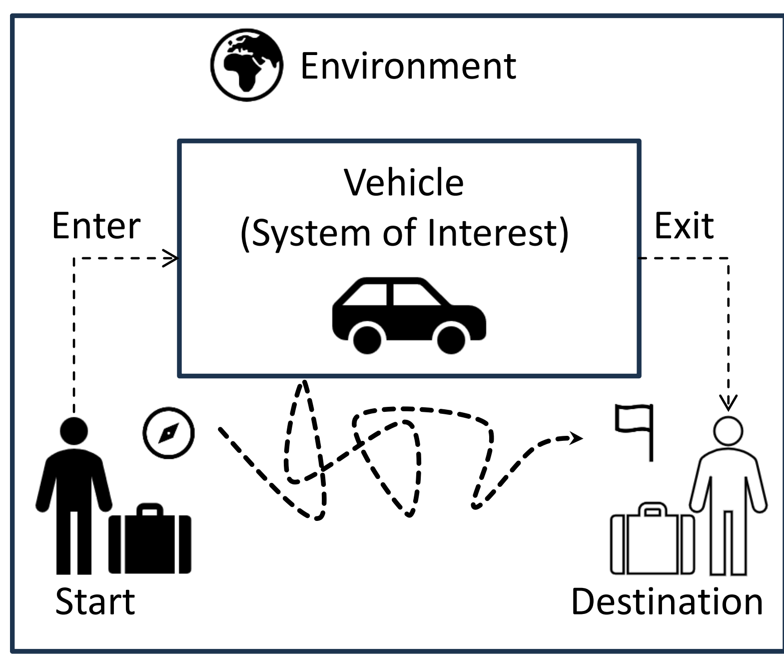


Figure 4.1: The automotive transportation scenario based on the simple automotive model in [FMS14a].

In this example, a simple vehicle model serves as a starting point to develop a simple driving feature, which transports persons and goods from a start to a destination. For a better understanding, Figure 4.1 informal depiction of this setup and task. The vehicle is the *system of interest*, which separates from the *environment*. As an actor, the *vehicle occupant* interacts with the vehicle as a means of transportation. To this end, the vehicle occupant enters and exits the vehicle, which requires the vehicle to have operable doors. Inside the vehicle, occupants demand several forms of accommodation, which the vehicle accessory provides. To regulate these accessories, the vehicle occupants want to control the accessories *i.e.,* they want to operate the climate control and the entertainment system. As an implementation of the transportation task, the vehicle requires a particular type of vehicle occupant, the *driver*, which drives the vehicle as another vehicle use case. As an explicit differentiation of the driver from the other occupants that do not drive the vehicle, the *passenger* is another specialization of the vehicle occupant that cannot

drive the vehicle.

The execution of the vehicle transportation task requires several steps. First, people must enter the vehicle to become occupants. Then, the driver must drive the vehicle to the desired destination, at which point the people who entered the vehicle exit the vehicle to finish the transportation task. While driving, the driver must be able to assess the vehicle's environment and driving state and give the vehicle the necessary control commands. In this example, it is assumed that the vehicle has steering, acceleration, braking, and different gears to ensure control.

To propel the vehicle, it is further assumed that the vehicle has a propulsion system to provide the drive energy to accelerate the vehicle, a steering system to implement the steering commands, a structural system to ensure the mechanical structure, passenger accommodation, and stability of the vehicle, an energy system to provide the required energy for all systems, and finally, an infotainment system to provide the amenities.

To specify this vehicle and transportation task using a model-driven systems engineering methodology, the following chapters address these aspects. First, chapter 5 investigate how two example document-based systems engineering projects developed vehicles in general and which problems are present in the legacy specifications of these systems. Then, subsection 6.1.2 specifies the stakeholder value using SysML use case diagrams and presents the feature split for the operating principle development according to the CUBE procedure as section 3.1 presented. As an implementation of this operating principle, subsection 7.1.3 presents the activity diagrams that describe the required actions to transport people. Having this operating principle subsection 8.2.3 presents the resulting logical architecture.

4.3 Interim Summary: How are Systems Engineering Projects Conducted in the Automotive Industry?

Based on the previously presented projects and the classification presented in Table 4.1, the answers for **RQ-2.4** ("How is the system decomposed in the project, which viewpoints are created, and which paradigm is applied?") are:

1. First, most considered projects follow a logical decomposition approach regarding how the project decomposes its system under development. However, one exception was becoming evident in the case of the Alpha project, which follows the system's physical decomposition. To differentiate between these forms of decomposition, the comparison chapter 5 presents a comparison between both ways of system decomposition.

2. Second, addressing the views created for the system in each considered project, the results from Table 4.1 indicate that the specific views vary depending on the

project scope. However, since all projects follow the CUBE methodology, all views are related to those presented in chapter 3.

3. Third, when considering the development and modeling paradigms the considered projects use, it is essential to note that all the projects applied a combination of various paradigms resulting from selecting the projects. Apart from this obvious observation that results from the selection, this indicates that the CUBE methodology is a flexible and adaptable approach with the potential to tailor the methodology to the specific needs of various projects.

To further validate the findings, the following chapters present and ensure the conclusions' robustness. This section has already addressed some potential threats to validity. Overall, the analysis of the projects presented in Table 4.1 provides valuable insights into the system development approaches employed and sets the stage for comprehensively exploring the subject matter in the following sections. By nature, however, these projects only cover a comparably small selection of projects conducted at the same engineering service provider, which might not represent all projects conducted in the automotive industry. As mentioned, the selection of projects addressed this issue by involving multiple OEMs as contractors in different regions. Additionally, the projects included in the analysis were selected based on the application of CUBE as systems engineering methodology, which could introduce a selection bias compared to a project selection, which also includes system design projects that do not use CUBE as systems engineering methodology. At least for related methodologies such as SMArDT or SPES, it is probable that similar observations are possible, as these methodologies suggest similar views, decomposition methods, and paradigms. Another potential threat lies in the availability and accuracy of the project data. As the analysis relies heavily on the information provided in the projects, any inconsistencies or missing data could affect the validity of the conclusions. Therefore, all analyses presented in the following chapters focus on defined baselines of the project, which might be a partial state of the system development for better comparability of the projects. Finally, the interpretation of the results is subjective. Since different project members may have varying perspectives on the categorization and classification of the projects, this could introduce a potential bias. Therefore, the results were additionally aligned between a team of parties involved in the project to have a similar understanding of the presentation made in Table 2.1.

Based on the investigation of the created views, the created views primarily focused on textual requirements, operating principles, and logical architectures. These views form the foundation for the subsequent chapters, which further investigate the system analysis and design of automotive systems.

Chapter 5

Modeling Textual Requirements using a Requirement DSL

As stated in [Bro06], vehicle OEMs typically perform the majority of the requirements engineering activities in the automotive industry, to which suppliers later add only small parts as an enhancement. In this process, textual requirements written in natural language are arguably still essential for the specification of automotive systems. As a result, studies such as [LTK19] show that textual requirements are the norm in many areas of the automotive industry, and only a few models exist. Even though [HP00] early identified natural language requirements as insufficient and [Bro06] highlights requirements engineering as a significant challenge in the automotive industry, concurrent systems engineering projects such as Alpha and Beta, still rely on natural language requirements for system specification. Relying on natural language requirements is not only a common practice in automotive requirements engineering but also becomes evident when looking at general software requirements used in different contexts [LMP04].

Consequently, methods for defining and applying system models in the automotive industry must deal to some extent with legacy requirements in natural language and a broad acceptance of these kinds of requirements and experience among the involved engineers. Moreover, some related works in developing system architectures and system requirements engineering concluded that (stakeholder and system) requirements and architecture specifications are equal and intertwined [Nus01]. Therefore, they conclude that considering both together is somewhat beneficial for the development, which manifests in the twin-peaks model [Nus01]. From the same community, other authors argue that this can enhance the traceability between concrete usage scenarios and the system architecture created in a business process [HBG15] or that a formalization of the requirements could bridge the gap between (textual) requirements and architecture [RTP+14]. In contrast to this approach, other authors argue that requirements and architecture design can and should not be separated [PDKvK02] and suggest a combination of these concerns as problem frames [HJL+02], or express the concern that especially in the automotive industry requirements engineering is not always performed before design and architecture in the system development process [EHKP15]. Therefore, systems engineers who aim to apply methods for the design of automotive systems and their architecture

must not only be aware that textual requirements (separated to architectural design models) might exist but also keep in mind that these requirements are not necessarily correct, aligned with all stakeholders, or complete.

As a result, other authors propose using models during the requirements engineering process. The SPES methodology [PHAB12, PBDH16], which is also a basis of CUBE as presented in chapter 3, suggests, for example, the use of models during all development phases [LTK19]. By this, the SPES framework complies with existing standards, such as ISO/IEC 15288 [ISO15] and ISO/IEC 12207 [ISO17a], as shown in [BHH+14]. While the following chapters handle this aspect in more detail, this chapter solely focuses on integrating natural language requirements into the definition and application of system models in the automotive industry and whether this integration benefits the system model.

Therefore, this thesis focuses on to which extent legacy requirements are integrateable into system architecture models. One of the main challenges in this regard is that natural languages are neither context-free [PG82, Shi85], nor regular [Cho57] in the Chomsky hierarchy of formal languages [CS59] and therefore until today no correct and complete formal grammar for all natural language could be provided [JR12]. Although the natural language formulation, according to this definition, seems to dominate the requirement specification in the automotive industry, making automated processing with traditional grammar-based parsing techniques impossible, many requirements that look like natural language, at first sight rely on some formal or semi-formal requirement descriptions. As a result, one could encode these sentence structures in the form of a DSL, as, for example, practiced in [DHH+20] as a solution for scheduling shows in a television program. To further investigate this possibility, section 5.2 investigates sentence pattern applied in the automotive industry and examines the particular sentence pattern applied in the evaluated Alpha and Beta project. Based on these insights, section 5.3 presents a DSL for this sentence pattern before section 5.4 evaluates the applicability of this DSL in the context of legacy requirements from the Alpha and Beta projects. Finally, section 5.5 summarizes the results concerning the research questions and derives lessons learned from this application to system requirements from the automotive industry.

5.1 Requirement Diagrams for Natural Language Requirement Modeling

To visualize natural language requirements diagrammatically, the SysML provides the requirements diagram in [Obj19]. For the use in this thesis, Table 5.1 presents a subset of the SysML specification [Obj19] tailored to the application in the context of the integration of natural language requirements into the other SysML models this thesis presents. With this definition, this section provides the required elements in the SysML and contributes to answering **RQ-3.1** (“How can requirements be expressed?”).

Table 5.1: Requirement diagram elements required to describe natural language requirements according to the CUBE methodology.

Element Name	Notation	Description
	Obligatory Diagram Elements	
Requirement	«requirement» **Requirement Name** id = "ID" text = "Requirement"	A SysML requirements element consists of three obligatory parts—first, the diagram name, which names the element for a more straightforward recognition. Second, a unique identifier is an attribute of the requirements element that differentiates the element from the other elements in the model. Third, the requirement text contains the natural language requirement as a string.
Derive Relationship	«deriveReqt» «requirement» **Derived Requirement**	The derive requirement relationship enables requirements engineers to derive one or more requirements from another SysML element, which is not necessarily a requirements element. According to [Obj19], a «deriveReqt» relationship describes a dependency between two requirements in which a client requirement can be derived from the supplier requirement. Consequently, the relationship just creates a dependency between those two elements without any further precisely defined semantics.

With these two elements, Table 5.1 provides the foundations to diagrammatically integrate natural language requirements into SysML models. By simply adding the text into boxes, together with an identifier and a name, the SysML provides a low-key method to integrate texts into the model and to relate/trace them to other elements using the derive requirements relationship. As this relation does not have a precise

semantics specification in [Obj19], this only describes that the client requirement or element is derivable from the supplier element without exactly specifying what this derivation means in this context and which intellectual or logical effort is required for this derivation. Consequently, this diagram is easy to use, but has the drawback that without any additional method, requirements engineers may put arbitrary, contradicting, or unrelated texts into these boxes, resulting in no additional value to the diagram compared to comments in notes. Therefore, this chapter addresses this challenge by providing a method to formulate the requirement in the requirement elements. As unique identifiers and names provide no considerable value in the context of the method this thesis provides, these elements are not addressed any further, as the name only serves to summarize the requirements. Many SysML tools already provide methods to set the identifier automatically.

5.2 Sentence Patterns for Requirement Formulation

Requirement DSLs enable the formalization of natural language requirements. For example, the graphical formalization language Simplified Universal Pattern [TBH16] enables requirements engineers to formulate safety-critical requirements in trigger action pairs, which has shown in an automotive application example that a large proportion of the requirements can be formalized with this context [BBB+18]. Moreover, [MSL15] proposes a specification language for the specification of automotive systems. To this end, [MSL15] specifies a DSL as a context-free grammar and an ontology to define the axiomatic terms and types to allow including typed terms within the requirements [BBK+22b, BBK+22a]. To further enhance the application area of this DSL, [MSL16] introduces a toolchain for requirement consistency checking based on SAT-solving.

Furthermore, the authors in [KC05] and [YCC15] provide methods for similar interdependence and consistency checks for natural language requirements but in the form of semi-formal requirements formulated in a requirement pattern, or so-called requirement boilerplates. Instead of providing a fully processable context-free grammar for parsing, these approaches aim at restricting the requirements engineers to simple sentence structures with a limited structural scope in terms of grammatical sentence structure and used vocabulary without defining a formal syntax or semantics for the language. For example, [EVFM16] proposes an approach to use sentence patterns to improve the completeness of performance requirements, for which [EVF16] provides an approach for quality requirements. Likewise, [KC05] provides a pattern for real-time specifications, which [PMHP12] extends to requirement patterns for automotive behavioral requirements in a Case Study conducted at BOSCH. Moreover, [FMK+11] provides a pattern and an ontology for embedded system requirements, whereas [MWHN09] provides a general requirements syntax pattern. Apart from this company and industry-specific template, the MASTER template based on [PR21] that is available in German and English. Especially in specific

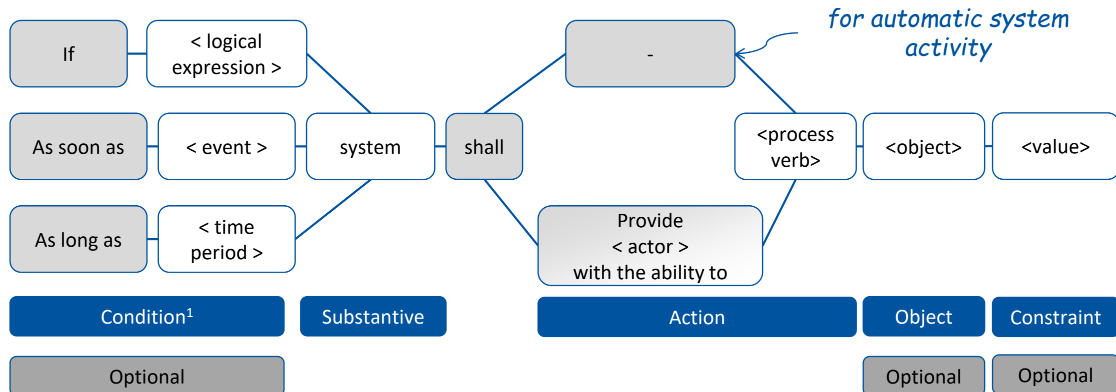
German-speaking countries, this template is also widely used [BBK+22b]. Instead of using arbitrary natural language, the investigated projects Alpha and Beta use a similar form of structured natural language in the form of a requirement template related to the one presented in [PR21].

Because sentence templates provide only a rough framework for requirement formulation and do not provide precisely defined syntax and semantics without further refinement, verifying compliance with a template is often tricky. Even though this informal definition brings the advantage that requirements engineers and template designers are not limited to formally defined sentence structures, this approach has the significant disadvantage that implementations of automated requirements processing cannot assume correct phrasing of the input [BBK+22b]. Therefore, several approaches in related works focused on extracting requirements in sentence templates by refining the original template. For instance, [EVF16] presents multiple sentence templates to specify quality requirements, and [FPQ+10] introduces a meta-model for software requirements. Moreover, a catalog of 29 QR patterns [RMBFQ09] and 37 non-functional patterns [PQF+12] serves as a basis of the PABRE framework [FQR+13], a method for using requirements templates. To identify such a DSL in unstructured requirements, [BBK+22b] and [BBK+22a] cluster unstructured textual requirements in natural language to detect formulation patterns in these requirements. As the requirements in the considered industry projects Alpha and Beta follow an already known pattern, this thesis presents a different approach by directly developing a DSL for these patterns. To this end, the following section first presents the sentence pattern in more detail and then presents the DSL usd for formalizing and processing the requirements.

5.2.1 A Reduced Pattern for Functional Requirements

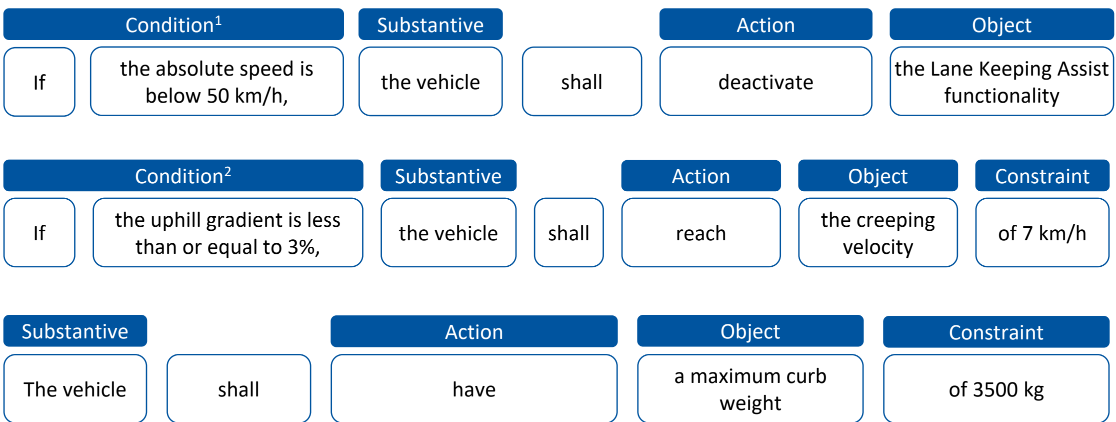
To ensure the consistency and the quality of the requirements written in the requirements engineering processes, the Alpha and Beta projects apply the so-called Specification guideline for Requirements Engineering (SPECTRE) to ensure consistency and quality of the requirements and requirement specification artifacts. To this end, the guideline contains, among other guides, the specification template Figure 5.1 presents. By providing this formulation template, SPECTRE aims to achieve a uniform appearance and quality of the requirements written by the company's employees. In addition to the template from Figure 5.1, the requirements engineers also received additional examples from Figure 5.2.

To write these requirements, SPECTRE defines the main characteristics for requirements specification. Hence, it sets targets for requirements specification and by defining practical rules, *e.g.,* in the form of the template Figure 5.1 presents and the use of specific additional requirement attributes, which this thesis does not consider. To ensure the requirement statements have the main characteristics SPECTRE defines, sentence templates/patterns for different requirement types. According to the template Figure 5.1



1 – It is also possible to attach the condition at the end of the sentence, as long as readability and understanding do not suffer

Figure 5.1: The Specification guideline for Requirements Engineering (SPECTRE) template used in the Alpha and Beta projects based on [PR21].



The examples are primarily intended to showcase the flexibility the template provides in formulating requirements.
1 – The requirement can also be formulated as… "Below the speed of 50 km/h, the vehicle shall deactivate…"
1,2 – It is also possible to attach the condition at the end of the sentence, as long as readability and understanding do not suffer

Figure 5.2: Additional examples for the application of the Specification guideline for Requirements Engineering (SPECTRE) template that the requirements engineers in the Alpha and Beta projects received.

presents, all requirements have a substantive and an action to express that the system, which is the substantive of each sentence, performs some action. This action is either a process verb *i.e.*, the system performs this action by itself or provides an actor with the ability to perform it. Moreover, all process verbs must appear in combination with the

modal verb "shall" to express the obligatory nature of the requirement. For example, a vehicle as a system of interest could either drive (by itself) or provide an actor driver with the ability to drive, which leads to the requirements first requirement that a vehicle shall drive:

Example 1 (SPECTRE Requirement I)**.** *The vehicle shall drive.*

Furthermore, the second refined requirement that the vehicle shall provide the driver with the ability to drive:

Example 2 (SPECTRE Requirement II)**.** *The vehicle shall provide the driver with the ability to drive.*

If a requirements engineer aims to refine the action further, optional objects in the template provide a mechanism to refine this aspect further. If the author of Example 2, for instance, wants to highlight that drive fevers to the vehicle as the system of interest, it is possible to write:

Example 3 (SPECTRE Requirement III)**.** *The vehicle shall provide the driver with the ability to drive the vehicle.*

Moreover, requirements engineers that use the SPECTRE template can specify conditions at the beginning and the end of each sentence and could, for example, write the following requirement to specify that driving the vehicle shall only be possible if the vehicle previously established the ability to drive:

Example 4 (SPECTRE Requirement IV)**.** *If the vehicle has established the ability to drive, the vehicle shall provide the driver with the ability to drive the vehicle.*

To express the occurrence of events, the requirements engineer can use the key phrase "as soon as", for example, by specifying:

Example 5 (SPECTRE Requirement V)**.** *As soon as the driver starts the engine, the vehicle shall provide the driver with the ability to drive the vehicle.*

Furthermore, the key phrase "as long as" could specify a time

Example 6 (SPECTRE Requirement VI)**.** *As long as the engine is running, the vehicle shall provide the driver with the ability to drive the vehicle.*

Finally, optional constraints provide further means to the requirements engineer to restrict the application of process verb *e.g.,* by writing the following requirement to specify that the only intended operational environment is an operation on public roads:

Example 7 (SPECTRE Requirement VI)**.** *As long as the engine is running, the vehicle shall provide the driver with the ability to drive the vehicle on public roads.*

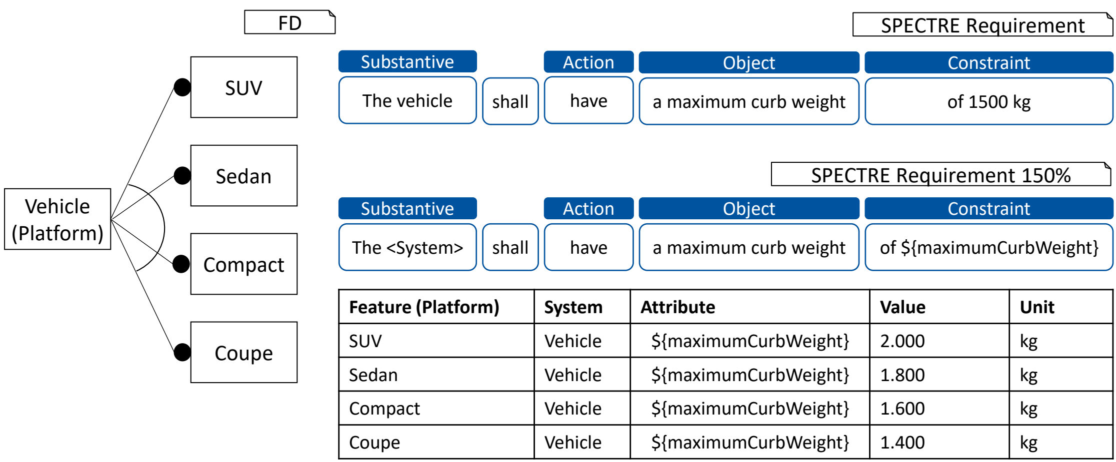
In addition to restricting functionalities expressed by the process verb, the condition, and the object are also applicable to express that the system shall have a specific characteristic or quality. For example, by writing the requirement

Example 8 (SPECTRE Requirement VIII)**.** *The vehicle shall have a maximum curb weight of 3500 kg.*

Because this requirement type also allows integration of system attributes and variability points, we developed additional techniques for applying these quality requirements in the Alpha and Beta projects.

5.2.2 A Reduced Pattern for Variability in Quality Requirements

As Example 8 shows, the SPECTRE language template from Figure 5.1 is applicable for quality as well as functional requirements and constraints. To express variability and reuse the requirements as modular requirements for multiple products in a platform (*cf.* section 2.4), the requirements engineers from the Alpha and Beta projects decided to enhance the requirements boilerplate for quality requirements according to the template Figure 5.3 presents.



Feature (Platform)	System	Attribute	Value	Unit
SUV	Vehicle	${maximumCurbWeight}	2.000	kg
Sedan	Vehicle	${maximumCurbWeight}	1.800	kg
Compact	Vehicle	${maximumCurbWeight}	1.600	kg
Coupe	Vehicle	${maximumCurbWeight}	1.400	kg

Figure 5.3: Additional applications of the Specification guideline for Requirements Engineering (SPECTRE) template for requirements engineers to integrate variability information in the Alpha and Beta projects.

To this end, the requirements engineers extended the initial template from Figure 5.1 with variables to define attribute values according to different variability models encoded in a feature model. If a vehicle platform is used as a basis for four exclusive vehicle types (*e.g.,* SUV, Sedan, Compact, and Coupe) a requirements engineer without this mechanism would have to write the following four requirements instead of writing the four requirements for different variables as follows,

Example 9 (SPECTRE Requirement IX)**.** *1. The vehicle (SUV) shall have a maximum curb weight of 2,000 kg.*
2. The vehicle (Sedan) shall have a maximum curb weight of 1,800 kg.
3. The vehicle (Compact) shall have a maximum curb weight of 1,600 kg.
4. The vehicle (Coupe) shall have a maximum curb weight of 1,400 kg.

Whereas a requirements engineer with this mechanism could only write the following requirement and specify the individual information in an attribute table (*cf.* Figure 5.3):

Example 10 (SPECTRE 150% Quality Requirements I)**.** *The <System> shall have a maximum curb weight of* ${maximumCurbWeight}

By this, a drastic reduction of written requirements is possible at the cost of maintaining an additional artifact that handles the variability-specific values of the requirement. Using this mechanism not only the attribute ${maximumCurbWeight}, but also the system <System> is a variation point for multiple requirements. Thus, the same requirement for one system could easily apply to multiple associated systems, *e.g.,* if a requirement should be valid for a previously known set of systems. While the first concept as shown in 10 was heavily used in the Alpha and Beta projects, the second concept was not implemented in the projects, as the requirements engineers wrote and managed the requirements per system and feared to lose track of the associated requirements if only one requirement is maintained for multiple systems. Consequently, this concept was not integrated into the DSL the following section develops.

5.3 A Requirements DSL for the Automotive Industry

To implement a DSL that processes these patterns, we developed and evaluated the SPECTRE DSL with a prototype implementation based on [Kat21], which processes these patterns using the language workbench MontiCore [Kra10, KRV10, HKR21]. As a foundation of this work, the thesis in [Kat21] developed eleven MontiCore grammars to parse different parts of the requirements, which are adapted slightly and presented fully in Appendix B and briefly summarized in this section. Furthermore, the work developed five context conditions to check some quality aspects of the requirements and a pretty printer to achieve a uniform appearance of the already parsed requirements. Therefore, the sentence template from Figure 5.1 was subdivided into parts for the condition, substantive of the sentence (*i.e.,* the system described in this requirement), the action, object, and constraint as Figure 5.4 shows. Based on this general structure, [Kat21] developed a further subdivision of the grammar based on the single components of the language. Therefore, this original structure served as a reference, which further evolved to the final structure Figure 5.5 presents. For increased reusability of the language components for different project structures, eleven grammars from the basis of the language for which Figure 5.5 depicts the relationship between these grammars. Since natural

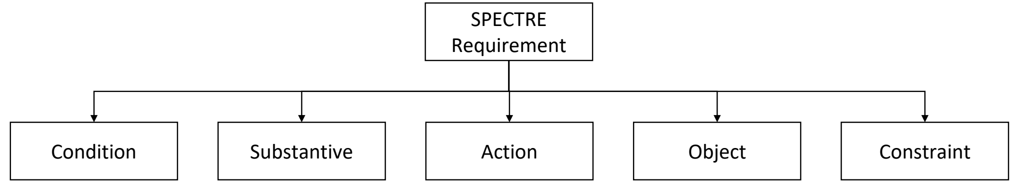


Figure 5.4: Basic sentence structure that formed the basis for grammar development.

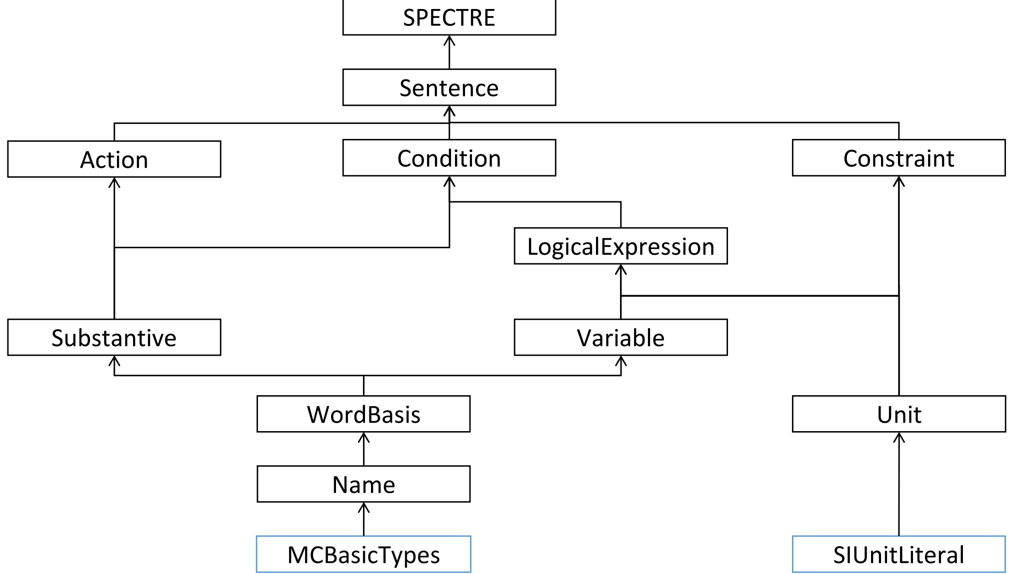


Figure 5.5: Final grammatical structure that forms the basis for the SPECTRE-DSL.

language texts typically combine multiple sentences, the SPECTRE-DSL grammar consists of a construction rule for texts that consist of an arbitrary number of sentences formulated according to the SPECTRE requirements boilerplate.

```
1 SpectreText = (SpectreSentence ".")* (SpectreSentence "."?);
```
MCG

Listing 5.1: The SpectreText grammar based on [Kat21].

As the high reusability of the language components for different project structures is beneficial for additional applications, the final language draws on eleven grammars structured as shown in Figure 5.5. Since natural language texts typically combine multiple sentences, the SPECTRE-DSL grammar consists of a construction rule for texts that consist of an arbitrary number of sentences formulated according to the SPECTRE requirements boilerplate.

```
1 interface SpectreSentence = SpectreMainSentence;                MCG
2 SpectreConditionFirstSentence implements SpectreSentence
3   = SpectreCondition "," SpectreMainSentence;
4 SpectreConditionLastSentence implements SpectreSentence
5   = SpectreMainSentence SpectreCondition;
6 SpectreNoConditionSentence implements SpectreSentence
7   = SpectreMainSentence;
```

Listing 5.2: SpectreSentence grammar based on [Kat21]

The main sentence rule then further constructs the main parts of the requirement. It starts with the non-terminal for the subject followed by a modal verb, which defines the legal obligation of the action, followed by an optional substantive and constraint. By this, the grammar construction follows the principles of LR grammar construction [Tom84] using additional non-terminals called preterminals, which correlate to the actual terminal defined in a separate place. Here, the construction rule delegates this to the `WordBasis` grammar, which handles these special non-terminals. The use of preterminals not only simplifies the language definition but also enables the language designer to easily extend the grammar later, as only one subgrammar that contains the dictionary of allowed non-terminals needs to be adapted. Thus, if future projects require additional concepts, such as additional modal verbs for different legal obligations, as mentioned in [PR21], only one grammar rule needs to be adapted.

```
1 SpectreMainSentence                                              MCG
2   = subject:SpectreSubstantives SpectreModalVerb SpectreNot?
3   SpectreAction SpectreSubstantives? SpectreConstraint?;
```

Listing 5.3: The SpectreMainSentence grammar based on [Kat21].

The action construction rule consists of construction rules for the predicates the subject or substantive of the sentence (*i.e.*, the system of interest) shall do. It starts with referring to the non-terminal that encodes the verbal phrase used for encoding that the system shall "provide a <actor> with the ability to". Again, the grammar uses preterminals to delegate the non-terminal definition to the `WordBasis` grammar.

```
1 interface SpectreVerbalPhrase;                                   MCG
2 SpectreProvidePhrase implements SpectreVerbalPhrase
3   = SpectreProvide SpectreSubstantive SpectreWithTheAbilityTo;
```

Listing 5.4: SpectreVerbalPhrase grammar based on [Kat21]

Since the original informal boilerplate (*cf.* Figure 5.1) marks this part as optional, its usage before the process verbs is also optional before the respective non-terminal in the grammar. Moreover, since a process verb can combine one or more names, the construction rule uses preterminals and refers to the SpecrteName non-terminal from the WordBasis grammar.

```
1 SpectreName = (name:Name | name:SpectreToken);
```
MCG

Listing 5.5: SpectreName grammar based on [Kat21]

Next, the substantive grammar follows the following structure. First, the following listing defines multiple alternatives for one or more substantives. Since the system and object definitions are similar, multiple substantives might occur, such as allowing a requirements engineer to define a requirement for more than one system by a list of systems. Hence, Boolean operators are allowed between the systems to define whether the requirement applies for one, both, or not for a system. Because of this, the following rule encodes these types.

```
1 SpectreSubstantives = ((SpectreSubstantive ",")?
2   SpectreSubstantive SpectreBooleanOperator)? SpectreSubstantive
```
MCG

Listing 5.6: The SpectreSubstantives grammar.

Based on this rule for defining one or more substantives, the following rule defines the rule for a single substantive used in combination with the previous rule.

```
1 symbol SpectreSubstantive
2   = SpectreQuantifier? SpectreArticle? SpectreName+
3   (SpectreOf SpectreSubstantive)? SpectreMarketingName?
4   SpectreComment?;
```
MCG

Listing 5.7: The SpectreSubstantive grammar based on [Kat21].

According to this non-terminal, a substantive consists of an optional qualifier, an optional article, and one or more words that define a named element in natural language, followed by optional relations to other substantives, marketing names, and comments, which were project-specific additions to the original boilerplate the Alpha project used as an extension.

The SpectreConstraint interface provides several ways to specify conditions as a part of a sentence in the DSL to express the optional constraint at the beginning or end of each sentence. The Alpha and Beta projects used three variants of the constraint.

```
1 interface SpectreConstraint;
```

MCG

Listing 5.8: SpectreConstraint grammar based on [Kat21]

First, where the requirements engineer specifies a unit or a variable as a constraint. Second, the constraint compares two variables, and third, where the constraint specifies a maximum or minimum value of the object in the sentence. For further information, Appendix B provides all project-specific implementations of this interface.

```
1 interface SpectreCondition;
```

MCG

Listing 5.9: The SpectreCondition grammar based on [Kat21].

Finally, the `SpectreCondition` interface, as Listing 5.9 presents, provides the framework for defining the condition as part of the general SPECTRE sentence boilerplate. Since the interface alone provides no implementation, [Kat21] developed several specifications for defining conditions. First, an object or action can have a constraint that specifies that reaching a specific quality is required *i.e.,* the obligation to reach a specific unit or variable. For example, the requirements engineer could specify constraints such that a vehicle should have a maximum curb weight "of 3,500 kg". Since it would also be possible to constraint that a specific vehicle characteristic shall have or be a specific value, a comparison without the "of" is also possible. For example, a requirements engineer could specify that the "vehicle maximum curb weight weight shall be 3,500 kg.". Therefore, the following rules serve to combine both cases. Second,

```
1 SpectreNoWordConstraint implements SpectreConstraint
2   = (SpectreUnit | SpectreVariable);
3 SpectreOfConstraint implements SpectreConstraint
4   = SpectreOf (SpectreUnit | SpectreVariable);
```

MCG

Listing 5.10: SpectreNoWordConstraintAndSpectreOfConstraint grammar based on [Kat21]

instead of using "of" to assign a specific value to an object or characteristic, a comparison operation such as "lower", "greater", or "less or equal" can specify a boundary as a constraint. To achieve this aim, the Listing 5.10 provides the respective concept. Third, and finally, a third way of expressing boundaries was defined for the application, which is a particular case of the second case. In this case, instead of defining a boundary in a comparing style, an upper or lower bound, the requirements engineer specifies upper and lower bounds explicitly, for example, by writing "a maximum of" or "a minimum" of as expressed in Listing 5.12.

```
1 SpectreComparisonOperatorConstraint                                    MCG
2       implements SpectreConstraint
3   = SpectreComparisonOperator (SpectreUnit | SpectreVariable);
```

Listing 5.11: SpectreComparisonOperatorConstraint grammar based on [Kat21].

```
1 SpectreWithAMinimumOfConstraint implements SpectreConstraint           MCG
2   = SpectreWithAMinimumOf (SpectreUnit | SpectreVariable);
3 SpectreWithAMaximumOfConstraint implements SpectreConstraint
4   = SpectreWithAMaximumOf (SpectreUnit | SpectreVariable);
```

Listing 5.12: The SpectreWithAMinimumAndMaximumOfConstraint grammar based on [Kat21].

Since none of these rules contains terminals, all non-terminals not defined in the previous grammar examples are specified in the `WordBasis` grammar. By this, all adoptions to the concrete implementations of the grammar are in a single source, which facilitates the reuse of the language definition in different projects without the necessity to adapt all grammar rules. These grammar rules introduce the primary contents required for the research questions discussed in this section. For further reference, Appendix B provides the complete grammar used for the evaluation based on a previous study conducted in [Kat21]. The presented grammar results from an iteration tested on all requirements of the Alpha and Zeta projects. Since natural language is not generally processable, as described in [JR12], it is not unrealistic to assume that the SPECTRE template from Figure 5.1 allows creating requirements that are not processible by the parser MontiCore generates. As the following section will show, the projects Alpha and Zeta contained these examples, and section 5.4 will discuss these threats in the context of these findings.

To additionally check the validity of the parsed requirements, [Kat21] developed five context conditions to cover the essential cases for the evaluation of the requirements used in the Alpha and Beta projects. Note that not all requirements and validity rules, as specified in [PR21], are included in the implementation, but only the most essential for a first evaluation of the requirements written in the project. As a first rule, the `DoesContainShallCoCo` checks whether the requirement contains "shall" as a modal verb and throws a warning if the requirement consists of another model verb specified in the `WordBasis`. Thus, the requirement "The vehicle must deactivate the Lane Keeping Assist functionality if the absolute speed is below 50 km/h." would be parsable but not conform to this context condition. As a particular case of this rule, other modal verbs apart from "must" and "shall" such as "might" are detected by the `DoesContainShallOrMustCoCo` condition, which causes an error to ensure that the requirements engineer uses a modal verb with some form of legal obligation according

to [PR21]. Both context conditions are required, as the grammar construction rules intentionally allowed other modal verbs in the construction, as references other than the SPECTRE, such as [PR21], allow these constructions to differentiate between different legal obligations. Thus, other modal verbs were allowed in the first place and later filtered using this context condition to detect whether the requirements engineers used these differentiations in their requirements. Since the same sources also state requirements to be formulated according to positivism, which means that requirements engineers shall formulate requirements in a positive instead of a negative formulation, the `NoNegationCoCo` warns the requirements engineer that a negative formulation is in use. Hence, the context condition triggers a warning for the requirement “The vehicle shall not activate the Lane Keeping Assist functionality if the absolute speed is below 50 km/h.” to hint that a formulation such as “The vehicle shall prevent activating the Lane Keeping Assist functionality if the absolute speed is below 50 km/h.” would be better suited in a positive requirement formulation. If a double negation is used, such as in “No vehicle shall not deactivate the Lane Keeping Assist functionality if the absolute speed is below 50 km/h.” the `NoDoubleNegationCoCo` triggers an error to prevent requirements engineers from using this kind of formulation.

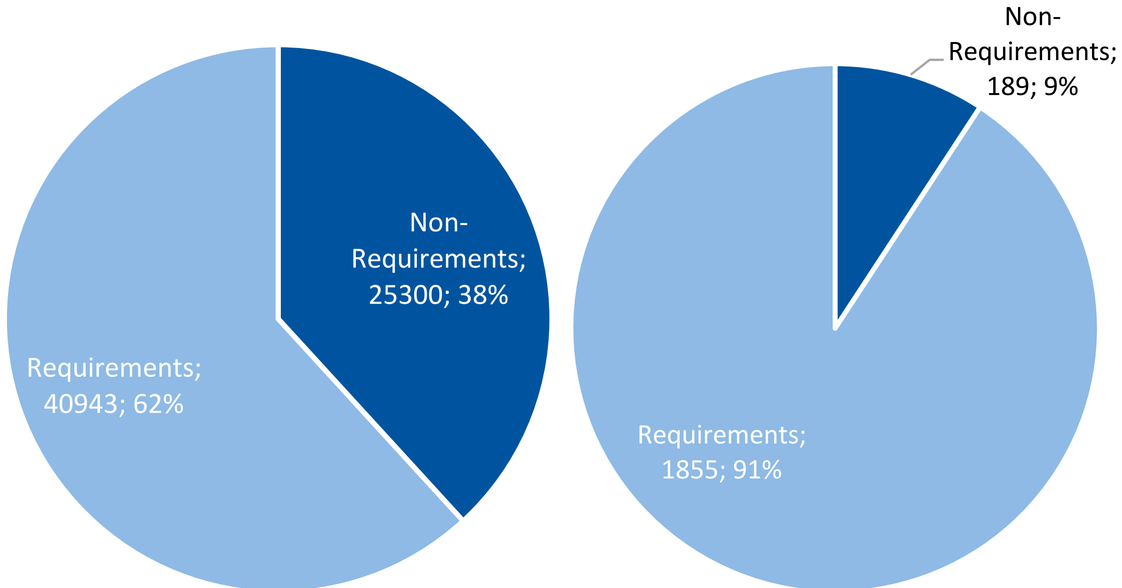
Finally, the SPECTRE-DSL from [Kat21] developed a PrettyPrinter to bring all parsed requirements into a unified form. By this, all requirements that could, for instance, have the condition at the end of the requirements are transformed to a requirement with the condition at the beginning, leading to a uniform style of requirements writing.

5.4 Evaluation Requirements in the Automotive Industry

To evaluate to which extend requirements in a requirements template are processable by a dedicated requirement DSL, the DSL developed in section 5.3 was applied to the requirements from the Alpha (subsection 4.1.1) and Beta project, which follow the already known pattern presented in section 5.2. By this, the evaluation aims to answer the following research questions:

- Is the developed requirements DSL applicable to legacy requirements from the Alpha and Beta projects to detect errors in the specification?
- How suitable is the developed SPECTRE-DSL in the context of legacy requirements from the Alpha and Beta projects to integrate legacy requirements into system architecture models?
- How applicable is the developed DSL in the context of legacy requirements from the Alpha and Beta projects?

To answer these questions, the SPECTRE-DSL served to process the legacy requirements from the Alpha (*cf.* subsection 4.1.1) and Beta (*cf.* subsection 4.1.2) projects. In



(a) During preprocessing excluded requirements in the Alpha project.

(b) During preprocessing excluded requirements in the Beta project.

Figure 5.6: Requirements excluded during the preprocessing in the Alpha and Beta projects.

this process, a preprocessing step filters the requirements and makes all requirements processible. Then, the SPECTRE-DSL served to parse the requirements, and this thesis rates the applicability of the DSL by analyzing the parser errors and results of the context-condition checks. As the automated processing of the requirements might miss some essentials, this thesis finally connects these results with an expert analysis performed on a comparable subset of the requirements in both projects to mitigate this risk.

5.4.1 Preprocessing

Since the requirement DSL from section 5.2 has some additional restrictions, some preprocessing steps were required to parse these legacy requirements. First, some requirement texts from the projects intentionally violated the SPECTRE template to formulate information without legal obligation. By convention, the requirements engineers of both projects marked these requirements by not using modal verbs in the sentence structures. Since these were part of the initially considered requirements, all 25300 requirements that did not contain any modal verbs are excluded from the total number of 66243 requirements leading to 40943 parsed requirements in the Alpha project. In the same way,

Table 5.14: Requirement preprocessing for the SPECTRE-DSL evaluation.

Preprocessing Step	Total Alpha	Total Beta
Unchanged requirements	25618	1625
At least one - was removed.	10589	89
At least one / was removed.	2262	116
At least one - and / removed.	2474	25
Total Requirements	40943	1855

Table 5.15: Overall evaluation results of the requirements in the Alpha and Beta projects using the SPECTRE-DSL.

Overall Evaluation	Alpha	Beta
Passed	2491	531
PrettyPrinterDeviation	2995	286
CoCo-Warning	5590	217
CoCo-Error	1312	10
ParserError	28555	811
Total	40943	1855

189 requirements in the Beta project were excluded from the total of 2044 leading to 1855 for evaluation in this project. Figure 5.6a and Figure 5.6b present the relation of excluded requirements in both projects.

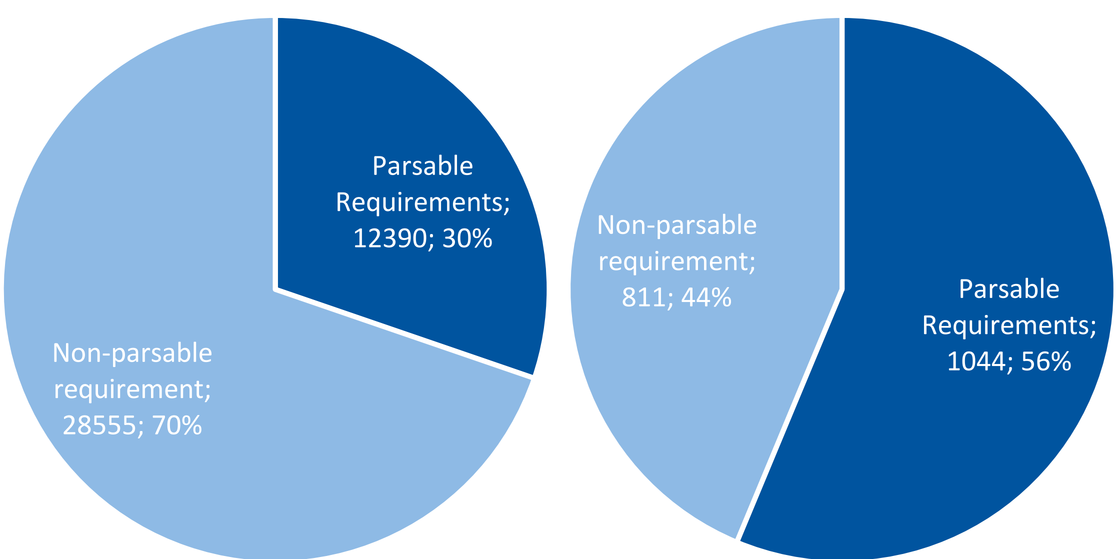
Second, hyphens and slashes had to be removed, as they are not allowed at all places in the requirements in the SPECTRE-DSL, whereas the original SPECTRE template did not make any restriction in this regard. Table 5.14 shows the exact number of requirements in which at least one illegal symbol was removed to evaluate the parser.

5.4.2 Parsing

After this preprocessing, the parser of the SPECTRE-DSL parsed the remaining requirements. During this process only 12390 (30.26%) of the requirements in the Alpha project and 1044 (56.28%) in the Beta project were parsable. Thus, 28553 (69.74%) of the requirements in the Alpha project and 811 (43.72%) in the Beta project were not parsable as Figure 5.7b additionally visualizes.

5.4.3 CoCos, PrettyPrinter and Overall Comparison

Additionally, all presented context conditions were applied, and the pretty printing algorithm was applied to identify deviations between the original and the pretty printed

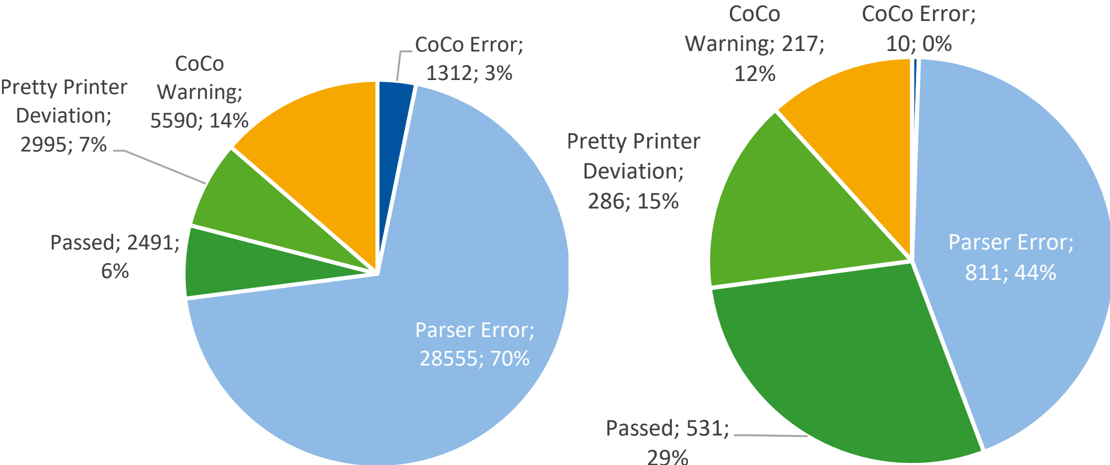


(a) Parsing results of the requirements in the Alpha project.

(b) Parsing results of the requirements in the Beta project.

Figure 5.7: The parsing results of the requirements in the Alpha and the Beta projects.

requirement. As the pretty printer, apart from fixing formatting issues, also formats the main and subordinate clauses into a unified form, a requirements engineer might want to cross-check deviating requirements for formulation errors. Based on these results, Table 5.15 presents the overall evaluation results, in which the requirements are classified according to the results the SPECTRE-DSL returned based on their severity. All passed requirements passed the parsing process and the context-condition check and are suitable for further processing in the context of further requirements correctness checking or integration into system architecture models. Next, all requirements where the pretty-printer returned a different result than the initial requirement might also be suitable for further processing. However, a manual quality check by a requirements engineer is required to decide whether the parsed requirement in the AST of the SPECTRE-DSL specifies the intended requirement. Then, all requirements with a warning did not pass all context-condition checks but failed at least one context-condition for which section 5.3 specified a warning. Since this might only indicate a problem but is not necessarily a violation of the requirements boilerplate, these requirements might also be suitable for further processing. The following requirements are erroneous, starting with the requirements for which the context conditions detected an error. While the requirements still have an AST and clear order to a violated guideline, all requirements that are not parseable are rejected for various reasons. As Table 5.15 shows, these requirements are the biggest group of requirements in both projects. The percentage distribution Figure 5.8 presents reveals



(a) Overall evaluation results of the requirements in the Alpha project.

(b) Overall evaluation results of the requirements in the Beta project.

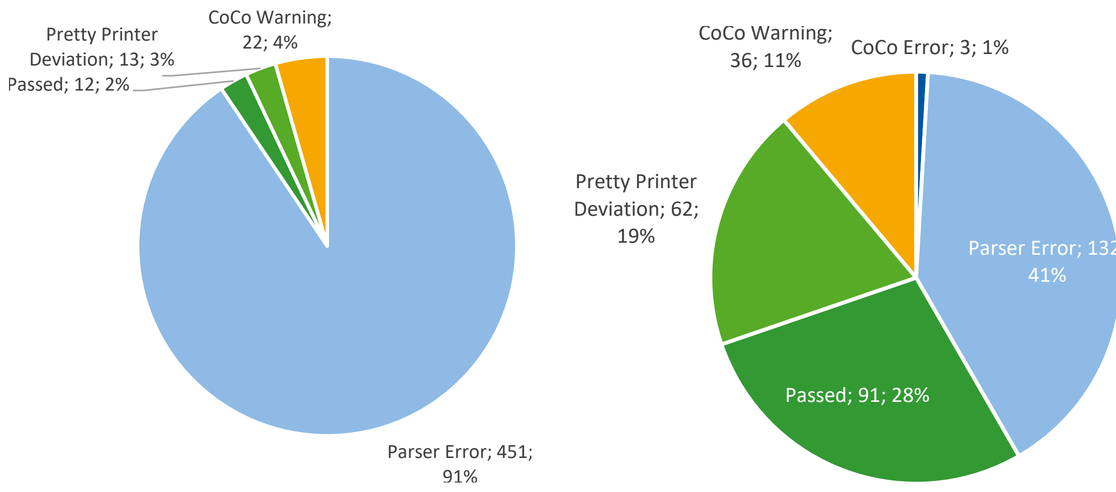
Figure 5.8: Overall evaluation results of the requirements in the Alpha and Beta projects.

further that in the Alpha project, the most significant proportion of the requirements could not be parsed, whereas in the Beta projects, more requirements were parsable than not-parsable. Consequently, all 38452 requirements in the Alpha project and all 1324 in the Beta project that were not evaluated as passed would require additional efforts to correct these errors.

Furthermore, both projects demonstrate that only a few requirements were successfully processed, which indicates that using the requirements DSL is more effective in detecting errors in the existing requirements than in processing them. The application identified more erroneous requirements than successfully processed requirements in the projects under evaluation. Thus, using a requirement DSL set is convenient in correcting legacy requirements in comparable projects. Nonetheless, it is not a suitable method for editing them when many errors are present in the legacy dataset, as the resources for automated requirements correction are unavailable.

5.4.4 Comparison with Expert Analysis

Since this evaluation only shows which requirements successfully parsed or failed a context condition but does not further explain why a specific requirement was not parsed, two subsets of the requirements were selected, in which two requirement experts manually classified the errors in the requirements. For improved comparability, the selected subsets refer to a set of requirements that address the same subsystem in both projects, but two different teams of requirements engineers specified the requirements. The appli-



(a) Overall evaluation results of the requirements of a subset of the requirements in the Alpha project.

(b) Overall evaluation results of the requirements of a subset of the requirements in the Beta project.

Figure 5.9: The overall requirement evaluation results.

cation of the SPECTRE-DSL leads to the requirement evaluation shown in Figure 5.9. As the comparison of Figure 5.9a and Figure 5.8a shows, there is a deviation in the subset from the overall trend, whereas the comparison of Figure 5.9b and Figure 5.8b reveals a comparable trend. Even though this deviation might indicate that the subset selection in the Alpha project is not representative of the overall project, a further analysis of the requirements in this subset revealed additional knowledge.

In this process, two requirements experts manually evaluated whether the requirements fulfilled the SPECTRE template and identified the errors in these requirements. After both engineers finished this manual evaluation, the requirements engineers consolidated their evaluations to an overall evaluation result. During this process, they discussed all deviations and developed a uniform classification scheme as presented in Table 5.16. With these results, comparing the manually rated requirements and the results from the SPECTRE-DSL enables a validity check of the passed requirements in the next step. As this comparison showed, all parsed requirements conform with the template, and all requirements that passed requirements were parsable in the DSL.

As the variable error shows, a large proportion of the Alpha requirements in the subset were not parsable since the requirements engineers specified the variable wrongly *e.g.*, by using the wrong brackets or by specifying the actual values inside the brackets, which is a violation of the grammar rules specified in the SPECTRE-DSL. Since they can easily be corrected, Figure 5.10 presents a new evaluation result using corrected requirements as input. As the comparison of Figure 5.10 and Figure 5.8a shows, the new requirements

Table 5.16: Manual evaluation results of the requirement subsets extracted from the Alpha and Beta project requirements.

Manual Classification	Total Alpha subset	Total Beta subset
Additional information	15	13
Comment Error	8	15
Constraint error	4	2
Correct	47	189
Punctuation error	8	1
Sentence structure error	168	103
Variable error	163	0
Wrong use of variable	85	1
Total	498	324

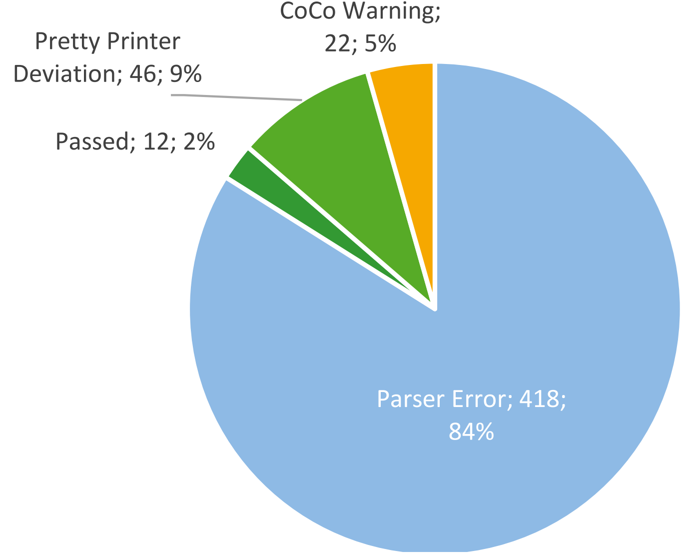


Figure 5.10: Overall evaluation results of the requirements in the Alpha subset with corrected variables.

are closer to the overall trend but do not follow it completely. Although slight deviations to the template application are easily overseen or accepted by a human reviewer, they cause significant deviations in the outputs using a parsing algorithm. Thus, requirement DSL are suited for error detection in requirements.

5.4.5 Reflection on Research Questions

In conclusion, the application reveals that the developed DSL is well suited to detect structural errors in the requirements. As Figure 5.8a and Figure 5.8b show, the application of the SPECTRE-DSL to the legacy requirements uncovered errors in a significant fraction of the requirements, which the manual requirement reviews performed in the projects prior to the application of the requirements DSL failed to uncover. Therefore, according to the applicability of the requirements DSL for structural error detection, the first research question can be answered with yes and backed up with these results. As the DSL by design only checks the structural consistency with the SPECTRE template, only these errors could be detected applying the DSL. Hence, an in-depth analysis of the unambiguity and semantical correctness could not be performed using the DSL due to a lack of formalization of the statements in the requirements template.

Regarding the second research question, as Figure 5.8a and Figure 5.8b indicate most legacy requirements were not parseable or required other corrections as indicated by the failed context condition checks. Thus, an easy integration without another massive correction effort within the legacy requirements is impossible. Thus, whether the efforts saved in processing these requirements overcompensate the efforts required to correct the legacy requirements is questionable. Therefore, it might be better to model new systems directly from scratch than integrate these legacy requirements.

In conclusion, concerning the third research question, all requirement templates were identified in the requirements. Moreover, in manually checked subsets, no requirements were discovered to be accurate in a manually formulated SPECTRE expression but were not parsed by the DSL. However, these project applications only covered some possible requirement formulations. Thus, further (and not parsable) formulations are possible in the requirements boilerplate, which cannot be processed using similar construction rules. Consequently, there remains a risk of not detecting correct requirements when applying the DSL on legacy requirements in the Figure 5.8a and Figure 5.8b and other unrelated projects.

5.5 Interim Conclusion, Discussion and Lessons Learned

The experimental evaluation presented in this thesis allows us to draw some conclusions and lessons. Since there are some risks and challenges in generalizing the results due to the nature of the experimental evaluation, they are first classified, discussed, and

evaluated in subsection 5.5.1 before subsection 5.5.2 presents the conclusions and lessons learned.

5.5.1 Threats to Validity

Regarding the validity of the results, the evaluation bears two significant risks. First, although the projects consider numerous requirements, they only consider projects at the same engineering service provider. Therefore, the results do not represent all the requirements in the automotive industry. Secondly, the partial reproducibility of the evaluation arises from the inability to publish the requirements alongside the DSL for assessment. As a result, other researchers may find the evaluation moderately reproducible.

Due to the first risk, the insights derived from this evaluation offer a limited perspective, capturing only a tiny portion of the overall industry. Nonetheless, despite this constraint, the case study provides valuable lessons that this thesis aims to convey in the concluding section of this chapter. As for the second risk, the grammars are accessible in Appendix B, enabling other researchers to conduct similar experiments on different requirements.

Moreover, some additional risks remain regarding the implementation. On the one hand, additional patterns might improve the requirements formulation results and lead to another result in the evaluation. To mitigate this risk, the manual evaluation of a representable subset as presented in section 5.4 was performed, which showed that the rules implemented in the DSL were sufficient to process the requirements in the subset. This assessment revealed that all parsed requirements conformed to the provided sentence template and vice versa. Hence, all requirements that passed the expert review were also parsable in the DSL. Consequently, the results revealed that even though manual checking is feasible, using the requirements DSL reveals the same structural errors without needing a dedicated expert evaluation. As revealed by the detected misuse of the template in the Alpha project, we can efficiently correct misused rules in the template or extend the DSL using the generative and extensible design of MontiCore. Because of the modular grammar design and the use of preterminals in the SPECTRE grammar, a straightforward extension is possible, mitigating concerns regarding potential improvements in the initial interface implementation used for the evaluation this thesis presents.

Finally, only five essential context conditions were implemented in the first place (*cf.* section 5.3), which means that additional context conditions might render additional requirements invalid. For a complete set of context conditions, the project internal guideline specified 11 additional rules that could be implemented as additional context conditions in future works. Since an essential proportion of the requirements requires additional effort to correct the results, which would also be the case when context conditions render additional requirements invalid, the results gathered in this evaluation are valid.

5.5.2 Lessons Learned from using the SPECTRE Requirement DSL in the Automotive Industry

Despite its limitations, the methodology yielded valuable lessons that apply to similar projects through its application in the context of the Alpha and Beta industrial projects. In order to continue to make these insights applicable, this paper will break them down into the following four lessons:

- L1: Legacy requirements in natural language requirements are partially unusable for automatic models or model element derivation.
- L2: Errors are more easily explained to the user if the elements are grammatically labeled.
- L3: Requirement DSLs are well suited to check the structural correctness of requirements but do not give many insights on the semantic quality.
- L4: Requirements engineers need to improve their ability to write syntactically correct requirements in a sentence template and recognize minor structural errors during a manual check.

The first finding resulting from the evaluation in section 5.4 is the observation that in both evaluated projects, a large part of the requirements do not meet the formal criteria of the sentence template and are therefore not suitable for further use (L1). For example, already during preprocessing 25300 requirements that did not contain modal verbs and were removed from the total number of 66243 requirements in the Alpha project, and 189 requirements in project Beta were excluded from the total 2044. Despite the intentional deviation from the requirements template in the applied methodology to make these requirements more informational, there is still a possibility that this wording stems from a specification error or that, despite their informational nature, these requirements contain essential information for the requirements specification. As a result, other techniques such as requirements classification of requirements and informal statements might be useful to process the requirements as suggested in [WV16]. In addition, numerous remaining requirements are also not processable due to other errors in using the requirements template. For example, only 12390 (30.26%) of the requirements in the Alpha project and 1044 (56.28%) in the Beta project were parseable. Of the parsed requirements, the context conditions subsequently revealed additional errors. In terms of the total 66243 in the project Alpha respectively 2044 in the project Beta, only 2491 (30.26%) of the requirements in the Alpha project respectively 531 (25.98%) of the requirements were in a state suitable for further processing without any manual corrections. Accordingly, a significant portion of the requirements that are not pareseable, contain errors or warnings have to be reworked. Thus, 38452 (58.05%) in the project Alpha or 1324 (64.77%) in the Beta project) require rework, which, due to the

additional effort that is required to rework the legacy requirements, reduces the benefit of the approach compared to completely new development with MBSE methods. Furthermore, independent of the experiment presented here, [AVP+19] also points out that when implementing MBSE, it is often not helpful to start with old requirements, but it is better to start from scratch for new MBSE projects. In addition, interviews in [LTK19] establish that fully formalized models, such as those needed for automated derivation of models from textual requirements, are not considered necessary by industry, and many legacy requirements are not reusable in new developments either due to their quality.

The second lesson results from the observation that although a requirement-DSL, as this thesis has presented in section 5.3, is well suited to check compliance with a sentence template, it is difficult for a user to see which steps are needed to translate the requirement into a correct requirement. The main reason is that the parsers cannot effectively communicate the errors to the relevant domain experts. Identifying specific actions to transform the provided text input into a valid request becomes challenging due to the sentence template in use. For example, the parser generator ANTLR used by MontiCore generates an LL(K) parser that is very good at identifying syntax errors due to its structure [MMMI16]. However, the text passages and excerpts are challenging to classify without a linguistic or grammatical context since the requirement template is oriented more toward the grammar of the English language than the formal language used for parsing, which also manifests in the results of the manual classification of the requirement errors (*cf.* Table 5.16) that indicates that most errors follow from incorrect sentence structures, which might be correct in natural language but violate the requirements template. Linguistic methods such as NLP [ZAF+21] or AI-based approaches such as LLMs could counter this problem by using different parsing algorithms in case of NLP or could provide means for automatic error correction [BKK+23] using LLMs or DSL extraction [BBK+22b, BBK+22a] suggested could be promising approaches for future research to extend to the methods presented in this chapter. Since natural language is neither context-free [PG82, Shi85], nor regular [Cho57] in the Chomsky hierarchy of formal languages [CS59], these methods based on natural language processing are not always correct, which might lead to false positives and false negatives. Therefore, if one uses NLP to check the performance, as presented, for example, in [ASBZ15] for the sentence template in [PR21], it still requires additional manual effort to rule out these errors.

As the results in section 5.4 show, the requirements specification DSL presented in this thesis is well suited for detecting errors in the requirements specification. This observation becomes evident not only in the number of errors detected in the projects considered but also in the comparison with the expert review of the requirements. As Table 5.15 shows, the majority of the requirements in the projects under consideration still contain structural errors that the requirements engineers did not recognize during the manual review. The comparison with a further in-depth expert evaluation in Table 5.16 shows that many structural errors in the requirements remained undetected in the first review.

Furthermore, this additional review showed that the use of the requirements DSL not only revealed all structural errors in the subset under consideration but also that all detected errors did not match the structural framework specified by the requirements template from section 5.3. Although the experts' capacity was insufficient to check all requirements manually, there could, therefore, exist further uncovered cases in which the requirements DSL failed in this task. Some conclusions for future projects can be drawn from the subset under consideration. Because even if this or a comparable DSL is capable of uncovering all structural inconsistencies, each evaluation comparable to the one in section 5.4 only provides a few insights regarding the semantic correctness of the requirements. Methods based on requirements templates only provide a structural framework for requirements formulation but offer no possibility to evaluate the logical correctness of the logical implications raised by the requirements. Therefore, the requirements could be correct (or incorrect) in the context of the system under development, regardless of their structural compliance with a requirements template DSL. Future projects that aim to mitigate this risk must provide additional mechanisms or transformations to check this aspect in addition to structural correctness. As presented in [BKK+23], possible extensions use requirement templates as a starting point for translating requirements in TCTL, in which an automatic consistency check is possible.

Since all requirements were manually reviewed by requirements engineers before, and still many structural errors were identified, the evaluation in section 5.4 also indicates that humans are not good at identifying structural errors and the fulfillment of a requirements template (L4). Therefore, the application of automated requirements-checking mechanisms enables requirements engineers to increase the quality of their requirements.

Based on these lessons learned during the application of the requirements DSL, the following suggestions are possible:

- As L4 shows, requirements engineers tend to overlook many structural errors in a manual review. Moreover, as L1 revealed, requirements DSLs and comparable NLP based approaches such as the one in [ASBZ15] reveal many additional errors in a short amount of time. Therefore, it is a good practice not only to hand a sentence template to the requirements engineers but also to establish mechanisms to automatically check the compliance of these structures.

- To aid in the requirements processing, it is helpful to aid the parser in detecting the individual sentence parts by adding brackets to the additional parts (for example, done in section 5.3 to mark the constraints). As another extension, requirements engineers could also label all parts of the requirements by different brackets or establish tabular requirement templates in which each part of the requirement is a cell in a requirements specification table.

- Do not use natural language for requirements specification. Even though this proposal seems misplaced in a chapter about methods for modeling textual require-

ments, the insights of the evaluation in section 5.4 show that an additional effort is required to correct two-thirds of the considered requirements. As a result, the efforts spent on legacy requirements that should not be part of the original model seem unreasonable. Methods such as SPES [PHAB12] or SPESXT [PBDH16] show that a complete model-based system specification does not require textual requirements formulated in natural language. Consequently, a specification and integration of natural language requirements might be unnecessary.

In the process of taking up the idea of the last insight, the following chapters aim to answer the following follow-up questions:

- How can elements that are typically described in natural language be encoded in SysML models?
- How can semi-formal textual requirements be derived from SysML models?
- Is it beneficial to use textual natural language requirements for reviewing purposes of the model?

To this end, section 6.1, section 7.1, and section 8.1 investigate how these elements are expressible as SysML Use Case Diagram (UCD), SysML Activity Diagram (AD), SysML State Machine Diagram (STM), and SysML Internal Block Diagram (IBD) diagrams and how these elements are usable to derive semi-formal requirements. Finally, subsection 6.4.3, subsection 7.3.2, and subsection 8.3.2 investigate how these perform in practical applications in the context of the Zeta, Delta, VTOL, and Epsilon projects to answer **RQ-4.4** ("Are the models that result from a model-driven specification comparable to document-based specifications?").

Chapter 6

Expressing Stakeholder Values using Use Case Diagrams

This chapter presents a method to express stakeholder values according to the CUBE methodology using use case diagrams, drawing on the previously published formalization of use case diagrams in [KRW22]. Since use case diagrams are a means to describe the functionalities of a system in terms of how its various users achieve their goals by using the system [FMS14b, KRW22], they are commonly used to express the functionality requirements [PR21] or to model system functionalities [FMS14b, JGW^{+}21].

The concept of describing a system's functionality based on simple models [KRW22], called use cases, originates in [Jac93] and further evolved over multiple approaches and representation forms [Coc01, Fow04] to its current standardized form in the UML [Obj17] and SysML [Obj19] specification. Apart from this standardized graphical representation, various ways to formulate use case specifications exist, such as tables, as mentioned in [Fow04, Coc01] or other UML/SysML diagrams [FMS14b]. In this context, two different approaches to modeling use case diagrams exist. First use case diagrams, as a means of describing how (*i.e.,* in which use cases) external actors interact with the system to achieve their goals [FMS14b], and, second, use case specifications, as a means to specify the system behavior in a given use case scenario [Whi06]. Since the first understanding is related to the corresponding use case diagram from the UML [Obj17] and SysML [Obj19] specification and follows the same ideas as the stakeholder value specification in CUBE, this section focuses on the first understanding, whereas the second understanding of a use case specification in chapter 7, as the latter understanding is more related to the operating principle specification in CUBE.

Hence, in the context of this thesis, the use case diagram aims to address three main concepts. First, the actors in this diagram help to visualize the system participants. Second, the system boundary aims to delineate the system from its environment from which the actors interact with the system. Third, the primary and namesake concept of the diagram, the use cases, serve to specify the system's intended behavior since the stakeholder values in CUBE focus on identifying high-level stakeholder requirements and how the system's features can contribute to specification.

To address **RQ-3.2** (How can stakeholder values be formulated?), this chapter is struc-

tured as follows. First, section 6.1 presents a method to express stakeholder values as use case diagrams. Then, section 6.2 defines the semantic basis for the use case diagrams used in this thesis based on [KRW22]. Drawing on this formal semantics definition, subsection 6.4.1 provides an extension of the modeling guideline section 6.1 in the context of the semantics definition from section 6.2 to derive natural language requirements from the use case models, which could ultimately replace natural language requirements in the stakeholder value specification. Finally, section 6.4 evaluates this method in the context of industry projects to demonstrate the approach's applicability in the automotive industry.

6.1 Use Case Diagrams for Stakeholder Value Specification

Apart from textual requirements, use case diagrams are a commonly applied means to formulate high-level requirements on a system [BS03, CHQW15]. In the context of the CUBE methodology, as summarized in section 3.1, the use case diagram targets several essential aspects in system analysis and design. First and foremost, the actors of a use case diagram aid in pinpointing the relevant system stakeholders at the current decomposition level. These stakeholders could include individuals or other systems that interact with or influence the system under design. Next, the system boundary draws a clear line between the system, where the use cases are located, and its environment, where the actors interacting with the system and serving as stakeholders are drawn. By this, the use case diagram assists in the definition of the *system boundary* as defined in subsection 2.1.1. Therefore, it facilitates a comprehensive understanding of the system's boundaries for the diagram reader and draws a first sketch of its interactions with the broader system context. Furthermore, the use case diagrams must reflect a stakeholder value model that assists systems engineers in articulating what stakeholders anticipate from the system, thus encapsulating their needs. Hence, use cases serve to represent these anticipated interactions and functionalities.

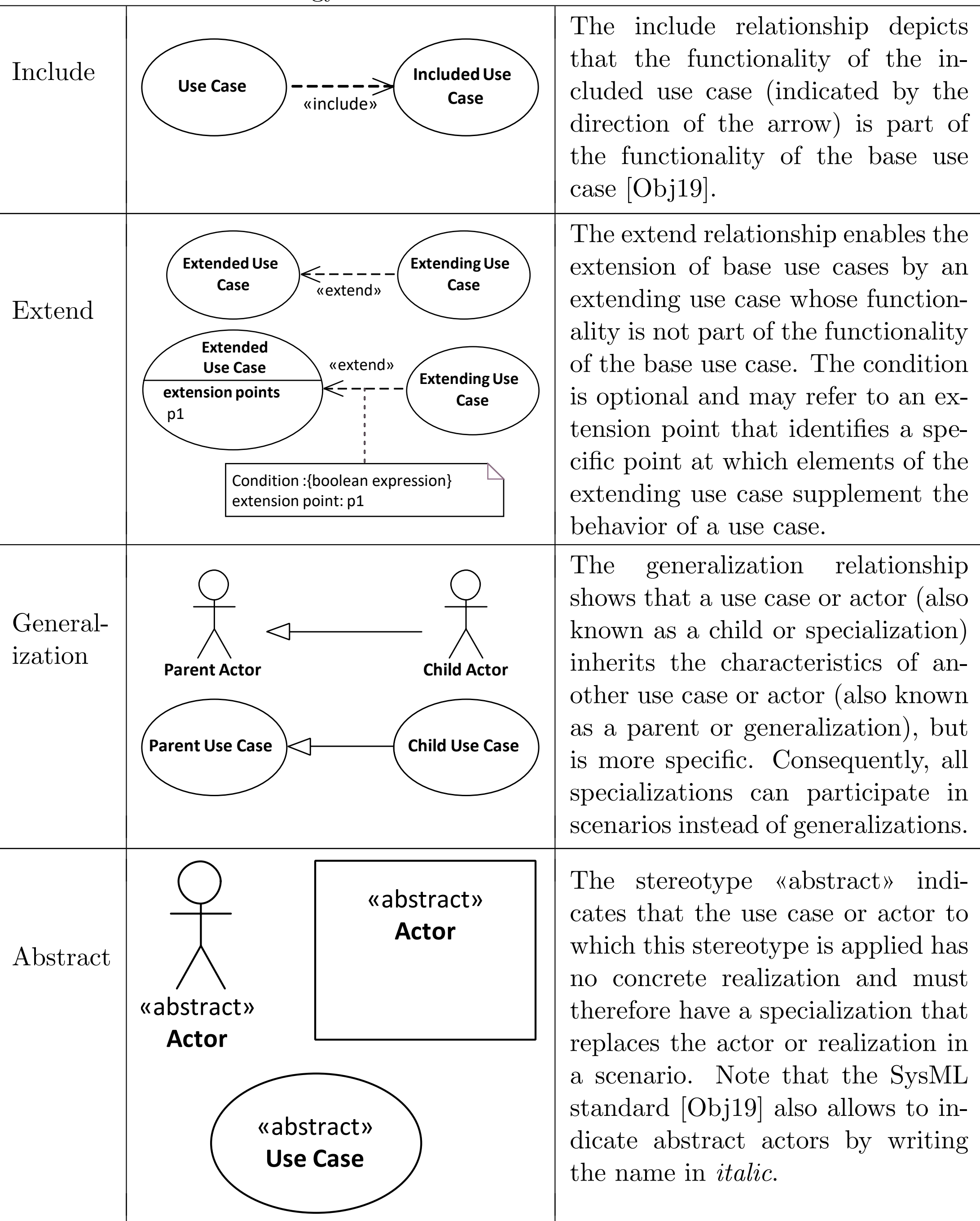
6.1.1 Diagram Elements to Model Stakeholder Values using SysML Use Case Diagrams

To systematically express stakeholder values independent of the specification method, with use case diagrams, Table 6.1 provides a subset of the SysML specification [Obj19] tailored to the application in the context of the stakeholder needs elicitation. By this, this section contributes to answering **RQ-3.2** ("How can stakeholder values be formulated?"), providing the required elements in the SysML.

Table 6.1: UCD elements required to describe the stakeholder values according to the CUBE methodology.

Element Name	Notation	Description
Obligatory Diagram Elements		
Use Case	Use Case	A use case describes a functionality or characteristic that the system offers. When associated with an actor, it refers to how the actors may use the system to achieve their goals.
Actor	Actor Actor	An actor represents a human (*e.g.,* a system user), an organization (*e.g.,* for standardization), or any system in the environment that interact with the system of interest).
Association	Actor Use Case	An association is a relationship between an actor and a use case that specifies the interaction between the actor and the system according to the specification of the use case [ISO17b].
System (Boundary)	System Boundary Use Case	A system boundary represents a mental boundary that separates the system under development from its environment. The use cases within this boundary express the ability of the system to offer these different functionalities to its stakeholders in its environment.
Optional Diagram Elements		

Table 6.1: UCD elements required to describe the stakeholder values according to the CUBE methodology.



Include	Use Case «include» Included Use Case	The include relationship depicts that the functionality of the included use case (indicated by the direction of the arrow) is part of the functionality of the base use case [Obj19].
Extend	Extended Use Case «extend» Extending Use Case Extended Use Case extension points p1 «extend» Extending Use Case Condition :{boolean expression} extension point: p1	The extend relationship enables the extension of base use cases by an extending use case whose functionality is not part of the functionality of the base use case. The condition is optional and may refer to an extension point that identifies a specific point at which elements of the extending use case supplement the behavior of a use case.
Generalization	Parent Actor, Child Actor Parent Use Case, Child Use Case	The generalization relationship shows that a use case or actor (also known as a child or specialization) inherits the characteristics of another use case or actor (also known as a parent or generalization), but is more specific. Consequently, all specializations can participate in scenarios instead of generalizations.
Abstract	«abstract» Actor «abstract» Actor «abstract» Use Case	The stereotype «abstract» indicates that the use case or actor to which this stereotype is applied has no concrete realization and must therefore have a specialization that replaces the actor or realization in a scenario. Note that the SysML standard [Obj19] also allows to indicate abstract actors by writing the name in *italic*.

As the name-giving element, use cases are the primary elements of the SysML use case diagram and consequently obligatory elements for a use case diagram that expresses the system's functionalities to its stakeholders. Since a use case only describes what happens but not who is required to participate in order to execute the functionality, nor which system executes this functionality, the actor, the association, and the system boundary are the other obligatory elements required in all SysML use case diagrams for expressing stakeholder values according to CUBE. Therefore, a system boundary (also called subject in the SysML standard [Obj19]) typically governs the diagrams, standing out as the most prominent element wherein and around all other diagram elements reside. It represents a mental boundary that separates the system under development from its environment. System boundaries might be nested according to the decomposition structure of the system, especially on the lower decomposition levels of CUBE. The use cases within each boundary express the ability of the system to offer these different functionalities to its stakeholders in its environment, which the diagram depicts as actors. According to the SysML specification, actors represent roles external to the system that may correspond to users, systems, and or other environmental entities [Obj19]. Since this definition overlaps with the stakeholder definition (*cf.* Def. 9), as the actors have a right, share, claim, or interest in a system or its possession of characteristics that meet their needs and expectations [ISO15] in the form of these use cases, the actors in the diagram represent the stakeholders of the system. When associated with an actor, a use case refers to how the actors may use the system to achieve their goals. Therefore, a use case associated with an actor depicts a stakeholder value where the actor is the stakeholder that requires the system to have a use case to achieve its goals. When not associated with an actor, a use case is either underspecified and might be associated with an actor of a lower decomposition layer later or executed by the system as an automatic system activity.

Apart from these obligatory elements, several optional elements might aid in further refining the diagrams. First, the include relationship indicates that the functionality of the included use case is part of the functionality of a base use case. Using the include relationship, a systems engineer can further refine the stakeholder value expressed by a base use case to additional steps. Following a similar idea, the extend relationship enables systems engineers to extend a base use case with an extend use case whose functionality is not part of the functionality specified in the base use case. With an optional condition, a systems engineer may further specify the required conditions for this extension or even combine this condition with a reference to an extension point that identifies a specific point at which elements of the extending use case supplement the behavior of a use case. Finally, the generalization relationship shows that a use case or actor (also known as a child) inherits the characteristics of another use case or actor (also known as a parent). By utilizing this relationship, a systems engineer can define additional, more specific, actors or use cases that may participate in all scenarios where the general use case or actor participates. Furthermore, the engineer can specify additional scenarios where only specialization is permitted.

Note that all elements are beneficial in a stakeholder value specification, but not all optional elements are necessarily required to create a valid stakeholder value specification.

6.1.2 Stakeholder Needs for a Simple Vehicle Expressed as a SysML Use Case Diagram

To apply the previously defined concepts, in the context of this thesis' running example from section 4.2, Figure 6.1 presents a use case diagram based on the simple vehicle example provided in [FMS14a]. As a boundary, the diagram specifies the 'Vehicle', which

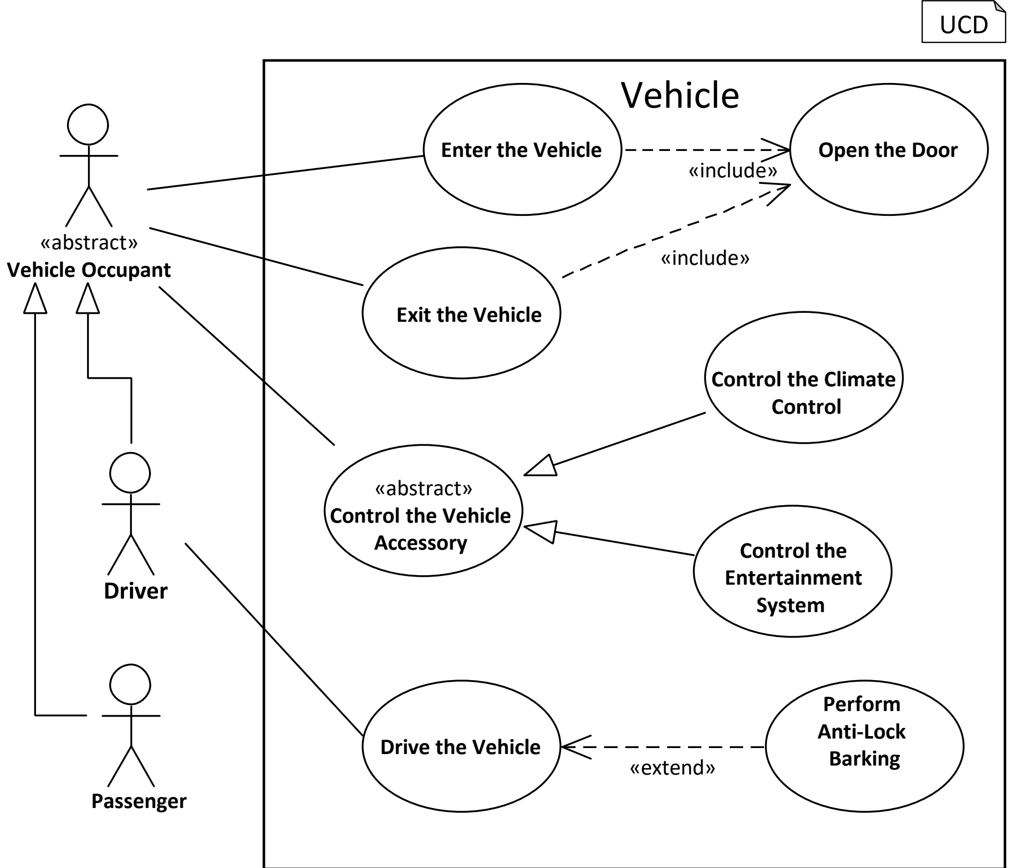


Figure 6.1: Automotive driving UCD based on an initial example from [FMS14a].

separates the vehicle as the system under development from the actors 'Passenger' and 'Driver' in the environment. Next, within the boundary, the UCD in Figure 6.1 specifies several vehicle use cases. First, persons want to become vehicle occupants, which requires

them to 'Enter the Vehicle'. As remaining in the vehicle is typically not the intended final state of using a vehicle, the vehicle occupants also want to 'Exit the Vehicle'. To feel well accommodated, vehicle occupants also express the need to 'Control the Vehicle Accessory' as a general use case that can, for example, concertize to 'Control the Climate Control' and 'Control the Entertainment System'. As the essential use case, a particular type of 'Vehicle Occupant' called 'Driver' additionally desires to 'Drive the Vehicle', which as a notable safety feature shall additionally allow to 'Perform Anti-Lock Braking' during driving scenarios.

Even though the diagram in Figure 6.1 follows the graphical syntax definition from the SysML standard [Obj19], this definition alone does not suffice to define the semantics of the diagram. Following the semantics definition of [KRW22], an intuitive description of the semantics of the diagram are possible depending on two different views: the closed and open-world semantics. As both relationships rely on the same fundamental concepts and only differ in the possible combinations of these concepts, this section introduces these fundamental concepts first. In both semantic interpretations, each use case in the diagram is executable if its preconditions are fulfilled. Moreover, executing a use case in both semantic interpretations introduces a scenario. As an extension to the original semantics definition, a system must execute these use cases in systems engineering. Hence, a scenario consists of systems executing use cases. If a system executes a use case in a scenario, then all use cases that extend the use case where the guard of the extend relation and the precondition of the use case are satisfied must also be executed by the executing system on the boundary in the scenario [KRW22]. This rule applies to all recursive executed use cases in the scenario. If we suppose there is an extend relationship from a use case to a use case of the scenario without a guard, the former use case may be executed in the scenario, but its execution is not required [KRW22]. Following the same logic, the system must execute all included use cases in the use case scenario and, recursively, all use cases this use case extends or includes. In the context of the example use case, Figure 6.1 presents the vehicle executes the 'Enter the Vehicle' and the 'Exit the vehicle' use cases only in combination with the 'Open the Door' use case. In contrast, the vehicle may or may not perform an 'Anti-Lock Braking' whenever the vehicle executes the 'Drive the Vehicle' use case, as the extend relation between the two use cases does not have a guard constraining the execution. In the scenarios, the actors participating in an executed use case are represented by the actors explicitly and implicitly associated with the use case [KRW22]. In this context, an actor is explicitly associated with a use case if the use case diagram contains an association between the actor and the use case [KRW22]. An actor is implicitly associated with a use case if the use case diagram contains an association between the actor and another use case that generalizes the use case [KRW22]. In the context of the use case diagram from Figure 6.1, this means that, for example, only the driver may participate in the 'Drive the vehicle' use case, and not every 'Vehicle Occupant' such as the 'Passenger', whereas the 'Driver' may also participate in all use cases in which the 'Vehicle Occupant' may participate.

Since the 'Vehicle Occupant' is an abstract actor, only concrete specializations of this actor may participate in the scenarios instead of the actors. As a result, only a 'Driver' or a 'Passenger' may participate in the 'Enter the Vehicle' use case. Following the same idea, each use case is replaceable by a use case that specializes the use case and has a satisfied precondition [KRW22]. Thus, the specialization inherits the properties corresponding to the extend relation of its generalization [KRW22]. Consequently, all actors that participate in the specialization must be determined using the same determination method. Furthermore, all use cases that must be recursively executed when executing the specialization (because of the extend relations) must also be executed [KRW22]. Hence, the 'Driver' and the 'Passenger', as specializations of the abstract 'Vehicle Occupant', may participate in the 'Control the Climate Control' use case and the 'Control of the Entertainment System' as they are specializations of the 'Control the Vehicle Accessory' use case. As the 'Control the Vehicle Accessory' is again abstract, this use case alone is not executable as part of a scenario, only its specializations. When looking at the difference between closed-world and open-world semantics, the closed-world semantics of a use case diagram only consists of scenarios of use case system pairs that relate to include or extend relationships contained in the diagram. Therefore, use cases not modeled in a use case diagram cannot be part of any scenario contained in the semantics of the use case diagram [KRW22]. Consequently, entering the vehicle and driving the vehicle would not be possible in the same scenario, as there is no connection between the 'Enter the vehicle' and the 'Drive the vehicle' use case modeled in the diagram (*cf.* Figure 6.1. In contrast, open-world semantics follows similar principles to the open-world semantics for feature models defined in [DKMR19]. Within this open-world semantics definition, different unrelated use cases a system executes are executable in the same scenario. In other words, adding the merged scenarios contained in the closed-world semantics is also part of the open-world semantics. Therefore, entering the vehicle and driving the vehicle would become possible in the same scenario, even though there is no connection between the 'Enter the vehicle' and the 'Drive the vehicle' (*cf.* Figure 6.1). Furthermore, the open-world semantics additionally allow the occurrence of systems executing use cases that the use case diagram does not contain. For example, in our simple vehicle use case diagram, the vehicle could close its doors and trunk or maintain its engine even though the original diagram in Figure 6.1 does not contain these use cases. Therefore, assuming a closed-world semantics implies that only the exact contents of the diagram are possible. In contrast, assuming open-world semantics implies that anything not explicitly constrained is possible. In the context of vehicle specifications, both approaches have advantages and disadvantages. Whereas the closed-world semantics has the advantage that only use cases of the diagram may participate in the scenarios [KRW22], it has the disadvantage that an underspecification that may leave room for later design decisions is no longer possible. In late development stages, when the development of the UCDs is nearly completed, this is often the expected behavior, as all intended functionalities are already part of the model. In contrast, when models often change in early development

stages [KRW22], adding model elements, such as use cases, is usually considered to refine the model in these stages. Then, an open-world semantics might be better suited.

6.2 Semantic Foundations of Use Case Diagrams based on [KRW22]

Based on the intuitive view of the semantics of the use case diagrams, this section introduces the formal basics required to apply use case modeling for stakeholder value specification in the automotive industry using CUBE. To this end, this section draws on the previously published semantics definition form [KRW22] and extends this understanding with concepts for integrating systems into the stakeholder value specification. As a basis, the definition made in this section orients on the graphical syntax the OMG standardized in [Obj19], with the restrictions made in Table 6.1. Since tabular specification techniques for use case scenarios as [Coc01] or [Fow04] present as specification methods prior to the UML definition are more related to operating principles for feature definitions, chapter 7 revisits these forms of use case specification in this context, as the method proposed in that usually focus on use case specifications.

Because use case diagrams are outside of the scope for which the OMG specified a formal semantics in the form of the fUML [Obj21], [KRW22] presents one of the few semantic foundations for use case diagrams, which either rely on additional intermediate languages or computation frameworks for semantics definition or instead focus on some aspects of the diagram or its contained elements including use case behavior specification within a single use case. Consequently, neither group of approaches is ideally suited as a foundation for modeling stakeholder values in the system development processes of the automotive industry. Falling in the first category of approaches that use additional intermediate frameworks or languages, [MDB14] aims at defining the semantics of use case diagrams based on the common CSAL. In contrast to the purely mathematical definition of the semantics of use case diagrams in [KRW22], the definition in [MDB14] requires the CSAL as an additional intermediate form for the semantics definition. Moreover, the approach presented in [MDB14] does not differentiate between include and extend relationships. In addition to this approach, [SB06] describes a method for translating use case diagrams into a Z notation scheme. Also following a translation approach, [SKNC17] transforms use case diagrams into Event-B [Abr10]. Even though this approach has its strengths in defining *systems* as contexts, the approach does not consider generalization and extension points, regardless of their relevance in defining stakeholder values. Falling into the second category of approaches that only satisfy parts of the diagram or focus on use case behaviors, [SL03] presents another method for formalizing the behavior of use cases. In contrast to the semantics [KRW22] defines, this approach primarily focuses on formalizing a use cases pre- and post-conditions and not on defining the semantic foundation of all elements in a use case diagram. Moreover, [Whi06] presents an ap-

proach to specify the system behavior during a use case execution precisely. In contrast to [KRW22], which focuses on the specification of the use case diagrams, this approach primarily focuses on use case specifications and combines sequence and activity diagrams to model use case executions. Even though this approach defines precise formal semantics, the syntax and semantics of the diagrams are closer related to activity and sequence diagrams than to use case diagrams. Hence, the semantics definition from [Whi06] better suites the semantics of sequence diagrams presented in [HM08, EFM+05] and activity diagrams [MRR11a, KR18, Kau21] than to the use case diagram semantics presented in [KRW22]. In the same way, [ORC+15] formally defines the semantics of a use case specification without focusing on the use case diagram and its relations between the diagram elements.

Since more than these approaches are needed to build the semantic foundation for modeling stakeholder values as use case diagrams, this section uses the semantic specification from [KRW22] as a basis and extends it with system boundaries to achieve all required concepts in this thesis. To this end, this thesis draws on modeling use case diagrams as relationships between actors, systems, and use cases as presented in subsection 6.1.1. From a semantic perspective, these relationships additionally depend on the assignment state of variables required to evaluate the expressions in extend relationships. As in [KRW22], this thesis assumes that all variables have a Boolean type, so a straightforward generalization to variables of arbitrary finite types is possible.

6.2.1 Basis Notation: Actors, Use Cases, Variables, Boolean Expressions

As a framework of basic notations, this thesis relies on the basis notation [KRW22] defines. Therefore, let in the following $\mathcal{V}$ denote an infinite set of variables, $\mathcal{U}$ denote an infinite set of use cases, $\mathcal{S}$ be an infinite set of systems, and $\mathcal{A}$ denote an infinite set of actors where $\mathcal{U} \cap \mathcal{A} = \emptyset$. Moreover, let $\mathbb{B} = \{t, f\}$ denote the set of Boolean values. Then, as in [KRW22], $val : \mathcal{V} \to \mathbb{B}$ describes the variable assignment function, which each variable to a Boolean value. Further, $\mathcal{V}^{\to}$ represents the set of all variable assignments and $Expr$ denotes the set of all well-formed and finite Boolean expressions over the variables $\mathcal{V}$ contains. Finally, $eval : Expr \times V^{\to} \to \mathbb{B}$ defines a function that maps each expression $e \in Expr$ to its truth value $eval(e, v)$ under the variable assignment $v \in V^{\to}$ as in [KRW22].

6.2.2 Abstract Use Case Diagram Syntax Definition

Based on these basis notations, the following definitions lay the groundwork for the mathematical definition of the abstract use case diagram syntax definition.

Definition 16 (Abstract Use Case Diagram Syntax)**.** *A use case diagram is a tuple* $d = (U, S, A, Abs, Exc, R, G_U, G_A, E, G, Con)$ *where*

- $U \subseteq \mathcal{U}$ *is a finite set of use cases,*
- $S \subseteq \mathcal{S}$ *is a finite set of systems,*
- $A \subseteq \mathcal{A}$ *is a finite set of actors,*
- $Abs \subseteq U \cup A$ *is a set of abstract use cases and actors,*
- $Exec \subseteq U \times S$ *are execution relationships to denote that a systems* $s \in S$ *executes a use case* $u \in U$*,*
- $R \subseteq U \times A$ *are associations between use cases and actors,*
- $G_U \subseteq U \times U$ *is a transitive, reflexive generalization relation over the set of use cases* U*,*
- $G_A \subseteq A \times A$ *is a transitive, reflexive generalization relation over the set of actors* A*,*
- $E \subseteq U \times U$ *is an extend relation between use cases,*
- $G \subseteq U \times U$ *is a guarded extend relation between use cases, and*
- $Con : G \cup U \to Expr$ *maps each guarded extend to its guard and each use case to its precondition.*

This definition extends the original concepts from [KRW22] and introduces systems as the third concept of the diagram (that the SysML diagram definition models as system boundaries). From an abstract viewpoint, a system executes the use cases within its boundary. Consequently, the boundary models two concepts in the same place. First is the system definition, and second is the use case definition. To extend the execution relationship, use cases and systems as participants of this relationship are needed. Thus, this thesis takes over the set U from [KRW22], which contains all use cases the diagram models. Then, the set S introduces a new set that contains all systems modeled in the diagram as boundaries. Similarly, as in [KRW22], the set A contains all actors, and the set Abs contains all abstract use cases and actors the diagram considers. Next, $Exec$ is a new concept this definition introduces, in contrast, to [KRW22], which the definition requires to allocate use cases to the system that executes the use case. As in [KRW22], the remaining sets denote the following relationships: R contains the associations between use cases and actors. Then, G_U is required as the use case generalization relation where $(u, v) \in G_U$ denotes that a use case u is a specialization of a use case v, respectively that the use case v generalizes the use case u as in [KRW22]. In the same way, the set G_A represents the actor generalization relation where $(a, b) \in G_A$ denotes that the actor a is a specialization of the actor b, in other words that the actor b generalizes the actor a [KRW22]. To express the extend relationships, the set E represents the extend relation

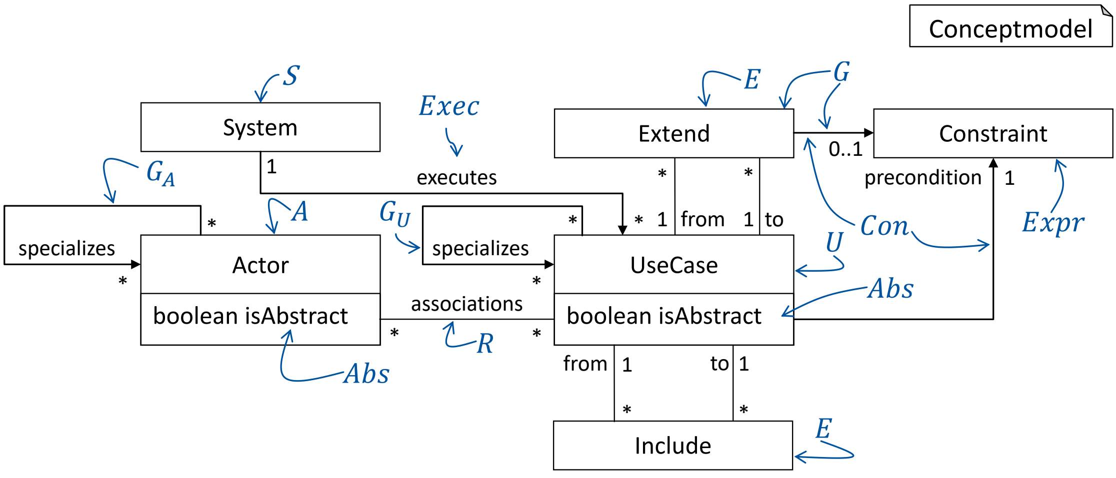


Figure 6.2: A class diagram to define the abstract syntax of the use case diagram language based on [KRW22].

between use cases where a tuple $(u, v) \in E$ represents that the use case u extends the use case v. It is important to note that the extend relation E models is unguarded. Whereas in contrast, the set G contains the guarded extend relations between use cases where $(u, v) \in G$ denotes that the use case u extends the use case v. Moreover, Con represents another function, which maps each guarded extend $e \in G$ to its guard $Con(e)$, a Boolean expression [KRW22].

As [KRW22] explains, adding the set I for include relationships is possible but not required, as a guarded extend $e = (u, v) \in G$ with $Con(e) = t$, *i.e.,* an always satisfied guard, can model each include association (v, u). As a result, whenever v is executed, u must also be executed, which precisely defines the idea of an include relationship. Therefore, this thesis also abstracts from this case to shorten the definition. Finally, function Con maps each use case $u \in U$ to its precondition $Con(u)$. Even though this concept is not yet required for stakeholder value specification, this concept becomes handy for the operating principle specification in chapter 7 and is therefore also taken over as a concept from [KRW22].

6.2.3 Mapping the Abstract Syntax to the Diagram Elements for Stakeholder Value Models in SysML Use Case Diagrams

Based on the previously defined concrete syntax (*cf.* Table 6.1) and the abstract syntax definition (*cf.* Def. 16), this section aims at combining both concepts in a unified understanding. To this end, Figure 6.2 depicts a class diagram defining the abstract syntax of the use case diagram language based on the previous work published in [KRW22]. Apart

from this influence, the diagram also refers to the UML definition of the abstract syntax of use case diagrams [Obj17] to which the SysML [Obj19] also refers as source of inspiration. As a result, the mapping reveals that the abstract syntax definition (*cf.* Def. 16) corresponds to all elements defined in the reduced abstract syntax from Table 6.1 including `UseCase` (U), `Actor` (A), `Extend` (E), `Include` E, and `Generalization` (G_A for actor generalization and G_U for use case generalization) and further extends these concepts with a `Constraint` that can extend use cases with preconditions that Table 6.1 currently not requires but will be introduced later in the context of operating principle definitions, and for expressions in extension points. Note that the set E realizes both the extend and the include relations, as the include relation is a particular case of an unguarded extend relationship as discussed in the context of Def. 16.

In contrast to the prior definition in [KRW22], this thesis additionally contains the system boundary in its definition and, therefore, can no longer express this essential aspect of the use case diagram. However, some aspects that use case diagrams for stakeholder value definitions do not contain are still not contained in the definition this thesis presents. Thus, for example, the abstract syntax still does not support associations between actors and cardinalities on the association ends between use cases. Hence, the shortcomings from [KRW22] concerning these aspects still apply to this thesis. However, as these elements are optional in the presented method for using use case diagrams to express stakeholder values, these shortcomings have a neglectable influence on the results of this thesis.

6.2.4 Semantic Use Case Domain

As the abstract syntax introduces the system and its execution of a use case into the diagram, an extended semantic domain definition is required. Consequently, the following definition extends the semantic domain definition from [KRW22] and introduces the system and the use case execution into the semantic domain.

Definition 17 (Semantic Use Case Diagram Domain)**.** *A scenario is a tuple* (val, Use, Rel, Exe) *where*

- $val \in \mathcal{V}^{\rightarrow}$ *is a valid variable assignment,*
- $Use \subseteq \mathcal{U}$ *is a finite set of use cases, and*
- $Rel \subseteq Use \times \mathcal{A}$ *is a finite set of links between use cases of the scenario and actors.*
- $Exe \subseteq Use \times \mathcal{S}$ *is a finite set of execution relations between the use cases of the scenario and the execution of the system of these use cases in this scenario.*

In conclusion of this definition, a scenario (val, Use, Rel, Exe) denotes that systems execute the use cases contained in Use according to Exe in which the actors participate

according to the relation Rel under the circumstances defined by the variable assignment val as in [KRW22]. Therefore, a scenario contains a set of use cases Use but no dedicated set of systems or actors because even tough use cases require a system to execute, a system cannot exist in a scenario without executing a use case [KRW22]. In the same way, a system may execute a use case without involving any actor. Hence, actors cannot exist in scenarios without being linked with use cases [KRW22]. As a deviation from [KRW22], Def. 17 additionally considers the execution relation $Exec$ to differentiate between the systems executing a use case in a scenario.

6.2.5 Formalization of the Use Case Diagram Semantics

As for the semantic domain, the formalization of the semantics from [KRW22] is no longer applicable after introducing systems into the use case definition. Hence, this section extends the definitions from [KRW22] to include systems in formalizing the semantics.

Thus let $d = (U, S, A, Abs, Exc, R, G_U, G_A, E, G, Con)$ be an arbitrary but fixed use case diagram according to Def. 16. Next, this section also relies on the auxiliary functions from [KRW22] defined as follows:

- For $u \in U$ this thesis defines the set of actors that are explicitly or implicitly (via generalization) associated to u by $Act_d(u) = \{a \in A \mid \exists v \in U : (u, v) \in G_U \wedge (v, a) \in R\}$ as in [KRW22].
- For $V \subseteq U$, this thesis defines the set of (explicit or implicit) associations between the use cases in V and actors in A by $R_d(V) = \{(v, a) \in V \times A \mid a \in Act_d(v)\}$ as in [KRW22].
- This thesis writes Act instead of Act_d and R instead of R_d if d is clear from the context.

Even though this thesis not only requires actors but also systems to participate in a system execution, the execution of a use case $u \in U$ under a variable assignment $val \in \mathcal{V}^{\rightarrow}$ can still rely on the definition from [KRW22], as the execution of the use case does not primarily depend on the associated actors and executing systems but on the (transitive) execution of (include and) extend relations with the satisfied guards under val, which is independent of the actors and systems that participate in the execution. Therefore, as in [KRW22], the execution can abstract from the actors associated with the use cases and the systems required from the execution, and all unguarded extends between use cases.

Definition 18 (Use Case Execution [KRW22])**.** *Let $u \in U$ be a use case, and let $val \in \mathcal{V}^{\rightarrow}$ be a variable assignment. The execution of u in d under val is the smallest set $exec(u, val)$ satisfying the following rules:*

1. *If $eval(Con(u), val) = t$ then $u \in exec(u, val)$.*

2. *If* $w \in exec(u, val)$ *and* $(v, w) \in G$ *and* $eval(Con(v), val) = t$ *and* $eval(Con((v, w)), val) = t$*, then* $v \in exec(u, val)$*.*

As the first rule states, if the precondition of the use case u is satisfied under the variable assignment v, then the use case u is contained in the execution [KRW22]. Moreover, whenever an execution includes a use case w, and another use case v extends this use case w, then v is also included in the execution if the guards of the extend association (v, w) and the precondition of v are satisfied under the assignment val [KRW22], as the second rule states.

To determine the set of possible executions of a use case under the generalization relation of use cases and under the unguarded extend relation between use cases, [KRW22] introduces the notion of a closure, which is also applicable in the context of the stakeholder value definition this thesis considers. The intuition behind this closure is twofold. First, it assumes that each generalized use case is replaceable by all specializations; second, unguarded execute relations may (or may not) lead to the execution of the related use case. Following the first intuitive description, the system executes the specialization instead of the generalization when replacing the generalization with the specialization. During this process, all use cases included in the specialization of the general use case are required in the execution under the variable assignment [KRW22]. In the same way, the execution of a use case associated via an unguarded extend relation also requires including all use cases contained in the execution of this use case under the variable assignment [KRW22]. As for the auxiliary functions, the closure is independent of the actors and the executing systems.

Definition 19 (Use Case Closure [KRW22])**.** *Let* $u \in U$ *be a use case, and let* $val \in \mathcal{V}^{\rightarrow}$ *be a variable assignment. The closure of* u *in* d *under* val *is the smallest set* $closure(u, val)$ *satisfying the following rules:*

1. *If* $exec(u, val) \neq \emptyset$*, then* $exec(u, val) \in closure(u, val)$*.*
2. *If* $C \in closure(u, val)$ *and* $w \in C$ *and* $(v, w) \in G_U$ *and* $eval(Con(v), val) = t$*, then* $(C \setminus \{w\}) \cup exec(v, val) \in closure(u, val)$*.*
3. *If* $C \in closure(u, val)$ *and* $w \in C$ *and* $(v, w) \in E$ *and* $eval(Con(v), val) = t$*, then* $C \cup exec(v, val) \in closure(u, val)$*.*

As described in [KRW22], the first rule of the use case closure auxiliary function expresses that the execution of the use case under the assignment is an element of the closure if the execution is possible, as its precondition is satisfied. Next, the second rule as in [KRW22] states that if (1) an execution C is already in the closure, (2) the execution also contains a use case $w \in C$, (3) the use case v is a specialization of the use case w, (4) and the precondition of the use case v is satisfied under the assignment val, then the resulting execution from replacing w with v and adding the use cases contained in the execution of v under val is also a valid execution that the closure must contain [KRW22].

Before completing the formalization of the semantics, the notion of the scenario is required to add the variable assignments, the systems executing the use cases, and the actors involved in the execution into a relationship under this variable assignment. As in[KRW22], if a use case v is a specialization of a use case w, then all actors associated with w are also implicitly associated with v.

Definition 20 (Use Case Scenario Definition)**.** *Let $u \in U$ be a use case and let $val \in \mathcal{V}^{\rightarrow}$ be a variable assignment. Then the smallest set $scn(u, val)$ that satisfies the following conditions defines the scenarios of of u in d under val:*

1. *If $C \in closure(u, val)$, then $(val, C, R(C), Exec(C)) \in scn(u, val)$.*
2. *If $(val, Use, Rel, Exe) \in scn(u, val)$ and $(v, a) \in Rel$ and $(b, a) \in G_A$, then $(val, Use, (Rel \setminus \{(v, a)\}) \cup \{(v, b)\}, Exe) \in scn(u, val)$.*

As an extension of [KRW22], the first rule adds the related actors of the use cases in the closure to the scenario and the related systems to the execution relation in the semantic domain. Then, as in [KRW22], the second rule states that all actors are replaceable by specializations of this actor.

Finally, the set of all scenarios of the use cases in the use case diagram under all possible variable assignments where the scenarios neither contain abstract actors nor abstract use cases defines the semantics of a use case diagram [KRW22].

Definition 21 (Use Case Diagram Semantics)**.** *$[\![d]\!] = \{(val, Use, Rel, Exe) \in scn(u, val) \mid val \in \mathcal{V}^{\rightarrow} \wedge u \in U \wedge Use \subseteq U \setminus Abs \wedge Rel(Use) \subseteq A \setminus Abs\}$ defines the semantics of the use case diagram d.*

With these definitions, this thesis provided the semantic groundwork for using use case diagrams for stakeholder value specification. As [KRW22] additionally provides a concrete textual syntax for use case diagrams, the following sections additionally present this syntax and the required extensions for using system definitions in this syntax before the remainder of this section introduces extensions and methods for applying these concepts in the context of automotive system development.

6.2.6 A Concrete Textual Syntax for Stakeholder Value Modeling

As this thesis does not only focus on the definition but also on the application of system models in the automotive industry, this section additionally provides a concrete textual syntax for the notation this thesis adds to [KRW22]. Therefore, this section summarizes the usage of the concrete syntax from [KRW22] and provides the required grammar extensions to introduce the required extensions based on a proposal from [Bla23], which provided a concrete syntax for system boundaries in the context of use case scenario specifications and chapter 7 revisits later for operating principle specification.

The original prototype implementation from [KRW22] implemented the concrete textual syntax of the use case diagram language using the language workbench MontiCore [HKR21]. The resulting language is publicly available as a stable release in a GitHub repository the MontiCore group provides (https://github.com/MontiCore/ucd) and summarized in this section before the extension with system boundaries. For a more detailed overview, the GitHub repository or [KRW22] provides additional insights. This stable release guarantees that all extensions are conservative, which is not the case for the extensions this thesis requires to define stakeholder values using use case diagrams. Therefore, this thesis starts with the original language and uses MontiCore to extend this language. The following listing provides an excerpt of the grammar.

```
MCG
1 grammar UCD {
2     // A use case diagram.
3     UseCaseDiagram = "usecasediagram" Name "{"
4         UCDElement*
5         "}";
6
7     // Elements contained in use case diagrams.
8     interface UCDElement;
9
10    // A use case.
11    symbol UCDUseCase implements UCDElement =
12    ["abstract"]? Name ("[" Expression "]")?
13    ("specializes" sup:(Name@UCDUseCase || ",")+)?
14    ("extend" (UCDExtend || ",")+ )?
15    ("include" incl:(Name@UCDUseCase || ",")+ )?
16    ";";
17
18    // An extend or an include used in UCDUseCase.
19    UCDExtend =
20    Name@UCDUseCase ("[" Expression "]")?;
21
22    // An actor.
23    symbol UCDActor implements UCDElement =
24    ["abstract"]? "@" Name
25    ("specializes" sup:(Name@UCDActor || ",")+)?
26    ("--" uc:(Name@UCDUseCase || ",")+)?
27    ";";
28 }
```

Listing 6.1: Original MontiCore grammar for use case diagrams from [KRW22].

This grammar implements the reduced abstract syntax that subsection 6.2.2 presents and provides a textual notation for the graphical representation in Table 6.1 accord-

ing to the MetaModel from Figure 6.2 as follows: The non-terminal UseCaseDiagram defines the context of the use cases diagram and provides a name to refer to this diagram. Inside a use case diagram, the MCG defines the interface UCDElements, an extendable interface for defining allowed elements within the diagram. To define the name-giving element of the diagram, the interface implementation UCDUseCase defines the non-terminal for the use case together with the allowed relationships. In the context of the extend relation, the non-terminal UCDExtend provides a mechanism to define expressions for the extend relationship. As for use cases, the non-terminal UCDActor defines the actors and their relations within the diagram as another implementation of the interface UCDElement.

Extending this existing implementation with systems and their boundaries as diagram elements is possible using the exact mechanisms. First, a new grammar that extends the original use case grammar is required to lay the groundwork. Then, a new implementation of the UCDElements in the context of this grammar becomes possible. Therefore, the following grammar describes the required extensions based on a proposal for the concrete textual syntax in [Bla23].

MCG

```
1 grammar StakeholderValueUCD extends UCD {
2     // A the system (boundary).
3     symbol UCDSystem implements UCDElement =
4     "system" Name "{"
5         UCDSystemElement*
6         "}";
7
8     // An element within the system boundary
9     UCDSystemElement =
10    (Name@UCDUseCase ";") | UCDUseCase;
11 }
```

Listing 6.2: Extenstion of the use cased diagram grammar based on a proposal of the concrete textual syntax from [Bla23].

Since the graphical representation of the system (boundary) Table 6.1 presents is twofold, the grammar provides two non-terminals that implement the system and its boundary in the context of a use case diagram for stakeholder value specification. First, the non-terminal UCDSystem defines the system and a scope in which use cases can be executed in braces. To denote the executable use cases, the non-terminal UCDSystemElement allows defining a list of use cases the system executes. Using this extension, the following model in textual syntax would describe the running example from Figure 6.1 as follows.

UCD

```
1 usecasediagram SimpleVehicle {
```

```
2     abstract @VehicleOccupant --
3     EnterTheVehicle,
4     ExitTheVehicle,
5     ControlTheVehicleAccessory;
6
7     @Passenger specializes VehicleOccupant;
8     @Driver specializes VehicleOccupant --
9     DriveTheVehicle;
10
11    system Vehicle {
12        EnterTheVehicle include OpenTheDoor;
13        ExitTheVehicle include OpenTheDoor;
14
15        abstract ControlTheVehicleAccessory;
16        ControlTheClimateControl specializes
              ControlTheVehicleAccessory;
17        ControlTheEntertainmentSystem specializes
              ControlTheVehicleAccessory;
18
19        PerformAntiLockBarking extend DriveTheVehicle;
20    }
21 }
```

Listing 6.3: An example use case diagram representing the example from Figure 6.1 in textual notation.

Since the non-terminal `Name` from the MontiCore library does not allow spaces, all spaces are replaced using a camel case. If this is required in concrete projects, alternative notations could be quickly introduced using the MontiCores extension mechanism to allow alternative names or representations that allow spaces *e.g.,* by writing brackets around names.

6.3 Feature-Driven Use Case Development

Although the previously presented concepts depend on the applied modeling methodology, some extensions can simplify the application in the context of CUBE. Since the stakeholder elicitation serves as a foundation for deriving a set of features, which is an essential aspect of the feature-driven development in CUBE [GKS+21], the derivation of features must also reflect in the application of use case diagrams for stakeholder value elicitation.

As a running example from the automotive industry, Figure 6.3 presents a diagram from the Delta project, which contains most of the concepts for which this section provides an extension as a part of a feature-driven use case development method for stake-

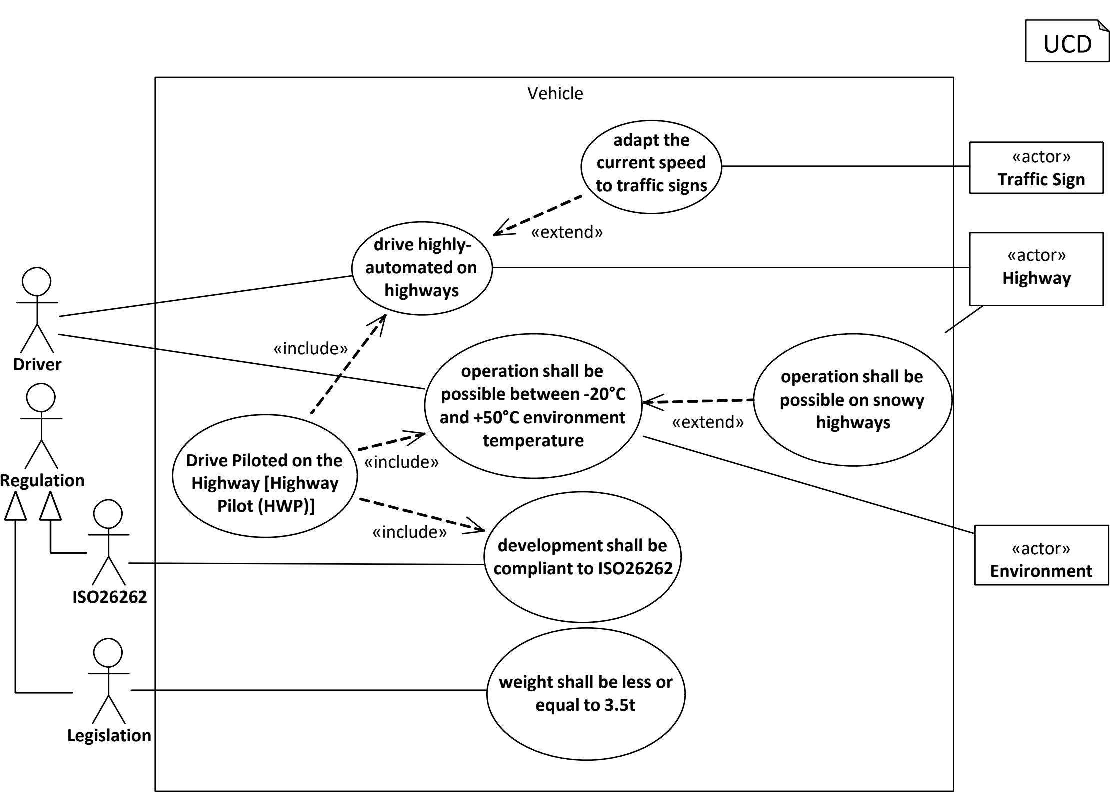


Figure 6.3: A stakeholder value use case diagram modeling the stakeholder value of a Highway Pilot Feature developed in the Delta project.

holder value elicitation.

To derive the features in CUBE, independent of the stakeholder value elicitation method, the systems engineers designing the system must select the features strategically to align with the stakeholder needs and requirements. Hence, the systems engineers must address all functional stakeholder values by expressing them in the feature. In the example diagram in Figure 6.3, this is expressed by the 'Drive Piloted on the Highway [Highway Pilot (HWP)]' use case, which describes a feature the vehicle provides to address most of the use cases in the diagram. By grouping the use cases into features, the method ensures a focused approach in the subsequent development phases, streamlining efforts toward the implementation of functionalities that directly contribute to satisfying stakeholder expectations and increasing reusability. As the use case that reflects the stakeholder value and the system feature that addresses this function are split, the identical use cases can be grouped differently in different systems, which increases the reusability of the individual use cases and, therefore, increases the modeling efficiency of the involved systems engineers.

On the other hand, even though the formulation within this thesis suggests that all features are functional, systems also offer other characteristics, such as attributes that do not provide functionality to an actor [WRF+15]. In the considered running example in Figure 6.3, this can be seen at the 'weight shall be less or equal to 3.5 t' use case, which does not precisely describe functionality that the system provides, but rather a constraint for the system under development. Applied to use case diagrams that specify the stakeholder value, this means that the set of derived features must realize and address all stakeholder values that address system functionalities and respect other system characteristics that are not necessarily realized by system behaviors. Consequently, the system must provide features, among other means, to implement all use cases that represent the functional stakeholder values.

6.3.1 Formulating System Functionalities in Use Case Notation

Since a feature is an independent end-to-end functionality of the system that benefits at least one stakeholder (*cf.* Def. 7), and use cases in this method typically express these needs, a feature must address one or multiple use cases. Because this thesis focuses on technical features, which are characterized by the fact that they define functional aspects of the system, a trace (directly or indirectly) between every use case and at least one feature must exist. Accordingly, a feature includes the use cases it realizes. By this, the execution of the feature implies addressing the stakeholder values expressed in the system's use cases. As the feature is an essential aspect of the system, all use cases and the feature must be executed by the same system or its subsystem. Moreover, for a successful system design specification, all system use cases must be traceable to system features. As stated before in the running example of this section, Figure 6.3

presents, the use case 'Drive Piloted on the Highway [Highway Pilot (HWP)]' expresses the considered feature. Note that the '[Highway Pilot (HWP)]' denotes a marketing name of the functionality, as also used in the SPECTRE DSL for natural language requirements (*cf.* section 5.3).

6.3.2 Formulating other System Characteristics in Use Case Notation

As a diagram primarily for behavior modeling [Obj19] (*cf.* Figure 2.4), most use cases a use case diagram contains typically address the functional characteristics of a system. According to the stakeholder definition in Def. 9, stakeholders not only request functionalities, but also characteristics concerning the system performance, specific qualities, or constraints [Gli07]. Therefore, this thesis additionally introduces a particular form of use cases, the boundary condition use case, to model characteristics as use case bubbles in stakeholder value models, although later abstraction levels might not realize these use cases as behaviors.

Definition 22 (Boundary Condition Use Case)**.** *A boundary condition use case describes performance concerns, specific qualities, or constraints to the solution space, the system development project, or the required system development process raised by the associated actors.*

In the example from Figure 6.3, the use cases 'operation shall be possible between $-20°C$ and $+50°C$ environment temperature', 'operation shall be possible on snowy highways', 'development shall be compliant to ISO26262', or 'weight shall be less or equal to $3.5t$' represent use cases of this form. Moreover, connected actors indicate an involvement of the actor in the fulfillment of the boundary condition, as for example in the case of the ISO26262, which sets the boundaries for the development of safe systems, or the legislation which formulates the weight limits in the example in Figure 6.3.

6.3.3 Refining Associations Based on the Role of the Actor

As, for example, stated in [Coc01], or [Fow04], actors can have different roles when participating in use cases. Consequently, the literature on use case modeling most commonly divides actors into two categories: primary and secondary. The primary actor, often called simply called actor, describes an actor that gets a value from the use case [Fow04] or interacts with the system to achieve its goal [Coc01]. In contrast to this primary actor, [Coc01] defines the secondary actor as an external actor who provides a service to the system under development.

In automotive system development, these definitions bear several significant issues and risks when applying these concepts. Regarding different automotive examples for

Table 6.2: Further UCD elements to aid in a feature definition based on stakeholder values according to feature-driven CUBE methodology.

Actor type	Expects UC	Involved in UC execution
Primary	x	x
Secondary	-	x
Tertiary	x	-
Unintended	-	-

the primary actors, neither definition seems applicable in these cases. First and foremost, regarding the definition from [Fow04], it is evident that not all stakeholders of automotive systems inherently benefit from the system. For example, the environment, or bystanders of a vehicle with combustion engines, are typically (negatively) influenced by the noise and toxic emissions of combustion engines. Moreover, regarding the definition from [Fow04], they are also not necessarily interacting with the system or aware that it interacts with the system. In the same example, the environment in general and the bystander in particular experience a negative influence by the combustion engine; however, they have no influence on the emissions. Consequently, the channel between the combustion engine and the environment is unidirectional. Regarding the secondary actor, not all actors involved in the execution of automotive systems inherently provide services for the system. A traffic sign, for example, participates in a traffic sign execution and might provide valuable information for a traffic sign recognition system; retrieving this information from the sign, however, is a task for the system itself in most concurrent traffic signs that are painted to the sign. Hence, the traffic sign only participates inactively in executing a traffic sign recognition use case. Even further from this definition, a regulatory entity might require the system to address a specific use case even though it is not involved in the system's execution nor influenced by the system execution.

To better reflect this aspect, this thesis uses a different definition of roles in which actors interact with the system, based on whether the actor expects the use case and its active involvement in the use case execution, which results in the matrix Table 6.2 reflects. Depending on whether or not the actor expects a use case or is involved in its execution, an actor can be a primary, secondary, tertiary, or unintended actor for this specific use case. A primary actor is defined as follows.

Definition 23 (Primary Actor)**.** *A primary actor is actively involved in the use case and expects the system to execute it.*

Because of this involvement and expectation, it represents the same role as [Coc01] and [Fow04] intend in most cases when primary actors are specified. In the context of the example in Figure 6.1 and Figure 6.3, the 'Driver' is a primary actor, for example,

in the 'Drive the Vehicle' use case in Figure 6.1.

In contrast to this definition of the primary actor, the secondary actor does not expect the system to execute the use case but is somehow involved in its execution, as the following definition states.

Definition 24 (Secondary Actor)**.** *A secondary actor is involved in executing the use case but does not expect the system to execute this particular use case to satisfy its needs.*

The previously mentioned traffic signs, or bystanders fall into this category, as both actors (which are also stakeholders in this case) are affected and therefore involved in the execution of the use case but do not have any expectations on the system to have this use cases, but might constraint the system in executing the system *e.g.,* by reducing unwanted influences. In the example that Figure 6.3 presents, the 'Traffic sign', 'Highway', and 'Environment' participate in all related use cases as secondary actors.

The 'Regulation' in this example, however, represents another type of actor, which expects the system to execute a use case but does not participate in its execution, as the following definition of the tertiary actor reflects.

Definition 25 (Tertiary Actor)**.** *A tertiary actor is not involved in the execution of the use case but expects the system to achieve a particular use case.*

Consequently, the 'ISO26262' and the 'Legislation' from the diagram in Figure 6.3 participate as tertiary actors in all associated use cases. When thinking of other examples of this kind of actor, not only regulatory entities but also service providers, owners, or other kinds of shareholders often expect characteristics of the vehicle even though they are not involved in the execution of the use case. For instance, the owner of a large fleet of rental vehicles is typically not involved in maintaining the vehicle. However, use cases that reduce the maintenance cost of all vehicles in the fleet are most beneficial for the owner, as it reduces the operation cost of the vehicles in the fleet.

Finally, the combination of not expecting the system to execute a specific use case and an explicit non-involvement requirement is sometimes required for system specification. Although it seems counterintuitive at first sight, this kind of use case with hostile intend [Ale03] has its place in systems engineering as the so-called misuse case [SO01a, SO01b]. This use case plays a vital role during the elicitation of security requirements [SO05], as a systems engineer concerned with the security of the system is often more interested in what the system shall not do when encountering an unintended actor or mis-actor [SO01a] than what it should do when everything is working as expected. Therefore, this fourth type of actor is defined as an unwanted or unintended actor as follows.

Definition 26 (Unintended Actor)**.** *An unintended actor may explicitly neither participate in the use case execution nor be able to expect a system's functionality.*

A criminal as an unwanted stakeholder of the system might, for example, want to steal the vehicle, for which the vehicle should provide countermeasures such as door locks that prevent the criminal from opening the door or an alarm system that informs the environment from a potential risk that a criminal is currently stealing the vehicle might be required as a countermeasure. To address these cases, [SO01a] proposes this kind of actor and introduces the misuse case and several additional relations to regular use cases to model these cases. Integrating all these concepts into the semantic definition goes beyond the scope of stakeholder value elicitation. This aspect is just mentioned for the actor, as it became relevant in evaluating the method in the context of the Zeta industry project. However, an implementation that addresses these conditions in the context of the semantic domain remains for future works.

As the application of these concepts in the Gamma, Epsilon, Delta, and Zeta projects showed, some additional best practices for how to relate use cases with actors are helpful when designing automotive systems. First and foremost, all actors used should be known on the currently shown decomposition level. Therefore, all actors a system engineer associates with a use case should be part of the system environment. Suppose the actor is, for example, a subsystem of the system under consideration (or part of another system in the environment). In that case, this might indicate that the currently considered use case should not be modeled on the current decomposition level and delegated to a later development phase. Next, when multiple modelers work on the same model, different modelers often use synonyms for the same actor. For example, the vehicle occupant and the passenger might be used differently as synonyms in different models. As a risk mitigation method, it is beneficial to define the actors with an informal description in a shared model library, which aids the modelers in choosing the right actors. Finally, when graphically modeling use case diagrams, placing primary actors above or left of the system boundary and the use case and secondary, tertiary, and unintended actors to the right or below the diagram is sometimes beneficial. By this, when a reader reads the diagram from the top left to the bottom right, first the primary actors, then the system that executes the use case, and then all secondary, tertiary, and unintended actors are read in the same order as a natural language sentence would formulate these concepts (*cf.* subsection 6.4.1). Though sticky figures, bubbles, and rectangles seem intuitive at first glance, some inexperienced readers are sometimes alienated by the idea that sticky figures also represent non-human actors. For that reason, the SysML standard [Obj19] introduced the rectangle notation (*cf.* Table 6.1) as an alternative representation for actors. Hence, it can be a good practice to represent non-human actors in rectangle notation and human actors as sticky figures when communicating with stakeholders unfamiliar with use case diagrams.

6.3.4 A Refined Use Case Syntax

Based on these observations, Table 6.3 extends the basic concepts for use case modeling from Table 6.1 to integrate the additional concepts required for a feature-driven use case development as part of a stakeholder value elicitation.

Table 6.3: Further UCD elements to aid in a feature definition based on stakeholder values according to feature-driven CUBE methodology.

Element Name	Notation	Description
	Obligatory Diagram Elements	
Feature Use Case	«feature» **Feature**	A feature is an independent end-to-end functionality of the system that benefits at least one stakeholder and addresses one or multiple use cases. Every use case must be traceable (directly or indirectly) to features via include relationships.
	Optional Diagram Elements	
Boundary Condition	«boundary condition» **Condition**	A boundary condition use case describes performance concerns, specific qualities, or constraints to the solution space raised by the associated actors.
Misuse Case	«misuse» **Misuse Case**	According to [SO05], a misuse case represents a sequence of actions, including variants, that a system performs when interacting with unintended actors of the systems and causing harm to some stakeholders if the sequence executes without prevention.

Table 6.3: Further UCD elements to aid in a feature definition based on stakeholder values according to feature-driven CUBE methodology.

Primary Actor Association	Primary Actor «primary» Use Case	A primary actor association indicates that the actor is actively involved in the use case and expects the system to execute the use case. Since primary actors are the typical unlabeled use cases, they are typically primary actor associations.
Secondary Actor Association	Secondary Actor «secondary» Use Case	A secondary actor association indicates that the actor is involved in executing the use case but does not expect the system to execute this particular use case to satisfy its need.
Tertiary Actor Association	Tertiary Actor «tertiary» Use Case	A tertiary actor association indicates that the actor is not involved in executing the use case but expects the system to achieve a certain use case functionality or boundary condition.
Misuse Association	Unintended Actor «misuse» Use Case	A misuse association indicates that the actor associated with the use case explicitly may not participate in the use case execution nor expect the system's functionality.

Even though the extensions made in Table 6.3 influence the syntax of the use case diagrams created using this method, the changes have a neglectable influence on the semantics. Since a feature use case is a valid use case on its own, even without the stereotype, introducing this stereotype is a semantics-preserving syntax extension from a process perspective with no influence on the semantics of the diagram. Although introducing a stereotype «boundary use cases» greatly stretches the notion of execution in the semantic definition, the interpretation of the result is retained in the semantic domain. Since a system exhibits these characteristics within its life cycle when it is used and these characteristics are associated with the external actor that appears here as a stakeholder, the meaning of the result, that the actor executes the use case under a given relation, is retained in the semantic domain, since the underspecification of the relation delegates the concrete interaction to a later phase of the system development.

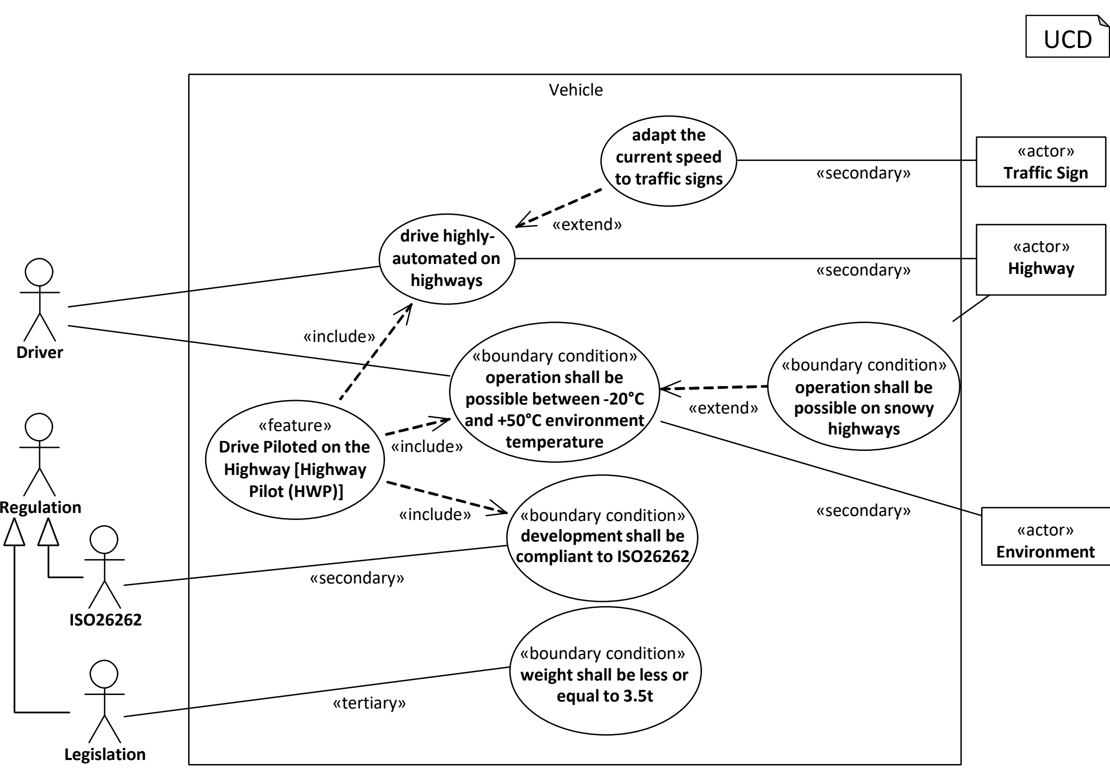


Figure 6.4: An extended version of the diagram from Figure 6.3 that integrates the syntax extensions from Table 6.3 into the diagram.

Hence, this extension also preserves semantics. The same argumentation also holds true for introducing the primary, secondary, and tertiary actor associations between use cases and actors, which further specify the relation kind. Therefore, introducing the primary, secondary, and tertiary actor is a refinement of the semantics definition in section 6.2. Finally, the misuse association is an inverted relation. Thus, in closed-world semantics, this relation must be removed prior to the semantic analysis, whereas in open-world semantics, this relation must be actively removed from the semantic domain. Hence, introducing this association has a minor impact on the definition of semantics. As this case only represents a corner case in the models this thesis primarily considers, this aspect remains for further analysis in future works.

6.3.5 Feature Driven Use Case Development of a Simple Vehicle

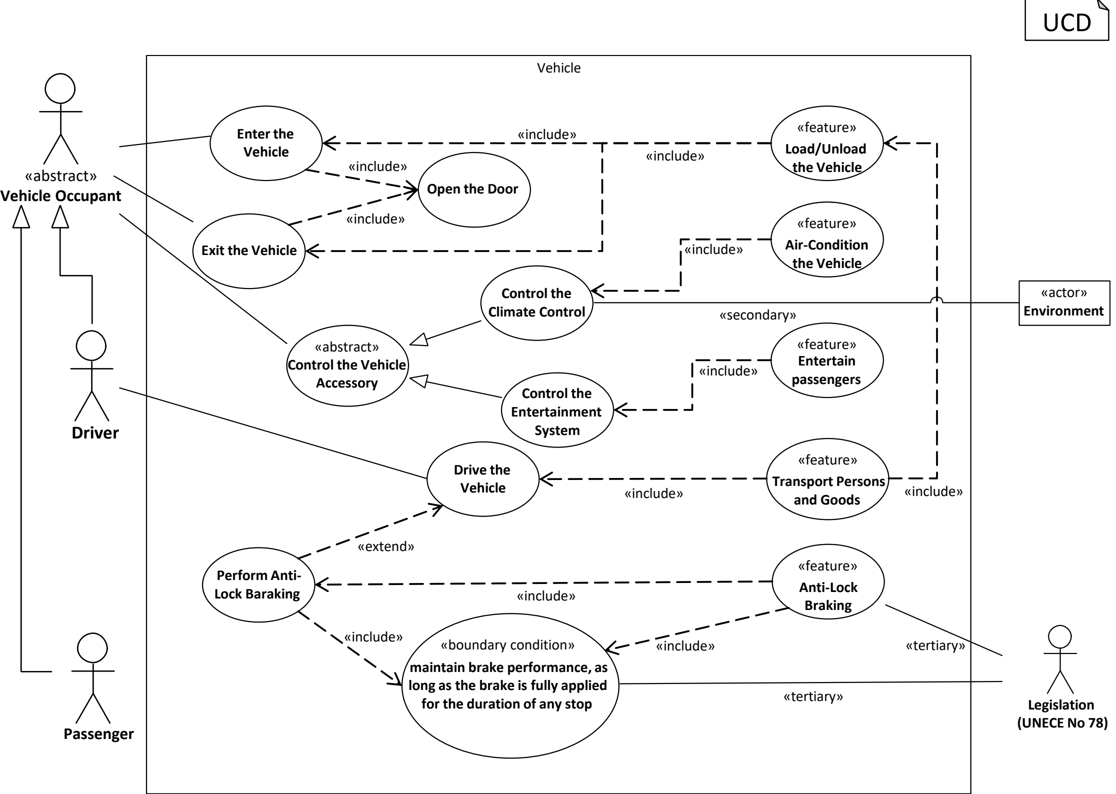


Figure 6.5: Automotive driving use case diagrams based on an initial example from [FMS14a].

Having the extensions Table 6.3 provides, the stakeholder values model from Figure 6.1 serves as an input to derive the vehicle features. To this end, Figure 6.5 introduces an example set of five features into the stakeholder value specification that addresses all stakeholder needs in the initial diagram in Figure 6.1. As a further extension of the initial diagram, this extended diagram also includes the environment as an influence on the climate control use case. It adds the UNECE No. 78 [fEUNEC04] as a regulatory entity with to example need. First, the vehicle may execute an anti-lock braking use case, and second, the vehicle maintains the brake performance as long as the brake is fully applied for the duration of any stop, which is stated as a requirement within the regulation. The five features Figure 6.5 presents describe the following functionalities:

1. The 'Load/Unload the Vehicle' feature serves to combine all functionalities the vehicle requires to enable vehicle occupants and their goods to enter and exit the vehicle. As the semantics definition from subsection 6.2.5 states, all use cases that are related to this use case via include or extend relations are included in the feature as well, thus 'Open Door' is included as well.
2. The 'Air-Condition the Vehicle' feature addresses all functionalities concerned with the vehicle's air conditioning. Thus, the use case 'Control Climate Control', which is a specialization of the 'Control the Vehicle Accessory' use case, is included in this feature.
3. The 'Entertain passengers' feature aims to address all use cases related to passenger entertainment. Consequently, the 'Control Entertainment System' as a specialization of the 'Control the Vehicle Accessory' use case is naturally part of the execution of this feature and, therefore, included in this feature. Note that two different features with seemingly unrelated implementations realize the abstract 'Control the Vehicle Accessory' use case in this example, which should demonstrate that the semantics and the inheritance of use cases defined in this thesis have no impact on the behavior specification. Since the semantic domain of use cases, associated actors, and executing systems is independent of the use case behavior specification, which chapter 7 defines later, use cases with completely different behaviors can specialize the same use case, as long as the associated actors and systems remain the same.
4. The 'Transport Persons and Goods' represents a typical transportation task many vehicles execute. Naturally, this feature includes the 'Drive the Vehicle' use case and the 'Load/Unload the Vehicle' use case. This feature should demonstrate that features may include other features to reduce the overall specification effort by enabling the systems engineers to model feature decomposition and reuse.
5. The 'Anti-Lock Braking' feature includes the 'Perform Anti-Lock Braking' as well as the 'maintain brake performance' boundary condition and is legally required for some vehicles on the European market according to UNECE No 78 [fEUNEC04] as

an extension of the 'Drive Vehicle' use case. With this feature split, the example should demonstrate three main aspects of feature relationships. First, the feature also (under some underspecified conditions) implies the execution of the 'Drive the Vehicle' and, therefore, the 'Transport Persons and Goods' feature, as the use case 'Perform Anti-Lock Braking' is unguarded extend relation and therefore this feature and all included features are part of the closure according to the definition in Def. 19. Second, tertiary actors might explicitly require the implementation of a feature. Third, features can also include boundary conditions, and it can be a good practice to explicitly connect features to other use cases that would nevertheless be in the closure to make relationships explicit for inexperienced readers.

Apart from the representation in Figure 6.5, it might also be a good practice to split the specification of stakeholder values later to feature specific diagrams, as it is easy to imagine that a single use case diagram for a massive system as a vehicle is no longer understandable for a human reader. Although large use case diagrams are often not published in scientific papers because of the limited available space, [JGS+20] presents an example of a comparably large use case diagram that specifies the stakeholder value of a VTOL system, which becomes more accessible to read in the follow-up publication in [JGW+21] where the authors split the original diagram to feature specific diagrams before defining feature specific operating principles according to the CUBE methodology.

As the next step of the system design, the systems engineers continue the stakeholder value elicitation on the next decomposition level. For this example, this section considers the refinement of the 'Enter the Vehicle' and the 'Exit the Vehicle' use case that the structure system, which contains all the vehicle's structural elements, addresses. Figure 6.6 describes the diagram on this decomposition level. Because the use cases on this decomposition layer partly follow from the operating principle specification, this section only presents the use case diagram, whereas subsection 7.2.2 the derivation of the use cases from the operating principles.

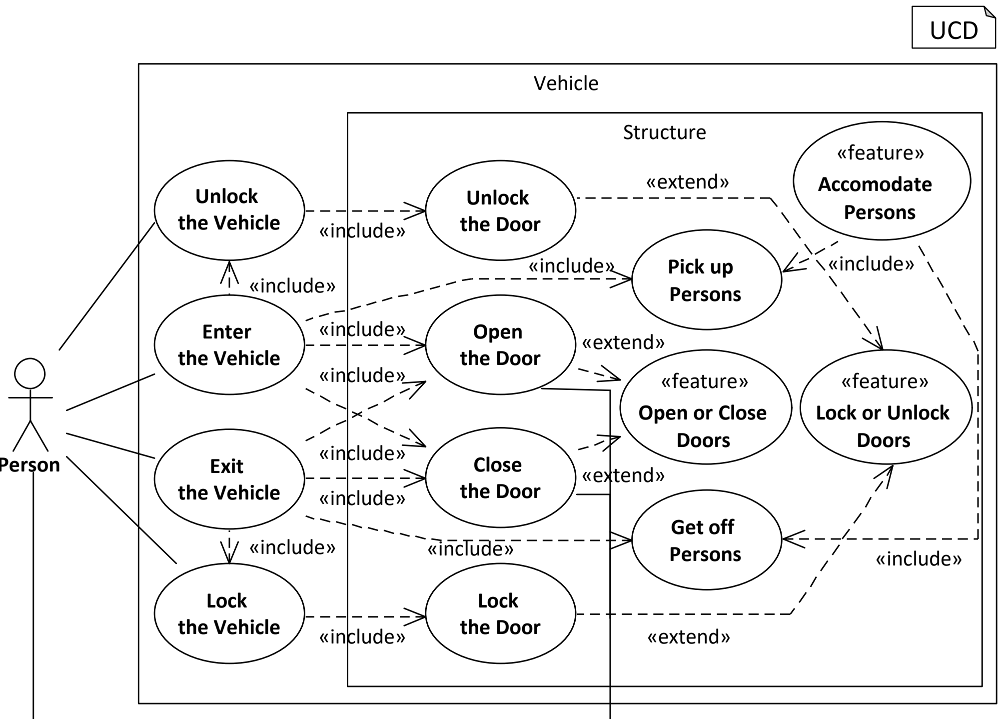


Figure 6.6: A stakeholder value use case diagram for the structure system on the next decomposition level.

6.4 Applying Use Case Diagrams and Generated Requirements in Automotive Systems Engineering Development Projects

In the process of evaluating the application of the methods presented in this chapter in the automotive industry, this section presents some insights from the Epsilon, Delta, and Zeta projects and relates them with the publicly available VTOL models from [JGS+20] on the application of stakeholder values modeling. Because previous related works already demonstrated the applicability of use case diagrams for stakeholder value specification in general [Rum94, ACD+01, Arl98, KG12], and several automotive [MHGK03, OM04, HGBS15] and CUBE specific applications demonstrated the applicability in particular [JGS+20, GKS+21, JGW+21], whereas the majority of the automotive industry still relies on natural language requirements [LTK19], this section takes up the follow-up questions from subsection 5.5.2 and aims to answer how well the models from these projects can hold up compared to textual requirements. For better comparability, subsection 6.4.1 provides a generation method to generate natural language requirements from use case diagrams and modeling guidelines for modeling use case diagrams as stakeholder value models that lead to (grammatically) correct system requirements. To evaluate the correctness of the generated requirements subsection 6.4.2 provides an evaluation scheme that requirements engineers could use to rate the correctness of the generated natural language requirements. As the Delta and the Zeta projects, as well as the VTOL model from [JGS+20] only developed a use case diagram for the stakeholder value and no natural language requirement specification, the evaluation of these projects focuses on the correctness rating of the resulting requirement and the possibility to correct the requirement by adapting the model for the next generation cycle. subsection 6.4.3 presents the results of this evaluation. Because the Epsilon project started with legacy requirements and additionally provided a model-based specification, the generated requirements this section additionally compares the generated requirements with the original requirements to determine the completeness of the derived requirements.

For the evaluation, this section draws on the evaluation results from [Zab23], and extends them with evaluations for the additional rules in section D.1.

6.4.1 Generating High-Level Requirements from Feature Use Cases

Since most of the automotive industry still relies on natural language requirements [LTK19], many stakeholders are more used to reading system models as natural language requirements. Consequently, it might be beneficial to provide some elementary concepts or even all model elements in the form of natural language requirements to stakeholders in addition to the model for reviewing purposes for these stakeholders. In addition to

this application, generating requirements from use case diagrams increases their comparability to legacy requirements. To this end, a master thesis [Zab23] developed a set of transformation rules and guidelines for modeling the respecting use case diagrams and applied them to the results from the Epsilon, Delta, and Zeta projects as well as the VTOL model published in [JGS+20], which this thesis extends with additional transformation rules and application insights. In addition to the theoretical concepts, the thesis developed a tool extension for the SysML modeling tool Enterprise Architect [Spa21] that all considered projects used, simplifying the application for the involved system modelers and reviewing requirements experts.

Because the stakeholder requirements a stakeholder value model represents a crucial backbone of the system development, as these requirements decide whether the right system for the right stakeholder is designed, the thesis in [Zab23] implemented 13 templates for almost all concepts and extensions Table 6.1, which this thesis extend with additional templates in Table D.1 to additionally address the concepts Table 6.3 provides. For this transformation, a set of transformation rules insert the respective fields of

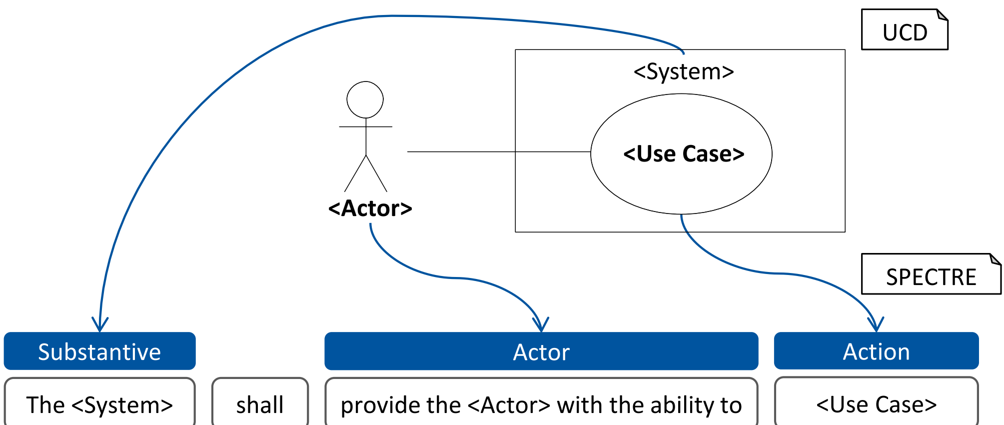


Figure 6.7: A straightforward transformation that inserts the names of the use case diagram element in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1).

the the SPECTRE template (*cf.* Figure 5.1) according to the basic intuition Figure 6.7 for a simple case. As the application reveals, this straightforward approach is only successful in combination with some modeling guidelines that aim to ensure a generation of grammatically correct sentences. For example, the transformation returns the following requirement when inserting the use case diagram from the automobile example from [FMS14b].

Example 11 (Generated SPECTRE Requirement from the Operate Vehicle UCD in [FMS14b])**.** *The Vehicle shall provide the Vehicle Occupant with the ability to Enter*

Vehicle.

Although this requirement is conceptually correct, an article is required to render the requirement grammatically correct. Therefore, the diagram in Figure 6.1 already contains many (but not all) required adaptions to formulate and generate grammatically correct requirements, such as the following requirement.

Example 12 (Corrected SPECTRE Requirement based on the Operate Vehicle UCD in [FMS14b])**.** *The Vehicle shall provide the Vehicle Occupant with the ability to Enter the Vehicle.*

Consequently, additional modeling guidelines are required to return correct natural language requirements. Because the names of the requirements do not influence the semantic domain, this adaption is possible without changing the semantics of the diagram. For the currently presented intuition, Figure 6.8 presents an example of such a guide-

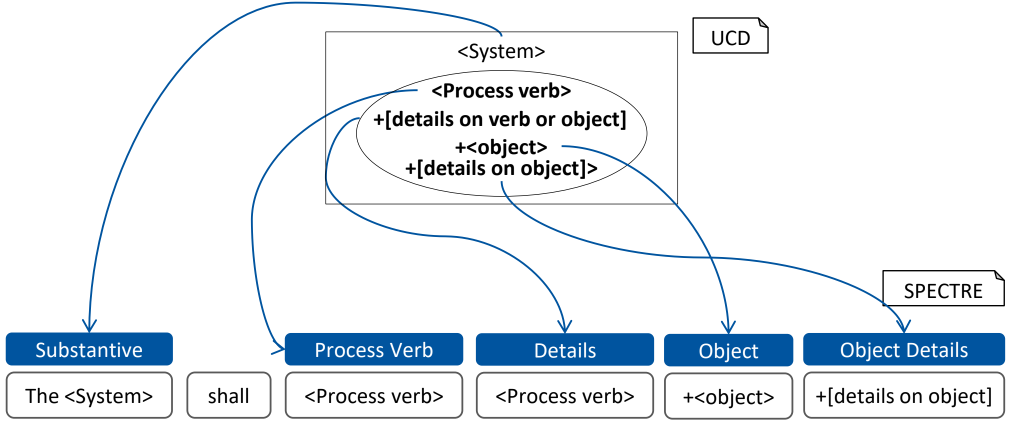


Figure 6.8: A modeling guideline to assure grammatically correct sentences in the SPECTRE template (*cf.* Figure 5.1).

line. As more detailed overview of the transformation rules section D.1 presents the full set of transformations this thesis applied during the evaluation. Moreover, Figure 6.9 provides another example based on the diagram in Figure 6.4 displaying the generated requirements together with the diagram elements. As the example in Figure 6.9 with a few use cases already indicates, it is often not good practice to display requirements and use cases in the same diagram, as examples from industrial applications with more use cases are otherwise no longer legible.

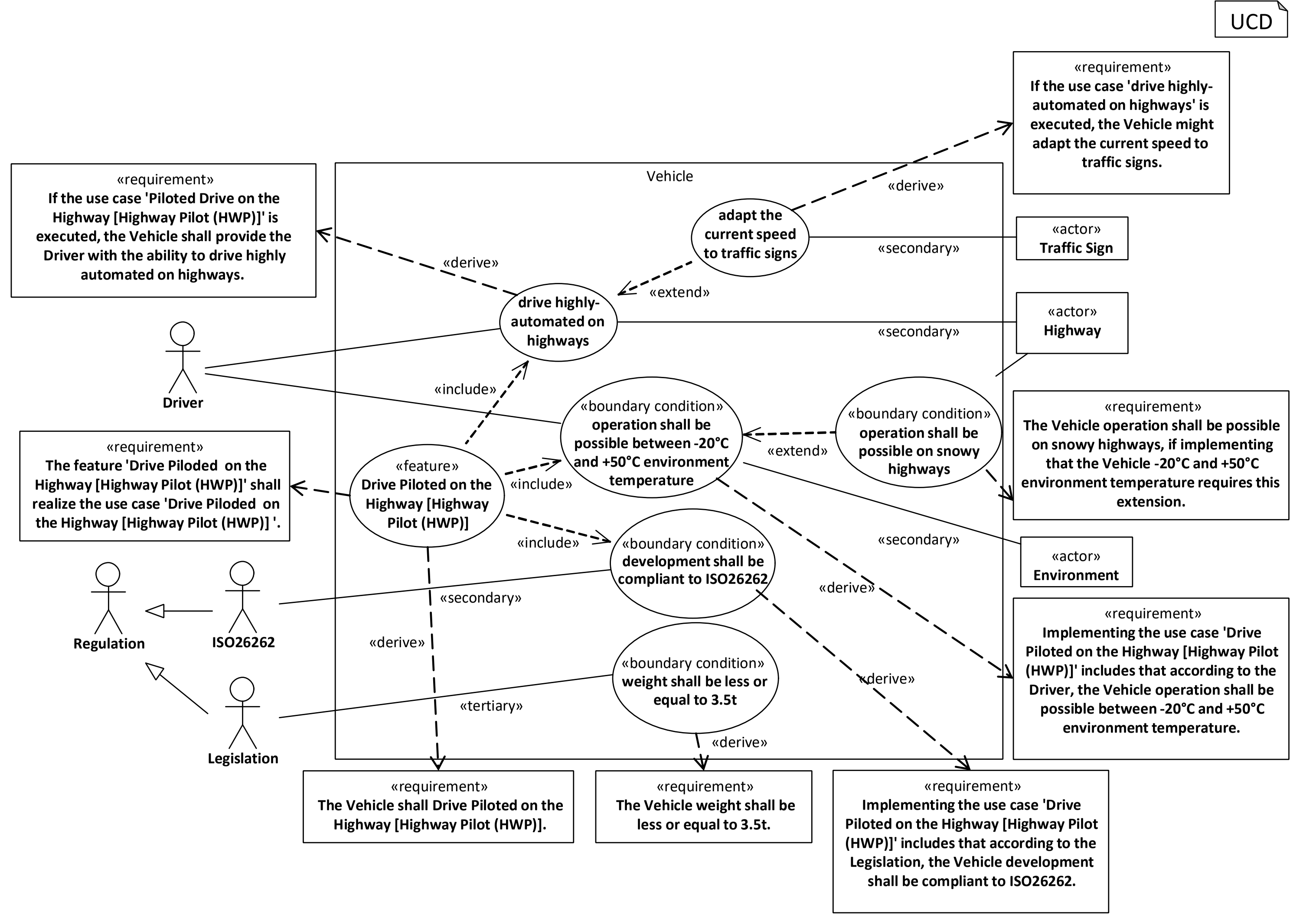


Figure 6.9: The use case diagram from Figure 6.4 together with the derived requirements using the transformation rules in section D.1.

6.4.2 An Evaluation Scheme for Generated Natural Language Requirements

To rate the generated requirements according to their syntactical and semantic correctness, this thesis relies on the rating scheme [BBK+22b, BBK+22a] presents. Table 6.4 shows the requirements ratings and shows example requirements from the evaluated project leading to this rating. In this context, a syntactical issue of a requirement is, for example, a misspelling or a small grammatical problem like a missing comma. Additionally, semantically incomplete or wrong requirements include but are not limited to requirements with logical problems, unclear or ambiguous use of verbs, or missing or wrong conditions. Since errors in capitalization often resulted from the duality of using the statements as names and as sentence parts, the reviewers were instructed not to include them in their assessment of grammatical correctness, even though they would constitute a grammatical error on closer inspection. A limitation of this evaluation system is its lack of ordination, which means that requirements in class 2 are not necessarily better than the requirements of class 3 or class 4 [BBK+22b]. Nevertheless, the findings from the evaluation suggest that, in many cases, a lower class requires less effort to correct a requirement.

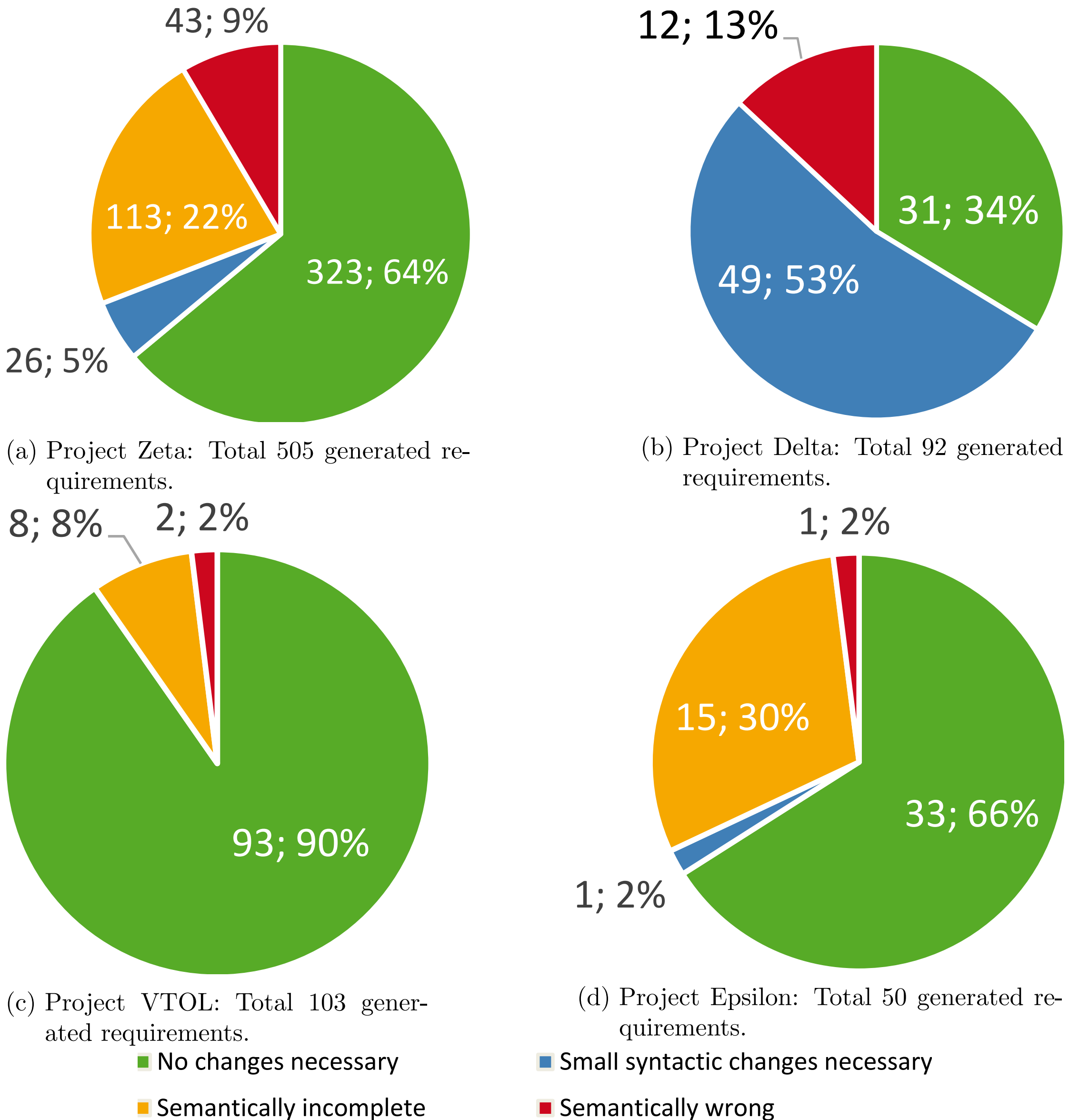
6.4.3 Correctness Evaluation of Generated Stakeholder Value Requirements in Model-Based Projects

As a measure of the applicability of the presented method, this section presents the results of the correctness evaluation of the requirements engineers performed based on the evaluation scheme Table 6.4 presents. To perform this evaluation, the transformation rules from subsection 6.4.1 were applied to the use case diagrams in the Zeta, Delta, VTOL, and Epsilon projects, before a group requirements engineers at FEV.io reviewed the generation result and mutually agreed on a rating of the requirements according to the rating scheme Table 6.4 presents.

Figure 6.10 presents the results of the evaluation, of which [Zab23] already presented the evaluation results from Figure 6.10a, Figure 6.10b, and Figure 6.10c, whereas the evaluation in Figure 6.10d is newly added in this thesis. The first finding that becomes evident in Figure 6.10 is the fact that although the rating scheme in Table 6.4 contains six classes for the requirements, the diagrams only consist of four classes of errors, which results from the nature of the generated requirements. As the performed classification predominantly revealed spelling errors and grammatical problems for which the evaluating requirements for which the evaluating requirements engineers agreed on classifying them in class 2 (or class 4). Because the class 4 evaluation was only present 12 times in all requirements evaluated in this section, subsection 7.3.2 and subsection 8.3.2, the evaluating requirements engineers agreed on reclassifying them in class 2. The second finding is that the four evaluated stakeholder value requirements seem not to follow a

Table 6.4: Requirement rating scheme based on the evaluation scheme use in [BBK+22b, BBK+22a].

Class	Description	Example	Explanation
1	Correct	The vehicle shall drive piloted on the highway.	No correction is required.
2	Syntactically Incorrect	The vehicle shall drive piloded on the Highway.	Spelling mistake in 'Piloded'.
3	Semantically Incomplete	The vehicle shall provide the driver with the ability to get information.	The kind of information is not specified in the requirement, rendering it semantically incomplete.
4	Semantically Incomplete and Syntactically Incorrect	The vehicle shall provide the passenger with the ability to get infromation.	The requirement contains a spelling mistake in 'infromation' and has the same incompleteness as the requirement before.
5	Major Syntactical Problems	The functions of the system are classified as safety relevant in with respect to ISO 26262.	Does not conform the guideline for requirement formulation Figure 5.1 defines.
6	Semantically Incorrect	The vehicle shall provide the criminal with the ability to steel the vehicle.	The requirement specifies the misuse case as intended behavior instead of the use case and uses 'steel' instead of 'steal'.



(a) Project Zeta: Total 505 generated requirements.

(b) Project Delta: Total 92 generated requirements.

(c) Project VTOL: Total 103 generated requirements.

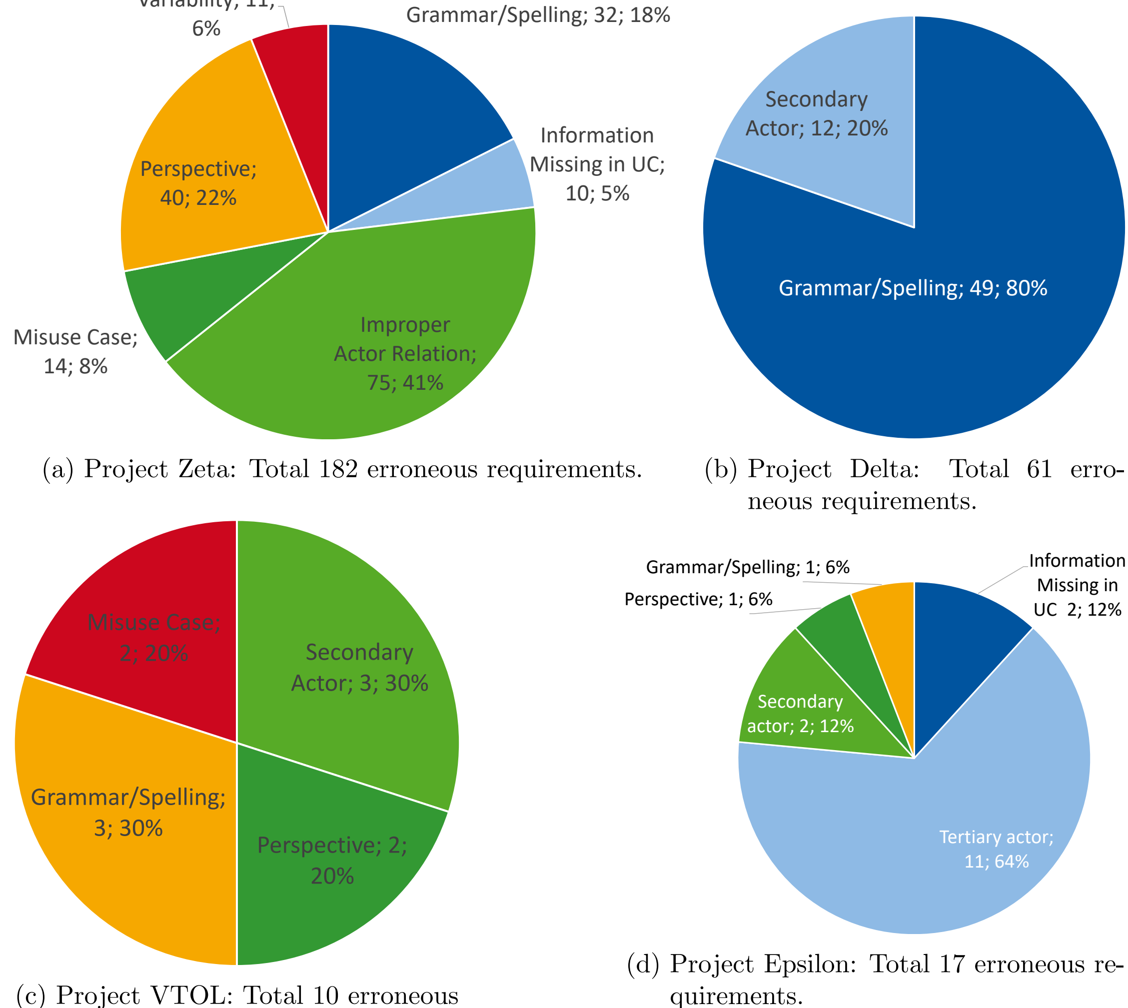
(d) Project Epsilon: Total 50 generated requirements.

Figure 6.10: Correctness evaluation results of the stakeholder value requirements generated in the Zeta, Delta, VTOL, and Epsilon projects.

clear project-independent trend, as, for instance, in the VTOL project, almost all requirements were generated correctly, whereas in the Delta project all generated requirements were erroneous. To further investigate whether there is a project-independent or project-specific trend within the error reason, the requirement experts further developed a correction for each erroneous requirement (*i.e.,* all requirements that were not in class 1 in the original evaluation), which they used to identify a reason for the error based on the part of the requirement that needed a correction. If a requirement requires more than one correction, the requirements engineers classify them according to the correction that fixes the highest requirement rating according to Table 6.4.

The results of this further error classification in Figure 6.11 reveal that, although the project distribution is different, the evaluating requirements engineers uncovered similar errors in all considered projects. In the Zeta project, as Figure 6.11a reveals, improper actor relationships, such as secondary and tertiary actors that were connected as primary actors, or missing and wrongfully connected actors form the biggest group of error reasons that lead to the generation of an erroneous requirement in the 'Improper Actor Relation' class. The next most significant error reason class 'Perspective' consists of requirements that result from use cases where the modeler used the wrong perspective to formulate the requirement *i.e.,* the requirement formulated what the system does and not which value the system has for the connected actor. For example, many use cases describe information that the system sends, signals, or displays for the user instead of the ability of the user to access or see the information. Though the models in the projects were carefully reviewed, the generation of natural language requirements revealed that some requirements still contained errors in grammar or spelling (in the 'Grammar/Spelling' class) that were sometimes only revealed in the generated requirement. For example, errors in the sentence structure were more easily uncovered in complete sentences than in the sentence fragments the use case names provided, such that sentence fragments that seemed correct in the use case led to incorrect stakeholder requirements in natural language formulation. Moreover, the generated requirements contained some semantically wrong requirements that described the opposite of the intended behavior (in the 'Misuse Case ' class). For example, the requirement that the vehicle shall provide the Criminal with the ability to steal the vehicle instead of the requirement that the vehicle shall hinder the Criminal from stealing the vehicle. All of these requirements were caused by not labeling misuse cases as such in the stakeholder value model. The next smaller class, 'Information Missing in UC', consists of requirements where the modelers forgot to specify essential information in the use case *e.g.,* which information the maintenance personnel should access during maintenance. Finally, the project consisted of a unique error class, 'Variability,' that was not present in the other projects, which consists of errors resulting from wrongly including variability information into the requirement *e.g.,* by writing 'Optional' before the use case.

In the Delta project, as Figure 6.11b presents, most requirements were erroneous as



(a) Project Zeta: Total 182 erroneous requirements.

(b) Project Delta: Total 61 erroneous requirements.

(c) Project VTOL: Total 10 erroneous requirements.

(d) Project Epsilon: Total 17 erroneous requirements.

Figure 6.11: Results of the error reason classification in the Zeta, Delta, and Epsilon projects.

they contained a grammatical error in the process verb. For example, the use case diagram contained use cases such as 'Activating function A', which resulted in requirements of the form "The vehicle shall 'provide the driver with the ability to activating function A" instead of the correct "The vehicle shall 'provide the driver with the ability to activate function A". These erroneous requirements result from a conflicting guideline in the context of use case scenario modeling as [MSG+22] presents, requesting the modelers to use the present progressive instead of the gerund in use case formulations. Therefore, most of the requirements in the project are in the 'Grammar/Spelling' class. Since some of the requirements additionally connected a secondary actor as a primary actor, the 'Secondary Actor' class summarizes these requirements, which is a subclass of the 'Improper Actor Relation' class in Figure 6.11a.

As Figure 6.11c depicts, using secondary as primary actors (in the 'Secondary Actor' class) and grammar or spelling mistakes (in the 'Grammar/Spelling' class) are responsible for most errors in the VTOL project. The remaining errors in the project equally distribute between the wrong use of perspective to formulate use cases (in the 'Perspective' class) and not labeling misuse cases as such (in the 'Misuse Case' class).

Finally, Figure 6.11d depicts the error reason classification of the requirements in the Epsilon project. As in the Zeta project (*cf.* Figure 6.11a), improper actor relationships cause most of the errors in the generated requirements. As the number of requirements in the 'Tertiary Actor' class shows, most of these errors in the Epsilon project result from using tertiary actors as primary actors. The remaining requirements with improper actor relationships result from secondary actors used as primary actors (in the 'Secondary Actor' class). The remaining errors result from information missing in the use case, grammar and spelling errors, and perspective errors in the use case diagram.

In conclusion, the manual classification of the generated requirements according to the rating scheme in Table 6.4 reveals that the derivation of stakeholder requirements in natural language based on use case diagrams is not only possible but also primarily correct (*cf.* Figure 6.10) when following the guidelines from subsection 6.4.1. Moreover, the analysis of the error reasons (*cf.* Figure 6.11) for the erroneously generated requirements enabled the reviewing requirements engineers to correct the errors their predecessors that modeled and reviewed the use case diagrams overlooked. Consequently, generating and reviewing natural language requirements for use case diagrams during a model review might assist inexperienced modelers in reviewing and improving their stakeholder value models. As the evaluation of the error reasons in Figure 6.11 revealed, an adaption of the use case diagram was sufficient to fix all errors in the generated natural language requirements. Therefore, the generation method from subsection 6.4.1 and the transformation rules from section D.1 are sufficient to model correct stakeholder requirements as use case diagrams in comparable projects in the automotive industry.

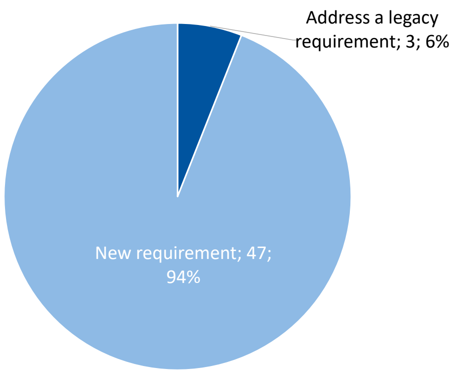
6.4.4 Completeness Evaluation of Generated Stakeholder Value Requirements in Model-Based Projects compared to a Document-Based Specification

Because the evaluation subsection 6.4.3 presents is only suited to demonstrate that use case diagrams are capable of providing correct stakeholder requirements in natural language but not sufficient to demonstrate that a complete set of natural language requirements is derivable, this section compares the generated requirements from the use case models in the Epsilon project with the document-based legacy requirements the project reworked.

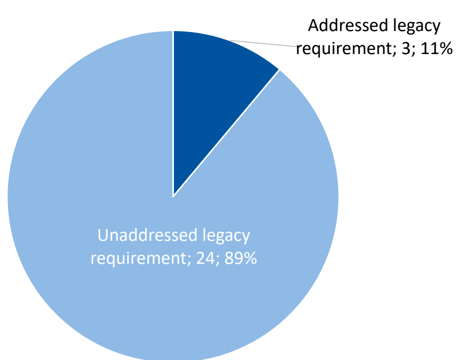
Because of the nature of the Epsilon project, two limitations of the results concerning the stakeholder value requirements require consideration to enable a meaningful evaluation of the completeness of the results. The first limitation of the available project is that the missing traceability to the stakeholders, their needs, and the value of the legacy requirements is one of the main reasons the customer contracted the project teams to apply a model-based systems engineering methodology to rework the specification. As a second limitation, the project team had the task to focus on the requirements concerning the functional characteristics of the system, as the customer considered a lack of functionality requirements as the second weakness of the document-based legacy requirements. In contrast, the customer considered the present project, process, performance, specific quality requirements, and constraints sufficient to retain natural language requirements orthogonal to the model. As a result, the project team had no obligation to model most boundary condition use cases during the project. Therefore, these requirements must be included in the initial generation of the natural language requirements. Consequently, this section explicitly compares the available stakeholder value requirements with the number of additionally generated stakeholder value requirements to highlight the additionally generated requirements or put them into perspective if modeling these connections does not realize an additional benefit to mitigate the first limitation. Regarding the second limitation, [Zab23] reworked the missing boundary condition use cases based on the document-based legacy requirements to evaluate whether the generation method from subsection 6.4.1 and the transformation rules from section D.1 suffice to express these legacy requirements.

To evaluate the completeness of the generated requirements, the same requirements engineers that conducted the correctness evaluation in subsection 6.4.3 mapped the generated requirements to the legacy requirements and the legacy requirements to the generated requirements. Then, they evaluated whether the different requirements specify the same system functionality or characteristic.

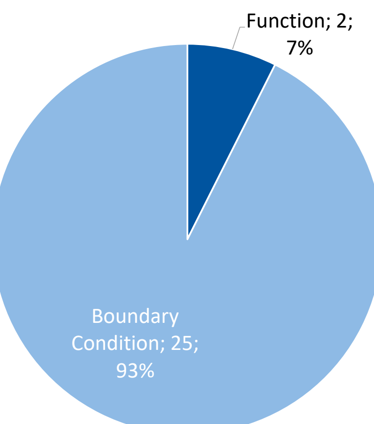
Figure 6.12a and Figure 6.12b depict the results of this mapping, of which all addressed requirements specified the same functionality or characteristic. These numbers reveal that the other requirement set covers only a fraction of the requirements. As



(a) Evaluation how many of the 50 generated requirements in the Epsilon project address one of the 27 legacy requirements.



(b) Evaluation how many of the 27 legacy requirements in the Epsilon project are addressed by one of the 50 generated requirements.



(c) Ratio of legacy requirements to be modeled as functional and as boundary condition use cases: Total 50 requirements.

Figure 6.12: Results of the completeness evaluation comparing the model-based stakeholder value specification from the Epsilon projects with the document-based legacy requirements the project used as input.

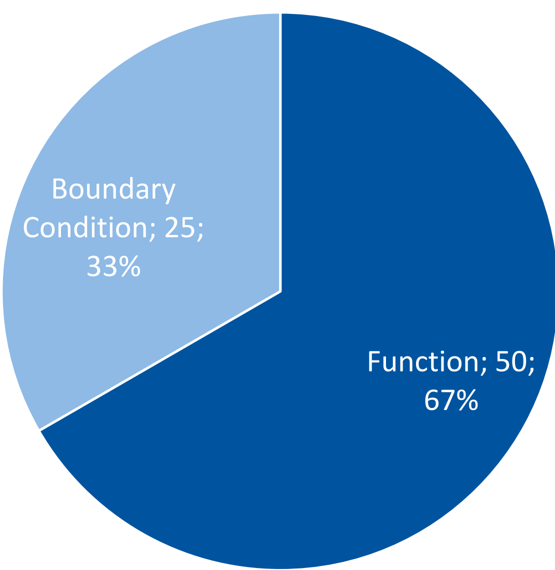


Figure 6.13: Number of functionality and boundary condition use cases in the reworked use case model: Total 75 requirements.

Figure 6.12c depicts, this is caused by the fact that the Epsilon project only modeled one boundary condition use case, whereas the legacy requirements contained almost only boundary condition use cases, as the analysis reveals. To mitigate this effect, a student reworked the original use case model in the context of his thesis [Zab23] to create a reworked use case model. As Figure 6.13 presents, the resulting model consists of 75 requirements that not only address the boundary conditions the stakeholders formulate as needs but also the functionalities these stakeholders require. Using this model, a correct set of stakeholder requirements (applying the corrections according to Figure 6.11d) is generable, that correctly addresses all requirements the legacy stakeholder requirement specification document contained. Therefore, after extensively reworking the model, a complete generation of the legacy stakeholder requirements was possible. Moreover, the additionally modeled functionalities the stakeholders require contribute to the customer's goal of achieving a better traceability of her requirements.

Because of the limited number of available projects, the limited comparability of the generated functional requirements, and the required rework of the input models due to neglecting the boundary conditions, this section's completeness evaluation is not representative of the automotive industry. Nevertheless, the results demonstrate that the presented method is applicable in the considered project to create a complete set of stakeholder value requirements that is at least as good (and here even better in terms of the number of stakeholder requirements the model makes explicit) than the legacy stakeholder requirements document. Because this thesis also publishes the generation

method and evaluation scheme so that other researchers may also aim to reproduce or contradict these results in similar projects, a better result might be achievable in other works in the future.

6.4.5 An Experiment to Measure the Effort Required to Maintain Use Case Diagrams next to Natural Language Requirements Manually

Because several stakeholders in the Zeta, Delta, and Epsilon projects suggested modeling use case diagrams and manually maintaining natural language requirements that describe the contents of the diagram as a feasible and easy-to-implement option, this section presents an experiment that aims at indicating the hidden costs and efforts behind this approach. The experiment was designed and conducted in the context of [Zab23] and demonstrated the time required to generate, review, and correct the requirements through automatic and manual approaches. This experiment investigates whether automatically generating requirements from use case diagrams increases the efficiency of system development processes by decreasing the time required to extract the requirements from the diagram, review the results, and correct them in the next step.

The experiment consists of three steps. In the first step, a test person with experience in systems engineering is asked to derive the requirements the diagram in Figure 6.14 specifies, having the transformation rules from section D.1. The test person will be asked to review the requirements in the next step. Then, the next step includes correcting the requirements, and ultimately, in the final step of the experiment, the model test person adapts the model such that the transformation yields the result. The required time is measured for each step, and the number of errors is counted.

When conducted at FEV.io in the context of [Zab23], this experiment revealed that the three participants that participated in the experiment took about 30 minutes to derive the requirements, between 20 seconds and 1 minute at an average of about 40 seconds per requirement to review the requirements, and about the same time per error to correct the errors in the textual requirements, and again 20 seconds to a minute on an average of about 45 seconds to correct them in the the model. When using these measurements as parameters (*cf.* Table 6.5) for extrapolating the times required for the models in the Zeta, CUBE, VTOL, and Epsilon projects result is a saving of 49,12% on average in all projects as Table 6.6 reveals. Therefore, the requirements generation reduces the overall effort to almost half of the required efforts for maintaining the same requirement in two representations, which also matches the general intuition that doing a task once instead of twice should be faster.

The flaws in the experiment and its example application are the size of the case study (one diagram), the limited number of participants during the experiment, and the need for more data about other requirement elicitation methods. Moreover, the extrapolation

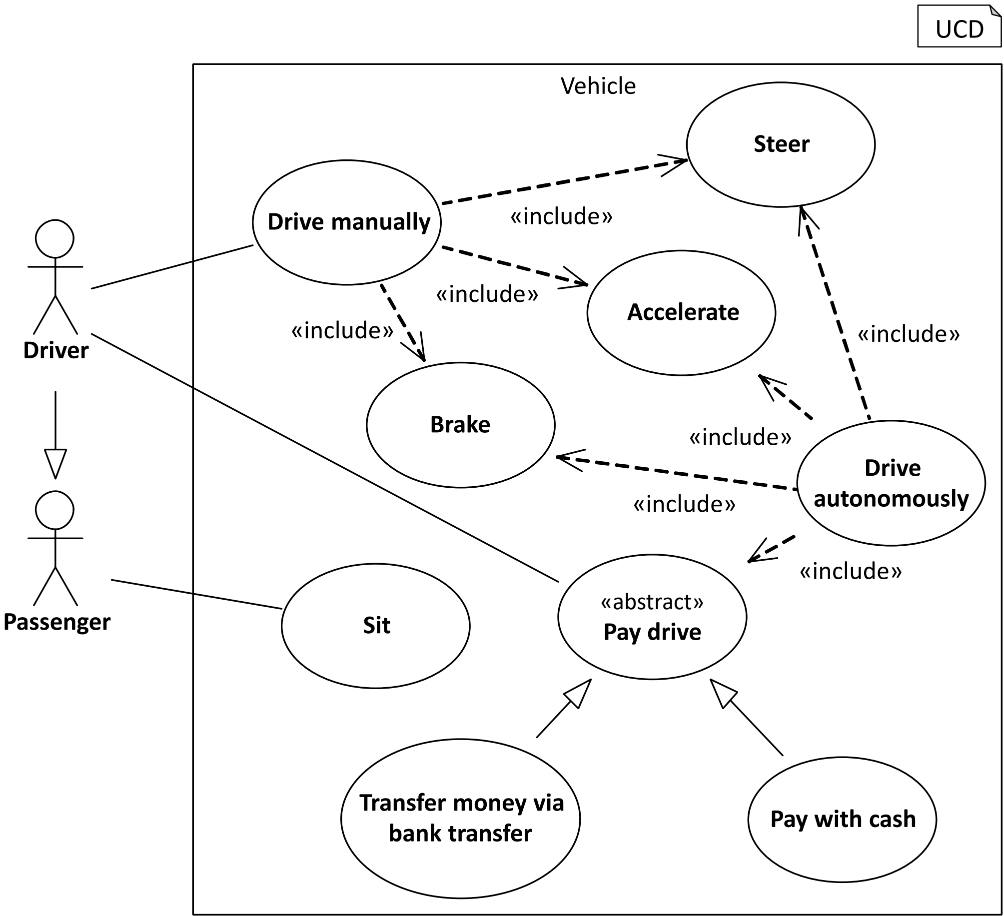


Figure 6.14: The example diagram for which requirement experts should derive textual requirements.

Table 6.5: Experimentally determined parameters for calculating the process times for automated and manual creation of textual stakeholder value requirements from use case diagrams.

Parameter	Value
Required time manual [s/req]	69.23
Required Time automated [s/req]	1.00
Average review time [s/req]	40.00
Average correction time model [s]	45.00
Average correction time requirement [s]	40.00

Table 6.6: Estimation of required times to manually/automatically derive stakeholder value requirements, their review, correction, and regeneration and re-review for generated requirements.

Project	Zeta	Delta	VTOL	Epsilon	Average
#Requirements	505	92	103	50	188
#Erroneous Requirements	182	61	10	17	68
Time for automated derivation [h]	0.140	0.026	0.029	0.014	0.052
Time for manual derivation [h]	9.712	1.769	1.981	0.962	3.606
Review Time [h]	5.611	1.022	1.144	0.556	2.083
Time for model correction [h]	2.275	0.763	0.125	0.213	0.844
Time for requirement correction [h]	2.022	0.678	0.111	0.189	0.75
Time for requirement regeneration [h]	0.051	0.017	0.003	0.005	0.019
Time for re-review [h]	2.022	0.679	0.111	0.189	0.75
Total process manual [h]	19.620	4.232	3.361	1.919	7.283
Totel process automated [h]	10.099	2.505	1.412	0.976	3.748
Required time for generation [%]	51.47%	59.20%	42.01%	50.85%	51.46%
Savings [%]	48.53%	40.80%	57.99%	49.15%	49.12%

neglects the additional efforts required in both projects, such as project coordination, alignment between the different involved reviewers, and the fact that the engineers are probably less efficient when working on hundreds instead of 26 requirements. Furthermore, these results do not reflect on the necessity of having natural language requirements following SysML models at all, as the results from subsection 6.4.4 indicate that use case diagrams alone can already sufficiently specify the stakeholder value in the considered project using the methods and extensions this thesis presents. However, the experiment demonstrates that the minimum time required to generate the requirements and maintain only one instead of two redundant places is much smaller than manually maintaining both views. Moreover, extrapolating these times to the projects reflects a general trend and might aid other systems engineers in planning their projects.

6.5 Interim Conclusion and Discussion

In conclusion, this chapter answers **RQ-3.2** (“How can stakeholder values be formulated?”) by providing the required SysML model elements in Table 6.1 and Table 6.3.

Moreover, this chapter introduces the method and the semantic foundations to model system stakeholders, their needs, and values using SysML use case diagrams as the

foundation to answer **RQ-4.1** ("Can operating principles be derived from use case scenario specifications?") in chapter 7. To this end, section 6.2 provides the semantic foundations for the UCD elements in Table 6.1 and extends the previous work in [KRW22] with the system boundary as a new element. Of course, more than this semantic foundation is required to derive operating principles. Therefore, chapter 7 revisits this concept and combines it with the concept of the use case specifications [Coc01, Fow04].

Additionally, subsection 6.4.3 and subsection 6.4.4 apply use case diagrams to replace natural language requirements as a means of expressing stakeholder requirements in document-based development projects to address **RQ-4.4** ("Are the models that result from a model-driven specification comparable to document-based specifications?"). Although the number of available projects was limited, especially for the evaluation of the completeness of the derived requirements, the numbers reveal that a generation of comparable natural language requirements from the use case diagram is possible using the methods and guidelines from subsection 6.4.1. Therefore, in the considered projects, the stakeholder value models could completely replace natural language requirements without sacrificing crucial information. Because of the threats to validity subsection 6.4.3 and subsection 6.4.4 already mentioned, the results cannot represent the complete automotive industry. Therefore, future works might aim to reproduce or disprove this chapter's results.

Finally, regarding the fulfillment of the methodical requirements from section 3.3, several aspects contribute to the fulfillment of the methodical requirements. Concerning **MR-1** ("Specification models created using the method must be reusable to realize reduced time-to-market constraints"), CUBE relies on solution-neutral stakeholder values for the elicitation of stakeholder requirements. Therefore, all use cases and actors are reusable in different contexts if the same stakeholders with the exact needs interact with a system. Moreover, with a broad system boundary definition, the system boundary and even system features might be applicable in a different context, making the complete diagram reusable for other development projects. Regarding **MR-2** ("Specifications must be adaptable to specific markets or vehicle projects"), this thesis groups the use cases into system features applicable in different contexts and configurations. Although this chapter does not integrate mechanisms to formulate dynamic architectures with different possible feature configurations as, for example, in [GJK$^+$21], the idea of feature-driven development that is integrated into CUBE [GKS$^+$21, Gra22] enables systems engineers to provide different individual functionalities that are configurable in different contexts. By providing a semantic foundation for use case modeling, this chapter creates a consistent specification by establishing a joint and semantically sound understanding of the diagram, as required by **MR-3** ("Specification models must be consistent"). In addition to this, it might also be beneficial to integrate additional measures for consistency checking into the modeling project *e.g.,* by implementing additional consistency checks after combining different use case diagrams in a system model or by implementing semantic

differencing to make system models or versions comparable [KRW22]. Concerning **MR-4** ("Specification models must be bidirectionally traceable"), the methods rely on different connections that allow the modeler to follow traces from actors to features and back using the connectors and associations Table 6.1 and Table 6.3 provide. Finally, section 6.3 addresses **MR-5** ("Creating specification models must be possible without redundant manual modeling tasks") by enabling the systems engineers to implement multiple use cases in the same feature. Future works can also integrate package and import structures into the concrete syntax for use case diagrams Listing 6.2 to further increase the reuse of use cases in different features and stakeholder value models.

Chapter 7

Specifying System Behaviors and Functionalities Based on Operating Principles

Although each SysML behavior diagram, apart from the UCD, is well suited to describe the operating principle of a feature [GKS+21], this chapter focuses on describing operating principles as activity diagrams. This is not only due to the fact most applications of CUBE [KRGK18, GKS+21, JGW+21, MSG+22] apply activity diagrams for this purpose, but also due to the usage in the industry projects Zeta, Delta, and Epsilon to which this chapter refers for the application of the presented methods.

As the activity diagram describes the possible order of action the system executes during an activity or process [AT01, Stö05, Esh06, Kau21], this diagram is an excellent match of the operating principle description, because as the activity diagram the operating principle that consists of actions that a system requires to execute to implement the feature. Because not only concrete syntax definitions for activity diagram exists [Obj17, Obj19], but also several semantics definitions are available in the literature [Obj21, GRR10, MRR11b, MRR11a, KR18, Kau21], this chapter solely focuses on the modeling aspect to model operating principles, without defining any further extensions to the various semantic interpretations that are available in the literature on this diagram. In connecting the execution of actions with the use cases in the stakeholder value models, this chapter additionally refers to methods for use case specifications [Fow04, Coc01] to bridge the gap between those concepts.

To this end, this chapter is structured as follows. section 7.1 presents a method to model operating principles as activity diagrams by providing the required diagram elements and concepts for which it also demonstrates the applicability in the context of this thesis' application example (*cf.* section 4.2). Then, section 7.2 connects these concepts with use case diagrams by introducing an extended template for use case specifications. Finally, section 7.3 presents insights from applying these concepts in the automotive industry.

7.1 Modeling Operating Principles as Activity Diagrams

In CUBE operating principles serve to describe the sequence of actions that a system design must provide to implement a feature (*cf.* subsection 3.2.4). Activity diagrams designed to model action sequences are suitable for modeling this execution. Therefore, the diagram provides several concepts that support a systems engineer in designing the operating principles of the system's features. First, activity diagrams provide activities expressing a dynamic behavior in a particular order of actions [AT01], which correspond to the features a system provides. Next, the activity models the actions and their execution sequence, for which the diagram provides actions and control flows, which correspond to the operating principles in the CUBE. Finally, activity diagrams express object flows within the diagram and activity parameters, which express the inputs and outputs the activity requires during its execution. Having these concepts, the systems engineers that use activity diagrams for operating principle specification may lay the foundations for defining the interfaces in the logical architecture by assigning the actions to their executing elements in the logical architecture from which the required and provided interfaces follow.

7.1.1 Diagram Elements to Model Operating Principles as Activity Diagrams

To systematically express operating principles with activity diagrams, Table 7.1 presents a subset of the SysML specification [Obj19] tailored to the application in the context of the operating principle specification. With this definition, this section provides the required elements in the SysML and therefore contributes to answering **RQ-3.3** ("How to model operating principles?").

Table 7.1: AD elements required to describe the operating principles according to the CUBE methodology.

Element Name	Notation	Description
Obligatory Diagram Elements		

Table 7.1: AD elements required to describe the operating principles according to the CUBE methodology.

Action	Action / callBehavior: Activity	Actions represent individual (atomic) steps within an activity. Thus, an activity describes a behavior composed of individual elements, such as actions. A call behavior action (on the right side) is a special action that invokes a behavior diagram of another activity.
Initial	Initial	An initial node represents the starting point for the behavior execution of an activity. Although the SysML standard [Obj19] allows both multiple or no initial nodes, having exactly one initial node in the diagram is a good practice.
Control Flow		A control flow determines the order in which the activity executes its elements. According to the SysML standard [Obj19], control flows are either solid or dotted lines consistently used in the diagram.
Final	Final	As soon as a control flow reaches the final node, the activity terminates the execution. Tough the SysML standard allows both multiple or no final nodes, having exactly one final node in the diagram is a good practice.
Optional Diagram Elements		
Activity	Activity / Activity	Activities express their behavior in a particular order of actions. Activity diagrams do not consist of activities but of actions but may contain the activity they describe as a boundary.

Table 7.1: AD elements required to describe the operating principles according to the CUBE methodology.

Activity Parameter	Activity activityParameter: Type	Activities can have parameters that describe the inputs the actions require and the outputs the actions create during the execution of the activity.
Object Flow	Action apOut: Type apIn: Type Action Action object: Type Action	Object flows describe objects that actions can exchange during their execution. Object nodes are rectangle-shaped and serve to define exchanged objects within an activity. Pins are object nodes used to specify inputs and outputs for actions [Obj17]. The pin symbol (on the top action) is a square at the outside edge of the action.
Decision	[guard] Decision [else]	Decision nodes are control nodes that accept one or two incoming (control or object) flows and select one outgoing flow with a fulfilled condition. An outgoing edge of a decision node may have a guard written as [condition]{constraint}.
Merge	Merge	Merge nodes are control nodes that combine multiple incoming flows into a single outgoing flow. All ongoing and outgoing flows must either be object or control flows.
Fork	Fork	Fork nodes split one incoming flow into multiple flows to realize a form of parallelism using this particular form of a control node.

Table 7.1: AD elements required to describe the operating principles according to the CUBE methodology.

Join	Join {joinspec = ...}	Join nodes are control nodes to join multiple incoming flows into one outgoing flow. By default, the join node requires all incoming flows to arrive before the flow proceeds with the outgoing flow. A join specification written as {joinspec = constraint} may serve to change this behavior definition.
Flow Final	FlowFinal	The actions entering the node terminate as soon as the control flow reaches the flow final node. However, the remaining control flow in the activity continues.
Interruptable Activity Region	reginon name interrupt flow	An interruptable activity region is a region within an activity diagram in which the execution of the actions in the region ends as soon as an interrupt flow is taken outside the region. If no interrupt flow is triggered and a regular flow leaves the region, the execution will continue regularly.
Send Signal Action	SendSignal	The send signal action is a special action that transmits a signal instance to all target objects [Obj17]. This target object can either be an accept event that triggers the execution of an activity or a trigger in a State Machine transition.

Table 7.1: AD elements required to describe the operating principles according to the CUBE methodology.

Accept Event Action	AcceptEvent	An accept event action is an action that waits for an event such as a send signal action [Obj17]. If the accept event action has no incoming flows, the action starts whenever the containing activity or region starts. Moreover, an accept event action without incoming flows remains enabled after it accepts an event [Obj17]. It only terminates together with the starting activity or region. In contrast, an accept event action with incoming flow behaves like a regular action.
Accept Time Event Action	AcceptEventTimer	The Accept Time Event Action is a specialization of the Accept Event Action that waits for the occurrence of time events.

Though activities are the name-giving elements of the diagram; the actions, which describe the steps within an activity, are the primary and obligatory elements of the diagram. Actions are either atomic steps within an activity or call action behaviors, a special form of actions invoking the diagram of another activity. When executing the diagram, the initial and final nodes determine the first diagram element for the execution and the final diagram element, which terminates the activity after its execution. Although the SysML standard [Obj19] allows multiple or no initial and final nodes in a diagram, having one initial and one final node is a good practice. As readers tend to read from the top left of a diagram to the bottom right, placing the initial node near the top left corner of the diagram and the final node near the bottom right corner is good practice. Since these nodes do not specify the execution order, the control flow determines the order in which the activity executes its elements. According to the SysML, standard [Obj19] and its underlying activity diagram of the UML standard [Obj17], multiple incoming and outgoing fork and join nodes are allowed as implicit merge and fork, which this thesis does not follow. Therefore, only one incoming and outgoing control flow is allowed (if not specified otherwise). To specify these control flows further, systems engineers applying the presented method may explicitly use the Decision, Merge, Fork, and Join node as Table 7.1 describes to formulate the control flow between the activity

diagram elements.

Apart from these elements that all operating principles require, the systems engineers that model the operating principles may use additional elements to express further concepts. Hence, an activity element may make the activity this diagram considers explicit as a boundary. Moreover, this activity may contain one or more activity parameters to define the inputs it requires and the output it produces. Object flows may be used to describe the exchanged objects, effectively connecting these parameters with the actions and expressing the internal processing of objects within the activity's actions. To this end, the SysML standard provides rectangle-shaped object nodes to define objects flowing within an activity between the actions of the activity. As a particular form of object nodes, pins may serve to specify inputs and outputs for actions. The Pin symbol is a square attached to the action block's outside flow. Moreover, this thesis restricts itself to using action pins for object flows to avoid confusing inexperienced diagram readers during the industrial application of the concepts. Furthermore, the author of this thesis refrains from using implicit control flows of object flows as defined in the SysML standard [Obj19] for the same reason. Moreover, as an additional guideline, all methods this thesis presents require that each object be provided only once with the same object, name, and type and that each pin that receives this object as input has the same name and type. Finally, Table 7.1 presents three specializations of actions a systems engineer may use to express asynchronous communication using the method this thesis presents. On the one side of the channel, a send signal action creates a signal instance, on which all accept event actions and triggers in STMs that currently wait for this signal to react. As a specific form of this behavior, an accepted time event action reacts to all defined time events independent of another external trigger apart from the progression of time.

7.1.2 Diagram Elements to Allocating Actions to their Executing Logical Element

Because the systems engineers that apply the CUBE methodology not only use operating principles to express the sequence of actions that must be executed but also the elements of the system under consideration that are responsible for executing these actions Table 7.2, provides additional optional diagram elements to realize this allocation.

Table 7.2: AD elements required to allocate actions to the logical element responsible for the execution according to the CUBE methodology.

Element Name	Notation	Description
	Optional Diagram Elements	

Table 7.2: AD elements required to allocate actions to the logical element responsible for the execution according to the CUBE methodology.

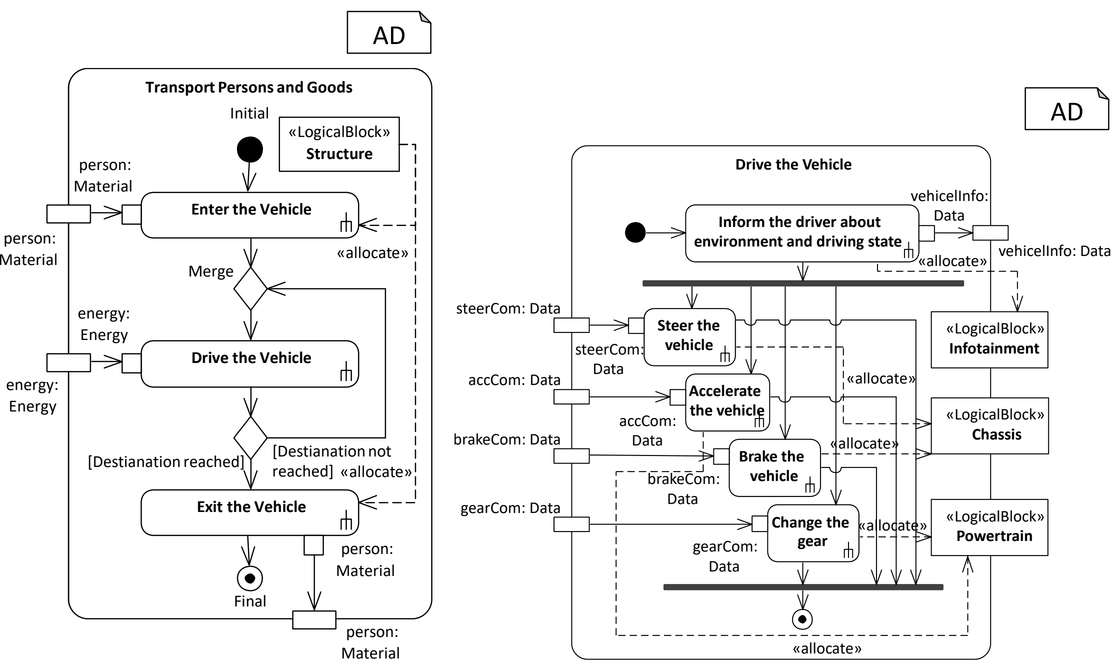
Allocate Partition	«allocate» name: ExecutingLogicalElement	Allocate partitions (or swim lanes) allocate behaviors based on responsibilities. The diagram element expresses that the partition's classifier executes all elements inside the partition.
Allocate Relationship	«allocate»	Allocate relationships are another form of expressing an allocation of behaviors based on responsibilities.
(Logical) System	«block» System	A (logical) system is a SysML block that represents the system. To allocate an element from the activity diagram elements in Table 7.1 to this system, systems engineers may use the Allocate Relationship that points at the system block.

Although Table 7.2 provides Allocate Partitions alongside the Allocate Relationship and the (Logical) System, it is a good practice to consistently only use one allocation type in a diagram. As an orientation, the application in the projects chapter 4 showed that small diagrams are often easier to model using allocate partitions, as the structure supports the modeler in structuring the actions according to the executing system, larger diagrams, or diagrams with complicated interactions between the actors may often benefit from using allocate relationships for the allocation, as the Allocate Partition no longer constrains the diagram layout. Moreover, it also increases the reusability of the operating principle in the context of a different system with other elements to completely separate the activity diagram from the allocation and to specify the operating principle and its execution in the context of the system in a separate development artifact such as an allocation matrix, as *e.g.,* [JGW+21] suggests.

Though these diagram elements are optional, allocating all actions to logical elements responsible for the execution is required to apply the CUBE methodology successfully.

7.1.3 An Example Operating Principle of a Transportation Task

As an essential part of the system design, according to CUBE, is the definition of an operating principle for all features defined in the stakeholder needs specification. As it



(a) A operation principle model for the ‘Transport Persons and Goods’ feature from Figure 6.1.

(b) A feature-specific operation principle for the ‘Drive Vehicle’ use case from Figure 6.1.

Figure 7.1: Operating principles for the ’Transport Person and Goods’ feature and the ’Drive Vehicle’ use case from Figure 6.1.

is the essential value the vehicle offers to its users, this section models the ‘Transport Persons and Goods’ feature considering all its related use cases in the context of the running example section 4.2, which is also used as an example in [GKM+25]. While some applications of CUBE would start with the analysis of which action the system requires from scratch [GKS+21, JGW+21] only considering the use cases the feature includes, this thesis implements an alternative method in which all included use cases must be addressed on all possible paths of the operating principle, and extend use cases on at least one possible path (under the condition the extension point and the constraint specify). As the stakeholder needs model is an underspecification, a system modeler might add further actions that do not correspond to a use case in the model during this process.

Figure 7.1a provides the operating principle of the ‘Transport Persons and Goods’ feature. To this end, the diagram contains the activity ‘Transport Persons and Goods’, which has the parameters person and energy as input and person as output. All param-

eters are typed according to the functional modeling paradigm (*cf.* section 2.3). As the initial node indicates, the activity starts with the call behavior action ‘Enter the Vehicle’, which calls the behavior of the corresponding use case and accepts the person that the feature operating principle accepts as input. Then, the ‘Drive the Vehicle’ action describes the behavior executed while driving the vehicle using the energy the activity requires. After a drive, it must be decided whether the vehicle reached its destination using a decision node. If this is the case, the occupants ‘Exit the Vehicle’ and provide the person as the output of the feature again. By this, the diagram models the intuitive flow of actions required to transport people as [GKM+25] presents: To transport people, first, a person must enter the vehicle. Then, the vehicle must drive (or be driven to) the desired destination of the passengers, at which the vehicle must enable all occupants to exit the vehicle. Regarding the systems responsible for executing these actions, the diagram displays that the ‘Structure’ system is responsible for executing the ‘Enter the Vehicle’ and the ‘Exit the Vehicle’ actions. Hence, the systems engineers who model this system can be made responsible for modeling the operating principles and features of this action. As an identification of the system responsible for the execution of the ‘Drive the Vehicle’ action is not possible, a further model for this operating principle is required.

Consequently, Figure 7.1b depicts the model of this use case. The diagram describes the activity 'Drive the Vehicle' based on the example published in [GKM+25]. In this operating principle, a driver (that can be either human or machine) must be informed about the environment and the vehicle's driving state in the first step. As the driver as an external entity is not part of the 'Drive Vehicle' feature, the model does not represent the steps required to make a driving input decision. To this end, activity the diagram considers provides the resulting steering inputs in the form of a 'Steering Command', 'Acceleration Command', 'Brake Command', and a 'Gear Selection' as outputs. The vehicle must respond accordingly by adjusting its steering, acceleration, braking, and gear shifting to ensure the effective operation of a vehicle that uses these inputs. Since the specifications of the diagram only considers the solution-neutral functionality of the vehicle, the physical cross-relationships between steering, accelerating, and braking are deliberately not modeled in this diagram, as they depend on the vehicle's technical realization. Therefore, the diagram does not consider the different behaviors a motorcycle would, for instance, implement at simultaneous steering and braking inputs. For that reason, this operating principle is independent of the technical solution and, for instance, is also reusable for multiple other sorts of vehicles, such as passenger cars, trucks, or buses.

Table 7.3: A tabular view to express the use case behavior inspired by [Coc98].

UC Name	Name of the use case for which this table specifies a behavior.
Trigger	Event that triggers the execution of the use case
Pre-condition	Pre-condition that shall be fulfilled before the execution of the Use Case
Desired Behavior	The action sequence that is or shall be executed during the use case execution collaborating with the associated actors.
Post-conditions	Post-conditions that shall be fulfilled after the successful execution of the use case.

7.2 Modeling Operating Principles as Scenario Specifications

Because use cases only provide a rough boundary for behavior specification, early approaches on use case diagrams [Jac93, Coc01, AM00, Fow04, KG12] already developed several concepts to bridge the gap between the rough description of an activity a use case provides and the actual action sequence that is or shall be observable during the use case execution. Most prominently, tabular representations [Coc01, Fow04] and templates for the elicitation of these tabular descriptions [Coc98, SO01b] provide systems engineers with the ability to express these aspects. In the process of applying these concepts in the context of the operating principles modeled using the CUBE, this section presents a CUBE specific template inspired by the template from [Coc98], which this section extends to an additional view on use cases and operating principles which aims at bridging the gap between use case diagrams specifying the stakeholder value and activity diagrams that specify the operating principles of system features in the CUBE.

7.2.1 A Scenario Specification Template for Operating Principle Modeling

Because the template in [Coc98] provides some concepts that are redundant when used in the context of the stakeholder value models section 6.1 introduced, this section provides a reduced tabular representation of use case behavior specifications. Table 7.3 defines a view for use case behaviors. First, this tabular view connects the use case specification to the use case by its name. Then, it defines a trigger that will trigger the execution of this use case. As the pre-condition field describes, a pre-condition must be fulfilled before the use case can be executed. The desired behavior specifies the system's action sequence during the use case execution. After the execution of the desired behavior, a post-condition must hold. Since use case behavior descriptions typically serve not only for specification purposes but also for testing and verification, this element can serve to check the successful execution of the use case. Note that though the syntax and

Table 7.4: A tabular view to express the desired behavior as an action sequence during the use case execution.

Desired Behavior							
#	Actor	Action	Input	Output	Condition	Execution	Next
1	Actor that performs the action	Action this step executes	name: Type	name: Type	Condition	Sequential Parallel Alternative Exception	2
2	...	...	...	...	...	...	End

semantics definition allow, this view intentionally provides an additional mechanism to describe pre- and post-conditions.

Because the definition of system behaviors in an unstructured text does not provide much guidance for modeling the actions, Table 7.4 provides an additional table that captures the desired use case behavior. As a primary concept, each row represents an action to be executed, with a number as an identifier (ID) and a next cell for the action. As someone or something needs to execute each action, similar to the allocation, each action has an actor that is responsible for the execution. As this diagram reflects the execution of a use case, the actor must either be the system the use case is allocated to, a subsystem of this system, or a primary or secondary actor connected to the use case. To clarify this relation, the template in [Coc98] enumerates the available primary and secondary actors in the template. To avoid redundancy to the use case diagram, the template this section presents deliberately refrains from enumerating them again in the template, but refers to the actors the use case diagram already defines. To express the use cases that require or provide objects, Table 7.4 has the input and output fields in each row. Finally, the condition and the execution fields enable systems engineers to specify conditions for the control flow, such as alternative executions, parallel execution with join specs and expeditions that interrupt the execution in an interruptable region of actions, and the standard sequential execution order. For further insights, subsection 7.2.3 describes how these concepts relate to the elements an activity diagrams describe.

As a simplification for the users in the automotive industry with heterogeneous backgrounds, most of the examples in this thesis use a natural language definition of the expressions and triggers. However, further formalizing the content allowed in these fields is beneficial according to [TG14]. Moreover, to further enhance this template for modeling misuse cases, future works could extend this template with capture points as [SO01b] proposes. As a combined representation of the views for use case specifications (*cf.* Table 7.3) and the tabular behavior description (*cf.* Table 7.4) representation as a template for use case behavior elicitation, section C.1 provides a template to describe

use case behaviors.

7.2.2 An Example Operating Principle of a Transportation Task Represented and Decomposed using Template-Based Use Case Specification

To relate the operating principles subsection 7.1.3 depicts as an activity diagram, this section presents example applications of the use case specification template Table C.1 of the feature ‘Transport persons and goods’ and the ‘Drive the Vehicle’ use case. Moreover, this section shows how this template assists in decomposing system features using a decomposition example of the ‘Open the Door’ and the ‘Close the Door’ use cases in the context of the stakeholder value of the ‘Structure’ system.

Table 7.5: A tabular use case specification of the feature operating principle ‘Transport persons and goods’ Figure 7.1a presents in the form of an activity diagram.

UC Name	«feature» Transport Persons and Goods							
Trigger	The vehicle owner wants to transport people to a destination using the vehicle.							
Pre-condition	-							
Desired Behavior	The vehicle transports people and goods to a destination.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1	Vehicle	Enter the Vehicle	person: Material	-	-	Sequential	2
	2	Vehicle	Drive the Vehicle	energy: Energy	-	-	Sequential	3
	3a	Vehicle	-	-	-	Destination reached	Alternative	Final
	3b	Vehicle	-	-	person: Material	Destination not reached	Alternative	2
Post-conditions	The vehicle occupants reached the destination and left the vehicle.							

As Table 7.5 shows, displaying a feature operating principle of the feature ‘Transport Persons and Goods’ is principally possible. Moreover, this view also enables systems

engineers to specify triggers, pre-conditions, and post-conditions as an additional element. Moreover, the tabular representation of the table might reduce the entry barrier into systems modeling for engineers with less experience in system modeling or support experienced engineers in eliciting the required actions without the need to create and maintain the layout of an activity diagram manually. As an additional example, Table 7.6 presents the use case specification of the 'Drive the Vehicle' in Figure 7.1b. Because the actions in the feature operating principle in Table 7.5 refer to actions with the same name as the executed use cases, these concepts are traceable between the different templates. As a result, the use case specifications may be reused for different features and projects, thus reducing the individual effort required for system development. Because use case behavior specification tables also specify the required behavior of the actors in the environment, use case specifications might contain elements that the operating principles in CUBE typically do not reflect. For example, Table 7.6 specifies the driver's actions, which are not contained in the operation principle Figure 7.1b depicts. As a result, additional table filters might be required to remove these elements from the operating principle specification.

Moreover, as the use case specifications of the 'Enter the Vehicle' (*cf.* Table C.3) and the 'Exit the Vehicle' (*cf.* Table C.2) use cases reveal, these specifications are not only suited to describe the operating principle of a use case, but also for deriving use cases for the next decomposition layer such as the 'Close the Door' and 'Open the Door' use case, or for detecting forgotten use cases or refining existing use cases such as the 'Unlock the Vehicle' and the 'Lock the Vehicle' use case.

As these examples show, the definition of use case scenarios is not only beneficial for feature use cases but also for all other functional use cases (*i.e.,* all use cases apart from boundary condition use cases), as they not only provide reusable building blocks for the feature operating principles but also provide the use cases for the next decomposition level.

7.2.3 Generating Operating Principles from the Scenario Specification Templates (and vice versa)

Since the tabular use case behavior descriptions and the activity diagrams representing the operating principles of the features in CUBE are highly related, a derivation of operating principles from the use case scenario templates is possible. In other related works, [WMS16], for example, already presents similar ideas to formalize the execution of use cases, or [Whi06] describes an approach to precisely specify the system behavior during a use case execution using this relationship.

Following the first intuition, each scenario corresponds to an activity in the activity diagram. Each row in the table then represents an action the actor executes. To represent

this relation, each row in the table corresponds to a diagram element in the activity diagram, and each entry in the 'Actor' column corresponds to a block in the logical system architecture. Therefore, the relation in the activity diagram is expressed by a relationship between the activity diagram element and the block of the executing system, subsystem, or actor. The behavior's initial action is represented by the first row of the table, so a control flow must connect the initial node of the activity diagram with the action that results from the first row in the table. For sequential execution, the current action in the table must have a control flow to the actions of the row specified in the next column. For parallel execution, all actions of consecutive rows with 'Parallel' execution correspond to forked actions in the activity diagram that are forked before continuing the execution. The first and last rows of the template may optionally encode these concepts to further specify a name in the fork node or a name and join specification in the join node. Following the same principle, the 'Alternative' execution corresponds to actions behind a decision node that need to be merged in a merge node before the execution can continue. As for the for and join nodes, additional optional rows may serve to specify names for decision and merge. In contrast, each row without a predecessor must have a condition to determine the condition under which the path is taken in the activity diagram after the decision node. To encode the object flows the table represents in the diagram, all inputs and outputs of the table correspond to action pins of the corresponding actions in the diagram, which have an object flow from the providing element with the same name and type to the requiring element with the same name and type. Finally, all unconnected inputs and outputs correspond to activity parameters in the activity diagram that provide those inputs and forward the outputs via activity parameters. As a more formal description of this intuitive view, section C.2 provides the transformation rules for transforming the behavior descriptions in tabular use case scenarios to activity diagrams.

Although the transformation rules are usable to derive activity diagrams from the desired behavior specification as well as desired behavior specification specifications from activity diagrams, the resulting diagrams may also contain actions the actors perform and are therefore not part of the system specification. As a result, removing the actions performed by the actors in the environment from the use case specification table is sometimes necessary, rendering the transformation unidirectional. Therefore, it is a good practice to either utilize use case specification templates only to describe both system behavior and inputs and reactions the actors provide and derive activity diagrams if required or to accept that only the elements the system performs are specifications. In contrast, the remaining actions of the environment are dummy actions that might become relevant for testing the system behavior in later phases of the behavior without any guarantee for the system.

7.2.4 A Concrete Textual Syntax for Scenario Specification Templates using Activity Diagrams

As the previous section demonstrated, because tabular use case specification and the activity diagram provide the same concepts, this idea was extended in a thesis to developed a concrete textual syntax to describe use case scenarios [Bla23]. Drawing on these results, this thesis extends integrates the additional concepts Table 7.1 provides, including interruptible activity regions, send signal actions, accept event actions, and accept event timer actions. section C.3 presents the grammar that specifies the activity diagrams used for the behavior specification in Listing C.1, and the grammar for the use case specification in Listing C.2. A systems engineer can use this concrete syntax to model the use case scenario from Table 7.6 as follows.

UCD-ST

```
1 scenariotemplate DriveTheVehicle {
2     primary Driver;
3
4     trigger "A driver starts the vehicle in order to move it
5              to a destination.";
6
7     pre "The driver is in a safe position within the vehicle,
8          all people and goods to be transported are safely
9          stored in the vehicle and the vehicle has a sufficient
10         energy level to reach the destination";
11
12    // The driver drives the vehicle to its destination
13    behavior activitydiagram DriveTheVehicle {
14        allocate Vehicle {
15            InformDriverAboutEnvironmentAndDrivingState
16             {
17                out vehicleInfo : Data;
18            }
19
20            fork StartDrive;
21
22            SteerVehicle {
23                in steerCom : Data;
24            }
25
26            AccelerateVehicle {
27                in accCom : Data;
28            }
29
30            BrakeVehicle {
31                in brakeCom : Data;
```

```
32              }
33
34              ChangeGear {
35                  in gearCom : Data;
36              }
37
38              join EndDrive;
39          }
40
41          allocate Driver {
42              DetermineControlCommands {
43                  in envInfo : Data;
44                  in vehicleInfo : Data;
45                  out steerCom : Data;
46                  out accCom : Data;
47                  out brakeCom : Data;
48                  out gearCom : Data;
49              }
50          }
51
52          initial -> InformDriverAboutEnvironmentAndDrivingState;
53          InformDriverAboutEnvironmentAndDrivingState
54              -> DetermineControlCommands;
55          DetermineControlCommands -> StartDrive;
56          StartEngine ->
57              SteerVehicle &
58              AccelerateVehicle &
59              BrakeVehicle &
60              ChangeGear;
61          SteerVehicle &
62          AccelerateVehicle &
63          BrakeVehicle & ChangeGear
64              -> EndDrive;
65          EndDrive -> final;
66      }
67
68      post "The vehicle reached its destination.";
69
70  }
```

Listing 7.1: An example use case scenario representing table from Table 7.6 in textual notation.

Note that the concrete syntax reverts the definition of primary and secondary actions to the use case specification template, as the concrete use case diagram (*cf.* subsection 6.2.6) does not include these concepts. Consequently, the example in Listing 7.1

starts with the keyword `primary` before it expresses the contents of the use case scenario.

Table 7.6: A tabular use case specification of the operating principle for the use case 'Drive the Vehicle' Figure 7.1b presents in the form of an activity diagram.

UC Name	Drive the Vehicle							
Trigger	A driver starts the vehicle to move it to a destination.							
Pre-condition	The driver is in a safe position within the vehicle, all people and goods to be transported are safely stored in the vehicle, and the vehicle has a sufficient energy level to reach the destination.							
Desired Behavior	The driver drives the vehicle to its destination.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1	Vehicle	Inform the driver about the environment and driving state	-	vehicleInfo: Information	-	Sequential	2
	2	Driver	Determine control commands	envInfo: Information; vehicleInfo: Data	steerCom: Data; accCom: Data; brakeCom: Data; gearCom: Data	-	Sequential	3
	3a	Vehicle	Steer the Vehicle	steerCom: Data	-	-	Parallel	Final
	3b	Vehicle	Accelerate the Vehicle	accCom: Data	-	-	Parallel	Final
	3c	Vehicle	Brake the Vehicle	brakeCom: Data	-	-	Parallel	Final
	3d	Vehicle	Change the Vehicle	gearCom: Data	-	-	Parallel	Final
Post-conditions	The vehicle reached its destination.							

7.3 Applying Operating Principle Models to Efficiently Design Automotive Systems

To refine and evaluate the application of the methods this chapter proposed for in the automotive industry, this section summarizes several results from the Epsilon, Delta, and Zeta projects and relates them with the publicly available VTOL models from [JGW+21], which presents the feature-driven operating principles for some features [JGS+20] elicited. Since previous works already demonstrated the applicability of activity diagrams for operating principle modeling [JGS+20, GKS+21, JGW+21, MSG+22] using the CUBE, and several publications show that activity diagrams are already established as a means to describe operating principles in the automotive industry [WMGT11, WMS16, BVR17, BV17, DGH+19], this section focuses on replacing natural language specifications with activity diagram models.

As for the stakeholder value application, subsection 7.3.1 provides a generation method to generate natural language requirements from activity diagrams and modeling guidelines for modeling activity diagrams as operating principle models that lead to (grammatically) correct system requirements. To evaluate the correctness of the generated requirements, this section builds on the evaluation scheme subsection 6.4.2 provides to evaluate the applicability of the generation method in the Delta, Zeta, and Epsilon projects as well as on the VTOL model from [GKS+21]. Since the Epsilon project additionally provides a model-based specification for a system initial build on a document-based approach, subsection 7.3.3 compares the results in both methodologies. As for the stakeholder value evaluation, this section draws on the evaluation results from [Zab23] for the evaluation of the natural language requirements generated in the Zeta and Delta project.

7.3.1 Generating Textual Functionality and Interface Requirements using Activity Diagrams

Based on the transformation rules [Zab23] presents, generating textual functionality and interface requirements in the context of feature operating principles is possible. Analogous to the generation of stakeholder value requirements subsection 6.4.1 describes, these transformation rules insert parts of the activity diagram models that express the operating principles as features into the respective fields of the SPECTRE template (*cf.* Figure 5.1) and provide naming guidelines to ensure the generation of grammatically correct requirements. Because of an agreement in the Zeta and the Delta project to only specify a core functional requirement for each action and interfaces of the operating principle and provide the execution order as a diagram, Table 7.7 provides templates for the requirements of these elements. The following section only evaluates the requirements

these templates generated for all projects. If a requirement generation of the other elements in Table 7.1, section D.2 provides additional transformation rules for additional textual requirements that describe the control flow.

Table 7.7: AD requirement generation templates to derive natural language requirements for functionality and interface requirements from activity diagram models based on [Zab23].

Requirement Generation Rules		
ID	Graphical Notation	Requirement Template
AT1	<Feature> AD <in1>: <In1Type> <inN>: <InNType> <action> <out1>: <Out1Type> <outN>: <OutNType>	The feature '<Feature>' shall <action> using <in1> of type <In1Type>, ..., and <inN> of type <InNType>, providing <out1> of type <Out1Type>, ..., and <outN> of type <OutNType>.
AT2	act [activity] ActivityDiagram [Feature]	The feature '<Feature>' shall execute all actions according to the execution order in <ActivityDiagram>.
AT3	<source feature> parameter: Type <destination feature> parameter: Type	The feature <source feature> shall provide the interface parameter <parameter> of type <Type> to feature <destination feature>.
AT4	<source feature> parameter: Type <destination feature> parameter: Type	The feature <destination feature> shall receive the interface parameter <parameter> of type <Type> from feature <source feature>.
Naming Conventions		

Table 7.7: AD requirement generation templates to derive natural language requirements for functionality and interface requirements from activity diagram models based on [Zab23].

ID	Naming Convention	Comment
AN1	**<Process verb>+ [details on the verb or the object]? <object>+ [details on the object]?**	This naming convention for activities and actions aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (*cf.* section 5.2)

Therefore, Table 7.7 only describes the generation templates that describe requirements for each action in an activity diagram (including their inputs and outputs) and templates for the generation of requirements that describe the interfaces between the different features of a system. As the resulting specification is intended to be split according to the CUBE artifact split in the process structure model in Figure 3.3, the requirement that feature A shall provide to feature 'B' with information 'I' is duplicated for the receive case in AT4 to make this requirement verifiable in a module test for the developers of the feature that shall receive 'I' as input. Though the generation of the requirements regarding the allocation is part of the logical architecture requirements, and, therefore, part of the generation subsection 8.3.1 describes, the transformation rules do not apply to actions that represent system use cases (or are allocated to the current system under consideration). Consequently, no additional requirements for the diagram Figure 7.1a may be generated as the requirements stakeholder value requirements already cover all requirements, and a generation of this requirement would result in the following erroneous requirement.

Example 13 (Erroneous SPECTRE Requirement for the 'Enter the Vehicle' action in Figure 7.1a)**.** *The feature 'Transport Persons and Goods' shall Enter the vehicle.*

Hence, the requirement in 12 from the stakeholder value requirements generation already covers the requirement in 13. Consequently, only a generation of the requirements from the diagram in Figure 7.1b activity diagram is required, leading, for example, to the action requirement requirement:

Example 14 (Operating Principle SPECTRE Requirement for the 'Inform the driver about environment and driving state' action in Figure 7.1b)**.** *The feature 'Drive the Vehicle' shall Inform the driver about environment and driving state providing 'vehcileInfo' of Type 'Data'.*

In addition to this requirement, the execution requirement for the whole diagram 'Drive the Vehicle' action in Figure 7.1b is then in 15.

Example 15 (Execution order requirement for the diagram in Figure 7.1b)**.** *The feature 'Drive the Vehicle' shall execute all actions according to the execution order in Figure 7.1b.*

Because the generation of requirements that refer to images of diagrams is challenging to evaluate from a semantic viewpoint without speaking about the diagram's contents, which is not part of the evaluation this thesis conducts, requirements of this form were excluded from the evaluation the next section presents. If a further specification of the execution is required, section D.2 provides a full-featured set of transformation rules that not only additionally reflect the execution order of the actions in the diagram but also take the executing elements from the logical architecture according to using the modeling elements in Table 7.2 into account. To this end, the transformation rules in section D.2 follow the intuition Figure 7.2 depicts. According to the structure in Figure 7.2, a requirement consists of three essential parts:

1. The event that triggers the execution of an action at the beginning of the SPECTRE requirement in Figure 7.2,
2. the actual action that is executed depending on inputs and providing outputs in the main sentence of the requirement, and
3. the additional logical conditions determining whether an execution is possible in the diagram.

Note that the transformation in Figure 7.2 does not formulate the definition of the object flows as a specific requirement, as they follow from the definition of the individual action requirements. Thus, the definition of object flows follows a provider, receiver logic, where all receivers receive an object as soon as the provider provides it.

7.3.2 Correctness Evaluation of Generated Operating Principle Requirements in Model-Based Projects

To demonstrate the applicability of the method to derive correct natural language requirements from operating principle models, as subsection 7.3.1 presents, this section presents an evaluation of the generated requirements in the context of the Zeta, Delta, VTOL publication in [JGW+21], and the models in the Epsilon projects. The evaluation was conducted analogous to the stakeholder value requirements in subsection 6.4.3 using the evaluation scheme from subsection 6.4.2 and the evaluation results from [Zab23] for the Zeta and the Delta project. In addition to these insights, this thesis also provides evaluation results for the VTOL and the Epsilon projects.

As the results of this evaluation in Figure 7.3 reveal, there is a clear project-independent trend for generating correct requirements on the operating principle level. First, most of the requirements are usable in the projects without any correction, as

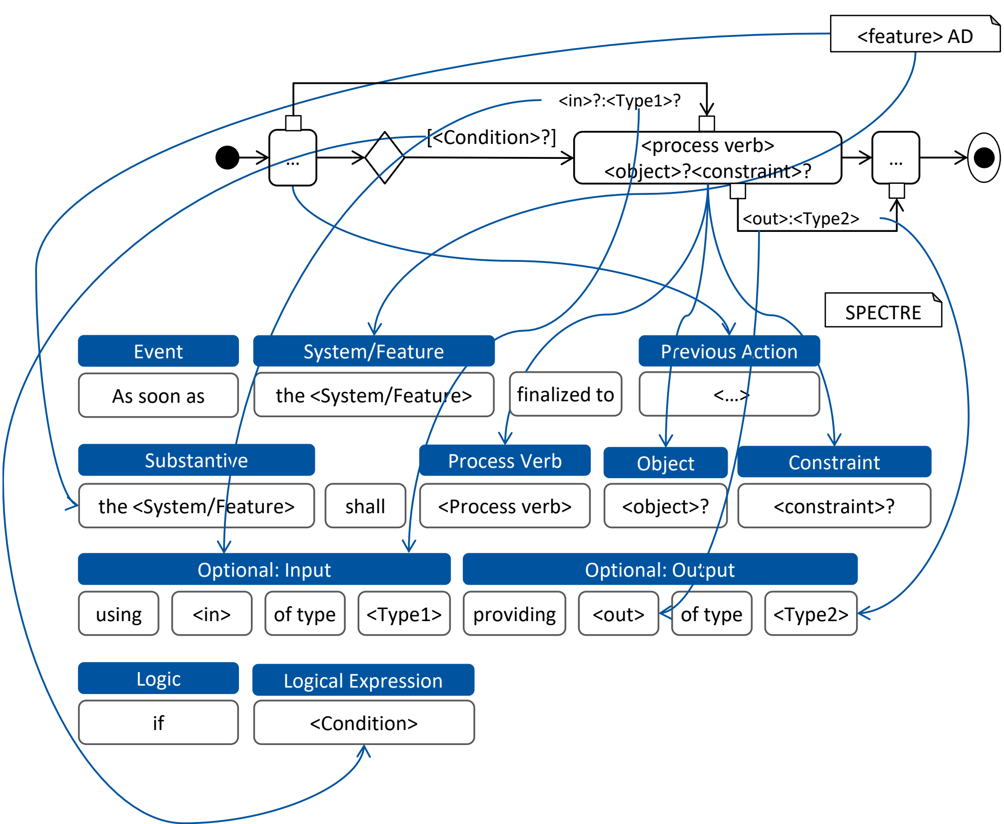
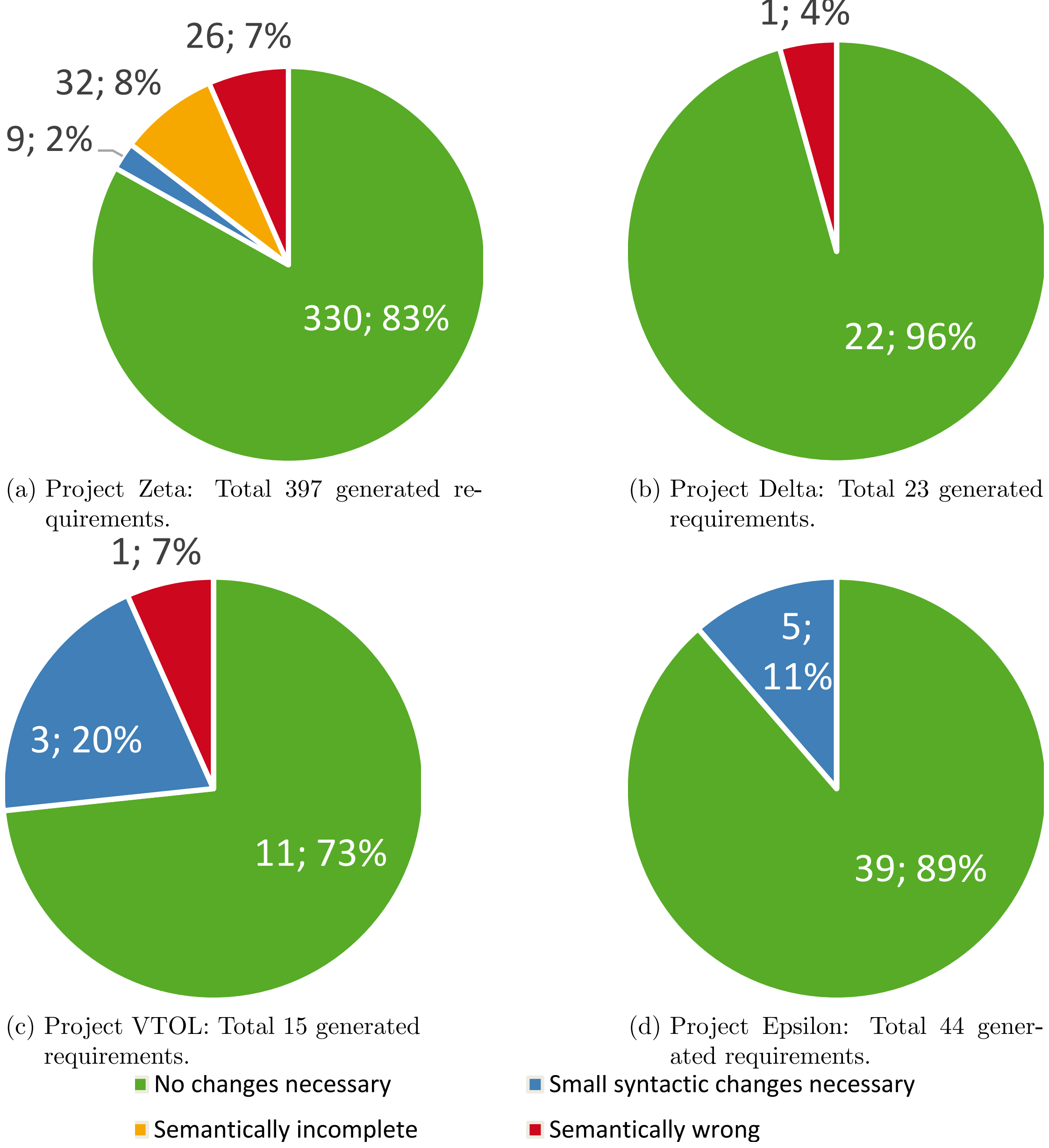


Figure 7.2: A straightforward transformation that inserts the names of the actions in the diagram and the execution order in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1).

the generation method from Table 7.7 resulted in correct requirements in most cases. Second, the number of semantically incomplete requirements decreased on this level. Third, fewer requirements with minor syntactic errors were generated, which is a solid contrast to the stakeholder value requirement evaluation in Figure 6.10, where no project-independent trend is observable and many requirements were erroneous. To explain this observation, [Zab23] assumes that operating principle requirements, compared to the stakeholder requirements, have a lower abstractness of the activities, which seems more accessible for the modelers to capture. Moreover, [Zab23] suspects that it might be easier for the system modeler to directly describe the actions a system performs in the activity diagram without changing the perspective between stakeholders and their needs from the system, which might eliminate this source of errors.

These assumptions also manifest in the error reason evaluation Figure 7.4 presents. As



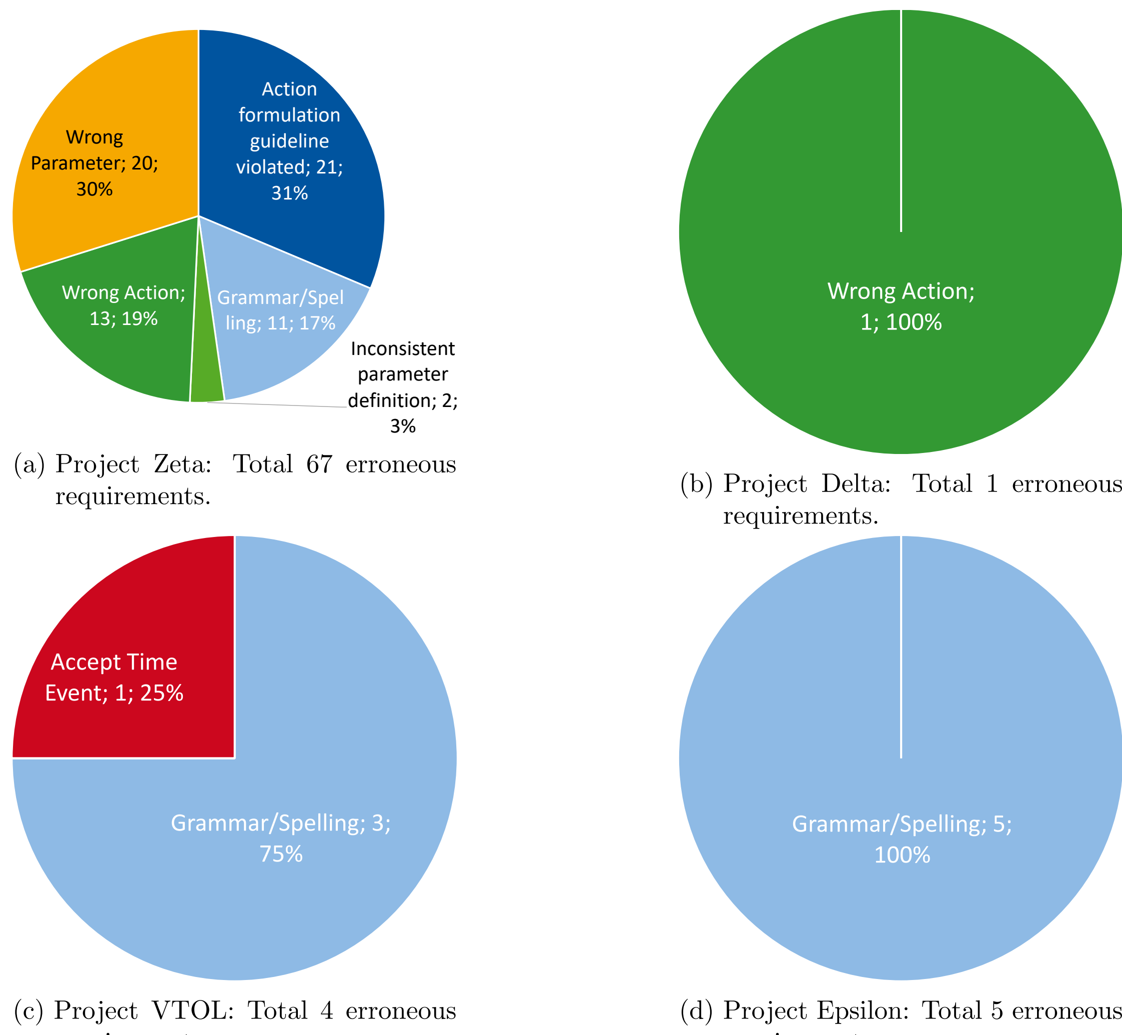
(a) Project Zeta: Total 397 generated requirements.

(b) Project Delta: Total 23 generated requirements.

(c) Project VTOL: Total 15 generated requirements.

(d) Project Epsilon: Total 44 generated requirements.

Figure 7.3: Correctness evaluation results of the operating principle requirements generated in the Zeta, Delta, VTOL, and Epsilon projects.



(a) Project Zeta: Total 67 erroneous requirements.

(b) Project Delta: Total 1 erroneous requirements.

(c) Project VTOL: Total 4 erroneous requirements.

(d) Project Epsilon: Total 5 erroneous requirements.

Figure 7.4: Results of the error reason classification in the Zeta, Delta, VTOL and Epsilon projects.

Figure 7.4a reveals, a violation of the guideline AN1 Table 7.7 of some modelers in the Zeta was responsible for the most significant class of errors in the 'Action formulation guideline violated' class. In this class of requirements, the modelers of the activity diagram failed to name actions using a process verb in the correct form and place writing, for example, that a feature shall "monitoring and checking incoming data' instead of the correct "monitor and check" data. The next most significant class of wrong requirements resulted from wrong or missing parameters in the model. For example, the modelers of a feature related to energy management failed to provide this energy to the action. In contrast, the modelers concerned with another feature failed to rename object flows the modeling tool generated by default. Furthermore, most misuse cases from the stakeholder value also lead to wrong actions. The operating principle of these features performed leads to the 'Wrong Action' class. As for stakeholder value requirements, the review of the model did not find some grammatical and spelling mistakes, leading to the errors in the 'Grammar/Spelling' class. Finally, an object flow with incompatible types was not found in the model's evaluation, leading to two requirements with inconsistent parameter definitions.

In the Delta project, as Figure 7.4b depicts, only one requirement was erroneous and caused by an action with a default name the modeling tool provided.

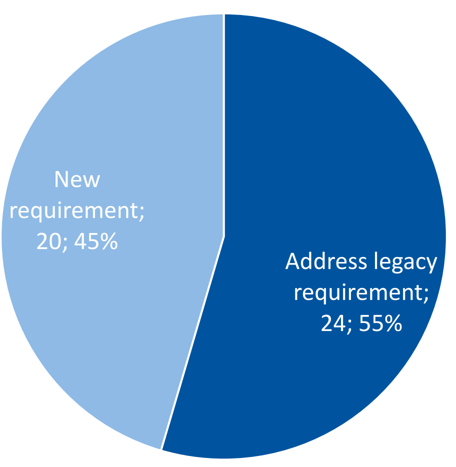
Finally, in the VTOL (*cf.* Figure 7.4c) and Epsilon (*cf.* Figure 7.4d), almost all erroneous requirements result from minor grammatical errors in the names of the actions. For example, the action "Initialize Communication System Check" does not lead to a correct requirement as an article is missing a grammatically correct sentence. As the only exception in the VTOL model, [JGW+21] presents, the time constraint of the accept event action is not valid in natural language and, therefore, named in a separate class 'Accept Time Event' in Figure 7.3c.

As these insights reveal, a derivation of operating principle requirements using the templates and guidelines from Table 7.7 is possible and aids the modelers in uncovering minor errors.

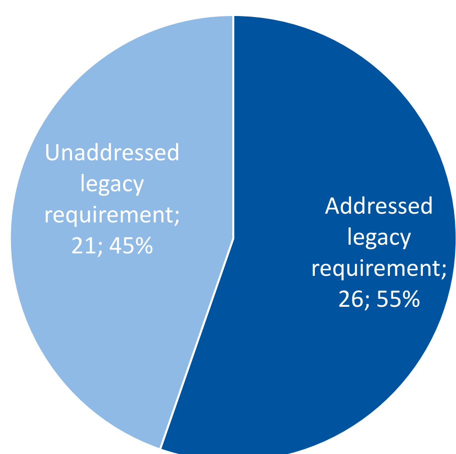
7.3.3 Completeness Evaluation of Generated Operating Principle Requirements in Model-Based Projects compared to a Document-Based Specification

As an additional demonstration that system models can replace natural language requirements in industry projects, this section compares the generated and corrected operating principle requirements in the Epsilon projects with the legacy requirements the project team received as input.

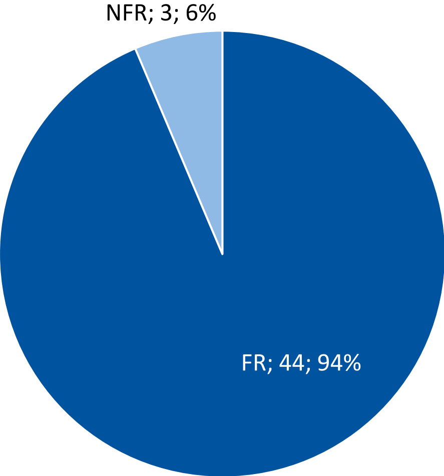
As the results in Figure 7.5 show, the generated requirements address almost as many requirements in the legacy requirements as they add to them and vice versa. Although



(a) Evaluation how many of the 44 generated requirements in the Epsilon project address one of the 47 legacy requirements.

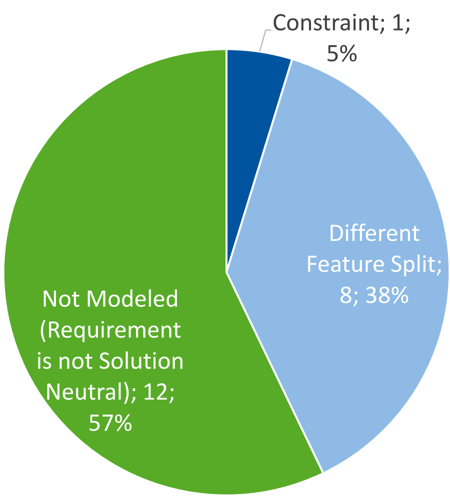


(b) Evaluation how many of the 47 legacy requirements in the Epsilon project address on of the 44 generated requirements.

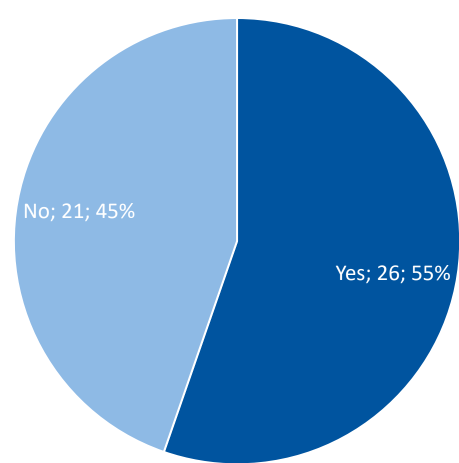


(c) Ratio of requirements reflecting functionalities (FR) versus requirements reflecting constraints, specific qualities or performance requirements (NFR) according to the requirement classification scheme in [Gli07]: Total 47 requirements.

Figure 7.5: Results of the completeness evaluation comparing the model-based stakeholder value specification from the Epsilon projects with the document-based legacy requirements the project used as input.



(a) Reason evaluation for the 21 not generated legacy requirements.



(b) Evaluation how many of the 44 generated requirements correctly cover one or more of the 47 legacy requirements.

Figure 7.6: Results from the interviews with the modelers in the project to determine the reasons for not modeling the requirement and to evaluate if the generated natural language requirement correctly covers the legacy specification.

more than half of the operating principle requirements in the legacy documents are addressed by the natural language requirements, the approach from subsection 7.3.1 generated a significant number of requirements that are still not addressed by these requirements. As Figure 7.5c illustrates, this cannot only be explained by constraints, specific quality, and performance requirements because only three of those requirements are contained in the legacy specifications. Therefore, the original modelers were asked to explain the differences, which led to the outcome Figure 7.6a presents. This interview revealed that one requirement, a constraint to the solution space of the operating principle, was modeled but not included in the activity diagrams, as the SysML specification [Obj19] requires these elements to be placed in SysML BDDs. Moreover, the project team deliberately did not address some of the requirements, as the input they received contained concrete solutions specific requirements in the operating principle, which are not only deliberately not modeled in the CUBE, but also no longer reflect the state of the art of this kind of systems today. Therefore, the project team discussed these requirements with the requirements engineers, who specified the requirements for the system in the document-based process and jointly decided not to integrate these aspects into the reworked model-based specification. Another group of requirements in the document-based specification referred to stakeholder values that a system architect with

the knowledge from the development of the existing system would no longer assign to the feature under development. As a result, the project team intentionally moved these aspects to a feature other engineers developed in another team outside of the project under consideration.

Comparing the generated requirements with the legacy requirements also revealed, as Figure 7.6b displays, that some requirements in the also described aspects related to the execution order of the actions in the operating principle. Consequently, these aspects can be modeled and generated using the extended generation method for natural language requirements Table D.2 presents. However, the evaluated requirement set only reflects this aspect with reference to the model, which the reviewing requirements engineers did not consider during their evaluation.

Although this correctness evaluation is not representative of all projects in the automotive industry, the results indicate that activity diagrams that specify operating principles are sometimes sufficient to replace natural language specification documents created by document-based systems engineering processes.

7.3.4 An Experience Report on Using Tabular Scenario Specification Templates and Activity Diagrams

To additionally provide insights on the application of the tabular specification as a means to represent operating principles early in the development as use case behavior description tables, this section provides some experiences from the 4 modelers that applied these models in the Zeta project.

Contrary to what was intended by the definition in section 7.2, the specification method was not used by the inexperienced system engineers in the project but by the more experienced system engineers, who found the specification method using tables to be an efficient way of collecting information about the application of features and to define reusable use case behaviors. This not only aimed at saving time in laying out the graphically represented activity diagrams but also to improve communication with stakeholders who had never seen one before.

Unfortunately, the project did not provide reliable data on these savings. Moreover, the small number of projects and participants and the missing environment for reproducing these results are also significant threats to the validity of these insights. Nonetheless, this experience report might motivate further research *e.g.,* in the context of the upcoming SysML v2 specification [Obj23], which provides a textual representation of both use case and activity diagrams comparable to the concrete textual syntax subsection 7.2.4 provides.

7.4 Interim Conclusion and Discussion

In summary, this chapter establishes SysML activity diagrams in Table 7.1 and Table 7.2, and use case scenario specifications in Table C.1 as a way to model the desired system behavior as the operating principle of a system. This section answers **RQ-3.3** (“How to model operating principles?”) by not only providing the required diagram elements in the SysML diagram but also by providing an additional method and textual syntax for use case scenario specifications in subsection 7.2.4. Although the suggested way of modeling operating principles was not applied in all projects this thesis considers, the experiences from subsection 7.3.4 indicate that the application in other projects might be beneficial for other projects and could be a relevant topic for future research.

Moreover, subsection 7.2.3 answers **RQ-4.1** (“Can operating principles be derived from use case scenario specifications?”) by providing a bidirectional transformation from operating principles to use case scenario specifications that connects both concepts and makes a derivation possible. Even though the considered use case specifications draw on the concept of use case specifications [Coc01, Fow04] that are more related to activity or sequence diagrams than to use case diagrams [KRW22], the transformation shows that use cases and system actions are closely connected. It provides additional concepts to increase the efficiency.

subsection 7.3.2 and subsection 7.3.3 demonstrate that a derivation of natural language requirements is possible using the method section 7.1 provides when combined with the additional guidelines from subsection 7.3.1. Tough Figure 7.6b reveals that the generated requirements with the legacy requirements also described aspects related to the execution order of the actions in the operating principle, which were not generated using the simple method from subsection 7.3.1, the expert interview and additional experiences in other projects indicate that the additional concepts section D.2 might be sufficient to close this gap. Because even the simplified requirements specify the execution order using the diagram itself, the results are still comparable to a document-based system specification as investigated by **RQ-4.4** (“Are the models that result from a model-driven specification comparable to document-based specifications?”), these documents often rely on images to describe these aspects as, for example, [Lev09] criticizes.

Finally, regarding the fulfillment of the methodical requirements from section 3.3, several aspects contribute to the fulfillment of the methodical requirements. Concerning **MR-1** (“Specification models created using the method must be reusable to realize reduced time-to-market constraints”), CUBE relies on solution-neutral operating principles for modeling system behaviors. For that reason, the systems engineer who models the actions within an operating principle should model them in a way that they are independent of the technical implementation of the system. Therefore, this method makes it possible and intended to reuse operating principles in other systems engineering projects

with comparable systems under development. As the experience in the Epsilon project indicates it is a good practice to find a suitable balance between solution neutrality and concrete applicability. Since CUBE introduces solution neutrality not only as a mean to increase re-usability, but also to leave freedom in the technical design decision, it is important that an operating principle neither impedes the reuse of the operating principle in another project nor unnecessarily depriving the engineers responsible for the technical realization of the freedom to find a suitable technical solution. However, a maximal solution neutral operating principle might not be the optimal solution for every case. In the Epsilon project for example, the systems engineers intentionally did not model the operating principle of the general chemical combustion reaction of the as a main feature of the powertrain, but decided modeling a specialization of this case, where fuel and air are already fixed as reaction partners, to simplify the operation in this particular use case. This is possible because automobile manufacturers have already introduced petrol and diesel engines with air as an oxidizer as a standard solution for the internal combustion engine and the inclusion of additional fuels and oxidizers in this reaction would not bring any further benefit in this application compared to the additional complexity that this consideration brings. To further enable the reuse of the diagram elements this chapter suggests in Table 7.1, section 7.1 intentionally splits the diagram elements to the system independent behavior description in Table 7.1 and the system allocation in Table 7.2. From a critical perspective, however, some challenges remain for future research. For example, how to split or tag the allocation from the operating principle from a language design point of view or how to package the operating principle for project-independent reuse and manage changes from an engineering perspective. Regarding **MR-2** ("Specifications must be adaptable to specific markets or vehicle projects"), the operating principle model does not support any variability mechanisms. However, the overall feature can be used or reused, or additional variability mechanisms can be integrated into the activity diagram or use case specification language in future works. Concerning **MR-3** ("Specification models must be consistent"), no additional methods are integrated into this section's concepts. Therefore, future works could take up this point for additional investigations. To support traceability as required by **MR-4** ("Specification models must be bidirectionally traceable"), each operating principle corresponds to a feature on the stakeholder value specification level, and a logical component in the logical architecture executes each action within the operating principle. Therefore, tracing features to actions and systems, as well as systems to actions to features, is possible within the model. Finally, **MR-5** ("Creating specification models must be possible without redundant manual modeling tasks") is implicitly addressed by providing multiple solution-independent and reusable feature operating principles and the instantiation of activities in actions the SysML already provides in [Obj19], but could further be improved by integrating additional concepts for activity reuse within an action instantiation and package or library concepts.

Chapter 8

Deriving Logical System Architectures from Operating Principles

This chapter presents methods to specify the logical architecture of a system according to the CUBE methodology. To this end, this section focuses on describing the structure of a system as a SysML IBD and the dynamic architecture in the form of a SysML STM as Figure 3.11 motivated based on [BGM+09] and [GORW23]. As a measure to increase the reusability of the logical architecture (*cf.* **MR-1** "Specification models created using the method must be reusable to realize reduced time-to-market constraints'), this chapter further introduces the concept of a Logical Reference Architecture (LRA).

This logical architecture concept draws on previously established concepts from the software product line engineering [PBL05, WLRW15], as well as additional concepts for software- and system architecture definition [RR02, GH00, Oqu04, RRW13, BHH+17]. Regarding the semantics of this architecture, the behavior definition this thesis uses draws on SysML state machine diagrams, which are related to Mealy [Mea55] and Moore [Moo56] automata, for which several semantic definitions exist [BCR07, NSC+07, DEKR19, Kau21]. In combination with the structural aspect defining ports and systems with automata, this type of model is well-suited to describe and specify the behavior of cyber-physical systems[Lee08, Alu15] the automotive industry develops [CAFC+16, WBJ08]. Although this thesis focuses on the SysML [Obj19] in **RQ-3.4**, several other languages exist to express system architectures. For example, AutoFocus [HF07, AVT+15], EmbeddedMontiArc [KRRvW17, KRSvW18, vW20, Kus21], MontiArc [HRR12, Hab16], or its extension MontiArcAutomaton [Rin14, Wor16, BKRW17, BKRW19] use the Focus [BS01, Bro10, RR11] framework as a semantic foundation for system modeling and specification.

This chapter is structured as follows to specify a logical architecture for a system. First, section 8.1 introduces the concept of a logical architecture and the required model elements to answer **RQ-3.4** ("How to model logical architectures?"). Next, section 8.2 addresses **RQ-4.2** ("Are logical architectures derivable from operating principles?") by introducing a transformation method to derive the logical architecture of a system from an operating principle specification. Finally, section 8.3 applies the concepts on the mod-

els and derived textual requirements in natural language from the Zeta, Delta, VTOL, and Epsilon projects to evaluate the applicability of the method in the context of the efficiency (*cf.* **RQ-4.3** “Is the efficiency increased by this automation?”) and compatibility to document-based specification (*cf.* **RQ-4.4** “Are the models that result from a model-driven specification comparable to document-based specifications?”).

8.1 A Logical Reference Architecture for Element Allocation

A reference architecture of a system fulfills a similar purpose to a reference architecture for software product lines as described in [PBL05]: Capturing the high-level design for the applications of the system. A critical aspect of a system architecture is the *core architecture* [PBL05] or *base architecture* [WLRW15]. In this architecture type, the implicit knowledge and design decisions from the top-level requirements should become explicit [WLRW15]. For example, an automotive company does not produce all possible means of transportation, only a reduced subset that fits its product line, such as cars, trucks, or motorcycles. Thus, when an automotive company develops new mobility concepts and features, the technical design decision that only one product from the company's product line will be accepted is implicit. In a reference architecture, these decisions manifest as a static or structural decomposition of a system, which provides standard components that all systems of this type should have. In addition, reference systems usually have a dynamic or behavioral part, which predestines the joint system states and transitions for the localization of actions within an operating principle. For example, the standardization of labeling the electric terminals in automotive wiring within vehicle electronics [DIN14] implicitly creates, in addition to the static solution space (the existence of circuits, specific terminals, and lines), a particular state space that a behavioral description of the logical reference architecture makes explicit. Therefore, this thesis defines the Logical Reference Architecture (LRA) as follows:

Definition 27 (Logical Reference Architecture (LRA))**.** *The Logical Reference Architecture (LRA) of a system of interest defines the logical structure and behavior of a system to make the high-level design explicit for the applications of the system.*

The logical reference architecture this thesis presents aims to fulfill **MR-1** and **MR-2** by providing predefined system components to which systems engineers can allocate actions from an operating principle. Since the logical reference architecture aims to be independent of the technical solution, the same logical architecture may serve as a reference for multiple technical architectures, allowing the required adaptability by **MR-2**. By generating further dependencies to existing components when implementing new features as described in section 8.2, a re-adoption to a different LRA also becomes possible, which increases the adaptability required in **MR-2**.

Like other system architectures [RR02, GH00, Oqu04, RRW13, BHH+17], the LRA as described in this thesis has a static *i.e.,* structural and a dynamic *i.e.,* behavioral part. The static part defines the system's logical decomposition and specifies the subsystems for the next decomposition level. The dynamic part of a system is expressible in the form of a state machine that describes the states and transitions of the system during its life cycle.

8.1.1 Structural Logical Reference Architecture

In the context of this thesis, the assumptions and constraints regarding the system decomposition become explicit by the structural Logical Reference Architecture (sLRA). The sLRA describes the structural components of a system and its subsystems. This reference architecture is a logical architecture [GHK+08a, BGK+09] that is reusable for many systems. Using logical components as a substitute for their actual product implementation makes the architecture reusable over project boundaries, as required by **MR-2**. For the scope of this thesis, the sLRA is defined as follows.

Definition 28 (structural Logical Reference Architecture (sLRA))**.** *The structural or structural Logical Reference Architecture (sLRA) of a system of interest defines the system's structural components from a logical perspective to define its subsystems. It is derived during or from previous system development projects, reusable for similar projects, and should facilitate the system decomposition.*

In contrast to the static architectures in variability analysis [Sch19, KNS+21], these components of the sLRA are not necessarily constant over the run-time of the system but predestine the structural components of a system and its subsystems. By this, the sLRA enables systems engineers to reuse their knowledge from previous system development projects with the systems. As mentioned in the definition, the sLRA is either developed during a new system development project or derived from the knowledge from previous system developments.

By this, implicit design decisions (*e.g.,* the constraint that automotive manufacturers produce cars) become explicit, and the implicit company knowledge on how to build cars becomes manifested. Since this embodiment of previously developed solutions has the risk of focusing too much on existing technical solutions, this architecture consists of logical components instead of technical ones. Following this philosophy, the integral aspect of the LRA is that it defines logical architecture components instead of technical components based on the product realization. By this, the risk of being stuck to existing technical implementations should be reduced, and the immense product portfolio within different development projects should become reusable.

Logical System Decomposition

Since natural language is ambiguous [Cho20], the logical meaning of a component is not uniquely identifiable by its name alone. Thus, there are multiple semantically equivalent ways to specify a logical system. For example, propulsion and drive systems may or may not result in the same set of possible implementations depending on the specifier's and the implementer's understanding. Apart from this fundamental ambiguity of natural language, different decomposition approaches exist in the literature. The following summarizes some of these understandings according to the different CUBE layers:

1. Customer-Oriented Logic: This understanding decomposes the system according to the system functionalities a system stakeholder can experience during the interaction with the system. It is, for example, applied in the identification of logical architectures for micro-serves as presented in [GKGZ16], or [THLL18] and results in a feature or scenario-oriented system architecture that decomposes the systems along the stakeholder value layer of CUBE on the logical architecture layer of CUBE.
2. Operating Principle Logic: According to this understanding that is *e.g.,* followed in [GHK+08a], a logical system component describes a functional block of a system that applies an operating principle to the system. Consequently, this approach decomposes the logical systems according to the operating principle layer of CUBE. This way of decomposition was partly applied in the Gamma projects, where the systems engineers split the system into part functions, for which independent feature teams were responsible.
3. System Domain Logic: This solution decomposes the logical components according to the domains required to execute a specific action of an operating principle. It is, for example, followed in [JGW+21], where it defines the subsystems of an VTOL. Thus, this approach introduces a new structure on the logical architecture CUBE layer that orients at logical system domains.
4. Technical Realization Logic: According to this philosophy, the logical components decompose according to a logical abstraction of their technical implementation. It is, for instance, followed in [Bro01a] in the component-based design structure matrix, where it not only describes the technical components of the system but also allocates them as clustered chunks to a development team. Hence, this system decomposition introduces an abstraction of the CUBE technical architecture layer on the logical system architecture layer. The abstraction from this solution manifests in categorizing multiple technical solutions as a logical system in this context.

Table 8.1: Different possibilities to define vehicle subsystems of a vehicles LRA using the example of vehicle propulsion.

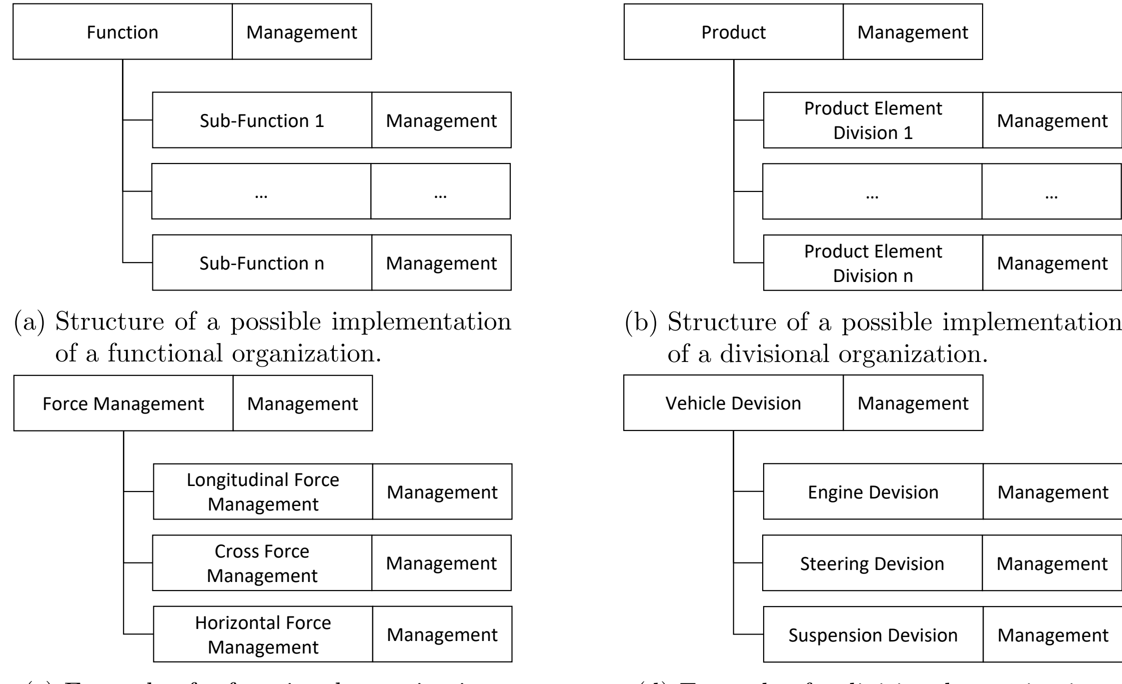
Decomposition Understanding	CUBE Layer	Logical Architecture Component	Description
Customer-Oriented Logic	A	Propulsion Energy Provision System	According to this understanding, the necessity of a vehicle's propulsion follows the stakeholder value for the system to provide propulsion energy. Thus, a system in this logic might have the name "Propulsion Energy Provision System".
Operating Principle Logic	B	Longitudinal Force Management System	On the operating principle level, the propulsion embodies longitudinal forces' management (provision, adjustment, transmission, and reuse). Consequently, the resulting logical system must execute these tasks and may exist under the name "Longitudinal Force Management System".
System Domain Logic	C1	Propulsion System	According to this way of decomposition, the logical name of the system follows from the name of the stakeholder value: Propulsion, and from the engineering domain that provides it. Since the provision of propulsion is an engineering domain, the name is word-for-word usable as a system name, and the system is called "Propulsion System".
Technical Implementation Logic	C2	Engine	From a technical perspective, a vehicle's propulsion requires an engine as a product solution. This implementation manifests in the LRA by this logical system definition. As the "Engine" represents a logical category without further specifying the engine type (*e.g.*, combustion or electrical), multiple degrees of freedom are still available in the technical realization.

To compare these approaches in a more accessible manner, Table 8.1 provides an example of different possibilities to define logical subsystems of a vehicle that implements propulsion. The example shows that for the same technical solution, multiple logical architectures are possible with fundamental differences in the understanding of the system that go beyond the pure linguistic difference of using synonyms for the same system. If the stakeholder value serves as a decomposition anchor for the logical architecture, a vehicle would need a “Propulsion Energy Provision System” since the need for the propulsion of a system requires that the vehicle provides propulsion energy to accelerate the vehicle. Following the functional principle logic, however, the logical subsystems are defined based on the operating principles. Since the propulsion of a vehicle operates by managing the longitudinal forces, the subsystem, according to this philosophy, could result in a “Longitudinal Force Management System” which is solely responsible for the task of longitudinal force management based on the corresponding operating principle. When using the system domain logic philosophy, not a single operating principle but an executing domain that may realize multiple operating principles serves as the logical subsystem. Following the example Table 8.1 presents, this means that the “Propulsion System” may not only realize the propulsion but may also be responsible for the thermal management or the transmission of the energy. Finally, the last example uses an “Engine” based on the technical implementation of the system. Based on the domain knowledge that road vehicles always use engines for the task of providing drive energy, the vehicle would have an engine as a logical subsystem, which in the technical architecture would need to be further specified to get a specific engine, such as an electrical-, diesel-, petrol-, or jet-engine.

As the example shows, all approaches to decompose systems serve to define the propulsion of a system. The generalization of these concepts to a general definition of a sLRA in the context of CUBE reveals several advantages and disadvantages of these approaches. As explained in section 3.1, defining the components in the sLRA is essential for the systems engineering methodology, as it defines the system considered in the next decomposition level. Because avoiding confusion in the systems development project is crucial, it is essential to define one common understanding for the definition of the sLRA and, consequently, a standard structure of the sLRA. For this identification, all approaches have several advantages and drawbacks. Identifying components for the sLRA according to stakeholder values assures that the sLRA addresses all of these needs. Moreover, the logical system architecture is directly derivable from the stakeholder value abstraction layer by defining a system for each stakeholder value. As a drawback, however, many systems are required, and the logical systems must realize all parts of the stakeholder value. Consequently, on the next decomposition level, many subsystems also need to be further decomposed, which results in a high effort in the systems engineering process and a complicated logical architecture with many components and connections. The second approach has the advantage over the first one in CUBE. The features implement groups

of stakeholder values and reduce the number of subsystems in this process. For each of these features, a feature operating principle characterizes the components of the sLRA, and thus, the sLRA is again obtainable from a higher abstraction level: The operating principle level. As a result, it reduces the number of subsystems, and the modeling effort for the different subsystems becomes manageable for feature teams for each operating principle. A drawback is that only a one-to-one mapping of the operating principle to the logical system is possible, as the operating principle directly defines the logical system, and the same logical system must perform all actions of the operating principle. The third understanding addresses this drawback, as logical execution domains are the basis of the definition of the logical system that can implement more than one operating principle. By this, not only can the number of subsystems be reduced, since a logical domain may realize multiple operating principles, but also parts in the form of single actions of an operating principle are addressable. A drawback of this approach is the challenging initial identification of the logical systems since no initial structure is directly obtainable from higher or lower abstraction layers. Consequently, the definition of a logical architecture requires an additional effort that a system development project must spend if no initial logical reference architecture exists. Finally, the fourth way presented in this thesis presents a bottom-up approach to defining logical architectures, which orients on the technical architecture to define a logical intermediate structure. It shares many advantages with the third approach *e.g.,* that a single logical element realizes multiple actions. As a drawback, it also has two significant risks: First, the additional initial effort for identification of the sLRA is still required as the architect needs to abstract away from the product implementation to obtain the logical architecture; second, this approach bears the risk that the manifestation of already existing products is becoming the key driver of the logical architecture development, which might prevent innovation. Since, for instance, the ongoing electrification of the automotive industry is one of the critical drivers of innovation in this area [MNAR22], new challenges arise, which the LRA of the system must cover. While the engine system, as presented as an example in this thesis, is uncritical, a simple extension of the solution space to include electrical engines renders it problematic. In the past, a fuel system for diesel and petrol engines was a standard solution for propulsion systems in registered road vehicles [PTWN18]. In the presence of electrical engines and fuel cells, as other advancing technical solutions [BHL21, PTY+24], it becomes harder to find a one-fits-all solution based on the technical implementation. As a result, either multiple sLRAs for all possible technical implementations must be found and maintained, or the fourth and the third understanding are mixed, resulting in an architecture that contains different system decomposition understandings *e.g.,* an ‘Engine’ connected to an ’Energy System’. Mixing the understandings requires constant alignment of the interfaces, as the engineers responsible for the system development must constantly exchange the understandings of both systems and their interfaces.

Impact of Logical System Decomposition on the Organization



(a) Structure of a possible implementation of a functional organization.

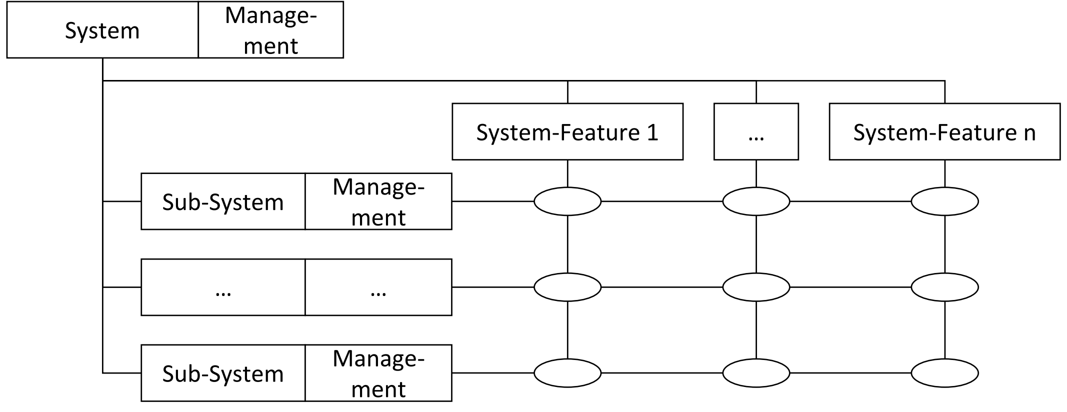
(b) Structure of a possible implementation of a divisional organization.

(c) Example of a functional organization.

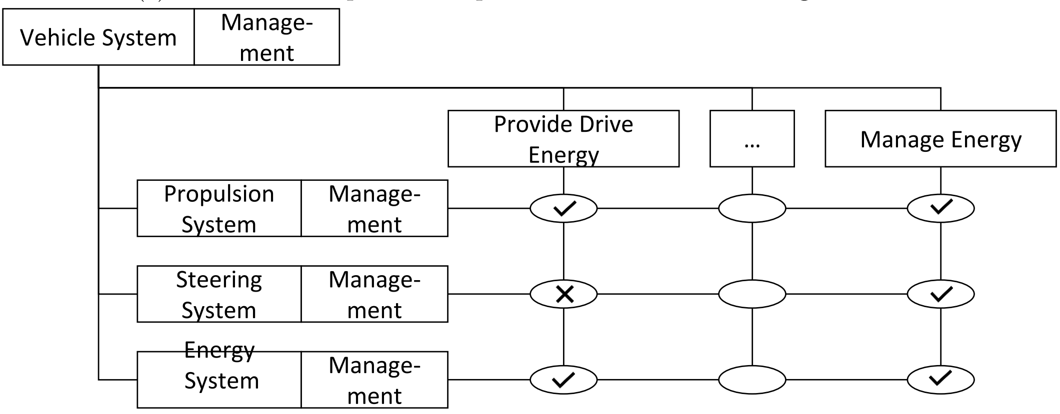
(d) Example of a divisional organization.

Figure 8.1: Impact on organization structure.

According to [Con68], the way of developing new systems also reflects in the organizational structure, which [CB16] even extends, stating that the same applies under specific conditions for the reverse direction. Following these hypotheses, the logical decomposition in the sLRA significantly impacts the enterprise organization and vice versa. Considering that the organization structure might impact the product and development processes, this thesis further analyzes three common approaches without considering the simple organization, which is defined by its informality, and the combinations of these approaches, for example, described in [FGT96]. The first two approaches decompose the logical architecture from a functional perspective and thus eventually reflect in a functional organization that Figure 8.1a sketches as a general pattern. Applied to a concrete example, a structure as displayed in Figure 8.1c structures the high-level vehicle function “Force Management” according to the geometrical dimensions to “Longitudinal Force Management”, “Cross Force Management”, and “Horizontal Force Management”. A similar pattern results when the fourth decomposition approach that structures components according to an abstraction of the technical architecture follows. Instead of the functions, in the organizational structure that Figure 8.1b sketches, the organization



(e) Structure of a possible implementation of a matrix organization.



(f) Example of a matrix organization.

Figure 8.1: Impact on organization structure (cont.).

is strictly structured into divisions based on the product structure. As shown in Figure 8.1d, this might lead to a structure in which a vehicle manufacturer organizes as divisions for the "Engine", "Steering", or "Suspension", which themselves possibly also distribute to different engineering areas such as development, research, or production. Finally, as presented in the third approach, a sLRA supports a matrix organization structure as sketched for a general structure in Figure 8.1e. As Figure 8.1f presents, the matrix organization structure enables the combination of two principles in the same enterprise organization: A functional and a divisional approach.

Logical System Decomposition

Out of these reasons, the "System Domain Logic" is selected as the decomposition philosophy of choice in the context of CUBE, as it not only provides several degrees of freedom when reusing the sLRA components in different contexts but also suits the understanding of the CUBE process the best, in which the logical systems from the sLRA are the considered systems on the following decomposition layers. Since the logical architecture created in this thesis is a reference architecture for multiple system development projects, the additional effort to create its logical architecture according to the different logical domains seems reasonable and beneficial when comparing the chances with the risks of the other approaches.

8.1.2 Modeling sLRA (structural Logical Reference Architecture) as SysML BDDs

Concerning **RQ-3.4** ("How to model logical architectures?"), this section focuses on how to model the sLRA using the SysML [Obj19]. Since the sLRA describes structural components, the structural diagrams of the SysML are suited because neither packages nor the internal structure of a block (which is defined by the sLRA and used to define the internal structure later), the BDD is a well-suited diagram for this modeling task, for this Table 8.2 shows the elements required to describe a static sLRA in a BDD.

Table 8.2: BDD elements required to describe the logical architecture of a system.

Element Name	Notation	Description
Obligatory Diagram Elements		
System	«block» System	In the sLRA model, a system is modeled as a block in a BDD.
Sub-system	«block» System ◆—— «block» Subsystem	The decomposition of system elements is modeled in a special kind of association: The composition. In a composition, the sub-objects strongly depend on the whole, also concerning the life cycle [Rum17].
Optional Diagram Elements		

Table 8.2: BDD elements required to describe the logical architecture of a system.

Association	«block» System 1 — Association Name (s1, s2) — «block» System 2	An association in a sLRA may have an association name and, for each of its two ends, an association role and a description of the possible navigation directions indicated by arrows. If the arrows are now shown, the direction is unspecified.
Generalization	«block» General System ◁— «block» Specialized System	If two systems of the sLRA are in an inheritance relationship, the 'Specialized System' inherits its attributes and methods from its 'General System'. The 'Specialized System' may modify or add these attributes and methods as the modifiers allow. In addition, the 'Specialized System' forms a sub-type of the 'General System', which, according to the substitution principle, allows instances of the 'Specialized System' to be used where instances of the 'General System' are required [Rum16].

Because the static decomposition only consists of systems and their subsystems, only two elements from the BDD are required: First, to describe the systems SysML blocks are suited. Second, to define the logical decomposition of the system, the "composition" relationship between system components enables the modeler of the sLRA to define the subsystems of a system. In addition to these two elementary building blocks of a sLRA model, several other elements are usable to constrain further or extend the model. One of these elements is the association relationship, which relates several blocks in the sLRA. Different elements and systems within the sLRA Model can be related using an association relationship. Since this can lead to particular constraints and complicated cross-connections that are not necessarily desirable in all variants derived from a reference architecture, this element should be used with special care when defining a sLRA. For example, in the context of a propulsion system and a power system of a vehicle, such a relationship could express the particular relationship between the two systems (the propulsion system generates the propulsion energy, and the power system provides the energy in a different form that is converted to propulsion energy) already at a high

level in the sLRA. Since the implementation of this relationship is ultimately manifested in interfaces, which may be defined differently depending on the operating principle applied, such a linkage also carries the risk that too concrete or contradictory a boundary condition is applied. Therefore, in such a case, it would not be advisable to additionally manifest such a constraint in the sLRA in the form of an association relation. Another relationship that is helpful when creating for a sLRA model, but is not necessary, is the inheritance relationship. The inheritance relation makes introducing logical components as super components possible, to which several sub-types exist. In particular, when defining a logical architecture based on the product structure, the generalization relation is helpful for conflict resolution. For example, an inheritance and association relationship can resolve the conflict between different energy systems for different engines (combustion engines and electric motors). For this, an energy system could serve as a general system and specialize into a fuel and battery system expressed by the inheritance relation. Combined with the association relation, the constraint is that an internal combustion engine must be related to the fuel system, and an electric motor can be formulated to a battery system. In addition to this type of use, however, the inheritance relation is also valuable for introducing various logical principles in the third decomposition understanding to divide an ample solution space into several small solution spaces without necessarily having to create subsystems via a composition relation. For example, a logical input system within the infotainment system may be specialized via an inheritance relation by a speech input, screen input, gesture input, button input, or machine input.

One of the main principles of the kind of sLRA presented in this thesis is that all elements of the sLRA shall be unique to allow an unambiguous one-to-one mapping of elements to their executing entity. Thus, a logical component is more of a type or class for multiple technical solutions. Consequently, redundancy is not allowed or possible in the structure of the system. This principle assures that all components provided in the sLRA are uniquely defined by their name. Thus, no cardinalities on the relations should be used since a one-to-one relationship is always used in this kind of diagram. One might argue that this description philosophy cannot describe redundancy for safety critical systems in [YZW+21]; however, this is elaborated further for aircraft design, and a second LRA is introduced with redundancy for those components where multiple redundant components are required. To distinguish between different redundant components, they introduce new components with different names and do not increase the number of cardinalities, which enables a better differentiation based on the name on a signal level in the IBD derived from the sLRA description in the form of a BDD as described in section 8.2. To implement this principle in the sLRA model, the cardinalities of the composition and the association relationships are not allowed, and an implicit one is assumed at all ends of the relationships.

Figure 8.2 presents an example BDD with a reduced selection of subsystems, describing

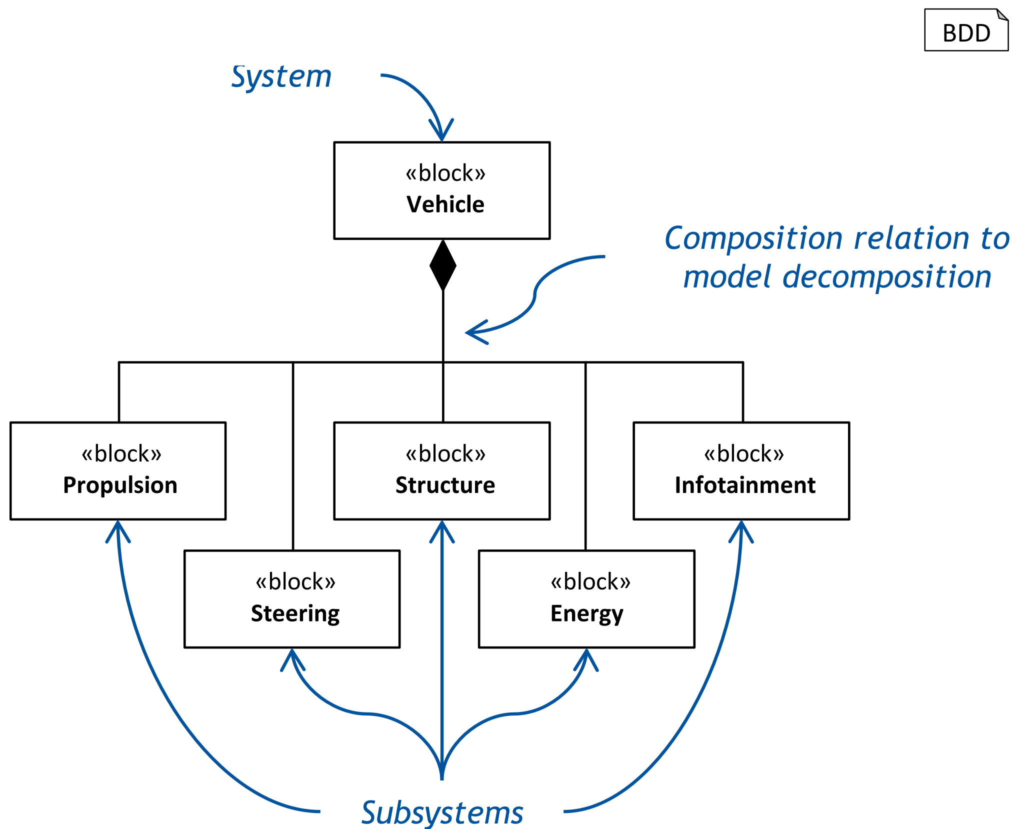


Figure 8.2: An example BDD modeling the sLRA of a vehicle using a reduced selection of possible subsystems.

the decomposition of a vehicle to its subsystems as part of the sLRA. In the example, the system 'Vehicle' has the subsystems 'Propulsion', 'Chassis', 'Structure', 'Energy', and 'Infotainment' modeled by the composition relationship. It is a standard design guideline in the SysML to combine multiple connectors of the same type into one connector for better readability.

8.1.3 Behavioral Logical Reference Architecture

In contrast to the sLRA, the behavioral Logical Reference Architecture (bLRA) describes the dynamic behavior of a system. In the context of this thesis, the following definition of a bLRA holds.

Definition 29 (behavioral Logical Reference Architecture (bLRA))**.** *The behavioral or behavioral Logical Reference Architecture (bLRA) of a system of interest describes the*

behavior of a system component defined in the sLRA. It is derived during or from previous system development projects, reusable for similar projects, and should facilitate the system decomposition.

Following this definition, the bLRA describes the joint system states and transitions for the localization of actions within an operating principle. In this regard, the bLRA is a reference model consisting of a static and a dynamic part in the context of variability. The static part of the state machine *i.e.,* the state and transition space which is independent of the realized system features is defined by the bLRA, the dynamic *i.e.,* the part that changes depending on the realized systems, is derived from the operating principles.

Since the bLRA describes a system behavior in terms of states and transitions, the state machine [BCR07] is a well-suited paradigm to model this aspect of the architecture. As mentioned in the introduction of this chapter, the implicit knowledge and design decisions from the top-level requirements should become explicit in a reference architecture [WLRW15]. In the context of bLRA, this means that the dynamic behavior of a system of a system is describable in terms of a state machine.

Regarding a vehicle's propulsion, which was also the running example in subsection 8.1.1, the implicit knowledge about the system states manifests in the different conditions of the propulsion in which its observed attributes fulfill a specific constraint (*cf.* Def. 1). For a logical system definition, it is a good practice to identify these constraints by considering the input- and output channels as a black box and defining the system states to gain a gray-/white-box view of the logical behavior without inferring too much solution-specific knowledge. Applied to a propulsion system, usually, the following input and output behaviors are observable (in the same sequence for the usual usage cycle of a propulsion system) at the ports when respecting the propulsion output and the energy and propulsion request input channels:

1. The propulsion system does not provide propulsion and consumes no or negligible energy for this task. The system does not react to propulsion requests.
2. The propulsion system consumes energy but cannot respond to propulsion requests by providing propulsion.
3. The propulsion system reacts to propulsion requests by providing propulsion and consumes the required energy on the propulsion energy channel.

Together with the knowledge of propulsion systems, these observable states are usable for a system state definition. In the first state, the system is switched off, leading to the first state of the bLRA: The off state. The second stage is the running state, in which the propulsion system provides propulsion as a reaction to the propulsion requests. When taking the domain knowledge of propulsion systems into account, two additional states are identified. First, to get from the off state to the starting state, there is an additional

state in which the propulsion system consumes energy but does not provide propulsion as a reaction to propulsion inputs. Based on the domain knowledge that this energy is used to startup and prepare the propulsion system for operation, another system state follows: The starting state. Following the same argumentation, a stopping state is also identifiable, only with the knowledge of propulsion systems that a different process is required to start propulsion from the off state rather than to shut down the propulsion from the running state. Based on these observations, most states and transitions are defined. Only the triggers are missing. The system requires a startup trigger from an external system to start the propulsion. The only constraint is that the propulsion system is ready to be started up (*i.e.,* the propulsion system does not have any errors preventing it from starting up in the past). Taking this transition, additional activity might be required to initiate the startup process. No external system is required to come from the starting to the running state, so the transition appears spontaneous from the outside. When looking at the condition for this state transition, it is observable that the system must be successfully started up in order to take this transition. From running to stopping, a shutdown trigger is required, which analogously to the startup trigger shuts the system down and stops its operation. Again, the transition between stopping and off state appears spontaneous from the outside since no other external trigger and only the condition that the system successfully stopped is required for this state change.

Two other bLRAs are possible from a modeling perspective. First, the starting and stopping states could be omitted, and the starting and stopping processes could be modeled either as activities on the state transitions or internally as part of the entry and exit conditions when entering or leaving the state. Second, the startup and stopping states could be merged to the same state since the externally observable behavior is the same when disregarding the domain knowledge that starting and stopping propulsion requires different transition processes. For the running example used in this chapter, the understanding described above was preferred over the other two options, as the resulting bLRA not only results in a didactically more appealing model but also represents the most sophisticated view of the system as a higher number of states and transitions is modeled.

When implementing the logical architecture of the propulsion system, not all of these states and triggers must necessarily be implemented in a one-to-one mapping of the logical representation of the product implementation. A modern diesel engine, for example, has an almost one-to-one mapping of the states from this bLRA to their operating states. In the off state, the combustion engine is switched off. To turn on the engine, the driver presses a start button to start the engine automatically. Depending on the outside temperature, the engine automatically enables glow-plugs to preheat the engine as a cold-start aid. Afterward, in the actual starting state, the engine uses an electric starter to start up the engine. It automatically switches to the running state when the engine runs without needing an additional starter engine. When running, the engine

reacts to the driver pressing the throttle by providing torque and consuming diesel from the vehicle's fuel system (serving as an energy system). As soon as the driver presses the on/off button again, the propulsion stops the combustion and stops the engine. Modern engines sometimes start a cooling fan at a standstill depending on the whether to increase the engine lifespan before reaching the off state. It is not that the startup and the shutdown triggers are the same triggers from a driver's perspective. However, they might be mapped to different signals sent to the engine depending on the internal interpretation of the button press event. Another valid technical implementation is, for example, the jet engine, having a much more complicated startup and shutdown process [CM93], or electrical engines with almost no startup and shutdown operations required.

8.1.4 Modeling the dynamic behavior in the LRA as State Machine Diagrams

Respecting **RQ-3.4** ("How to model logical architectures?"), this section defines how to model the dynamic part of the LRA that is concerned with executing the operating principles in the context of the logical architecture components. To this end, the following elements are required as described in Table 8.3.

Table 8.3: STM elements required to describe the behavioral part of the logical architecture of a system.

Element Name	Notation	Description
Obligatory Diagram Elements		
State Machine	State Machine	A state machine is a modeling element for state machine models. It represents behavior in terms of a system's transitions and states.
State	State	According to Def. 6, a state describes a system's condition in which its observed attributes fulfill a specific constraint for a meaningful time.

Table 8.3: STM elements required to describe the behavioral part of the logical architecture of a system.

State Activities	State do / doActivity entry / entryActivity exit / exitActivity	States may have entry activities that the state machine executes when the state is entered, do activities the state machine executes while in the state, and exit activities the system executes when the state machine leaves the state.
Initial Pseudostate	Initial	An initial state declares the first state to be visited in a state machine (or region).
Final Pseudostate	Final	The final state marks the last state a state machine (or region) must reach for execution completion. When this state is reached, the state machine is stopped, and all tokens are destroyed.
Transition	State1 Trigger [Guard] /Effect State2	Models a lifecycle connection between two states based on their operational conditions. A transition may include a trigger, a guard (constraint), and an effect (activity).
	Optional Diagram Elements	
Entry/ Exit	Entry/Exit EntryPoint ExitPoint	Each state may have entry and exit points.
Region	Region [Region1] [Region2]	A state machine may have regions containing states, pseudostates, and transitions. Regions are a means of defining nested states and transitions.

Table 8.3: STM elements required to describe the behavioral part of the logical architecture of a system.

History State	History Deep History	History is pseudostates that serve to model points where reentering the last state of a state machine owning region that was active prior to its interruption is possible. The SysML defines two types of history states depending on the inclusion of the sub-states: Shallow History (H) and Deep History (H*)
Terminate Node	Terminate	When a state machine reaches a terminate node, the current execution path is terminated, but the state machine (or region) continues operating.
Choice Node	Choice	A choice node is a pseudostate that decides which outgoing transition is taken based on an evaluation of the constraints attached to the transitions. The choice is made after the state machine reaches the choice node.
Junction State	Junction	A junction declares a decision point at which a Transition may branch out into multiple guarded, alternative paths between states. In contrast to the choice node, the decision on which path to take is made before the junction is visited.

Table 8.3 differentiates between two types of elements. The obligatory elements and the optional elements. The obligatory elements are not required *i.e.,* obligatory to be used in each diagram describing a bLRA, but need to be part of the repertoire of a systems engineer defining a bLRA. To model a state machine, the state machine element is the first and most important element to define the context and environment of the state machine. Moreover, states and transitions are required to model a system's states and their transitions between the elements. Transitions have a trigger, guard, and effect as specified in the SysML [Obj19]. A trigger defines the event, signal, or input that triggers

the state change specified by a transition. The guard constraints the applicability of the transition. Only if the constraint from the guard is fulfilled can a state change be performed. Finally, the effect specifies an activity that is performed during the state change. Note that all elements in the transition are optional and can occur in all combinations. Moreover, UML [Obj17] and SysML [Obj19] allow multiple triggers to be drawn at the same transition as a shorthand notation for multiple transitions with the same guard and effect but different trigger. In addition, the initial and final pseudostates define a state machine's initial and final state. Specifying the actions that are executable within a state and entry, do, and exit activities are also part of the obligatory elements of the methodology presented in this thesis. Note that entry and exit states are already syntactic sugar since they can be modeled using effects at transitions [KS10]. A vast spectrum of possible bLRAs are expressible using these elements. In addition to these elementary elements, also Table 8.3 provides some optional extensions from the SysML for the specification of a bLRA as syntactic sugar.

Semantically, the state machines as specified in the SysML are based on Mealy [Mea55] and Moore [Moo56] automata. The semantic interpretation of the automaton remains unchanged from the definitions provided there for simplicity of the description. For more insights on the semantics and implications for UML state machine [BCR07], the semantic definition and semantic differencing [DEKR19], or matching and merging [NSC+07], the referenced sources should build the context for the semantics definition used in this thesis.

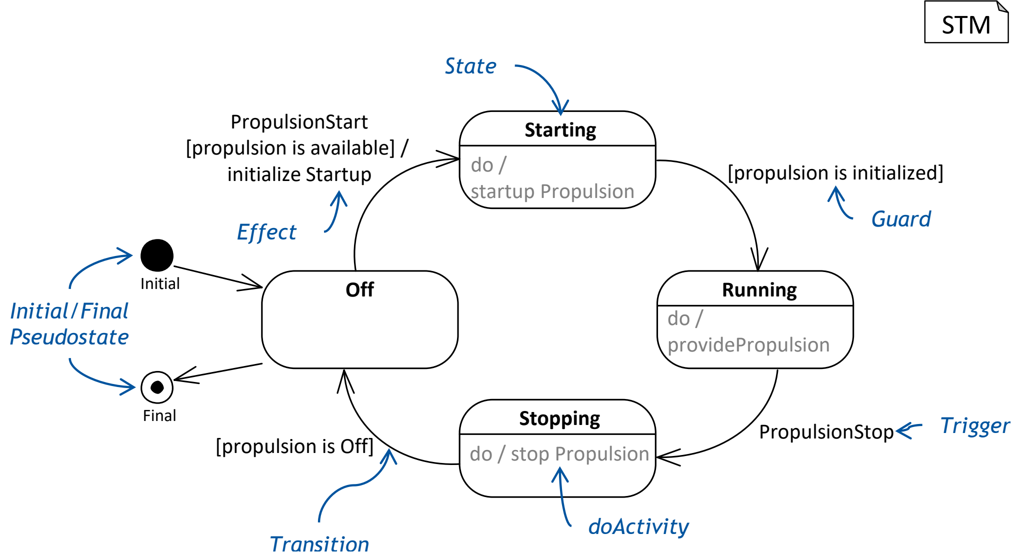


Figure 8.3: An example STM modeling the bLRA of a propulsion system.

Figure 8.3 presents an example STM modeling the bLRA of a propulsion system. The

STM describes the internal behavior of the 'Propulsion' System specified in Figure 8.2. Initially (and finally), the propulsion system must be switched off as indicated by the initial and final pseudostates. When a 'PropulsionStart' trigger is received, the propulsion system is ready (indicated by the 'PropulsionReady' guard), and the propulsion system shall 'initStartup' to initialize the startup of the system and change to the 'Starting' state. This state describes the state in which the propulsion starts up. The concrete action performed during this stat is modeled in the 'doStarting' activity. When the startup is finished, indicated by the fulfillment of the '[PropulsionSystemRunning]', the system may switch to the 'On' state, which describes the behavior of the propulsion system when the propulsion is running. As soon as the system receives a 'PropulsionStop' trigger, which indicates that the propulsion system is required to stop its functionality, the system shall go to the 'Stopping' state. This state models the stopping procedure of the propulsion system in the 'doStopping' activity. Finally, when the propulsion is stopped, as indicated by fulfilling the '[PropulsionOff]' guard, the propulsion system shall re-switch to the off state.

8.2 Generating Logical Architectures from Operating Principles

The transformation between operating principles to logical architectures was already published in [GKM+25].

8.2.1 Modeling Process for Semi-Automated Logical Architecture Generation Based on Operating Principles

Feature dependencies are one of the main challenges and a major source of erroneous and deficient behavior in automotive systems [VF13]. Thus, one of the core tasks of the development of a feature realization in the logical architecture is the identification of the work-split and further subsystems of the system development process [GKM+25]. Even when assuming that, the subsystems are already defined in the LRA, an executing element for each action in the architecture is to be identified for a complete system specification. For this purpose, there must exist an unambiguous mapping of all actions from all operating principles to their executing element from the LRA [GKM+25]. As a result, only one component from the LRA is allowed to execute an action, to assure that an unambiguous mapping of all inputs and outputs required to execute this action is possible.

Considering the propulsion examples from the previous sections, this means that for each step of each action required to get propulsion, a systems engineer needs to identify a system capable of performing this action to specify an operating system. So, for example, the action to 'provide propulsion energy' might be allocated to the 'Propulsion System',

the action to 'provide energy' to the 'Energy System' from the LRA. If multiple mapping of actions to systems was allowed, a systems engineer might also allocate propulsion energy to the same system, the 'Propulsion System'. The problem is that it is no longer possible to differentiate the interfaces between the systems. Since the 'provide propulsion' action requires energy as input and returns another form of energy, the propulsion energy, it is still being determined which of the two systems provides or requires which interface. By allocating the actions to a single system only, this conflict is resolved, and it becomes clear that the 'Propulsion System' provides the propulsion energy as an interface and requires the energy for this task from the 'Energy System' that must provide this energy. By following this approach, two steps are assured: First, by allocating all actions to their executing element in the LRA, the actions become traceable to the systems responsible for the execution, and the automation from the next section is enabled. Second, the operating principle is rechallenged, as an action that cannot be uniquely mapped to an executing element must be further decomposed until a unique mapping is possible. Concerning **MR-5**, the semi-automation proposed in this section is not a redundant task but some additional effort that could serve as a starting point for further automation.

With respect to the efficiency of the system development process, however, there is still room for improvement. One of the most significant aims of the development would be an automated assignment suggestion and a logical architecture definition. Because of the ambiguity of natural language and the different philosophies on decomposing the logical architecture as described in subsection 8.1.1

In principle, this generation method can also be used to generate the technical architecture since the technical architecture model consists of the same model elements: parts, ports, and connectors. The only requirement for generating a correct technical architecture with the derivation proposed in the following sections is that technical interfaces from the technical architecture must be part of the operating principle. Consequently, an operating principle used to generate a technical architecture is no longer solution-neutral since, for example, concrete communication channels or energy forms must already be part of the operating principle. As a result, when considering the technical architecture of a drive system, all final interfaces must be known in advance, which is unrealistic in an industrial development project and a stumbling block for innovation and reuse since all these interfaces must be the same in all development projects, variants, and products.

8.2.2 Modeling Logical Architectures

Respecting the modeling aspect from **RQ-3.4** ("How to model logical architectures?") this section provides the elements required for the definition of a logical architecture. For this, subsubsection 8.2.2 introduces guidelines and modeling elements for modeling the structural elements of the logical architecture. In addition, subsection 8.1.3 explains

the models and constraints for a behavioral model.

Modeling the Architecture Structure as IBD (SysML Internal Block Diagram)

The logical architecture models an inside or white-boy view of a system, defining its subsystems and elements.

Table 8.4: IBD elements required to describe the logical architecture of a system.

Element Name	Notation	Description
	Obligatory Diagram Elements	
Part Property	name: Type	A part property is a property of a block that has another block as type, which is part of the block indicated by a composition relationship. By using parts in an IBD, a white box view is created. Since the logical architecture, in contrast to the technical architecture, abstracts away from redundancy, all elements of the LRA shall only be instantiated as part property once.
Port	portName: Type; «proxy» proxyPort; «full» fullPort; name: Type; p: InIF; p: OutIF; p: InOutIF	According to [Obj19], ports define points to enable external entities to connect to and interact with a block in different or more limited ways than connecting directly to the block itself. The SysML consists of multiple ports. Ports are properties with a type specifying features available to the external entities via connectors to the ports [Obj19].

Table 8.4: IBD elements required to describe the logical architecture of a system.

Connector	p1: T p p p2: T	A connector establishes a communication link between two parts (since ports are also Parts, they are the most important use case for connectors in the Logical Architecture) and enables an interaction between those parts.
Item Flow	p1: T p ConveyedItem «flow» p p2: T	Item flows are a means of specifying things that float between blocks and/or parts across associations or connections [Obj19]. For better traceability, item flows can be connected to object nodes from activity diagrams specifying the operating principle of the flow.

Table 8.4 describes the elements required to model a logical architecture of a system in the form of an IBD. Most importantly, the part property describes the elements in a logical architecture. As mentioned in Table 8.4, all properties that are part of the logical architecture of a system must also be connected in the LRA of the system. By this, it is assumed that the logical system architecture diagrams are always derived from the LRA and describe a white box view of the considered system. As important as the part properties, the ports also play an essential role in the definition of a logical system architecture since they describe the interfaces of a system. There are different types of ports defined in [Obj19]. The first type summarizes a usage pattern where ports act as proxies for their owning blocks or its internal parts (proxy ports) [Obj19], and the second type identifying ports specify separate elements of the system (full ports) [Obj19]. As [Obj19] defines, a proxy ports defines a boundary by specifying which features of the owning block or internal parts are visible through external connectors. In contrast, a full port according to [Obj19] defines a boundary with their features. From a usage point of view, proxy ports are always typed by interface blocks which are specialized kinds of blocks that have no behaviors or internal parts [Obj19]. In contrast, full ports cannot be behavioral in the UML sense of standing in for the owning object because they handle features themselves rather than exposing their owners' features or internal parts of their owners. Ports not specified as a proxy or full are called "ports" [Obj19]. Despite of the differences, the SysML aims at enabling modelers to chose between proxy or full ports at any time in the development lifecycle, or not at all, depending on the applied methodology [Obj19]. Finally, the connector and flow properties are required to

model the connections between the interfaces between the systems.

8.2.3 An Example Logical Architecture of a Transportation Task

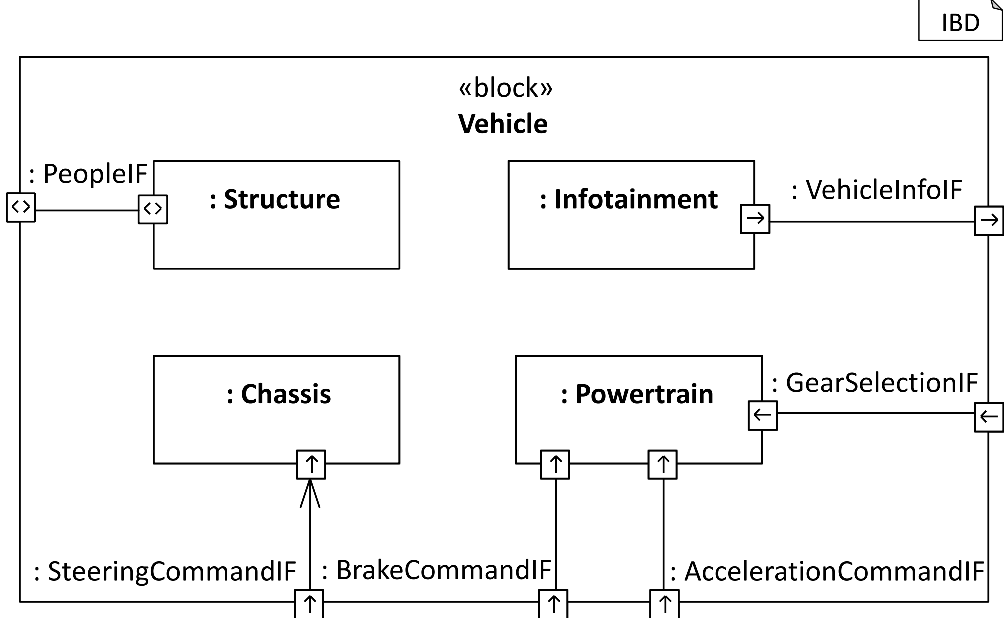


Figure 8.4: An example IBD modeling the logical architecture of a simplified vehicle system.

Figure 8.4 presents an example IBD describing the logical architecture of a simplified vehicle system. The vehicle consists of the same elements defined in the sLRA defined in Figure 8.2. However, not only are their blocks used in this view, but concrete part properties represent concrete instances of these blocks. In the architecture presented in this figure, all systems are mounted to the structure as indicated by the different mount ports. Since all connections shall be based on material connections, they convey a material as datatype, even though no material will constantly flow between those ports in the final system implementation. Moreover, the energy system provides drive energy to the propulsion and operation energy to all other systems required to provide their kinematic or electromagnetic behavior.

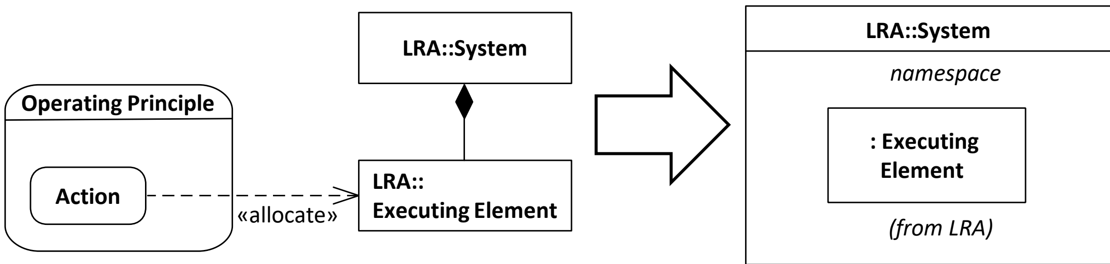
It is easy to imagine that the diagram is no longer readable for a complete vehicle specification since thousands of ports are connected between the different system elements. To prevent this and keep the diagrams readable, it is a good practice to create multiple views on the vehicle system based on an external criterium. Following the different CUBE layers, it is, for example, possible to create a diagram for each feature or

stakeholder value, operating principle, or logical architecture element. In practice, these views are usable for a specific stakeholder, but only a few are easy to understand. As a result, the methodology used to generate these diagrams should be capable of creating multiple views depending on the reader's needs.

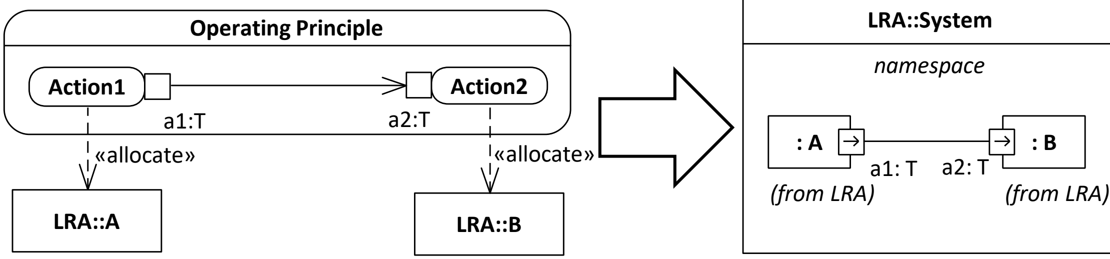
8.2.4 Transformation Rules from Operating Principles to Logical Architectures

The transformation described in this section takes one or more operating principles as input and produces a logical architecture based on an element allocation from the actions in the operating principle to the executing element from the sLRA as described in subsection 8.2.1. Therefore, the operating principles must comply with the modeling constraints and guidelines formulated in section 7.1. In addition to these syntactic modeling rules, all actions in the activity diagrams modeling the operating principles must ensure that all elements are mapped to an executing element from the LRA [GKM+25]. Furthermore, it is required that the mapping of action to the executing element must be unique (*i.e.*,, it must be ensured that an action is only executed by one element from the logical architecture).

When these preconditions are fulfilled, the transformation is based on six compara-

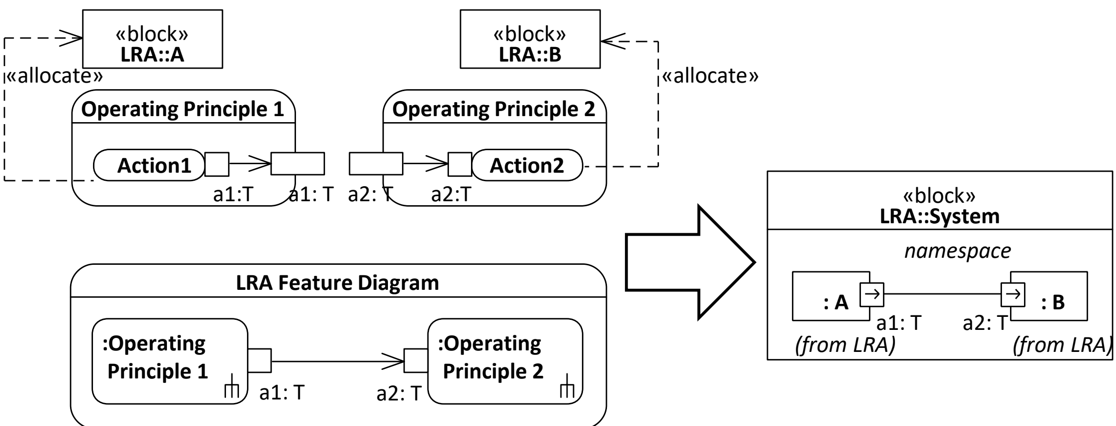


(a) Transformation rule for action execution in the logical architecture based on [GKM+25].

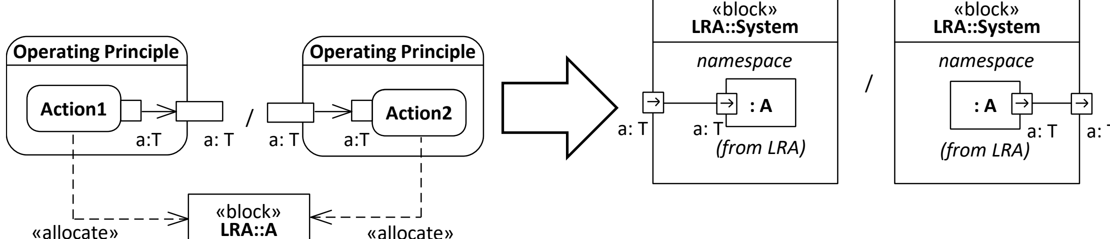


(b) Action interface transformation rule creating interfaces for actions connected with an object flow based on [GKM+25].

Figure 8.5: Transformation rules from operating principles to logical architectures.



(c) Feature interface transformation rule to connect features using parameters provided in other features in the logical architecture based on [GKM+25].

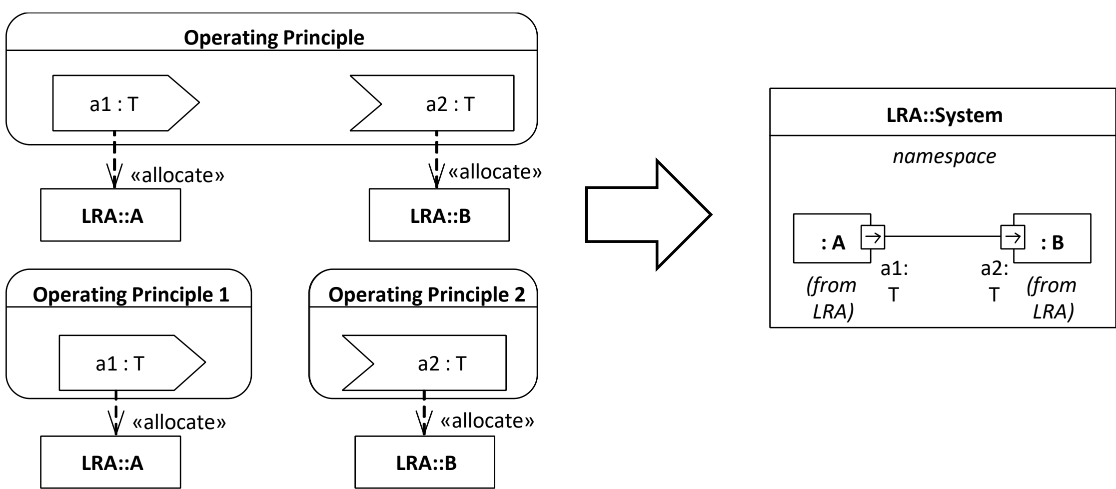


(d) Input/Output rule to create system interfaces for objects that are not provided by other features based on [GKM+25].

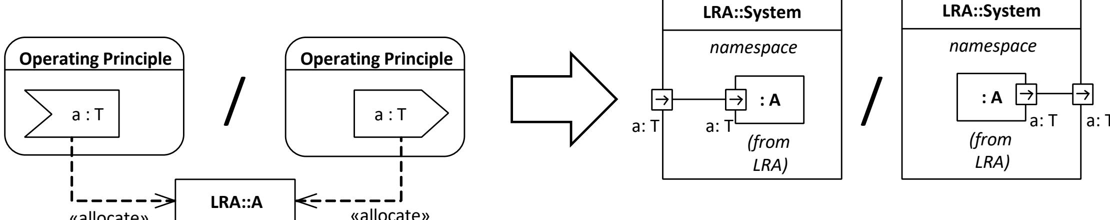
Figure 8.5: Transformation rules from operating principles to logical architectures (cont.).

bly simple formal transformation rules, as described in the section. To perform the transformation, the rules are applicable to the operating principles returning the logical architecture. Since the transformation rules only require the existence of the elements of each rule for a mapping to the logical architecture, all other elements from the AD are usable in the diagrams for the transformation but not regarded in the transformation. For the transformation, the following rules exist that draw on and extend the concepts [GKM+25] presented:

1. Execution Rule: Figure 8.5a presents the first rule, which describes that the LRA component, an action is allocated to, executes this system. Following this rule, a system that needs to perform this action to implement the operating principle requires the LRA component for the execution. Consequently, the logical architecture of the system requires this executing element to perform this action.



(e) Feature interface transformation rule to connect features using signals provided in other features in the logical architecture



(f) Input/Output rule to create system interfaces for signals that are not provided by other features based on [GKM+25]

Figure 8.5: Transformation rules from operating principles to logical architectures (cont.)

2. Action Interface Transformation Rule: This rule as presented in Figure 8.5b describes how to regard the object flows to or from an action in the LRA. Therefore, the rule states that if two actions are allocated to different elements 'A' and 'B' from the LRA and have an object flow between two action pins 'a1' and 'a2' with the same or a compatible type 'T', the information flow is part of the resulting logical architecture [GKM+25]. In the logical architecture, this is expressed by ports with the same name, type, and direction as Figure 8.5b presents. Because the SysML specification [Obj19] allows multiple ways to specify activities and parameters as well as several ways for port specification, this rule has the precondition that the ports are typed with an interface block 'T' having 'a1' respectively 'a2' as flow property and the same direction (in-going/outgoing) as indicated by the object flow [GKM+25].
3. Feature Interface Transformation Rule: Figure 8.5c presents a rule to resolve inter-

feature dependencies during the logical architecture generation. These interdependencies are caused since the feature operating principles are not stand-alone but connected in the dynamic part of the LRA as described in subsection 8.1.3. The rule resolves these dependencies by connecting the systems allocated to actions in the context of different activities in the logical architecture. If an 'Action1' has an action pin connected to an outgoing parameter of its 'Operating Principle 1' activity and there is a second activity, 'Operating Principle 2', which has an action pin connected to an ingoing parameter and both operating principle activities are connected in invoked actions allocated to different elements 'A' and 'B' from the LRA, then 'A' and 'B' are connected in the generated logical architecture [GKM+25].

4. Input/Output Rule: Since every object, modeled in an object flow, must have a source, the rule that Figure 8.5d presents creates interfaces to the systems environment to get this information from external sources.
5. Feature Interface Transformation Rule for Signals: Figure 8.5e presents a rule to introduce channels into the logical architecture to realize the communication via send signals and receive signals. With the introduction of this rule, this thesis extends the concepts [GKM+25] presents to make the rules applicable in the context of the concepts this thesis additionally introduces in Table 7.1. The application is comparable to the rule in Figure 8.5c.
6. Input/Output Rule for Signals: As for the object flows. Also, each sent or received signal needs a provider and a recipient. Thus, Figure 8.5f introduces a transformation rule to introduce additional channels to or from the environment for all signals not used or provided within the system. As the interface transformation rule for signals, this rule extends the concepts from [GKM+25] to make them applicable to the concepts this thesis additionally uses in Table 7.1. Apart from this distinction, the application is analogous to the rule Figure 8.5d presents for object flows.

Since these rules rely on several restrictions and assumptions, several implications arise from the transformation rules, as also discussed in [GKM+25]:

- The first rule implies that allocation actions to executing elements are only possible (and allowed) if the element is a subsystem of the currently regarded system. While this implication matches the understanding of CUBE as described in section 3.1, this implication might be too strong for the general application of the transformation rules in other contexts. In these cases, it might be possible to reformulate the rule such that a new 'Supersystem' is introduced for executing elements that do not have an (*e.g.,* as they are the root element of the LRA).
- From the second rule follows that object flows between actions allocated to the same LRA element is not regarded in the logical architecture (as no interface between the sub-elements will be generated) and, therefore, not visible in the

logical architecture. Knowing which objects are available in the subsystems is a good indicator for allocating new actions, so generating a list of available objects for each component is also a good practice. With this list of 'capabilities,' the systems engineer that allocates new actions to systems is aided in the decision-making process. An advanced implementation could also use this for automated element allocation for an automated error correction or computer-aided system design.

- The rules only transform the existing operating principles to a logical architecture based on the object flows modeled in the operating principles activity diagram and the allocation of action to elements in the LRA, without any assurance of additional constraints to the validity of the input models. Especially in the context of the third transformation rule, this might result in errors since all elements created there are based on the inputs from other operating principles often developed by independent feature teams. Common mistakes are *e.g.,* that a port receives multiple inputs from different systems. Therefore, the resulting logical architecture might consist of illegal port connections that must either be later found and resolved in a validation phase of the logical architecture or corrected upfront by validating the operating principles. Since the logical architecture needs to be regenerated in the first approach, the latter approach is preferred over the first one.
- Finally, the implications from the fourth rule may lead to resulting architectures that need to be reviewed again. Since the fourth rule interprets all unconnected activity parameters as system interfaces, forgotten or falsely modeled objects from an object flow might also become system interfaces. While the general approach of making everything that cannot be provided within a system a system interface allows high reuse of the features concerning MR1 and is technically correct (since not provided objects must inherently be provided from the system environment) [GKM+25], it must be assured that this information is and can be provided to the system by the system environment. Consequently, this must also be assured upfront or after the logical architecture generation in the validation phase of the generated logical architecture.

Since an automated application of these rules is possible, the requirements from MR3-5 follow from the design of the transformation rules [GKM+25].

8.3 Applying Logical Architecture Models to Efficiently Design Automotive Systems

As an applicability evaluation of the methods in this chapter, this section presents several results from the application of these methods in the disclosed Epsilon, Delta, and

Zeta automotive projects and relates them with the publicly available VTOL models from [JGW+21]. To this end, this section provides methods and insights for the generation of natural language requirements similar to the application results section 6.4 presented for stakeholder values and subsection 8.3.1 provided for operating principles in these projects. Moreover, this section provides additional application insights on the architecture generation method section 8.2 presents in the context of its application in the Zeta project.

To achieve these aims, subsection 8.3.1 provides a generation method for the generation of static system and interface requirements in natural language, as well as dynamic state machine models that relate the feature operating principles with the system states and its transition in the bLRA according to the method section 6.4 presents. In the process of evaluating the correctness of the generated requirements, this section again applies the evaluation scheme subsection 6.4.2 presents to evaluate the applicability of the natural language requirement generation method in the Delta, Zeta, and Epsilon projects as well as on the VTOL model from [GKS+21] in subsection 8.3.2. Since the document-based documentation from the system the Epsilon project did not provide legacy requirements for the logical architecture as a reference, but provides a technical architecture specification subsection 8.3.3 investigates whether the interface and system requirements are also applicable to derive correct and complete technical architecture requirements using the document-based specification the Epsilon project as an input and the VTOL model [JGW+21] as reference implementation. Finally, subsection 8.3.4 presents the insights and lessons learned from applying the logical architecture generation method subsection 8.2.4 introduced in the Zeta project.

8.3.1 Generating Textual System and Interface Requirements using Logical Architecture Diagrams

Using the transformation rules [Zab23] presents, a generation of textual system and interface requirements and behavior requirements in a system state machine is possible. As for the generation of stakeholder value and operating principle requirements, the transformation templates rely on the SPECTRE template (*cf.* Figure 5.1) and insert the respective parts of the IBD and the STM in the according places in the template. As an extension to this process, section D.3 and section D.4 provide additional transformation rules for which this section provides an evaluation.

Generating Textual System and Interface Requirements using IBDs (SysML Internal Block Diagrams)

Since the logical system architecture only uses and provides objects according to the features the operating principles describe, most of the requirements and generation

templates are analogous to the templates subsection 7.3.1 provides. Consequently, the behavioral requirements on the logical architecture and the operating principle layer are identical if the system is decomposed according to the feature split (*cf.* subsubsection 8.1.1).

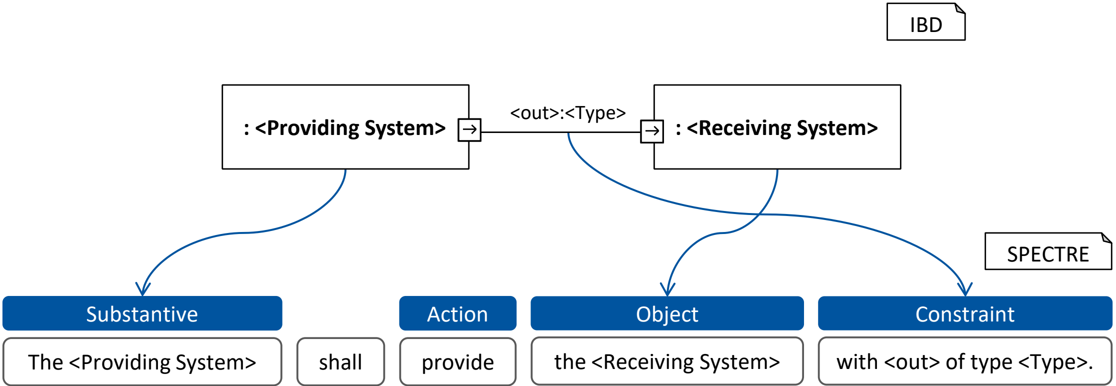


Figure 8.6: A straightforward transformation that inserts the names of the systems and ports of the internal block diagram elements in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1).

From a structural point of view, the requirements are derivable following a similar intuition as for operating principles as subsection 7.3.1. As Figure 8.6 depicts, the basic intuition for requirements describing the logical architecture follows the object flows in the IBDs. Thus, a providing system provides data, material, or energy according to the functional modeling paradigm that section 2.3 presents at a named port to a receiving system at its port with name and type. Consequently, the respective requirement is derivable as in Figure 8.6. Following the same idea, a similar requirement is derivable for the receiving system that receives data, material, or energy at a named port from a providing system.

As subsection 7.3.1 already discussed, the first template of Table 7.7 provides the system execution requirement this section delegated to the logical architecture templates. As section 8.2 describes, these interfaces rely on the definition of the logical architecture and are therefore derivable using the transformation rules Figure 8.5 defines. Also, the system requirements might be split according to the process structure model Figure 3.3 describes, Table D.3 contains a requirement for each port of the architecture—one for sending the object, and one for receiving the object. The application of the rules Table D.3 provides results, for example, in the following requirement to address the 'Inform the driver about environment and driving state' action in the 'Infotainment' system.

Example 16 (Logical Architecture SPECTRE Requirement for the 'Inform the driver about environment and driving state' action in Figure 7.1b)**.** *The 'Infotainment System'*

shall Inform the driver about environment and driving state providing 'vehcileInfo' of type 'Data'.

As an interface requirement, this would then, for example, lead to the requirement:

Example 17. *The Vehicle shall provide the Driver with the 'vehicleInfo' of type 'Data'.*

For the driver, the respective requirements would be generated.

Example 18. *The Driver shall receive the 'vehicleInfo' of type 'Data' from the Vehicle.*

Generating Textual System and Interface Requirements using STMs (SysML State Machine Diagrams)

Because the STM in the logical architecture model is a crucial model for the behavior specification of the logical architecture, the thesis in [Zab23] implemented six templates to address the Table 8.3 provides for the specification of system behaviors in the logical architecture. Based on this thesis, Table D.4 provides improved versions of these templates and further templates to address the concepts Table 8.3 provides. The other

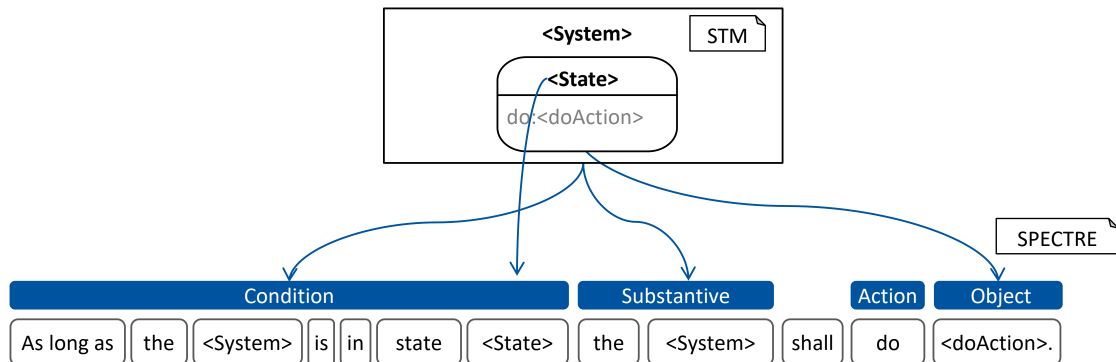


Figure 8.7: A straightforward transformation that inserts the names of the systems and states of a state machine diagram in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1).

templates for natural language requirement generation from system requirements only apply to the generation of grammatically correct requirements if the system modelers follow additional naming conventions for states and transition guards.

The application of the rules Table D.4 provides results, for example, in the following requirement to address the startup of the propulsion system.

Example 19 (Logical Architecture SPECTRE Requirement for the 'Starting' state in the propulsion state machine Figure 8.3)**.** *As long as the Propulsion System is in the state 'Starting', the 'Propulsion System' shall 'startup Propulsion'.*

To enter this state the system requires the following relation:

Example 20. *As soon as the Propulsion System receives the trigger 'PropulsionStart' AND the Propulsion System is in the state 'Off' AND propulsion is available, the Propulsion System shall enter the state 'Running' AND initialize startup.*

Moreover, the propulsion must be initially in the off state:

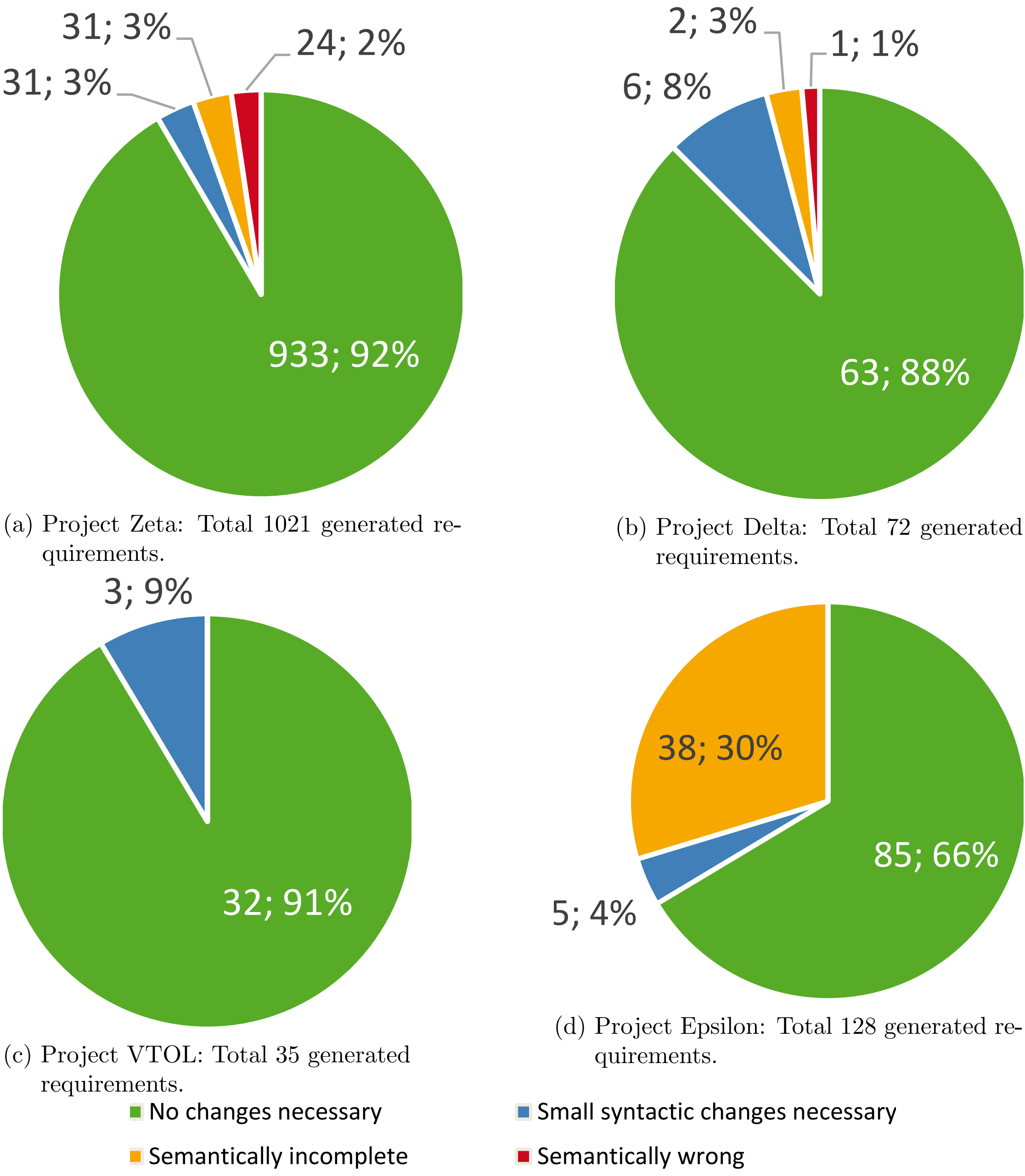
Example 21. *As soon as the Propulsion System becomes operational, the Propulsion System shall enter the state 'Off'.*

8.3.2 Correctness Evaluation of Generated Logical Architecture Requirements in Model-Based Projects

As a demonstration of the application of the method to derive correct natural language requirements from logical architecture models, this section presents an evaluation of the generated requirements in the context of the Zeta, Delta, VTOL publication in [JGW+21], and the models in the Epsilon projects based on evaluation results [Zab23] presented for the Zeta and Delta project, and new results for the VTOL and Zeta projects. The evaluation was conducted using the same evaluation scheme (*cf.* subsection 6.4.3) and evaluation process as for the stakeholder value and operating principle requirements in subsection 6.4.4 and subsection 7.3.3. Note that the Epsilon and VTOL model in [JGW+21] do not provide a state machine to describe the system behavior on the operating principle. Therefore, the evaluation results in these projects focus on the interface and activity requirements, which requires that the transformation templates from Table D.3 be generated for the system.

The evaluation results in Figure 8.8 show a project-independent trend for generating correct requirements on the logical architecture level. Therefore, the overwhelming majority of the natural language requirements are usable in the projects without any correction, as the generation method from Table 7.7 resulted in correct requirements in most cases. Figure 8.8d is the only outlier of this trend, as though a significant majority of the requirements are correct in natural language representation, but a group of incorrect requirements remains.

As the further error classification in Figure 8.9 depicts, most of these errors are comparable in all projects and often result from errors already present on the previous abstraction level. The most significant cause of errors in the generated natural language



(a) Project Zeta: Total 1021 generated requirements.

(b) Project Delta: Total 72 generated requirements.

(c) Project VTOL: Total 35 generated requirements.

(d) Project Epsilon: Total 128 generated requirements.

Figure 8.8: Correctness evaluation results of the logical architecture requirements generated in the Zeta, Delta, VTOL, and Epsilon projects.

requirements was caused by grammar and spelling errors, as Figure 8.9a depicts. The next most significant group of errors then resulted from wrong parameters, from which most propagated from the errors in operating principle models. However, some only became evident on this level, as the reviewers could apply their implicit knowledge of the LRA that a specific system may not receive or provide a specific object. As a result of these findings, additional implications on the LRA can be drawn *e.g.,* that additional constraints might be beneficial to make this implicit knowledge explicit. For example, to describe and automatically check that only the 'Energy' shall provide energy to other systems or that the 'Infotainment' system shall not provide control commands to the 'Propulsion' System. As Figure 8.9a shows, the violation of the guideline AN1 Table 7.7 of some modelers in the Zeta also propagated to this abstraction level and caused the errors in the 'Action formulation guideline violated' class. Furthermore, some misuse cases from the stakeholder value also lead to wrong actions on the logical architecture level in the 'Wrong Action' class. Interestingly, not all wrong actions reached this level, as the modelers on the operating principle level recognized this error and decided not to allocate these actions to a system. Finally, the requirements in the 'Wrong Action' and 'Wrong Type' classes also resulted from errors on the previous abstraction level. As the reduced number of requirements in the 'Wrong Action' class reveals, some modelers noted their errors and did not allocate them to an executing system. In contrast, other actions were wrong in the context of their executing feature but correct when considered on the system level.

In the Delta project, most erroneous requirements resulted from grammar and spelling mistakes, as Figure 8.9b depicts. Moreover, one erroneous action from the operating principle level (*cf.* Figure 7.4b) that a default name the modeling tool caused also propagated to this level, leading to the 'Wrong Action' error.

The same observation is possible in the VTOL project (*cf.* Figure 8.9c), where all errors propagated from minor grammatical errors in the names of the actions on the system level. As the publication does not contain a state machine to describe the logical system behavior, no additional errors in these requirements were introduced.

Finally, as Figure 8.9d depicts, also all grammatical errors in the operating principle requirements lead to erroneous system requirements. Furthermore, the modelers expressed underspecification in defining types by omitting the type at the port. Because this leads to erroneous requirements and misleading architecture models, it is a good practice to use general types as the functional modeling paradigm provides (*cf.* section 2.3) in these cases.

In conclusion, the analysis of the natural language requirement concerning their correctness reveals that a derivation of correct logical architecture requirements using the templates and guidelines from Table D.3 is possible. Moreover, the generated requirements also aid the modelers in uncovering minor errors in the model; however, because

of the low percentage of errors in the models at this abstraction level, detecting these errors in the models might be more efficient in the considered projects.

8.3.3 Completeness and Transferability Evaluation of Generated Physical Architecture and Interface Requirements in Model-Based Projects compared to a Document-Based Specification

Because the document-based specification of the system in the Epsilon system did not contain a logical architecture but directly mapped the actions from the operating principles to their realizing components, this section uses the same templates for natural language requirement generations from the logical architecture to generate requirements for the technical architecture. By this, this section not only aims at answering the question of whether a complete set of natural language requirements is derivable from a SysML architecture model but also to gain additional insights on the transferability of the methods this thesis provides for the logical architecture in the context of the physical architecture specification, which is outside of the scope of topics this thesis investigates. Regarding the logical architecture, however, no definite answer is possible based on these results.

As the modelers in the Epsilon project modeled this technical architecture based on the product documentation, this thesis adds the following template to the templates Table D.3 based on previous works in [Zab23] and uses them to generate natural language requirements for the project architecture.

Template 1. *The <TechnicalComponent> shall realize the <LogicalSystem>.*

Moreover, using the templates from Table D.3, additional requirements from implementing the actions from the operating principle must also be generated for the technical component. Using the same process as in subsection 8.3.2 requirements, engineers first evaluated the correctness of the derived requirements according to the evaluation scheme in Table 6.4. In comparing the enclosed requirements with publicly available models, the same method was also applied to the technical architecture in the VTOL model from [JGW+21] to provide comparable examples for other researchers.

As Figure 8.10 reveals, most of the generated requirements in both projects were correct. However, some requirements with grammatical errors that resulted from minor grammatical errors in the action specification of the operating principles also propagated to this abstraction level. Moreover, the underspecification of the types in the logical architecture leads to undefined types in the technical architecture, as the modelers needed to derive these types from the provided documentation. Because these model errors are not a methodological weakness but caused by known issues, the results in Figure 8.10 show that the templates from the logical architecture are transferable to the technical

architecture of the considered projects and suited to derive correct natural language requirements.

As the further error classification in Figure 8.11 reveals, all of the errors in the VTOL project result from minor grammar and spelling mistakes that are easily correctable. In the Epsilon project, however, the error classification reveals that the grammar and spelling mistakes are only responsible for the minority of errors in the generated requirements. In contrast, most of the errors result from unspecified types at the ports of the technical architecture. Although this form of underspecification would not be problematic at many other parts of the system architecture, this might become an issue for the technical architecture, as this architecture determines the actual implementation of the system for which a detailed and refined specification of all implemented elements is not only beneficial but also required. Therefore, the systems engineers of the Epsilon project should spend additional effort on correcting these requirements and investigating why the types of these ports in the technical architecture were unspecified. Consequently, the errors are comparable in all projects and often result from errors already present on the previous abstraction level.

To further evaluate the completeness of the generated requirements for the technical architecture, the reviewing requirements engineers mapped the results of the natural language specification to the legacy requirements of the document-based systems engineering process specified for this abstraction levels in the inputs of the Epsilon project. They decided whether the generated requirements fully cover the legacy requirements they replace. As the results in Figure 8.12 show, the generated requirements address most of the legacy requirements (*cf.* Figure 8.12). In comparison, the generated requirements not only address the legacy requirements but also provide additional requirements that the legacy requirements did not contain (*cf.* Figure 8.12b). Contrasting the previous correctness evaluations, more generated requirements are required to address one legacy requirement, as the legacy requirements were not atomic; therefore, multiple atomic requirements from the generated requirements were needed to address the requirement in the legacy specification.

As tracing the requirements to the operating principle level revealed, all of the unaddressed requirements in the technical architecture were derived from requirements that the operating principle level of the Epsilon project deliberately excluded from their new model-based specification, which Figure 8.13a additionally presents as a pie chart. Moreover, the evaluation of whether the generated requirements fully reflect the contents that the legacy requirements contain (*cf.* Figure 8.13b) reveals that, although most requirements in the legacy requirements were correctly addressed some requirements additionally contained additional conditions. Most of these requirements were already incorrect on the operating principle level; the remaining requirements, however, refined the requirements from the operating principle by adding additional information

for the realization as an additional condition to the requirement. Consequently, modeling technical architectures requires further mechanisms to refine requirements and models from previous abstraction layers.

Although this correctness evaluation only addresses one application project and is therefore not representative of all projects in the automotive industry, the results already reveal that some aspects of the logical architecture modeling are transferable for the technical architecture model, additional concepts for the refinement of operating principles for the new constraints the product implementation sets are, however, required for a successful application. Therefore, future works on this topic might relate to the results from this evaluation to additionally address these topics in their work.

8.3.4 A Critical Reflection and Lessons Learned from Generating Logical Architectures in the Automotive Industry

Respecting the efficiency and applicability of the approach this thesis presented in this chapter, and to address **RQ-4.3** ("Is the efficiency increased by this automation?"), this section presents several insights from applying the logical architecture generation in the Zeta project. The results are based on the prior publication in [GKM+25]. Following the same structure as in this publication, this section first describes the application of the transformation rules in Figure 8.5 in the Zeta project and summarizes the lessons learned from the application in this project. Next, this thesis presents the analysis results from [GKM+25] to determine whether this approach was capable of increasing the efficiency in the project as an answer to **RQ-4.3** before this section reflects on the fulfillment of the methodical requirements section 3.3 defined. As the applied approach has some limitations, this section then discusses these limitations in the context of the results this section summarized, as presented in [GKM+25].

Efficiency Evaluation of the Project Team Applying a Logical Architecture Evaluation

As the method for generating the logical architecture section 8.2 may contribute to the reduction of the efforts that the involved systems engineer spent into the creation of the system model, **RQ-4.3** aims to address this aspect to determine whether this kind of automation reduces the effort in the project. To this end, [GKM+25] investigates the impact of this approach in the context of the Zeta project. As a method to determine this increase in the team's efficiency, [GKM+25] compares the performance of the project team with a benchmark from previous projects that do not rely on this automation. To gain insights on the efficiency, [GKM+25] compares the hours the modelers spent in the logical architecture modeling project phase with the hours spent in comparable projects. Therefore, [GKM+25] applied this chapter's approach in the automotive development

Table 8.5: Number of elements modeled and transformed for the 16 features modeled in the automotive development project as presented in [GKM+25]

Operating Principle	
Actions 380	Parameters 749

Logical Architecture	
Sub-Systems 6	Generated Interfaces 549

project Zeta to create a high-level vehicle architecture for software and hardware development. In the Zeta project, as also summarized in subsection 4.1.6, a relatively small team of 7 systems engineers created a model-based specification for 16 vehicle features. As an implementation, the Zeta project relied on Enterprise Architect as a modeling tool and used a project-specific implementation of the transformation rules from section 8.2 [GKM+25].

To respect the thought schedule of the Zeta project, the analysis in [GKM+25] relies on an initial benchmark and estimation known from the previous projects, from which it was known that modelers integrate each feature into the already existing logical architecture as Figure 3.3 specifies. To coordinate this work with colleagues working on individual features, [GKM+25] respects additional efforts during the logical architecture creation project phase. Then [GKM+25] presents that the remaining effort in this phase contributes to the creation of the logical architecture model the logical architecture, the creation of diagrams for alignment with the technical expert, and the alignment of the result with all relevant stakeholders [GKM+25]. As the automation section 8.2 inherently only contributes to the creation of the logical architecture but not the remaining efforts in this project phase, not all efforts are automated. To respect this effort, [GKM+25] assumes that only (50%) of the overall time required in this project phase is saved using this automation. Based on this estimation, the Zeta project management planned its milestones accordingly, and [GKM+25] revisited the results from the initial planning after the completion of the project and compared the initial planning with the actual efforts the modelers reported during this project phase. As a result, [GKM+25] concludes that the modelers only required (46.87%) of the hours than in comparable projects (*e.g.,* the Gamma, Delta, or Epsilon projects) that did not use the automation as presented in section 8.2.

To improve the overview of the provided elements, [GKM+25] additionally provides

the number of generated elements, which Table 8.5 also presents to answer in **RQ-4.3** in the context of this thesis. Table 8.5 shows that 16 with combined 380 and 749 when distributed to 6 results in the generation of 549 when applying the transformation rules from subsection 8.2.4. Thus, the numbers reveal that not all parameters in the logical architecture lead to generated interfaces in the logical architecture. This is caused by the fact that adjacent actions in the operating principle model do not lead to interfaces in the logical architecture, if the actions are performed by the same system. Consequently, the object flows stays within the system and does not generate an interface in the logical architecture. Moreover, the numbers reveal that already a small number of initial functions and operating principles may already have a significant impact on the number of model elements on later decomposition levels. Because the considered scope of [GKM+25] only represents the first decomposition level and a small scope of vehicle functions, it is easy to imagine that the number of interfaces almost grows exponentially for the following decomposition levels as soon as these interfaces are refined as [GKM+25] already stated. Therefore, [GKM+25] additionally concludes the considerable importance of interface handling during the application of the methods [GKM+25] and this section presented. Moreover, it is also possible to imagine that the observed efficiency even increases in later project phases when the systems engineers consider further decomposition levels.

For **RQ-4.3**, these numbers show that automation potential subsection 8.2.4, when applied in a project such as the Zeta project, actually increases the efficiency of industry projects. Thus, the results from [GKM+25] reflect the qualitative potential of the approach. However, quantitatively, [GKM+25] argued that these results might threaten validity. First and foremost, the project estimated the saved efforts in advance and planned the project accordingly. As this planning was known to the project members, this might have influenced their performance and time reporting, as the tighter schedule communicates an expectation on the project team to create the models faster, which results in increased pressure in the project [GKM+25]. Since other studies have already observed increased project productivity by communicating higher demands, an increased efficiency of the project might also be an effect of the project team striving to meet these expectations on the costs of other negative aspects in quality or productivity. This effect was, for example, observed in [NPS06] in the context of construction performance. Another impact [GKM+25] reports is the effort required to fix errors in the diagrams. As the errors the domain experts found in the derived logical architecture result in errors in the underlying operating principle, the modelers additionally spend time correcting these errors during the logical architecture phase, which also impacts the overall efficiency estimation. As this thesis, however, focuses on the application of software and systems engineering to effectively and efficiently utilize system models, the evaluation results from [GKM+25] in the context of the Zeta project are sufficient to show that an application of the method this chapter contributes to this goal.

Moreover, [GKM+25] also observes that the bottleneck in the project shifted from the

availability of the modeling systems engineers to the availability of the functional domain experts, whose availability and response time limited further improvements in the timeline. Furthermore, [GKM+25] also states that the creation of a correct diagram is often also one of many time-consuming aspects of modeling when using graphical modeling languages such as the SysML [Obj19]. However, the layout of the diagram elements also dramatically impacts the understanding of the diagram according to [Stö11]. Because the modeling tool (Enterprise Architect [Spa21]) needs to provide sufficient methods for creating satisfactory diagram layouts, the modelers still spend considerable effort to lay out the generated diagrams. As the correction of the underlying operating principle diagrams also influences the efforts observed in the logical architecture modeling phase, two additional implications are derivable from the insights in [GKM+25]. First, that the acceptance of the model as a development artifact in an industry project not only relies on its correctness but also on its presentation, and second, that the actual implementation of CUBE sometimes does not follow a linear process, but instead structures the created artifacts a-posterior.

Lessons Learned from Generating Logical Architectures in the Automotive Industry

Apart from the observed efficiency, [GKM+25] additionally provides lessons learned from applying the methods in the Zeta project, which also applies to the methods this thesis presents. These lessons primarily address the execution times, collaboration, and traceability, summarized in the following three lessons that are relevant in the context of this thesis:

- Execution Time & Space Complexity: Although the execution times and space complexity of the transformation rules are more than sufficient for the industrial application, the number of generated model elements is concerning, as many features could easily reach a high number of elements. Although this is not critical for the automated interface generation, this becomes problematic for the manual creation of operating principles on the next decomposition levels, which is typically more costly than the automated generation. Therefore, it is a good practice to critically review the interfaces created on higher decomposition levels to prevent the models from containing unnecessarily many interfaces and manually maintained actions on lower decomposition levels.
- Collaborative Modeling and Variant Handling: In the applied modeling tool Enterprise Architect, collaborative modeling and variant handling was additionally problematic, as the native transformation rules [GKM+25] used and this thesis also presents and extends in section 8.2 provide no other mechanisms for variant handling. As a result, the model elements the transformation creates contain no additional mechanisms for variant handling as section 2.4 identified as an essential practice for automotive system development. Therefore, the lesson learned from

the application in [GKM+25] is that additional mechanisms for variant handling, as well as creating a packaging and import structure for collaboration and model reuse is required to improve further the implementation concerning the other goals of the method requirements from section 3.3.

- Traceability: Moreover, [GKM+25] identifies that creating additional links between the parameters and action pins and the ports in the IBD simplifies the navigation and supports traceability. Therefore, the learned lessons for this thesis are analogous to the generation of natural language requirements: The creation of links between the original and generated artifacts enables traceability and contributes to the usability of the model.

As the first lessons from [GKM+25] states, the implementation of the transformations performed well in the modeling tool; the underlying database, however, quickly reached its limitations when multiple features were considered, and multiple modelers modeled different features in parallel [GKM+25], which results from the implementation of the modeling tool, which by design always aims to access the full view of the model, although the specific application would only require the modeling tool only to access a small part of the model. With this observation, [GKM+25] also runs into a problem that the earlier work in [GHK+08b] presents, which states that modeling tools typically only provide total views of the systems, which does not scale well on large amounts of functions.

Critical Reflection on Assumptions, Limitations, and Guidelines

The approach this thesis extends in section 8.2 inherits some of the limitations from the approach [GKM+25] presents. First, the application and logic of CUBE only allow systems engineers to allocate actions from a system operating principle to subsystems of this system, as only individual components of the system can perform actions so that the system executes the function under consideration. As a result, material, energy, or data provided or received within the system is not visible or accessible from outside the system and, therefore, not considered in the logical architecture. Although this is reasonable from a methodical point of view, this consideration has the inherent limitation that these flows are not represented in the logical architecture. Therefore, system engineers unfamiliar with the system might need to be aware that specific data, material, or energy is already available. As a result, they might allocate actions to executing systems that are suboptimal for the overall system architecture, as new interfaces must be implemented to realize components that might not be required in another system. To mitigate this risk, which results from the design of the second transformation rule (*cf.* Figure 8.5b), [GKM+25] suggest integrating some form of capability model into the logical architecture to make the systems engineers that work with the logical architecture aware of the other actions and object flows that are already available within a system.

Regarding the third and the fifth transformation rule (*cf.* Figure 8.5c and Figure 8.5e), [GKM+25] identifies the risk that the rules do not eliminate illegal connections. For that reason, if a systems engineer models, for example, an illegal connection where an input pin receives inputs from multiple output pins or parameters without a merge node in-between or incompatible types, the resulting logical architecture would also contain these errors. As a mitigation, the Zeta project used additional context condition checks to identify these inconsistencies beforehand to improve the overall quality of the resulting logical architecture [GKM+25].

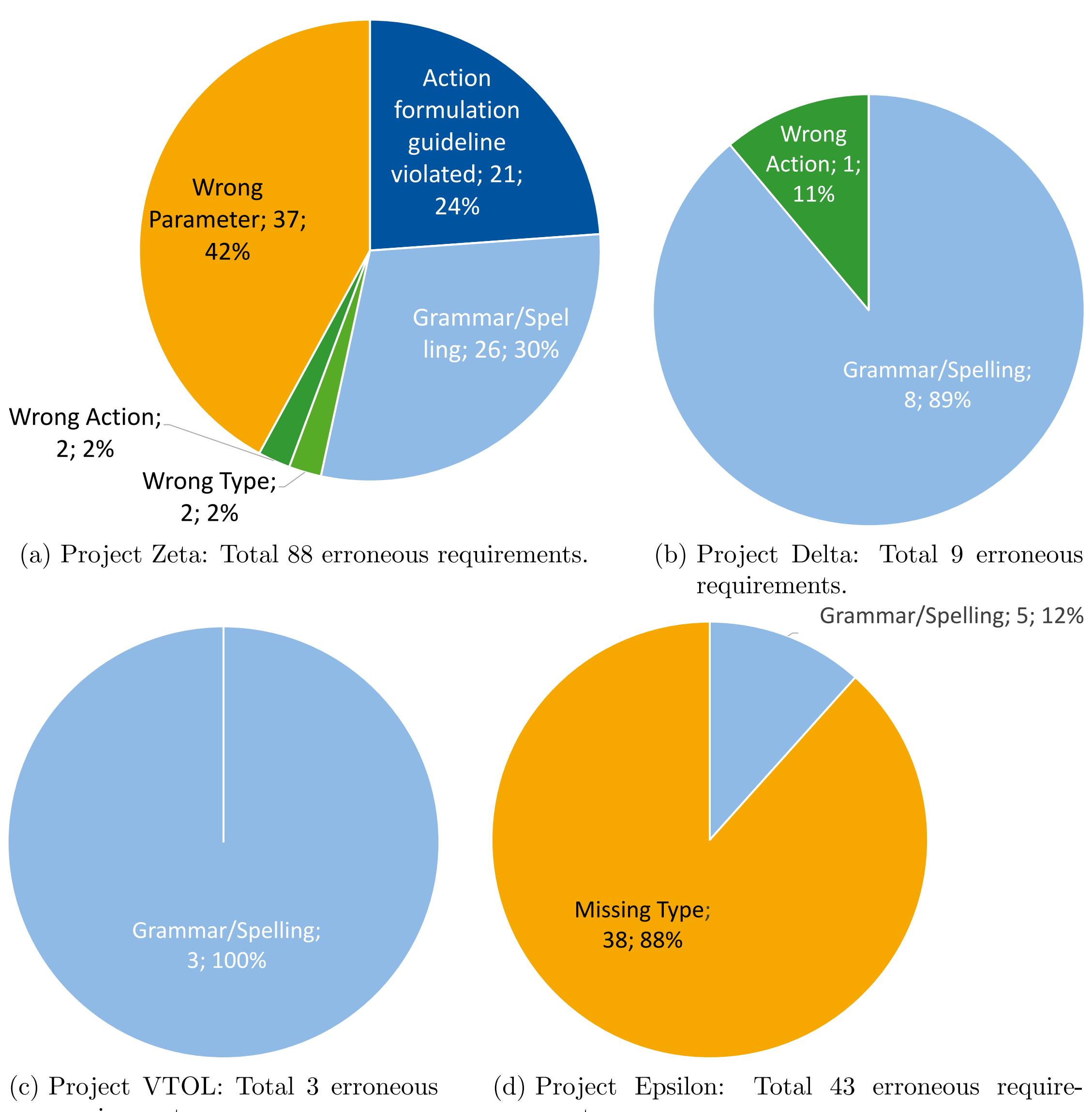
Moreover, the fourth and the sixth transformation rule (*cf.* Figure 8.5d and Figure 8.5f) inherit from [GKM+25] that all unresolved interfaces are automatically delegated to the environment, as interfaces the system cannot provide or receive itself consequently interact with systems outside of the system boundary. As stated in [GKM+25], the Zeta project aimed at a higher reuse of the operating principles and resulting logical architecture as required in MR1. With this decision, the transformation rules, however, have the risk that modeling errors in the operating principles, where modelers forgot to model the information flows, manifest in the design of the logical architecture, as it is no longer possible to differentiate unintentional missing providers or receivers from explicit environment interfaces. As a mitigation, the Zeta project implemented a validation phase of the generated logical architecture models in which domain experts reviewed the generated logical architecture concerning the environment interfaces, after which the systems engineers reworked and regenerated the logical architecture for all errors the domain experts identified [GKM+25].

As already stated concerning **RQ-4.3**, the insights from the Zeta project, as also published in [GKM+25], only provide a qualitative statement about the increased efficiency and a functionality demonstration for the used tools, which is not an exhaustive tool evaluation. Therefore, the results this thesis and [GKM+25] presented only focus on the methodological feasibility of the transformation and deliberately excluded the aspect of tool evaluation [GKM+25]. Therefore, future works might focus on this aspect in more detail. Moreover, an exhaustive execution and timing behavior analysis was not conducted in the context of the Zeta projects, as the used implementation was already sufficient for an industrial application and not reported as critical by the users apart from the tooling aspect of this thesis already summarized in the lessons learned from the industrial application of the transformation rules.

Another aspect this thesis also did not cover in the application of the transformation from section 8.2 in the Zeta project: the execution of the operating principles in the logical architectures. As the behavior of a logical architecture following the understanding Figure 3.11 depicts based on [BGM+09] is state-based whereas the operating principle is action or flow-based, an additional transformation might be required, or additional concepts to cover timing behaviors are applicable. Previous works already addressed some

of the challenges by providing methods for the derivation of test cases from activity diagrams [MXX06, DGH+19] or providing means for a scenario-based specification that makes the systems executable using scenario simulations [MSG+22]. In addition, the modeling approach for modeling functionalities is based on modeling and decomposing activity diagrams, which itself is an exciting research and application topic for which several approaches exist [GRR10, CJHL18]. Since the methods this thesis presents still has to consider several possible decomposition strategies, the approach relies on the modeler of correctly modeling and decomposing the activity diagrams in the operating principles of a feature [GKM+25]. Because the system's dynamic behavior was not part of the considered project phase in the Zeta project, the execution of the operating principles was not covered in the context of the Zeta project.

Another limitation of the approach section 8.2 presented was already mentioned in [GKM+25] and concerns allocating actions to logical architecture elements. While it is a reasonable assumption or simplification for the development of a logical architecture that action may only be executed by a single system and not distributed over multiple systems, as it simplifies the system decomposition as subsubsection 8.1.1 motivates. The organization of the project or enterprise, according to [Con68] (*cf.* subsubsection 8.1.1), still limits the possible applications. As a first limitation, cardinalities or redundancies are complicated to model using the approach, as executing an action in multiple systems would require allocating the action to multiple systems. This makes the approach unsuitable for deriving the technical architecture, where multiple parts are often responsible for executing an action or redundantly performing an action. Imagine, for example, the four tires of a passenger car. All of these tires act to transmit the driving force to the road. In the approach this thesis proposes, however, it is not allowed to integrate the aspect of four redundant tires into the logical architecture of the vehicle, although this aspect is a classifying aspect of passenger cars in the European Union. To solve this issue, a similar automation would have to address these aspects and extend the set of transformation rules to handle redundancy and cardinalities.



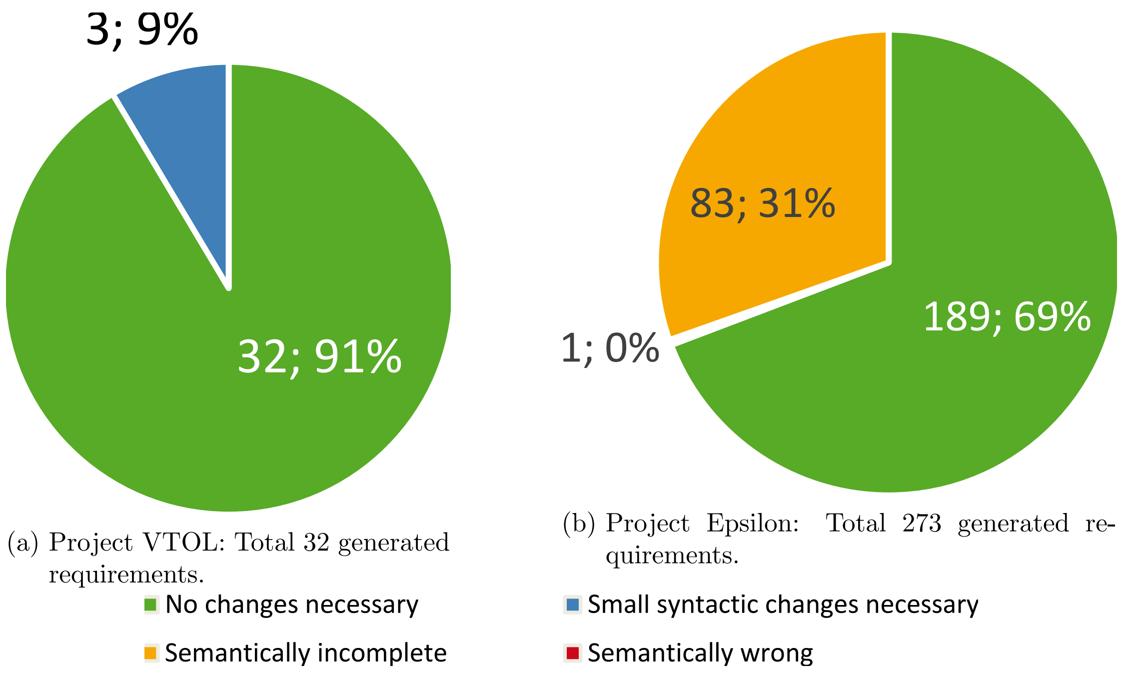
(a) Project Zeta: Total 88 erroneous requirements.

(b) Project Delta: Total 9 erroneous requirements.

(c) Project VTOL: Total 3 erroneous requirements.

(d) Project Epsilon: Total 43 erroneous requirements.

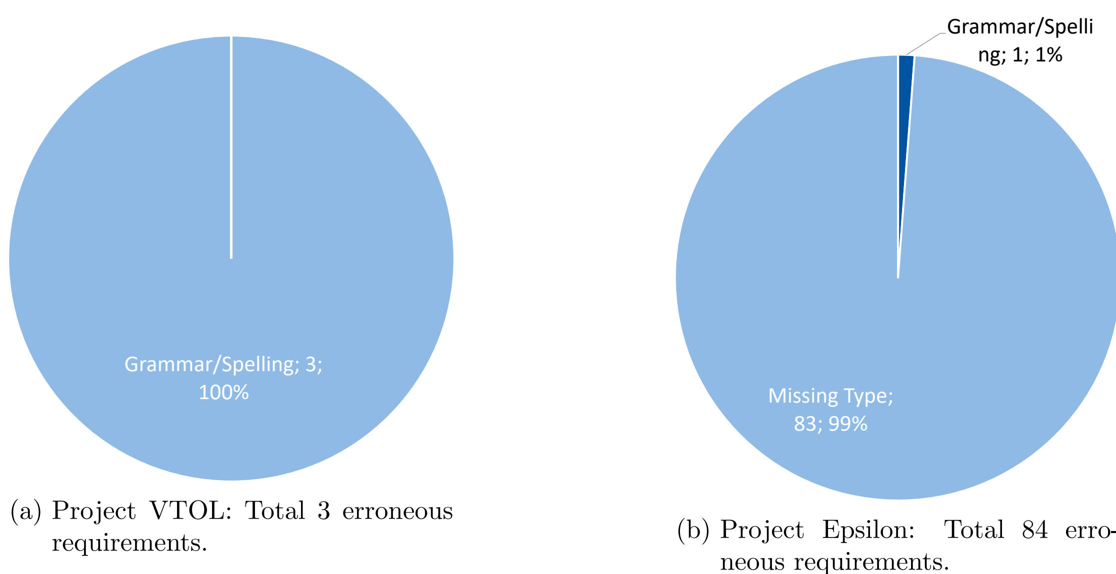
Figure 8.9: Results of the error reason classification in logical architecture requirements in the Zeta, Delta, VTOL and Epsilon projects.



(a) Project VTOL: Total 32 generated requirements.

(b) Project Epsilon: Total 273 generated requirements.

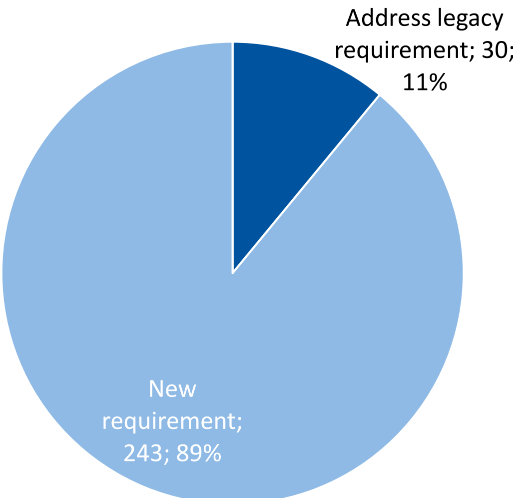
Figure 8.10: Correctness evaluation results of the technical architecture requirements generated in the VTOL, and Epsilon projects.



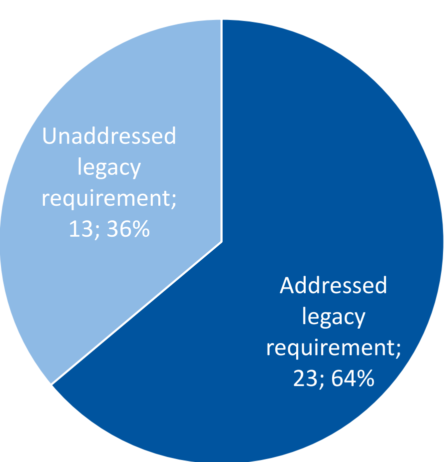
(a) Project VTOL: Total 3 erroneous requirements.

(b) Project Epsilon: Total 84 erroneous requirements.

Figure 8.11: Results of the error reason classification in technical architecture requirements in the VTOL and Epsilon projects.

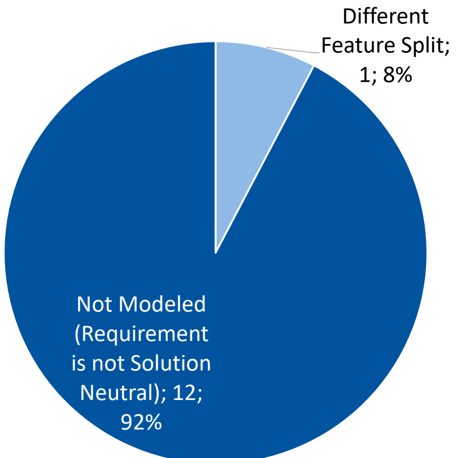


(a) Evaluation how many of the 273 generated requirements in the Epsilon address one of the 36 legacy requirements.

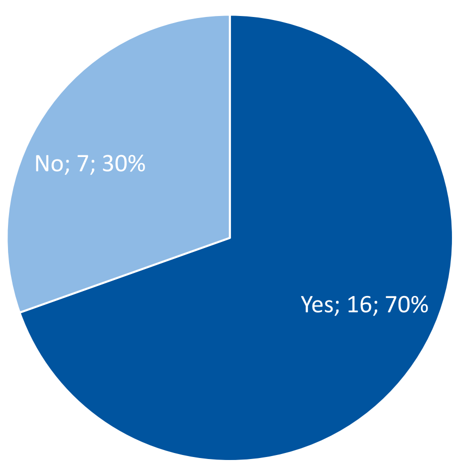


(b) Evaluation how many of the 36 legacy requirements in the Epsilon project address one of the 273 generated requirement.

Figure 8.12: Results of the completeness evaluation of the technical architecture requirements comparing the model-based stakeholder value specification from the Epsilon projects with the document-based legacy requirements the project used as input.



(a) Reason evaluation for the not generated legacy requirements: Total 13 unaddressed legacy requirements.



(b) Evaluation if the generated requirements cover the legacy requirements: Total 23 generated requirements.

Figure 8.13: Results from the interviews with the modelers in the Epsilon project to determine the reasons for not modeling the requirement and to evaluate if the generated natural language requirement correctly covers the legacy specification.

8.4 Interim Conclusion and Discussion

In conclusion, this chapter answers **RQ-3.4** (“How to model logical architectures?”), **RQ-4.2** (“Are logical architectures derivable from operating principles?”), **RQ-4.3** (“Is the efficiency increased by this automation?”), and **RQ-4.4** (“Are the models that result from a model-driven specification comparable to document-based specifications?”).

Therefore, chapter 8 addresses **RQ-3.4** and provides SysML Internal Block Diagram (IBD) elements to describe the systems and their interfaces in Table 8.4. In addition to this static structure, Table 8.3 provides the elements of a SysML State Machine Diagram (STM) that are required to describe the state-based behavior of these systems.

Answering **RQ-4.3**, section 8.2 provides transformation rules to translate operating principles to system and system interface rules using a structural Logical Reference Architecture (sLRA). Moreover, state-based behavior follows an additional assignment of all activities, from the operating principles to actions at the states' transitions or entry, do, and exit activities. By this, this chapter does not provide its derivation of a unique logical architecture with structure and behavior. However, it relies on a manually created reference architecture with the required states and transitions. Future research might focus on an automated transformation that [KR18] suggests to transform activity diagrams to state charts that are closely related to state machines in SysML, as elaborated with a prototypical implementation in [Bla23] in the broader context of this thesis. Another method is also discussed in [DH01], where life sequence charts that are related to sequence diagrams serve as a basis to bridge the gap between the flow/message-based behavior of a

Concerning **RQ-4.3**, subsection 8.3.4 analyses the impact of this method on the efficiency of the project team in the Zeta project. Although an actual efficiency increase was observable, subsubsection 8.3.4 addresses some limitations and threats to the validity of the observations.

Respecting **RQ-4.4**, section 8.3 provides additional transformation rules to derive natural language requirements in subsection 8.3.1 and evaluates the correctness and completeness of the generated requirements in the considered projects in subsection 8.3.2 and subsection 8.3.3. As the results indicate, the derived requirements are comparable to a document-based specification, and deviations result from the shortcomings of the simplified transformation in subsection 7.3.1 used for the evaluation. Therefore, future work might focus on improving the models and transformations at the operating principle level.

Finally, regarding the methodological requirements, subsection 8.2.4 already mentioned their fulfillment.

Chapter 9

Conclusion and Future Work

As a conclusion of this thesis, this chapter summarizes the primary results and contributions. To this end, section 9.1 outlines the main results from each chapter and relates them to the research question this thesis aimed to tackle. As a dedicated summary of the generation of the natural language requirements this thesis provided in section 6.4, section 7.3, and section 8.3, section 9.2 creates an overall view on the evaluation results the individual sections created concerning the respective method this thesis presented in the respective chapter. Next, section 9.3 summarizes these lessons and relates them with the respective method and application to share the condensed insights of applying the presented methods revealed in the automotive industry. Finally, section 9.4 summarizes ideas for future research directions.

9.1 Summary and Research Questions

This thesis focused on applying several established methods and principles from software and systems engineering in the automotive industry to define and apply system models effectively and efficiently to specify automotive systems. The following list summarizes the contributions of this thesis concerning the initial research questions from section 1.1:

Established Methods for Automotive Systems Engineering chapter 2 summarizes the foundations of system modeling, identifies the characteristics of an automotive system model, and examines the state-of-the-art systems engineering methodologies for developing automotive systems. With these contents, this chapter answers **RQ-1** ("Which established methods software and systems engineering methods exist that are applicable to specify automotive systems?") and provides the foundations for the remainder of this thesis. In particular, section 2.1 investigates the contribution of related works in systems engineering to condense a systems engineering understanding for this thesis (*cf.* Def. 10) and answers **RQ-1.1** ("What are the foundations of systems engineering?"). Next, section 2.2 focuses on extending this understanding to a system model in the automotive industry based on an original definition from [Sta73] (*cf.* Def. 13) and answers **RQ-1.2**

("What are the characteristics of a system model in the automotive industry"). Finally, section 2.6 investigates system decomposition, viewpoints, and paradigms in a selection of systems engineering methodologies (*cf.* Table 2.1) to answer **RQ-1.3** ("Which systems engineering methodologies exists, how is the system decomposed in this methodology, which viewpoints are created, and which paradigm is applied?").

Model Driven Systems Engineering using CUBE chapter 3 presents a model-driven systems engineering method based on CUBE and examines the processes and structures that systems engineers follow when applying this method as an answer to **RQ-2** ("Is there a common practice for creating, interpreting, analyzing and using system specifications in the considered projects in the automotive industry"). Since CUBE is the shared basis for the system design in the considered projects, the CUBE specification process as subsection 3.1.1 presents in Figure 3.2 answers **RQ-2.1** ("Which system specification process is applied in these projects?"). Moreover, Figure 3.3 highlights the artifacts the systems engineers create using CUBE and therefore answers **RQ-2.2** ("Which artifacts are created during this process?"). Knowing these artifacts, section 3.2 focuses on the derivation of the required system elements for a system architecture specification based on [BGK+09] and [BGM+09], leading over several individual views to the combined overview in Figure 3.14 as an answer to **RQ-2.3** ("Which elements are required to specify a system in the automotive industry?"). Furthermore, the chapter provides a general overview of how these elements relate to SysML diagrams as a first step towards answering **RQ-3** ("Which elements of the SysML support the creation of the elements developed under **RQ-2**?").

Apply System Models in the Automotive Industry chapter 4 summarizes the projects and highlights how the methods this thesis investigates were applied in the context of this project. As a result of this chapter, Table 4.1 answers **RQ-2.4** ("How is the system decomposed in the project, which viewpoints are created, and which paradigm is applied?").

Processing Natural Language Requirements chapter 5 examines a method for the specification of systems using structured natural language and investigates the status of requirements in two projects for developing automotive systems. By this, this chapter contributes to **RQ-3.1** ("How can requirements be expressed?") by providing the required SysML elements in Table 5.1 and the SPECTRE-DSL (*cf.* section 5.3 and Appendix B) to structure the requirement texts in natural language.

Modeling Stakeholder Values as Use Case Diagrams chapter 6 answers **RQ-3.2** ("How can stakeholder values be formulated?") by providing the required SysML model ele-

ments in Table 6.1 and Table 6.3. Furthermore, the chapter introduces a method and the semantic foundations to model system stakeholders, their needs, and values using SysML use case diagrams as the foundation to answer **RQ-4.1** ("Can operating principles be derived from use case scenario specifications?") in chapter 7. Finally, the chapter applies use case diagrams to replace natural language requirements as a means of expressing stakeholder requirements in document-based development projects in subsection 6.4.3 and subsection 6.4.4 showing, that the results are comparable to a document-based system specification as **RQ-4.4** ("Are the models that result from a model-driven specification comparable to document-based specifications?") investigates.

Modeling Operating Principles as Activity Diagrams chapter 7 establishes SysML activity diagrams (*cf.* Table 7.1 and Table 7.2) and use case scenario behavior templates (*cf.* Table C.1) to model the desired system behavior as the operating principle of a system answering **RQ-3.3** ("How to model operating principles?"). In addition, subsection 7.2.3 answers **RQ-4.1** ("Can operating principles be derived from use case scenario specifications?") by providing a bidirectional transformation from operating principles to use case scenario specifications that connects both concepts. Moreover, subsection 7.3.2 and subsection 7.3.3 show that a derivation of natural language requirements is possible using the method section 7.1 provides when combined with the additional guidelines from subsection 7.3.1. Therefore, the results are comparable to a document-based system specification as investigated by **RQ-4.4** ("Are the models that result from a model-driven specification comparable to document-based specifications?").

Deriving Logical System Architectures from Operating Principles chapter 8 provides further answers concerning **RQ-3.4** ("How to model logical architectures?"), **RQ-4.2** ("Are logical architectures derivable from operating principles?"), **RQ-4.3** ("Is the efficiency increased by this automation?"), and **RQ-4.4** ("Are the models that result from a model-driven specification comparable to document-based specifications?"). To answer **RQ-3.4** chapter 8 provides elements from the SysML SysML Internal Block Diagram (IBD) in Table 8.4 to describe the systems and their interfaces, whereas Table 8.3 provides the elements of a SysML SysML State Machine Diagram (STM) that are required to describe the state-based behavior of these systems. As an answer to **RQ-4.3**, section 8.2 then provides transformation rules to translate operating principles to system and system interface rules using a structural Logical Reference Architecture (sLRA). The state-based behavior is then followed by assigning all activities, from the operating principles to actions at the states' transitions or entry, do, and exit activities. Concerning **RQ-4.3**, subsection 8.3.4 analyses the impact of this method on the efficiency of the project team in the Zeta project with the result that an actual efficiency increase was observable. Finally, respecting **RQ-4.4**, section 8.3 provides additional transformation rules to derive natural language requirements in subsection 8.3.1 and evaluates the correctness

and completeness of the generated requirements in the considered projects in subsection 8.3.2 and subsection 8.3.3 showing that the derived requirements are comparable to a document-based specification.

9.2 Combined View on Generating Textual Requirements from Models

The evaluations of the natural language requirements in section 6.4, section 7.3, and section 8.3 render several abstraction layer-specific views on the elements the system model elements describe. To condense an overall result from these individual views, Table 9.1 presents an overview of the number of generated requirements on all abstraction levels, independent of the system decomposition dimension. As these numbers reveal, a majority of the requirements result from the Zeta project, because the most requirements were generated in this project. Moreover, most requirements were generated on the logical architecture level, where operating principle execution requirements and interface requirements merge into a system specification. Unsurprisingly, the orating principle requirements and the product architecture requirements represent the smallest group of requirements, as the first does not contain the interface requirements due to the simplification in Table 7.7 and the latter were only generated in two projects (*cf.* subsection 8.3.3). Consequently, the number of requirements on the operating principle and logical architecture level must be considered in combination, when comparing them to the previous abstraction levels. Interestingly, this number of generated requirements increases from the stakeholder value to the combined operating principle and logical architecture abstraction levels in the Zeta and the Epsilon project, remains on a comparable level in the Delta project, whereas it decreases in the VTOL project. While an increase of generated requirements also follows the intuition that more requirements are needed to specify a refined model of an abstract stakeholder need, a reduction of

Table 9.1: Number of requirements generated on the abstraction levels of the CUBE in the analyzed projects.

Project	Stakeholder Value	Operating Principle	Logical Architecture	Product Architecture	Total
Zeta	505	397	1021	Missing	1923
Delta	92	23	72	Missing	187
VTOL	103	15	35	32	185
Epsilon	50	44	128	273	495
Total	750	479	1256	305	2790

Table 9.2: Correctness evaluation results according to the evaluation scheme from Table 6.4 in the analyzed projects.

Class	Zeta	Delta	VTOL	Epsilon	Total
1	1586	116	165	346	2213
2	66	55	9	12	142
3	178	2	8	136	324
6	93	14	3	1	111
Total	1923	187	185	495	2790

requirements in the VTOL project seems counterintuitive at the first sight. An analysis of the use cases in [JGS+20] and the operating principle specification in [JGW+21] reveals, that the operating principle and logical architecture specification in [JGW+21] only covers a part of the use cases, which is less than half of the use cases. Thus, a further increase of generated requirements is expected for an overall model specification.

The summary of the correctness evaluations on the different abstraction views Table 9.2 further presents an overview of the requirement grades according to the evaluation scheme Table 6.4 presents. To display these results in a more accessible manner, Figure 9.1 provides the ratings in each project as pie charts. As the numbers in Figure 9.2 reveal, the majority of the generated requirements is correct, which is also the case in the individual views in Figure 9.1. Regarding the semantically incomplete requirements and requirements that require small syntactic changes, there are some project specific outliers, that result from project-specific effects section 6.4, section 7.3, and section 8.3 already discussed.

Following from the aforementioned shortcomings, the combined results do not provide a extensive overview of all possibly generated requirements in these projects, but nonetheless provide some considerable insights. First of all, that the majority of requirements the approach generates in correct in all projects, as Figure 9.2 highlights. Second, that there are similar errors in all projects with one outlier in the stakeholder needs specification in the Delta project (*cf.* Figure 6.10), which indicates that systems engineers tend to make similar modeling errors in the considered projects using the methodology chapter 3 proposes. And finally, that the generated textual notation may aid in the detection of these errors, at least for inexperienced modelers, but as subsection 6.4.4, subsection 7.3.3, and subsection 8.3.3 already discussed it can neither be sufficiently confirmed nor disproved that a complete set of requirements is generated. However, the considered examples in the Epsilon project indicate, that a derivation of a complete set of requirements without the need for additional natural language requirements might be possible.

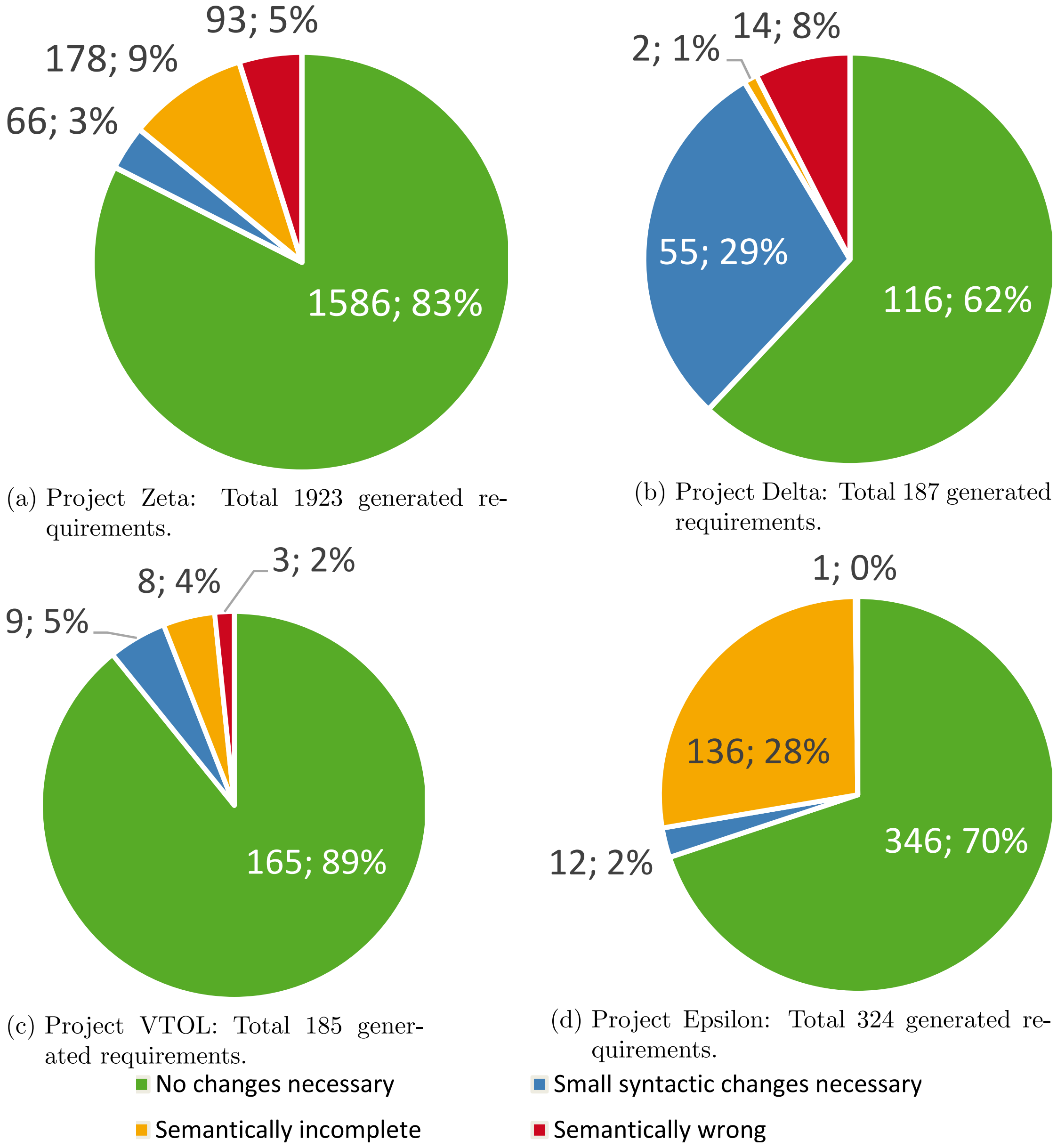


Figure 9.1: Evaluation results of the requirements generated in the Zeta, Delta, VTOL, and Epsilon projects.

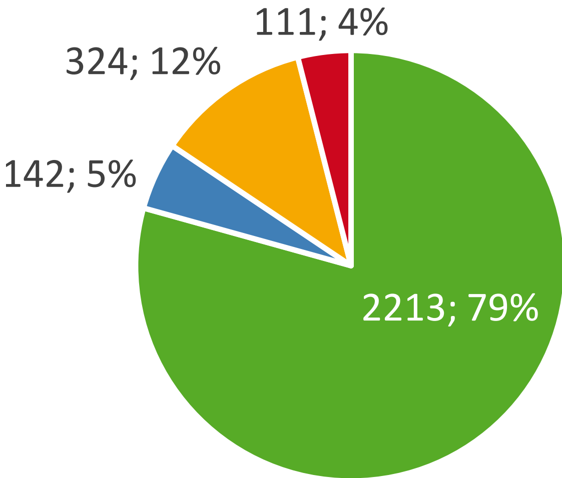


Figure 9.2: Project-independent evaluation results of the generated requirements: Total 2790.

9.3 Lessons Learned from the Industrial Application

Several lessons were learned in the industry projects' context during the applications of this thesis's methods, some of which chapter 4 presented. As a condensed version of these lessons, [GORW23] presents several insights into this application, for which this section provides references to the methods and application results this thesis presents.

9.3.1 Lesson 1: Systems Thinking

As [GORW23] formulates as a first lesson, the success of an automotive systems engineering project depends not only on the applied methods but also on the mindsets of the involved systems engineers. Therefore, several additional measures are required in some organizations and projects to challenge and encourage systems thinking [Che99] as a mindset during the systems development project. In this context, systems thinking describes a mindset that requires the systems engineer to transform implicit knowledge into explicit knowledge *i.e.,* a mindset to apply the modeling and elicitation methods for systems modeling chapter 6, chapter 7, and chapter 8 present. Although systems thinking offers these opportunities, [GORW23] reports that this way of thinking is not necessarily practiced in every organization and every project, but must be actively demanded and promoted. Therefore, it is a fundamental prerequisite for the success of a development project, which is why establishing this necessary mindset in the entire development team must be a key objective at the start of the project. As a result, systems engineers who aim to apply the methods presented in this thesis must be aware that

these methods exist and have the support of their colleagues and management when applying them.

9.3.2 Lesson 2: Analytical Modeling

As a second lesson, [GORW23] concludes that modeling requires analytical thinking and the ability to structure complex information. To incorporate this lesson in automotive system development, [GORW23] derives the best practice to present abstract concepts in visual form and infers that this visualization is not just about visualizing existing information but also about creating a system model that reflects both the current project situation and includes generic specification parts for future company developments.

To support this, this thesis not only provides a method that supports the reuse objective of CUBE [GKS+21] methodically during the creation of solution neutral operating principles in subsection 3.2.4, but also in the feature-driven use case modeling methodology in section 6.3, the suggested use of scenario use case behavior templates in subsection 7.2.2, and the project-independent LRA section 8.1 introduces for the specification of logical architectures. Moreover, [GORW23] recognizes this project-independent and uniform language and an understanding of the systems engineering processes. To address this challenge, **RQ-3** identifies the elements of the SysML a uniform application of the model-driven CUBE variant requires. Therefore, section 5.3 provides a requirements template and DSL to express natural language requirements uniformly and structured. Next section 6.1 provides the language to express and section 6.2 the semantic foundations to understand and communicate stakeholder values during automotive system development based on a previous work published in [KRW22]. Furthermore, section 7.1 provides the uniform language to model solution-neutral operating principles. Finally, section 8.1 sets the boundaries to define reusable logical architectures as SysML IBDs (*cf.* subsection 8.1.2) and their behavior as SysML STMs (*cf.* subsection 8.1.4).

Apart from the definition of the language, [GORW23] also highlights the importance of documenting the applied systems engineering process and its outcomes. By this, [GORW23] aims at improving the understanding of the relationships between the artifacts to enable the project management to plan and track systems engineering activities in a structured manner. To address this, section 3.2 defines a CUBE specific system architecture framework that specifies the process, its structure, and resulting artifacts in the process-structure models in Figure 3.5, Figure 3.9, Figure 3.10, Figure 3.11, Figure 3.12, and Figure 3.13 that extend the shorter summary [GORW23] presents.

9.3.3 Lesson 3: Difference between Syntax and Semantics

As the third lesson, [GORW23] concludes that a system modeling language is not only defined by its syntax ("what we see"), but above all by its semantics ("what it means") [BR23]. Based on this lesson, [GORW23] derives the best practice that all systems engineers involved in the systems engineering process must clearly distinguish between these concepts to make precise communication possible and avoid misunderstandings. To implement this learning in the methods this thesis presents, the SysML-based modeling languages this thesis uses have precisely defined semantics. As no standard semantic definition for use case diagrams was known to the author, section 6.2 presented the semantic foundation for this diagram based on [KRW22]. Moreover, the operating principle specification draws on activity diagrams for which other related works [Obj21, GRR10, MRR11b, MRR11a, KR18, Kau21] already provided a semantic foundation. Finally, the behavior description of logical architectures uses state machines for which for which also several semantic foundations are available [Mea55, Moo56, Har87, BCR07, NSC+07, DEKR19].

9.3.4 Lesson 4: Degree of Formalization

Although the third lesson requires a certain degree of formalization to enable a precise semantics definition, the fourth lesson from [GORW23] states that a high degree of formalization also might bring several disadvantages in heterogeneous development teams in which team members from different domains and with different knowledge work together. Therefore, [GORW23] suggests that the degree of formalization of a system modeling language and a system model should be adaptable to the application areas and the intended uses of the model. Thus, when deciding on the degree of formalization for a specific model, the relationship between benefits, costs, and effects on acceptance in the development project must contribute to the decision process. Several measures are possible to achieve this in the methods this thesis presents. First, the derivation process of natural language requirements already suggests integrating natural language in the naming of the components to facilitate the model's readability for stakeholders who aim to avoid reading mathematically precise statements, which does not affect the constraints on the execution order in the rest of the diagram sets. Moreover, the scenario behavior specification template section 7.2 simplifies the diagrams' presentation without affecting the defined behaviors' precision. Finally, the architecture generation procedures section 8.2 supports the final statement from [GORW23] that if something can be automated efficiently, this potential should be used to avoid errors, save time, and increase quality and, therefore, the required formalization is acceptable.

9.3.5 Lesson 5: Reusable Models

The fifth lesson [GORW23] derives is concerned with reusing system models across and within different system development projects. The lesson identifies that a reusable model requires a clear separation between a function-oriented/solution-neutral view and a product-oriented/solution-specific view. As a way to implement this lesson in the methods, this thesis supports the principle of CUBE to specify operating principles as a solution-neutral [GKS+21] action sequence and supports splitting the definition of an operating principle (*cf.* subsection 7.1.1) from the allocation subsection 7.1.2. As another way of reusing parts of a system model, the logical architecture definition from chapter 8 supports the definition of a project-specific and project-independent logical reference architecture. By this, the logical reference architecture ensures that the same (logical) systems are available in both projects, and a project-independent logical reference architecture enables cross-project reuse. As for the degree of formalization, however, [GORW23] additionally suggest that the systems engineers that develop the solution neutral weight up the degree of solution, similar to the degree of formalization, in order to determine whether the benefits of possible reuse in another solution justify the effort involved in creating and understanding the model. subsubsection 8.1.1 considers this aspect, for example, in the different methods to decompose a system into logical components. In addition, during the definition of operating principles, it is also a good practice to discuss the actions and the degree of solution independence with the stakeholders that require and implement the operating principles. When considering a combustion process, for example, the solution-independent description of the chemical reactions in a combustion process is the general reduction-oxidation (redox) reaction, in which in two parallel but interconnected processes, a reducing agent is oxidized, and an oxidant or oxidizing agent is reduced. Although this process defines the most reusable operating principle specification, many automotive companies only develop systems in which fuel is the reducing agent and air is the oxidizing agent. Therefore, companies restricting themselves to only these two participants might directly encode this in the operating principle specification to facilitate comprehensibility.

9.3.6 Lesson 6: Modular System Models

As the sixth lesson [GORW23] states, high modularity of the model elements (*cf.* subsection 2.4.2) not only facilitates reuse but also contributes to improving the performance of the modeling tools used. Apart from the methods this thesis presents, [GORW23] therefore suggests using version management, variant management, and library concepts when setting up the models on the tooling side. By this, the principle of modularity and the building block system is effective, which is also a universal principle in developing automotive systems [Mac13].

To achieve this in a project, [GORW23] defines additional best practices that are also valid in the context of this thesis's methods. First, [GORW23] acknowledges distinguishing between solution-neutral and product-specific interfaces and creating corresponding packages on the decomposition levels or system sections to promote system-oriented thinking. Moreover, [GORW23] suggests dividing the model into reusable packages (artifact containers) to increase flexibility.

9.3.7 Lesson 7: Correlation Between Organizational Structure and System Model

As the seventh and final lesson, [GORW23] observed that the organization's structure influences the system model and vice versa, as [Con68] states. Therefore, [GORW23] suggests that systems engineers must consider this interconnection during the system development. To address this challenge, subsubsection 8.1.1 considers this aspect in the definition of the LRA and discusses this impact on the organization structure in the context of several examples.

9.4 Potential Future Research

Although this work successfully addressed the research questions of section 1.1, it also uncovered several gaps in the current state of research on automotive systems engineering and provided methods that could be further developed in future research. Therefore, the following non-exhaustive list aims to envision other researchers with directions for further research:

Deriving Models from Natural Language Requirements As mentioned in subsection 5.5.2, linguistic methods such as NLP [ZAF+21] or AI-based approaches such as LLMs could counter the problems with the extraction from information from natural language requirements by using different parsing algorithms in case of NLP or could provide means for automatic error correction [BKK+23] using LLMs or DSL extraction [BBK+22b, BBK+22a] suggested. Moreover, as presented in [BKK+23], possible extensions could also use requirement templates as a starting point for translating requirements in TCTL, in which an automatic consistency check is possible.

Generating State Machines into the Logical Architecture To extend the concepts for logical architecture generation chapter 8 presents, future research might focus on integrating the state-based behavior model according to subsubsection 3.2.5 into the generation approach. To this end, future works could explore an automated transforma-

tion from activity diagrams to state machines analogous to the transformation [KR18] suggests to define the semantics of activity diagrams and transform activity diagrams to state charts that are closely related to state machines in SysML. An early version of this was a prototypical implementation in [Bla23] for MontiArc [Hab16] in the broader context of this thesis.

Technical Architecture Derivation As this thesis did not address the model creation of the technical architecture and only provides the basis artifacts required in subsubsection 3.2.5, future works could focus on the derivation of these solutions from and for the logical architecture.

Adaptions for SysML v2 Since the SysML v2 standard in [Obj23] introduces new concepts and replaces the SysML standard this thesis refers to [Obj17] future works could aim at transferring the concepts this thesis provides to this standard, making the methods applicable in future system models. Moreover, as the SysML v2 provides a textual representation of all diagrams, the concrete textual syntax could further improve some of the methods as subsection 7.3.4 exemplary discussed for the activity diagram syntax subsection 7.2.4 provides.

AI-Driven System Model Creation Apart from the already suggested utilization of LLMs for processing natural language requirements, artificial intelligence, particularly generative artificial intelligence, provides several other possibilities to increase the efficiency and effectiveness of the methods this thesis suggests. Therefore, future works may focus on improving the efficiency and effectiveness of systems modeling with methods that *e.g.,* layout diagrams, explain models to their readers and copilot system modelers, or ultimately generate new models from a given set of inputs and constraints.

Contribute Additional Insights from other Companies or Domains The results of this thesis cover different industry projects. This number is comparably small compared to the overall impact of the projects in the automotive industry. Therefore, the results in subsection 6.4.3, subsection 6.4.4, subsection 7.3.2, subsection 7.3.3, subsection 8.3.2, or subsection 8.3.3 cannot represent the complete automotive industry. Therefore, future works might additionally aim to reproduce or disprove the results this thesis provides in these and all other affected chapters.

Bibliography

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[ACD+01] Annie I. Antón, Ryan A. Carter, Aldo Dagnino, John H. Dempster, and Devon F. Siege. Deriving goals from a use-case based requirements specification. *Requirements Engineering*, 6:63–73, 2001.

[AK94] C. v. Altrock and B. Krause. Multi-criteria decision making in German automotive industry using fuzzy logic. *Fuzzy sets and systems*, 63(3):375–380, 1994.

[Ale03] Ian Alexander. Misuse cases: Use cases with hostile intent. *IEEE software*, 20(1):58–66, 2003.

[Alu15] Rajeev Alur. *Principles of cyber-physical systems*. MIT press, 2015.

[AM00] Frank Armour and Granville Miller. *Advanced use case modeling: software systems*. Pearson Education, 2000.

[Arl98] Jim Arlow. Use cases, UML visual modelling and the trivialisation of business requirements. *Requirements Engineering*, 3(2):150–152, 1998.

[ARS+14] Manar H. Alalfi, Eric J. Rapos, Andrew Stevenson, Matthew Stephan, Thomas R. Dean, and James R. Cordy. Semi-automatic identification and representation of subsystem variability in simulink models. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 486–490, 2014.

[ASBZ15] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. Automated Checking of Conformance to Requirements Templates Using Natural Language Processing. *IEEE Transactions on Software Engineering*, 41(10):944–968, 2015.

[ASN00] Dave Alford, Peter Sackett, and Geoff Nelder. Mass customisation — an automotive perspective. *International Journal of Production Economics*, 65(1):99–110, 2000.

[AT01] Jauhar Ali and Jiro Tanaka. Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams. *Journal of Computer Science and Information Management*, 2(1), 2001.

[ATZ19] ATZextra. Engineering-Unternehmen I Top 50 Ranking. *ATZextra*, 24:14–17, 2019.

[AVP+19] Tiago Amorim, Andreas Vogelsang, Florian Pudlitz, Peter Gersing, and Jan Philipps. Strategies and best practices for model-based systems engineering adoption in embedded systems industry. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 203–212. IEEE, 2019.

[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&WUCOR@MoDELS 2015*, CEUR Workshop Proceedings, pages 19–26. CEUR-WS.org, 2015.

[Bal08] Carliss Young Baldwin. Where do transactions come from? Modularity, transactions, and the boundaries of firms. *Industrial and corporate change*, 17(1):155–195, 2008.

[BBB+18] Jan Steffen Becker, Vincent Bertram, Tom Bienmüller, Udo Brockmeyer, Heiko Dörr, Thomas Peikenkamp, and Tino Teige. Interoperable Toolchain for Requirements-Driven Model-Based Development. In *ERTS 2018*, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, January 2018.

[BBCM06] Laurent Balmelli, David Brown, Murray Cantor, and Michael Mott. Model-Driven Systems Development. *IBM Systems Journal*, 45(3):569–585, 2006.

[BBK+22a] Vincent Bertram, Miriam Boß, Evgeny Kusmenko, Imke Nachmann, Bernhard Rumpe, Danilo Trotta, and Louis Wachtmeister. Neural Language Models and Few-Shot Learning for Systematic Requirements Processing in MDSE. In *International Conference on Software Language Engineering (SLE'22)*, pages 260–265. ACM, December 2022.

[BBK+22b] Vincent Bertram, Miriam Boß, Evgeny Kusmenko, Imke Helene Nachmann, Bernhard Rumpe, Danilo Trotta, and Louis Wachtmeister. Technical Report on Neural Language Models and Few-Shot Learning for Systematic Requirements Processing in MDSE, November 2022.

[BC00] Carliss Young Baldwin and Kim Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[BCR07] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Semantics of UML: Towards a System Model for UML–The State Machine Model. 2007.

[BCSLS05] Stephan Biller, Lap Mui Ann Chan, David Simchi-Levi, and Julie Swann. Dynamic pricing and the direct-to-customer model in the automotive industry. *Electronic Commerce Research*, 5(2):309–334, 2005.

[BDG95] Eric Bonabeau, Jean-Louis Dessalles, and Alain Grumbach. Characterizing emergent phenomena (1): A critical review. *Revue internationale de systémique*, 9(3):327–346, 1995.

[Ben83] Herbert D. Benington. Production of Large Computer Programs. *Annals of the History of Computing*, 5(4):350–361, 1983.

[Ber69] Ludwig von Bertalanffy. *General System Theory: Foundations, Development, Applications.* George Braziller, Inc, revised edition, 1969.

[BG21] Beate Bender and Kilian Gericke, editors. *Pahl/Beitz Konstruktionslehre.* Springer Vieweg Berlin, Heidelberg, 9 edition, 2021.

[BGK+09] Manfred Broy, Mario Gleirscher, Peter Kluge, Wolfgang Krenzer, Stefano Merenda, and Doris Wild. Automotive architecture framework: Towards a holistic and standardised system architecture description. Technical report, 2009.

[BGM+09] Manfred Broy, Mario Gleirscher, Stefano Merenda, Doris Wild, Peter Kluge, and Wolfgang Krenzer. Toward a holistic and standardized automotive architecture description. *Computer*, 42(12):98–101, 2009.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+14] Wolfgang Böhm, Stefan Henkler, Frank Houdek, Andreas Vogelsang, and Thorsten Weyer. Bridging the gap between systems and software engineering by using the SPES modeling framework as a general systems engineering philosophy. *Procedia Computer Science*, 28:187–194, 2014.

[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.

[BHL21] Marie Rajon Bernard, Dale Hall, and Nic Lutsey. Update on electric vehicle uptake in european cities. *International Council on Clean Transportation (ICCT): Hungary, Budapest*, 2021.

[BKK+23] Vincent Bertram, Hendrik Kausch, Evgeny Kusmenko, Haron Nqiri,

Bernhard Rumpe, and Constantin Venhoff. Leveraging Natural Language Processing for a Consistency Checking Toolchain of Automotive Requirements. In Kurt Schneider, Fabiano Dalpiaz, and Jennifer Horkoff, editors, *IEEE 31st International Conference on Requirements Engineering (RE)*, pages 212–222. ACM/IEEE, September 2023.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)*, 149:437–461, March 2019.

[Bla23] Markus Blanke. A Scenario Template Modeling Language for Use Case Driven Generation of System Architectures. Bachelor Thesis, RWTH Aachen University, Software Engineering Group, 2023.

[BLH+13] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabò, Sandra Torchiaro, Sara Tucci, et al. EAST-ADL: An architecture description language for automotive software-intensive systems. In *Embedded Computing Systems: Applications, Optimization, and Advanced Design*, pages 456–470. IGI Global, 2013.

[BMA+23] Boris Böhlen, Oliver Meyer, Bassam Alrifaee, Julius Beerwerth, Alexandru Kampmann, Stefan Kowalewski, Marco Konersmann, Bernhard Rumpe, and Felix Steinfurth. Software-Defined Vehicle–Herausforderungen in der Diagnose dienstorientierter Fahrzeugarchitekturen. In *Tagungsband-Diagnose in mechatronischen Fahrzeugsystemen XVI: Software-Defined Vehicle, SOVD, Maschinelles Lernen und KI, Standardisierung, HU und ADAS*. TUDpress, 2023.

[BMM09] Romain Beaume, Remi Maniak, and Christophe Midler. Crossing innovation and product projects management: A comparative analysis in the automotive industry. *International Journal of Project Management*, 27(2):166–174, 2009. European Academy of Management (EURAM 2008) Conference.

[Boe84] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75, 1984.

[BR01] Peter Braun and Martin Rappl. A model-based approach for automotive software development. In *OMER-Object-oriented Modeling of Embedded*

Real-Time Systems, GI-Workshops OMER-1 & OMER-2. Gesellschaft für Informatik e. V., 2001.

[BR23] Manfred Broy and Bernhard Rumpe. Development Use Cases for Semantics-Driven Modeling Languages. *Communications of the ACM*, 66(5):62–71, May 2023.

[Bro01a] Tyson R. Browning. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering management*, 48(3):292–306, 2001.

[Bro01b] Manfred Broy. Refinement of time. *Theoretical Computer Science*, 253(1):3–26, 2001.

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42, New York, NY, United States, 2006. Association for Computing Machinery.

[Bro10] Manfred Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, 2010.

[Bro12] Manfred Broy. System behaviour models with discrete and dense time. *Advances in Real-Time Systems*, pages 3–25, 2012.

[Bro15] Manfred Broy. Rethinking nonfunctional software requirements. *Computer*, 48(5):96–99, 2015.

[Bro18] Manfred Broy. A logical approach to systems engineering artifacts: Semantic relationships and dependencies beyond traceability–from requirements to functional and architectural views. *Softw. Syst. Model.*, 17(2):365–393, may 2018.

[BS01] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer-Verlag, 2001.

[BS03] Kurt Bittner and Ian Spence. *Use case modeling.* Addison-Wesley Professional, 2003.

[BV17] Martin Beckmann and Andreas Vogelsang. What is a good textual representation of activity diagrams in requirements documents? In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pages 56–63. IEEE, 2017.

[BVR17] Martin Beckmann, Andreas Vogelsang, and Christian Reuter. A case study on a specification approach using activity diagrams in requirements documents. In *2017 IEEE 25th International Requirements Engineering*

Conference (RE), pages 253–262. IEEE, 2017.

[BW22] Caroline Behle and Kim Woods. Ranking 2022. *ATZextra*, 27:24–27, 2022.

[CAFC+16] Samarjit Chakraborty, Mohammad Abdullah Al Faruque, Wanli Chang, Dip Goswami, Marilyn Wolf, and Qi Zhu. Automotive cyber–physical systems: A tutorial introduction. *IEEE Design & Test*, 33(4):92–108, 2016.

[Can03] Murray Cantor. *Rational Unified Process for Systems Engineering, RUP SE Version 2.0.* IBM Corporation, May 2003. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/aug03/f_rupse_mc.pdf, Accessed 2022-03-25.

[CB16] Lyra J. Colfer and Carliss Y. Baldwin. The mirroring hypothesis: theory, evidence, and exceptions. *Industrial and Corporate Change*, 25(5):709–738, 2016.

[CBK+13] Raman Chitkara, Werner Ballhaus, Bernd Kliem, Stan Berings, and Boris Weiss. Spotlight on automotive: PwC semiconductor report. *Technical report, PwC Technology Institute*, 2013.

[CH11] Ashraf Ferdouse Chowdhury and Mohammad Nazmul Huda. Comparison between adaptive software development and feature driven development. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 363–367. IEEE, 2011.

[Che99] Peter Checkland. Systems thinking. *Rethinking management information systems*, pages 45–56, 1999.

[Cho57] Noam Chomsky. *Syntactic Structures.* De Gruyter Mouton, Berlin, Boston, 1957.

[Cho20] K. R. Chowdhary. *Natural Language Processing*, pages 603–649. Springer India, New Delhi, 2020.

[Cho21] Joel Choi. A Model-Driven Development Model for Component Structures and Part List. Bachelor Thesis, RWTH Aachen University, Software Engineering Group, 2021.

[CHQW15] Thorsten Cziharz, Peter Hruschka, Stefan Queins, and Thorsten Weyer. *Handbook of Requirements Modeling IREB Standard.* 09 2015.

[CJHL18] Huifeng Chen, Jian-min Jiang, Zhong Hong, and Ling Lin. Decomposition of uml activity diagrams. *Software: Practice and Experience*, 48(1):105–122, 2018.

[CKS11] Mary Beth Chrissis, Mike Konrad, and Sandra Shrum. *CMMI for development: guidelines for process integration and product improvement.* Pearson Education, 2011.

[CM93] M.A. Chappell and P.W. McLaughlin. Approach of modeling continuous turbine engine operation from startup to shutdown. *Journal of Propulsion and Power*, 9(3):466–471, 1993.

[CN02] Paul Clements and Linda Northrop. *Software product lines.* Addison-Wesley Boston, 2002.

[Coc98] Alistair Cockburn. Basic use case template. *Humans and Technology, Technical Report*, 96:28, 1998.

[Coc01] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley Longman Publishing Co., Inc., 75 Arlington Street, Suite 300 Boston, MA, United States, 2001.

[Con68] Melvin E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[CS59] Noam Chomsky and Marcel P. Schützenberger. The algebraic theory of context-free languages. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 118–161. Elsevier, 1959.

[CTE+22] Kelly X. Campo, Thomas Teper, Casey E. Eaton, Anna M. Shipman, Garima Bhatia, and Bryan Mesmer. Model-based systems engineering: Evaluating perceived value, metrics, and evidence through literature. *Systems Engineering*, 2022.

[DBF01] Giovani Da Silveira, Denis Borenstein, and Flávio S. Fogliatto. Mass customization: Literature review and research directions. *International Journal of Production Economics*, 72(1):1–13, 2001.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[Dep07] Department of Defense. DoD Architecture Framework, Version 1.5. `https://dodcio.defense.gov/Portals/0/Documents/DODAF/DoDAF_Volume_I.pdf`, 04 2007. Accessed: 2023-01-02.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on*

Software Engineering and Advanced Applications (SEAA'18), pages 146–153, August 2018.

[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. *Journal on Software: Practice and Experience*, 49(2):301–328, February 2019.

[DGS+14] Yanja Dajsuren, Christine M. Gerpheide, Alexander Serebrenik, Anton Wijs, Bogdan Vasilescu, and Mark G.J. van den Brand. Formalizing correspondence rules for automotive architecture views. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, page 129–138, New York, NY, USA, 2014. Association for Computing Machinery.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal methods in system design*, 19:45–80, 2001.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DIN14] DIN 72552-2. Terminal markings for vehicles. Technical report, DIN Deutsches Institut für Normung e. V., 07 2014.

[DJK+19] Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, and Andreas Wortmann. Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext. In Jeff Gray, Matti Rossi, Jonathan Sprinkle, and Juha-Pekka Tolvanen, editors, *International Workshop on Domain-Specific Modeling (DSM'19)*, pages 40–49. ACM, October 2019.

[DJR+19] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, Louis Wachtmeister, and Andreas Wortmann. Model-Driven Systems Engineering for Virtual Product Design. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MPM4CPS*, pages 430–435. IEEE, September 2019.

[DJR+22] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, Manuel Wimmer, and Andreas Wortmann. A cross-

domain systematic mapping study on software engineering for Digital Twins. *Journal of Systems and Software (JSS)*, 193, November 2022.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DLM10] Dov Dori, Chen Linchevski, and Roke Manor. OPCAT–An Object-Process CASE Tool for OPM-Based Conceptual Modelling. In *1st International Conference on Modelling and Management of Engineering Processes*, pages 1–30. University of Cambridge Cambridge, UK, 2010.

[Dor02] Dov Dori. *Object-Process Methodology: A Holistic Systems Paradigm.* Springer Berlin Heidelberg, 2002.

[Dor16] Dov Dori. *Model-Based Systems Engineering with OPM and SysML.* Springer Publishing Company, Incorporated, 1st edition, 2016.

[Dou16] Bruce Powel Douglass. *Agile systems engineering.* Morgan Kaufmann, 2016.

[DRBS03] Dov Dori, Iris Reinhartz-Berger, and Arnon Sturm. Developing complex systems with object-process methodology using OPCAT. In *Conceptual Modeling-ER 2003: 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003. Proceedings 22*, pages 570–572. Springer, 2003.

[DRY01] Terry D. Day, Sydney G. Roberts, and Allen R. York. SIMON: A new vehicle simulation model for vehicle design and safety research. *SAE paper*, (2001-01):0503, 2001.

[DS17] Joseph D'Ambrosio and Grant Soremekun. Systems engineering challenges and MBSE opportunities for automotive system design. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2075–2080. IEEE, 2017.

[DW15] Wolfgang Dröschel and Manuela Wiemers. *Das V-Modell 97: der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz.* Walter de Gruyter GmbH & Co KG, 2015.

[EDG+06] Huáscar Espinoza, Hubert Dubois, Sébastien Gérard, Julio Medina, Dorina C. Petriu, and Murray Woodside. Annotating UML Models with Non-functional Properties for Quantitative Analysis. In Jean-Michel

Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 79–90, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[EF17] Christof Ebert and John Favaro. Automotive software. *IEEE Software*, 34(03):33–39, 2017.

[EFM+05] Christoph Eichner, Hans Fleischhack, Roland Meyer, Ulrik Schrimpf, and Christian Stehno. Compositional semantics for UML 2.0 sequence diagrams using Petri Nets. In *International SDL Forum*, pages 133–148. Springer, 2005.

[EHKP15] Ulf Eliasson, Rogardt Heldal, Eric Knauss, and Patrizio Pelliccione. The need of complementing plan-driven requirements engineering with emerging communication: Experiences from volvo car group. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 372–381. IEEE, 2015.

[EHPL15] Ulf Eliasson, Rogardt Heldal, Patrizio Pelliccione, and Jonn Lantz. Architecting in the automotive domain: Descriptive vs prescriptive architecture. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 115–118. IEEE, 2015.

[Esh06] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38, 2006.

[Est07] Jeff A. Estefan. Survey of model-based systems engineering (MBSE) methodologies. *Incose MBSE Focus Group*, 25(8):1–12, 2007.

[EVF16] Jonas Eckhardt, Andreas Vogelsang, and Daniel Méndez Fernández. Are" non-functional" requirements really non-functional? an investigation of non-functional requirements in practice. In *Proceedings of the 38th International Conference on Software Engineering*, pages 832–842, 2016.

[EVFM16] Jonas Eckhardt, Andreas Vogelsang, Henning Femmer, and Philipp Mager. Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 46–55, 2016.

[fEUNEC04] United Nations Economic Commission for Europe (United Nations Economic and Social Council). Regulation No 78 of the Economic Commission for Europe of the United Nations (UN/ECE) — Uniform provisions concerning the approval of vehicles of category L with regard to braking. *Official Journal of the European Union*, L 95:67–88, 03 2004.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report,

Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[FGT96] Kirk Dean Fiedler, Varun Grover, and James T.C. Teng. An empirically derived taxonomy of information technology structure and its relationship to organizational structure. *Journal of Management Information Systems*, 13(1):9–34, 1996.

[Fix05] Sebastian Fixson. Product architecture assessment: a tool to link product, process, and supply chain design decisions. *Journal of operations management*, 23(3-4):345–369, 2005.

[FLS65] Richard P. Feynman, Robert B. Leighton, and Matthew Sands. The feynman lectures on physics; vol. i. *American Journal of Physics*, 33(9):750–752, 1965.

[FMK+11] Stefan Farfeleder, Thomas Moser, Andreas Krall, Tor Stålhane, Herbert Zojer, and Christian Panis. DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 271–274. IEEE, 2011.

[FMS14a] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[FMS14b] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML, Third Edition: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2014.

[Fow04] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[FPQ+10] Xavier Franch, Cristina Palomares, Carme Quer, Samuel Renault, and François Lazzer. A metamodel for software requirement patterns. In *Requirements Engineering: Foundation for Software Quality*, 2010.

[FQR+13] Xavier Franch, Carme Quer, Samuel Renault, Cindy Guerlain, and Cristina Palomares. *Constructing and Using Software Requirement Patterns*, pages 95–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[FS03] Ronald S. Farrell and Timothy W. Simpson. Product platform design to improve commonality in custom products. *Journal of Intelligent Manufacturing*, 14:541–556, 2003.

[GGTP13] Hugo G. Chalé Góngora, Thierry Gaudré, and Sara Tucci-Piergiovanni. Towards an Architectural Design Framework for Automotive Systems Development. In Marc Aiguier, Yves Caseau, Daniel Krob, and Antoine Rauzy, editors, *Complex Systems Design & Management*, pages 241–258, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GH00] J. Grundy and J. Hosking. High-level static and dynamic visualisation of software architectures. In *Proceeding 2000 IEEE International Symposium on Visual Languages*, pages 5–12, 2000.

[GHK+08a] 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.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHZM17] Kristina Gruber, Jakob Huemer, Armin Zimmermann, and Ralph Maschotta. Integrated description of functional and non-functional requirements for automotive systems design using SysML. In *2017 7th IEEE International Conference on System Engineering and Technology (ICSET)*, pages 27–31. IEEE, 2017.

[GJ10] Rickard Garvare and Peter Johansson. Management for sustainability–a stakeholder theory. *Total quality management*, 21(7):737–744, 2010.

[GJK+21] Malin Gandor, Nicolas Jäckel, Lorenz Käser, Alexander Schlie, Ingo Stierand, Axel Terfloth, Steffen Toborg, Louis Wachtmeister, and Anna Wißdorf. Architectures for Dynamically Coupled Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 95–124. Springer, January 2021.

[GKGZ16] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings 5*, pages 185–200. Springer, 2016.

[GKM+25] Christian Granrath, Christopher Kugler, Judith Michael, Bernhard Rumpe, and Louis Wachtmeister. Generating Logical Architectures from SysML Behavior Models. *INCOSE Systems Engineering*, 28(6):762–778, November 2025.

[GKS+21] Christian Granrath, Christopher Kugler, Sebastian Silberg, Max Meyer, Philipp Orth, Johannes Richenhagen, and Jakob Andert. Feature-driven systems engineering procedure for standardized product-line development. *Systems Engineering*, 24, 08 2021.

[Gli07] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26, 2007.

[Gog96] Joseph A. Goguen. Parameterized programming and software architecture. In *Proceedings of Fourth IEEE International Conference on Software Reuse*, pages 2–10, 1996.

[GORW23] Christian Granrath, Philipp Orth, Bernhard Rumpe, and Louis Wachtmeister. Optimierungspotentiale zur effizienten Systemmodellierung im Kontext der Automobilen Systementwicklung. In Walter Koch, Daria Wilke, Stefan Dreiseitel, and Rüdiger Kaffenberger, editors, *Tag des Systems Engineering 2023*, pages 307–313. Gesellschaft für Systems Engineering (GfSE) e.V., November 2023.

[Gra22] Christian Granrath. *Feature-getriebene Systementwicklung von Produktlinien mittels Referenzarchitektur für Simulationsmodelle elektrischer Automobilantriebe.* Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, Aachen, 2022. Veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Dissertation, Rheinisch-Westfälische Technische Hochschule Aachen, 2022.

[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.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HASH07] Sami Haddadin, Alin Albu-Schäffer, and Gerd Hirzinger. Safety evaluation of physical human-robot interaction via crash-testing. In *Robotics: Science and systems*, volume 3, pages 217–224. Citeseer, 2007.

[HBG15] Erik Hebisch, Matthias Book, and Volker Gruhn. Scenario-based architecting with architecture trace diagrams. In *2015 IEEE/ACM 5th International Workshop on the Twin Peaks of Requirements and Architecture*, pages 16–19. IEEE, 2015.

[HE88] Vladimir Hubka and W. Ernst Eder. *Theory of Technical Systems.* Springer Berlin Heidelberg, 1988.

[HF07] Florian Hölzl and Martin Feilkas. AutoFOCUS 3-A scientific tool pro-

totype for model-based development of component-based, reactive, distributed systems. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pages 317–322. Springer, 2007.

[HGBS15] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. Applying product line Use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 338–347, 2015.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJL+02] Jon G. Hall, Michael Jackson, Robin C. Laney, Bashar Nuseibeh, and Lucia Rapanotti. Relating Software Requirements and Architectures using Problem Frames. In *Proceedings IEEE Joint International Conference on Requirements Engineering*, pages 137–144. IEEE, 2002.

[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.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLMSN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 45–66. Springer, 2015.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software & Systems Modeling*, 7(2):237–252, 2008.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July

2019.

[HMST07] Ryan Hutcheson, Daniel Mcadams, Robert Stone, and Irem Tumer. Function-based systems engineering (FUSE). In *DS 42: Proceedings of ICED 2007, the 16th International Conference on Engineering Design, Paris, France, 28.-31.07. 2007*, pages 715–716, 2007.

[HNZ+23] Gregor Hoepfner, Imke Nachmann, Thilo Zerwas, Joerg K. Berroth, Jens Kohl, Christian Guist, Bernhard Rumpe, and Georg Jacobs. Towards a Holistic and Functional Model-Based Design Method for Mechatronic Cyber-Physical Systems. *Journal of Computing and Information Science in Engineering (JCISE)*, 23(5), March 2023.

[HP00] Frank Houdek and Klaus Pohl. Analyzing requirements engineering processes: a case study. In *Proceedings 11th International Workshop on Database and Expert Systems Applications*, pages 983–987. IEEE, 2000.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37(10):64–72, October 2004.

[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.

[HS19] Tomas Huldt and Ivan Stenius. State-of-practice survey of model-based systems engineering. *Systems engineering*, 22(2):134–145, 2019.

[HU14] Peter E. Harland and Zakir Uddin. Effects of product platform development: fostering lean product development and production. *International Journal of Product Development 18*, 19(5-6):259–285, 2014.

[HW91] Herman M.H. Hegge and Johan C. Wortmann. Generic bill-of-material: a new product model. *International Journal of Production Economics*, 23(1):117–128, 1991.

[IBM10] IBM. *Model-Based Systems Engineering with Rational Rhapsody and Rational Harmony for Systems Engineering – Deskbook 3.1.2*, February 2010. `https://www-01.ibm.com/support/docview.wss?uid=swg27023356`, Accessed 2022-03-25.

[IBM20] IBM. *IBM Engineering Systems Design Rhapsody*, 04 2020.

[IEE94] IEEE 830-1993. IEEE Recommended Practice for Software Requirements Specifications. Technical report, ISO/IEC/IEEE, 1994.

[Imh20] Christiane Imhof. Ranking 2020 Neue Auf- und Absteiger. *ATZextra*,

25:10–13, 2020.

[Imh21] Christiane Imhof. Ranking 2021. *ATZextra*, 26:12–15, 2021.

[INC07] INCOSE. *Systems Engineering Vision 2020*, September 2007.

[IRBM05] Michel D. Ingham, Robert D. Rasmussen, Matthew B. Bennett, and Alex C. Moncada. Engineering Complex Embedded Systems with State Analysis and the Mission Data System. *Journal of Aerospace Computing Information and Communication*, 2, December 2005.

[ISO11a] ISO 26262:2011. ISO 26262: Road Vehicles : Functional Safety. Standard, ISO, 2011.

[ISO11b] ISO/IEC/IEEE 42010:2011. ISO/IEC/IEEE 42010: systems and software engineering — architecture description. Standard, ISO/IEC/IEEE, 2011.

[ISO11c] ISO/IEC/IEEE 42010:2011(E). Systems and software engineering — Architecture description. Technical report, ISO/IEC/IEEE, 2011.

[ISO15] ISO/IEC/IEEE 15288:2015(E). Systems and software engineering — System life cycle processes. Standard, ISO/IEC/IEEE, 2015.

[ISO17a] ISO/IEC/IEEE 12207:2017(E). Systems and software engineering — Software life cycle processes. Technical report, ISO/IEC/IEEE, 2017.

[ISO17b] ISO/IEC/IEEE 24765:2017(E). Systems and software engineering — Vocabulary. Standard, International Organization for Standardization and International Electrotechnical Commission and Institute of Electrical and Electronics Engineers, 2017.

[ISO21] ISO/SAE 21434:2021. ISOO/SAE 21434: Road Vehicles : Cybersecurity engineering. Standard, ISO, 2021.

[Jac93] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach.* Pearson Education India, 1993.

[JCA04] F. Robert Jacobs, Richard B. Chase, and Nicholas J. Aquilano. Operations management for competitive advantage. *Boston: Mc-Graw Hill*, 64:70, 2004.

[JGS+20] Nicolas Jäckel, Christian Granrath, Robert Schaller, Malte Grotenrath, Maximilian Fischer, Philipp Orth, and Jakob Andert. Integration of VTOL air-taxis into an existing infrastructure with the use of the Model-Based System Engineering (MBSE) concept CUBE. 10 2020.

[JGW+21] Nicolas Jäckel, Christian Granrath, Louis Wachtmeister, Abdulsamed Karaduman, Bernhard Rumpe, and Jakob Lukas Andert. Feature-Driven Specification of VTOL Air-Taxis with the Use of the Model-Based Systems Engineering (MBSE) Methodology CUBE. In *77th Annual Vertical*

Flight Society Forum and Technology Display (FORUM 77)), pages 2776–2784. Curran Associates, Inc., May 2021.

[JR12] Gerhard Jäger and James Rogers. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970, 2012.

[Kat21] Christina Katzer. A Domain Specific Language for Requirements Management in an Industrial Context. Bachelor Thesis, RWTH Aachen University, Software Engineering Group, 2021.

[Kau21] Oliver Kautz. *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Aachener Informatik-Berichte, Software Engineering, Band 46. Shaker Verlag, April 2021.

[KC05] Sascha Konrad and Betty HC Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381, 2005.

[KFD23] Tamás Kolossváry, Dániel Feszty, and Tibor Dőry. Systems engineering in automotive product development: A guide to initiate organisational transformation. *Journal of Open Innovation: Technology, Market, and Complexity*, 9(4):100160, 2023.

[KG12] Daryl Kulak and Eamonn Guiney. *Use cases: requirements in context.* Addison-Wesley, 2012.

[KK98] Rudolf Koller and Norbert Kastrup. *Prinziplösungen zur konstruktion technischer produkte.* Springer Berlin Heidelberg, 1998.

[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.

[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 *Inter-*

national Conference on Software Engineering: Software Engineering in Practice (ICSE'18), pages 172–180. ACM, June 2018.

[KNS+21] Jörg Christian Kirchhof, Michael Nieke, Ina Schaefer, David Schmalzing, and Michael Schulze. Variant and Product Line Co-Evolution. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 333–351. Springer, January 2021.

[Köb21] Ronja Köberlein. From Functional Behavior Specification to Logical Architecture: A Model-Driven Approach. Bachelor Thesis, RWTH Aachen University, Software Engineering Group, 2021.

[KR18] Oliver Kautz and Bernhard Rumpe. Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In *Proceedings of MODELS 2018. Workshop ME*, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering.* Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRGK18] Stefan Kriebel, Johannes Richenhagen, Christian Granrath, and Christopher Kugler. Systems Engineering with SysML The Path to the Future? *MTZ worldwide*, 79:44–47, 05 2018.

[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.

[KRSvW18] 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 B. Kruchten. The 4+1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

[Kru04] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Professional, 2004.

[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.

[KRW22] Oliver Kautz, Bernhard Rumpe, and Louis Wachtmeister. Semantic Differencing of Use Case Diagrams. *Journal of Object Technology (JOT)*, 21:3:1–14, July 2022.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. Wiley Publishing, 1st edition, 1998.

[KS10] Adam S. Klimovich and V. V. Solov'Ev. Transformation of a Mealy Finite-State Machine into a Moore Finite-State Machine by Splitting Internal States. *J. Comput. Syst. Sci. Int.*, 49(6):900–908, dec 2010.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.

[KWV+16] Haneen Khreis, Karyn M. Warsow, Ersilia Verlinghieri, Alvaro Guzman, Luc Pellecuer, Antonio Ferreira, Ian Jones, Eva Heinen, David Rojas-Rueda, Natalie Mueller, Paul Schepers, Karen Lucas, and Mark Nieuwenhuijsen. The health impacts of traffic-related exposures in urban areas: Understanding real effects, underlying driving forces and co-producing future directions. *Journal of Transport & Health*, 3(3):249–267, 2016.

[LAB+11] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168. IEEE, 2011.

[Lee08] Edward A Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*, pages 363–369. IEEE, 2008.

[Lev09] Anatoly Levenchuk. SysML is the Point of Departure for MBSE, Not the Destination. *INCOSE Insight*, 12(4):54–56, December 2009.

[LFM00] Howard Lykins, Sanford Friedenthal, and Abraham Meilich. Adapting uml for an object oriented systems engineering method (oosem). In *INCOSE International Symposium*, volume 10, pages 490–497. Wiley Online Library, 2000.

[LGTL85] Wen-Shing Lee, Doris L. Grosh, Frank A. Tillman, and Chang H. Lie. Fault Tree Analysis, Methods, and Applications - A Review. *IEEE transactions on reliability*, 34(3):194–203, 1985.

[LHM+14] Feng-Lin Li, Jennifer Horkoff, John Mylopoulos, Renata S. S. Guizzardi, Giancarlo Guizzardi, Alexander Borgida, and Lin Liu. Non-functional requirements as qualities, with a spice of ontology. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 293–302, 2014.

[LMP04] Mich Luisa, Franch Mariangela, and Novi Inverardi Pierluigi. Market

research for requirements analysis using linguistic tools. *Requirements Engineering*, 9:40–56, 2004.

[LS12] David Long and Zane Scott. *A Primer for Model-Based Systems Engineering*. Lulu. com, 2012.

[LSJM96] Henry George Liddell, Robert Scott, Henry Stuart Jones, and Roderick McKenzie. σύστημα. In *A Greek–English Lexicon: Ninth Edition with Revised Supplement*. Calderon Press, 1996.

[LTK19] Grischa Liebel, Matthias Tichy, and Eric Knauss. Use, potential, and showstoppers of models in automotive requirements engineering. *Software & Systems Modeling*, 18:2587–2607, 2019.

[Mac13] John Paul MacDuffie. Modularity-as-property, modularization-as-process, and 'modularity'-as-frame: Lessons from product architecture initiatives in the global automotive industry. *Global strategy journal*, 3(1):8–40, 2013.

[Mal22] Urvesh Arvindbhai Malani. Analysis and Optimization of Software Architectures created using the CUBE Methodology. Master Thesis, RWTH Aachen University, Software Engineering Group, 2022.

[Mar94] James Martin. The pmte paradigm : Exploring the relationship between systems engineering process and tools. *INCOSE International Symposium*, 4:176–183, 08 1994.

[Mar22] Matthias Markthaler. *Modellbasierte Methode für die automatisierte Testfallerstellung in der Automobilindustrie auf der Grundlage eines durchgängigen Systems Engineering Ansatzes*. Aachener Informatik-Berichte, Software Engineering, Band 52. Shaker Verlag, November 2022.

[MBRB10] Cem Mengi, Önder Babur, Holger Rendel, and Christian Berger. Model-driven Configuration of Function Net Families in Automotive Software Engineering. In *Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010)*, pages 49–60, Paris, France, June 2010.

[MDB14] Bhaswati Mondal, Barun Das, and Prasenjit Banerjee. Formal Specification of UML Use Case Diagram–A CASL based Approach. *International Journal of Computer Science and Information Technologies*, 5(3):2713–2717, 2014.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MGWJ20] Max Meyer, Christian Granrath, Louis Wachtmeister, and Nicolas Jäckel. Methods for the Development of Collaborative Embedded Systems in

Automated Vehicles. *ATZelectronics worldwide*, 15(12):58–63, December 2020.

[MHGK03] Martin Mutz, Michaela Huhn, Ursula Goltz, and Carsten Kroemke. Model Based System Development in Automotive. Technical report, SAE Technical Paper, 2003.

[ML97] Marc H. Meyer and Alvin P. Lehnerd. *The power of product platforms*. Simon and Schuster, 1997.

[MLM+12] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2012.

[MMMI16] André Murbach Maidl, Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. Error reporting in Parsing Expression Grammars. *Science of Computer Programming*, 132:129–140, 2016. Selected and extended papers from SBLP 2013.

[MNAR22] Subbarao Mopidevi, Rajanand Patnaik Narasipuram, Sreedhar Reddy Aemalla, and Hemachander Rajan. E-mobility: Impacts and analysis of future transportation electrification market in economic, renewable energy and infrastructure perspective. *International Journal of Powertrains*, 11(2-3):264–284, 2022.

[MNS96] Hausi Muller, Ronald J. Norman, and Jacob Slonim, editors. *Computer Aided Software Engineering*. Springer US, 1996.

[Moo56] Edward F. Moore. *Gedanken-Experiments on Sequential Machines*, pages 129–154. Princeton University Press, Princeton, 1956.

[MP19] Azad M. Madni and Shatad Purohit. Economic Analysis of Model-Based Systems Engineering. *Systems*, 7(1), 2019.

[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.

[MSG+22] Max-Arno Meyer, Sebastian Silberg, Christian Granrath, Christopher Kugler, Louis Wachtmeister, Bernhard Rumpe, Sebastien Christiaens, and Jakob Lukas Andert. Scenario- and Model-Based Systems Engineering Procedure for the SOTIF-Compliant Design of Automated Driving Functions. In *2022 IEEE Intelligent Vehicles Symposium (IV'22)*, pages 1599–1604. IEEE, June 2022.

[MSL15] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. ReSA: An ontology-based requirement specification language tailored to automotive systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

[MSL16] Nesredin Mahmud, Cristina Seceleanu, and Oscar Ljungkrantz. ReSA Tool: Structured requirements specification and SAT-based consistency-checking. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 1737–1746, 2016.

[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.

[Muf99] Moreno Muffatto. Introducing a platform strategy in product development. *International Journal of Production Economics*, 60-61:145–153, 1999.

[MWHN09] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy Approach to Requirements Syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322, 2009.

[MWJ+22] Junda Ma, Guoxin Wang, Lu Jinzhi, Hans Vangheluwe, Dimitris Kiritsis, and Yan Yan. Systematic Literature Review of MBSE Tool-Chains. *Applied Sciences*, 12:3431, 03 2022.

[MXX06] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic Test Case Generation for UML Activity Diagrams. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, page 2–8, New York, NY, USA, 2006. Association for Computing Machinery.

[NLF+15] Claus Ballegaard Nielsen, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. *ACM Comput. Surv.*, 48(2), sep 2015.

[NP95] Kai Nagel and Maya Paczuski. Emergent traffic jams. *Physical Review E*, 51(4):2909, 1995.

[NPR07] NPR 7123.1. NASA Systems Engineering Processes and Requirements, April 2007.

[NPS06] Madhav Prasad Nepal, Moonseo Park, and Bosik Son. Effects of schedule pressure on construction performance. *Journal of Construction Engineering and Management*, 132(2):182–188, 2006.

[NSC+07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In

29th International Conference on Software Engineering (ICSE'07), pages 54–64. IEEE, 2007.

[Nus01] Bashar Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, 2001.

[NW17] Thomas Noll and Louis Wachtmeister. Analysing cryptographically-masked information flows in mils-aadl specifications. In *MILS*, 2017.

[Obj17] Object Management Group. *OMG Unified Modeling Language (OMG UML)*, version 2.5.1 edition, December 2017.

[Obj19] Object Management Group. *OMG Systems Modeling Language*, version 1.6 edition, March 2019.

[Obj21] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models*, version 1.5 edition, June 2021.

[Obj23] Object Management Group. *OMG Systems Modeling Language*, 2.0 beta edition, July 2023.

[Obj25] Object Management Group (OMG). OMG SysML, 2025.

[OM04] Hannes Omasreiter and Eduard Metzker. A context-driven use case creation process for specifying automotive driver assistance systems. In *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.*, pages 334–339. IEEE, 2004.

[ON15] Jens Otto and Oliver Niggemann. Automatic parameterization of automation software for plug-and-produce. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[Oqu04] Flavio Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[ORC+15] Marcos Oliveira, Leila Ribeiro, Érika Cota, Lucio Mauro Duarte, Ingrid Nunes, and Filipe Reis. Use Case Analysis Based on Formal Methods: An Empirical Study. In Mihai Codescu, Răzvan Diaconescu, and Ionuț Țuțu, editors, *Recent Trends in Algebraic Development Techniques*, pages 110–130, Cham, 2015. Springer International Publishing.

[PBDH16] Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hönninger. *Advanced Model-Based Engineering of Embedded Systems*. Springer International Publishing, Cham, 2016.

[PBFG07] Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Engineering Design: A Systematic Approach*. Solid mechanics and its

applications. Springer London, 2007.

[PBL05] Klaus Pohl, Günter Böckle, and Frank Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques.* 01 2005.

[PDKvK02] Barbara Paech, Allen H. Dutoit, Daniel Kerkow, and Antje von Knethen. Functional requirements, non-functional requirements, and architecture should not be separated-a position paper. *REFSQ, Essen, Germany (September 2002)*, 2002.

[Pen97] Jean-Michel Penalva. *La modelisation par les systemes en situation complexes.* PhD thesis, Paris 11, 1997.

[PG82] Geoffrey K. Pullum and Gerald Gazdar. Natural languages and context-free languages. *Linguistics and Philosophy*, 4:471–504, 1982.

[PHAB12] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-Based Engineering of Embedded Systems.* Springer Berlin, Heidelberg, 2012.

[PKH+16] Patrizio Pelliccione, Eric Knauss, Rogardt Heldal, Magnus Ågren, Piergiuseppe Mallozzi, Anders Alminger, and Daniel Borgentun. A proposal for an automotive architecture framework for volvo cars. In *2016 Workshop on Automotive Systems/Software Architectures (WASA)*, pages 18–21, 2016.

[PKH+17] Patrizio Pelliccione, Eric Knauss, Rogardt Heldal, S. Magnus Ågren, Piergiuseppe Mallozzi, Anders Alminger, and Daniel Borgentun. Automotive architecture framework: The experience of volvo cars. *Journal of Systems Architecture*, 77:83–100, 2017.

[Pla03] Max Planck. *Treatise on Thermodynamics.* Longmans, Green, and Co., 1903.

[PMHP12] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive behavioral requirements expressed in a specification pattern system: a case study at BOSCH. *Requirements Engineering*, 17:19–33, 2012.

[PQF+12] Cristina Palomares, Carme Quer, Xavier Franch, Cindy Guerlain, and Samuel Renault. A catalogue of non-technical requirement patterns. In *2012 Second IEEE International Workshop on Requirements Patterns (RePa)*, 2012.

[PR21] Klaus Pohl and Chris Rupp. *Basiswissen requirements engineering: Aus- und Weiterbildung nach IREB-Standard zum certified professional for requirements engineering foundation level.* 2021.

[PTWN18] Kate Palmer, James E. Tate, Zia Wadud, and John Nellthorp. Total cost of ownership and market share for hybrid and electric vehicles in the UK, US and Japan. *Applied Energy*, 209:108–119, 2018.

[PTY+24] Ruoqing Peng, Justin Hayse Chiwing G. Tang, Xiong Yang, Meng Meng, Jie Zhang, and Chengxiang Zhuge. Investigating the factors influencing the electric vehicle market share: A comparative study of the European Union and United States. *Applied Energy*, 355:122327, 2024.

[RFB12] Ana Luísa Ramos, José Vasconcelos Ferreira, and Jaume Barceló. Model-Based Systems Engineering: An Emerging Approach for Modern Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(1):101–111, 2012.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems.* Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.

[RMBFQ09] Samuel Renault, Oscar Mendez-Bonilla, Xavier Franch, and Carme Quer. A pattern-based method for building requirements documents in call-for-tender processes. *International Journal of Computer Science and Applications*, 6, 2009.

[RR02] Claudio Riva and Jordi Vidal Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 47–55. IEEE, 2002.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRW13] 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.

[RTP+14] Neha Rungta, Oksana Tkachuk, Suzette Person, Jason Biatek, Michael W. Whalen, Joseph Castle, and Karen Gundy-Burlet. Helping System Engineers Bridge the Peaks. In *Proceedings of the 4th International Workshop on Twin Peaks of Requirements and Architecture*, pages 9–13, 2014.

[Rum94] James E. Rumbaugh. Getting started: Using use cases to capture requirements. *Journal Object Oriented Programming*, 7:8–12, 23, 1994.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum06] Bernhard Rumpe. Agile Test-based Modeling. In *Proceedings of the 2006 International Conference on Software Engineering Research & Practice. SERP'2006*. CSREA Press, USA, June 2006.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[SB91] Simon Saunders and Harvey R. Brown. *The Philosophy of Vacuum*. Oxford University Press, 1991.

[SB06] Sabnam Sengupta and Swapan Bhattacharya. Formalization of UML use case diagram-a Z notation based approach. In *2006 International Conference on Computing Informatics*, pages 1–6, 2006.

[Sch00] Melissa Schilling. Toward a general modular systems theory and its application to interfirm product modularity. *Academy of management review*, 25(2):312–334, 2000.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sch19] Christoph Schulze. *Agile Software-Produktlinienentwicklung im Kontext heterogener Projektlandschaften*. Aachener Informatik-Berichte, Software Engineering, Band 40. Shaker Verlag, May 2019.

[She17] Garrett Shea. NASA Systems Engineering Handbook Revision 2, 2017. NASA Systems Engineering Handbook Revision 2 | NASA.

[Shi85] Stuart M. Shieber. Evidence against the context-freeness of natural language. In *The Formal complexity of natural language*, pages 320–334. Springer, 1985.

[Sim91] Herbert A. Simon. *The Architecture of Complexity*, pages 457–476. Springer US, Boston, MA, 1991.

[SKNC17] Thiago Sousa, Luciano Kelvin, Constantino Neto, and Carlos Carvalho. A Formal Semantics for Use Case Diagram Via Event-B. *Journal of Software*, 12:189–200, 04 2017.

[SL03] Wuwei Shen and Shaoying Liu. Formalization, testing and execution of a use case diagram. In *International Conference on Formal Engineering*

Methods, pages 68–85. Springer, 2003.

[SO01a] Guttorm Sindre and Andreas L Opdahl. Capturing security requirements through misuse cases. *NIK 2001, Norsk Informatikkonferanse 2001, http://www. nik. no/2001*, 74, 2001.

[SO01b] Guttorm Sindre and Andreas L Opdahl. Templates for misuse case description. In *Proceedings of the 7th International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ'2001), Switzerland*, 2001.

[SO05] Guttorm Sindre and Andreas L Opdahl. Eliciting security requirements with misuse cases. *Requirements engineering*, 10:34–44, 2005.

[Spa21] Sparx Systems. *Enterprise Architect*, 02 2021.

[SPI15] The SPICE User Group: Automotive SPICE Process Assessment/Reference Model V3.0. *Automotive SPICE Process Assessment/Reference Model V3.0*, July 2015.

[SRW98] Zahed Siddique, David W. Rosen, and Nanxin Wang. On the Applicability of Product Variety Design Concepts to Automotive Platform Commonality. volume 3: 10th International Conference on Design Theory and Methodology of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 09 1998. V003T03A007.

[SSV98] Ian Sommerville, Peter Sawyer, and Stephen Viller. Viewpoints for requirements elicitation: a practical approach. In *Proceedings of IEEE International Symposium on Requirements Engineering: RE'98*, pages 74–81. IEEE, 1998.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2):1–42, 2009.

[Sta73] Herbert Stachowiak. *Allgemeine modelltheorie*. Springer, 1973.

[Sta03] Diomidis H. Stamatis. *Failure Mode and Effect Analysis*. Quality Press, 2003.

[Stö05] Harald Störrle. Semantics and verification of data flow in UML 2.0 activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35–52, 2005.

[Stö11] Harald Störrle. On the impact of layout quality to understanding UML diagrams. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 135–142, 2011.

[SZ16] Joerg Schäuffele and Thomas Zurawka. *Automotive Software Engineering: Principles, Processes, Methods, and Tools*. SAE International, 2016.

[TBH16] Tino Teige, Tom Bienmüller, and Hans Jürgen Holberg. Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In *MBMV 2016 : 19. GI/ITG/GMM-Workshop "Methoden und Beschreibungsprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, pages 118–161. Deutsche Nationalbibliothek, 2016.

[TG14] Saurabh Tiwari and Atul Gupta. Does increasing formalism in the use case template help? In *Proceedings of the 7th India Software Engineering Conference*, pages 1–10, 2014.

[THLL18] Shmuel Tyszberowicz, Robert Heinrich, Bo Liu, and Zhiming Liu. Identifying Microservices Using Functional Decomposition. In Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 50–65, Cham, 2018. Springer International Publishing.

[TMW+24] Till Temmen, Max-Arno Meyer, Louis Wachtmeister, Mohammadsadegh Zabihi, Christopher Kugler, Sebastien Christiaens, Bernhard Rumpe, and Jakob Andert. Application of Model-Based Systems Engineering Methods in Virtual Homologation Procedures for Automated Driving Functions. In Alexander Heintzel, editor, *Automatisiertes Fahren 2024*, pages 1–14, Wiesbaden, 2024. Springer Fachmedien Wiesbaden.

[Tom84] Masaru Tomita. LR parsers for natural languages. In *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics*, pages 354–357, 1984.

[TVRN01] Qiang Tu, Mark A. Vonderembse, and T.S. Ragu-Nathan. The impact of time-based manufacturing practices on mass customization and value to customer. *Journal of Operations management*, 19(2):201–217, 2001.

[TVRNRN04] Qiang Tu, Mark A. Vonderembse, T. Sv. Ragu-Nathan, and Bhanu Ragu-Nathan. Measuring modularity-based manufacturing practices and their impact on mass customization capability: a customer-driven perspective. *Decision Sciences*, 35(2):147–168, 2004.

[TW05] Kevin Turner and Geoff Williams. Modelling complexity in the automotive industry supply chain. *Journal of Manufacturing Technology Management*, 2005.

[Ulr95] Karl Ulrich. The role of product architecture in the manufacturing firm. *Research Policy*, 24(3):419–440, 1995.

[vB50] Ludwig von Bertalanffy. The Theory of Open Systems in Physics and Biology. *Science*, 111(2872):23–29, 1950.

[VDI21] VDI/VDE 2206:2021. Development of mechatronic and cyber-physical systems. Standard, VDI/VDE, 2021.

[VF13] Andreas Vogelsang and Steffen Fuhrmann. Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 267–272, 2013.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 249–262, 2001.

[vR17] Michiel R. van Ratingen. The Euro NCAP Safety Rating. In *Karosseriebautage Hamburg 2017: 15. ATZ-Fachtagung*, pages 11–20. Springer, 2017.

[vW20] 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. Shaker Verlag, March 2020.

[Wac17] Louis Wachtmeister. Analysing Cryptographically-Masked Information Flows in MILS-AADL Specifications. Bachelor Thesis, RWTH Aachen University, Software Modeling and Verification Group, 2017.

[Wac19] Louis Wachtmeister. Model-Driven Systems Engineering for Vitrual Product Design. Master Thesis, RWTH Aachen University, Software Engineering Group, 4 2019.

[WBJ08] Daniel Work, Alexandre Bayen, and Quinn Jacobson. Automotive cyber physical systems in the context of human mobility. In *National Workshop on high-confidence automotive cyber-physical systems*, pages 3–4, 2008.

[Web09] Julian Weber. *Automotive development processes: Processes for successful customer oriented vehicle development*. Springer Science & Business Media, 2009.

[Wei84] Carl Friedrich von Weizsäcker. *The Unity of Nature*, pages 239–254. Springer Netherlands, Dordrecht, 1984.

[Whi06] Jon Whittle. Specifying precise use cases with use case charts. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 290–301, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[WLDM20] Qun Wu, Kun Liao, Xiaodong Deng, and Erika Marsillac. Achieving

automotive suppliers' mass customization through modularity: Vital antecedents and the valuable role and responsibility of information sharing. *Journal of Manufacturing Technology Management*, 31(2):306–329, 2020.

[WLRW15] Tim Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker. *Architecture Patterns and Principles*, chapter 7, pages 49–74. John Wiley & Sons, Ltd, 2015.

[WMGT11] Ernest Wozniak, Chokri Mraidha, Sebastien Gerard, and Francois Terrier. A guidance framework for the generation of implementation models in the automotive domain. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 468–476. IEEE, 2011.

[WMO+16] Stephan Weyer, Torben Meyer, Moritz Ohmer, Dominic Gorecky, and Detlef Zühlke. Future modeling and simulation of CPS-based factories: an example from the automotive industry. *Ifac-Papersonline*, 49(31):97–102, 2016.

[WMS16] Swee Yin Wong, Edwin Mit, and Jonathan Sidi. Integration of use case formal template using mapping rules. In *2016 Third International Conference on Information Retrieval and Knowledge Management (CAMP)*, pages 25–31. IEEE, 2016.

[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.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021.

[WRF+15] David D. Walden, Garry J. Roedler, Kevin Forsberg, R. Douglas Hamelin, and Thomas M. Shortell, editors. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. Wiley, Hoboken, NJ, 4 edition, 2015.

[WV16] Jonas Winkler and Andreas Vogelsang. Automatic Classification of Requirements Based on Convolutional Neural Networks. In *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, pages 39–45, 2016.

[Wym93] Wayne A. Wymore. *Model-Based Systems Engineering*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1993.

[YCC15] Rongjie Yan, Chih-Hong Cheng, and Yesheng Chai. Formal consistency checking over specifications in natural languages. In *2015 Design, Au-*

tomation & Test in Europe Conference & Exhibition (DATE), pages 1677–1682, 2015.

[YG08] Om Prakash Yadav and Parveen S. Goel. Customer satisfaction driven quality improvement target planning for product development in automotive industry. *International Journal of Production Economics*, 113(2):997–1011, 2008.

[YZW+21] Hao Yang, Chao Zhan, Haomin Wu, Chao Cui, and Renyu Hu. Research on modeling of aircraft-level high-lift system architecture based on sysml. In *Journal of Physics: Conference Series*, volume 1827, page 012096. IOP Publishing, 2021.

[Zab23] Mohammadsadegh Zabihi. Automatische Generierung von natürlichsprachlichen Anforderungen aus Modellen des modellbasierten Systems Engineerings im Automobil. Master Thesis, RWTH Aachen University, Lehr- und Forschungsgebiet Mechatronik in mobilen Antrieben, 2023.

[ZAF+21] Liping Zhao, Waad Alhoshan, Alessio Ferrari, Keletso J. Letsholo, Muideen A. Ajagbe, Erol-Valeriu Chioasca, and Riza T. Batista-Navarro. Natural language processing for requirements engineering: A systematic mapping study. *ACM Comput. Surv.*, 54(3), apr 2021.

[ZHCL18] Wang Zhe, Jerome Hugues, Jean-Charles Chaudemar, and Thierry LeSergent. An integrated approach to model based engineering with sysml, aadl and face. In *SAE Technical Paper*. SAE International, 10 2018.

[ZVC06] Qingyu Zhang, Mark A. Vonderembse, and Mei Cao. Achieving flexible manufacturing competence: the roles of advanced manufacturing technology and operations improvement practices. *International Journal of Operations & Production Management*, 26(6):580–599, 2006.

Acronyms

AADL Architecture Analysis & Design Language.

AD SysML Activity Diagram.

ADAS Advanced Driver-Assistance System.

Alpha Vehicle development project for Asian OEM.

AST Abstract Syntax Tree.

BDD SysML Block Definition Diagram.

Beta Powertrain development project for Asian OEM.

bLRA behavioral Logical Reference Architecture.

BOM Bil of Materials.

CAD Computer-Aided Design.

CAE Computer-Aided Engineering.

CASE Computer-Aided Software Engineering.

CPS Cyber-Physical System.

CrESt Collaborative Embedded Systems.

CSAL Algebraic Specification Language.

CUBE Compositional Unified system-Based Engineering.

Delta Function development for autonomous driving funded by the EU.

DSL Domain Specific Language.

DVP&R Design Verification Plan and Report.

Epsilon Control system development in the context of a powertrain for a German OEM.

EU European Union.

FDD Feature-Driven Development.

FuSE Function-based Systems Engineering.

Gamma ADAS function development project for autonomous driving on the highway.

GB Guobiao (chin. for national standard).

IBD SysML Internal Block Diagram.

INCOSE International Council on Systems Engineering.

JPL NASA Jet Propulsion Laboratory.

LLM Large Language Model.

LRA Logical Reference Architecture.

MBE Model-Based Engineering.

MBSE Model-Based Systems Engineering.

MCG MontiCore Grammar.

MDA Model-Driven Architecture.

MDD Model-Driven Development.

MDE Model-Driven Engineering.

MDSE Model-Driven Systems Engineering.

MIB Modularer Infotainment Baukasten (German For Modular Infotainment Toolkit).

MILS Multiple Independent Levels of Security.

MQB Modularer Querbaukasten (German for Modular Transverse Toolkit).

NASA National Aeronautics and Space Administration.

NLP Natural Language Processing.

OEM Original Equipment Manufacturer.

OMG Object Management Group.

OOSEM INCOSE Object-Oriented Systems Engineering Method.

OPL Object-Process Language.

OPM Object-Process Methodology.

PMTE Process, Methods, Tools and Environment.

RUP Rational Unified Process.

RUP-SE Rational Unified Process for Systems Engineering.

SA JPL State Analysis.

SE Systems Engineering.

sLRA structural Logical Reference Architecture.

SMArDT German term for "Spezifkationsmethode für Anforderungen, Design und Test" (Engl. specification method for requirements, design, and test).

SoI System of Interest.

SoS System of Systems.

SOTIF Safety Of The Intended Functionality.

SPECTRE Specification guideline for Requirements Engineering.

SPES Software Platform Embedded Systems.

SPESXT SPES Extended.

SPL Software Product Line.

STM SysML State Machine Diagram.

SysML Systems Modeling Language.

TCTL Timed Computation Tree Logic.

UCD SysML Use Case Diagram.

UML Unified Modeling Language.

UNECE United Nations Economic Commission for Europe.

US United States.

VTOL Vertical Take-Off and Landing.

Zeta Vehicle development project for German OEM.

Appendix A

Diagram and Listing Tags

Tag	Description
AD	Activity Diagram
BDD	Block Definition Diagram
CD	Class Diagram
IBD	Internal Block Diagram
MCG	MontiCore Grammar
STM	State Machine Diagram
SD	Sequence Diagram
UCD	Use Case Diagram

Table A.1: Explanation of the used tags in listings and figures.

Stereotype	Description
«EXT»	External elements
«GEN»	Generated elements
«HC»	Handcoded elements
«RTE»	Run-time Environment elements
«RT-IF»	Run-time Infrastructure elements

Table A.2: Explanation of the used stereotypes in listings and tags.

Appendix B

SPECTRE Grammars

In the following, this chapter presents the grammars chapter 5 uses for the processing of legacy requirements in natural language and the formulation of new structured requirements. As discusses in this chapter, the grammars are the result of a student thesis project [Kat21] and therefore not exclusively developed by the author of this thesis. For a better application of this grammar in the considered projects, however, most of the grammars were reworked by the author to achieve a better and more general applicability.

B.1 Action

MCG

```
1 /**
2  * This grammar defines actions used in sentences.
3  */
4 component grammar Action extends fev.spectre.WordBasis, fev.
     spectre.Substantive {
5   // Rule for defining an action
6   SpectreAction
7     = SpectreVerbalPhrase? processVerbs:SpectreSimpleName+;
8
9   // Interface for defining a verbal phrase in an action
10  interface SpectreVerbalPhrase = actor:SpectreSubstantive;
11
12  // Rule for a "Provide" phrase in a verbal action, with an actor
        and the phrase "With the ability to"
13  SpectreProvidePhrase implements SpectreVerbalPhrase
14    = SpectreProvide actor:SpectreSubstantive
         SpectreWithTheAbilityTo;
15
16  // Rule for an "Enable" phrase in a verbal action, with an actor
        and the phrase "To"
17  SpectreEnablePhrase implements SpectreVerbalPhrase
18    = SpectreEnable actor:SpectreSubstantive SpectreTo;
```

```
19 }
```

Listing B.1: The Action grammar.

B.2 Condition

MCG

```
1 /**
2  * This grammar defines conditions used in sentences.
3  */
4 component grammar Condition extends fev.spectre.Substantive, fev.
     spectre.LogicalExpression {
5   // Interface declaration for conditions
6   interface SpectreCondition;
7
8   // Interface for conditions involving logical expressions
9   interface SpectreLogicalExpressionCondition extends
       SpectreCondition;
10
11  // Rule for an "If" condition with an optional substantive and a
        logical expression
12  SpectreIfCondition implements SpectreLogicalExpressionCondition
13    = SpectreIf SpectreSubstantive? SpectreLogicalExpression;
14
15  // Rule for a "Below" condition with an optional substantive and
        a combination of names, optional "Of," and a unit or
       variable
16  SpectreBelowCondition implements
       SpectreLogicalExpressionCondition
17    = SpectreBelow SpectreSubstantive? SpectreName* SpectreOf? (
         SpectreUnit | SpectreVariable);
18
19  // Rule for an "Above" condition with an optional substantive
       and a combination of names, optional "Of," and a unit or
       variable
20  SpectreAboveCondition implements
       SpectreLogicalExpressionCondition
21    = SpectreAbove SpectreSubstantive? SpectreName* SpectreOf? (
         SpectreUnit | SpectreVariable);
22
23  // Rule for a "Between" condition with an optional substantive
       and a combination of names, followed by two units or
       variables separated by "And"
24  SpectreBetweenCondition implements
```

```
        SpectreLogicalExpressionCondition
25      = SpectreBetween SpectreSubstantive? SpectreName* (unit1:
          SpectreUnit | variable1: SpectreVariable) SpectreAnd (unit2
          : SpectreUnit | variable2: SpectreVariable);
26
27    // Interface for conditions involving events
28    interface SpectreEventCondition extends SpectreCondition;
29
30    // Rule for an "As Soon As" condition with an optional
        substantive and an event
31    SpectreAsSoonAsCondition implements SpectreEventCondition
32      = SpectreAsSoonAs SpectreSubstantive? SpectreEvent;
33
34    // Interface for conditions involving time periods
35    interface SpectreTimePeriodCondition extends SpectreCondition;
36
37    // Rule for an "As Long As" condition with an optional
        substantive and a time period
38    SpectreAsLongAsCondition implements SpectreTimePeriodCondition
39      = SpectreAsLongAs SpectreSubstantive? SpectreTimePeriod;
40
41    // Rule for a "During" condition with an optional substantive
        and a combination of names
42    SpectreDuringCondition implements SpectreTimePeriodCondition
43      = SpectreDuring SpectreSubstantive? SpectreName*;
44 }
```

Listing B.2: The Condition grammar.

B.3 Constraint

MCG

```
1 /**
2  * This grammar defines constraints used in sentences.
3  */
4 component grammar Constraint extends fev.spectre.Unit, fev.spectre
     .Variable {
5    // Interface declaration for constraints
6    interface SpectreConstraint;
7
8    // Rule for a constraint with no specific word, consisting of
        either a unit or a variable
9    SpectreNoWordConstraint implements SpectreConstraint = (
        SpectreUnit | SpectreVariable);
```

```
10
11   // Rule for a constraint using the word "Of," followed by either
         a unit or a variable
12   SpectreOfConstraint implements SpectreConstraint = SpectreOf (
        SpectreUnit | SpectreVariable);
13
14   // Rule for a constraint using the word "To," followed by either
         a unit or a variable
15   SpectreToConstraint implements SpectreConstraint = SpectreTo (
        SpectreUnit | SpectreVariable);
16
17   // Rule for a constraint with a comparison operator, followed by
         either a unit or a variable
18   SpectreComparisonOperatorConstraint implements SpectreConstraint
         = SpectreComparisonOperator (SpectreUnit | SpectreVariable);
19
20   // Rule for a constraint using the phrase "With a Minimum of,"
        followed by either a unit or a variable
21   SpectreWithAMinimumOfConstraint implements SpectreConstraint =
        SpectreWithAMinimumOf (SpectreUnit | SpectreVariable);
22
23   // Rule for a constraint using the phrase "With a Maximum of,"
        followed by either a unit or a variable
24   SpectreWithAMaximumOfConstraint implements SpectreConstraint =
        SpectreWithAMaximumOf (SpectreUnit | SpectreVariable);
25 }
```

Listing B.3: The Constraint grammar.

B.4 LogicalExpression

MCG

```
1 /**
2  * This grammar defines logical expressions used in conditions.
3  */
4 component grammar LogicalExpression extends fev.spectre.Unit, fev.
     spectre.Variable, fev.spectre.Substantive {
5   // Interface declarations
6   interface SpectreLogicalExpression;
7   interface SpectreEvent;
8   interface SpectreTimePeriod;
9
10  // Interface combining logical expression, event, and time
       period
```

```
11   interface SpectreLogicalStatement extends
         SpectreLogicalExpression, SpectreEvent, SpectreTimePeriod;
12
13   // Below expression logical statement
14   SpectreBelowExpression implements SpectreLogicalStatement
15     = SpectreName+ SpectreBelow (SpectreUnit | SpectreVariable);
16
17   // Greater expression logical statement
18   SpectreGreaterExpression implements SpectreLogicalStatement
19     = SpectreName+ (SpectreAbove | SpectreGreater) (SpectreUnit |
          SpectreVariable);
20
21   // Comparison operator expression logical statement
22   SpectreComparisonOperatorExpression implements
         SpectreLogicalStatement
23     = SpectreName+ SpectreComparisonOperator (SpectreUnit |
          SpectreVariable);
24
25   // Between expression logical statement
26   SpectreBetweenExpression implements SpectreLogicalStatement
27     = SpectreName+ SpectreBetween (unit1: SpectreUnit | variable1:
           SpectreVariable) SpectreAnd (unit2: SpectreUnit |
          variable2: SpectreVariable);
28
29   // No logical expression logical statement
30   SpectreNoLogicalExpression implements SpectreLogicalStatement
31     = SpectreSubstantives+;
32 }
```

Listing B.4: The Logical Expression grammar.

B.5 SPECTREName

```
1 /**
2  * This grammar defines the SpectreName which is used for
       arbitrary words in requirements.
3  */
4 component grammar Name extends de.monticore.types.MCBasicTypes, de
     .monticore.literals.MCCommonLiterals {
5   // Definition of a token for standard symbols used in Spectre
        names
6   token SpectreStandardSymbol = (':' | '-' | '/' | '\\');
7
```

MCG

```
8   // Definition of a simple name in Spectre
9   SpectreSimpleName = name:Name;
10
11  // Definition of a SpectreName, which can include a combination
       of names and numbers separated by standard symbols
12  SpectreName = (name:Name) (Name SpectreStandardSymbol?)* (number
       :NatLiteral SpectreStandardSymbol?)*;
13 }
```

Listing B.5: The SPECTREName grammar.

B.6 Sentence

```
1 /**
2  * This grammar defines sentences used in texts
3  */
4 grammar Sentence extends fev.spectre.Condition, fev.spectre.Action
     , fev.spectre.Substantive, fev.spectre.Constraint {
5   // Interface declaration
6   interface SpectreSentence = SpectreMainSentence;
7
8   // Sentence starting with a condition
9   SpectreConditionFirstSentence implements SpectreSentence
10    = SpectreCondition "," SpectreMainSentence;
11
12  // Sentence ending with a condition
13  SpectreConditionLastSentence implements SpectreSentence
14    = SpectreMainSentence SpectreCondition;
15
16  // Sentence without any condition
17  SpectreNoConditionSentence implements SpectreSentence
18    = SpectreMainSentence;
19
20  // Main structure of a sentence
21  SpectreMainSentence
22    = subject:SpectreSubstantives SpectreModalVerb SpectreNot?
         SpectreAction SpectreSubstantives? SpectreConstraint?;
23 }
```

MCG

Listing B.6: The Sentence grammar.

B.7 Substantive

MCG

```
/**
 * This grammar defines substantives used at multiple places in
     the DSL.
 */
component grammar Substantive extends fev.spectre.WordBasis {

  // Rule for combining multiple substantives with a boolean
      operator
  SpectreSubstantives
    = (SpectreOneSubstantive | SpectreTwoSubstantives |
        SpectreMultipleSubstantives);

  // Rule for multiple substantives separated by commas and joined
      with a boolean operator
  SpectreMultipleSubstantives
    = (multiple:SpectreSubstantive ",")+ second:SpectreSubstantive
        SpectreBooleanOperator first:SpectreSubstantive;

  // Rule for two substantives joined with a boolean operator
  SpectreTwoSubstantives
    = second:SpectreSubstantive SpectreBooleanOperator first:
        SpectreSubstantive;

  // Rule for a single substantive
  SpectreOneSubstantive
    = first:SpectreSubstantive;

  // Symbol definition for a substantive
  symbol SpectreSubstantive
    = SpectreQuantifier? SpectreArticle? SpectreName+ (SpectreOf
        SpectreSubstantive)? SpectreMarketingName? SpectreComment?;

  // Rule for a comment enclosed in parentheses
  SpectreComment
    = "(" SpectreName+ ")";

  // Symbol definition for a marketing name enclosed in square
      brackets
  symbol SpectreMarketingName
    = "[" SpectreName+ "]";

  // AST rule for retrieving the name of a substantive
```

```
35   astrule SpectreSubstantive =
36     method public String getName () {
37       String res="";
38       for(fev.spectre.name._ast.ASTSpectreName name:this.
            spectreNames) {
39         res = res+" "+ name;
40       }
41       return res;
42     };
43
44   // AST rule for retrieving the name of a marketing name
45   astrule SpectreMarketingName =
46     method public String getName () {
47       String res="";
48       for(fev.spectre.name._ast.ASTSpectreName name:this.
            spectreNames) {
49         res = res+" "+ name;
50       }
51       return res;
52     };
53 }
```

Listing B.7: The Substantive grammar.

B.8 Unit

```
1 /**
2  * This grammar defines units used in constraints and conditions.
3  */
4 component grammar Unit extends de.monticore.SIUnitLiterals {
5   // Rule for defining a unit in the DSL
6   SpectreUnit
7     = SIUnitLiteral;
8 }
```

MCG

Listing B.8: The Unit grammar.

B.9 Variable

```
1 /**
```

MCG

```
 * This grammar defines variables used in constraints and
     conditions.
 */
component grammar Variable extends fev.spectre.WordBasis {
  // Symbol definition for a variable in the DSL
  symbol SpectreVariable
    = SpectreQuantifier? SpectreArticle? "{" SpectreName+"}";

  // AST rule for retrieving the name of a variable
  astrule SpectreVariable =
    method public String getName () {
      String res="";
      for(fev.spectre.name._ast.ASTSpectreName name:this.
          spectreNames) {
        res = res+" "+ name;
      }
      return res;
    };
}
```

Listing B.9: The Variable grammar.

B.10 WordBasis

MCG

```
/**
 * This grammar defines multiple terminals used in multiple places
     in the DSL.
 */
component grammar WordBasis extends fev.spectre.Name {
  // Interface for defining articles
  interface SpectreArticle;
  SpectreNo implements SpectreArticle
    = ("No"| "no");
  SpectreThe implements SpectreArticle
    = ("The" | "the");
  SpectreA implements SpectreArticle
    = ("A" | "a");
  SpectreAn implements SpectreArticle
    = ("An" |"an");

  // Interface for defining boolean operators
  interface SpectreBooleanOperator;
  SpectreAnd implements SpectreBooleanOperator
```

```
19      = ("And" | "and" | "AND");
20    SpectreOr implements SpectreBooleanOperator
21      = ("Or"| "or" | "OR");
22
23    // Interface for defining comparison operators
24    interface SpectreComparisonOperator;
25    SpectreSmallerEqual implements SpectreComparisonOperator
26      = ((("Lower"|"Smaller") | ("lower"|"smaller")) key("than") key
           ("or") key("equal") key("to") | "<=");
27    SpectreSmallerThan implements SpectreComparisonOperator
28      = ((("Lower"|"Smaller")  | ("lower"|"smaller")) key("than") |
           "<");
29    SpectreGreaterEqual implements SpectreComparisonOperator
30      = (("Greater" | "greater") key("than") key("or") key("equal")
           key("to") | ">=");
31    SpectreGreaterThan implements SpectreComparisonOperator
32      = (("Greater" | "greater")  key("than") | ">");
33
34    // Interface for defining modal verbs
35    interface SpectreModalVerb;
36    SpectreCan implements SpectreModalVerb
37      = ("Can" | "can");
38    SpectreCould implements SpectreModalVerb
39      = ("Could" | "could");
40    SpectreMay implements SpectreModalVerb
41      = ("May"| "may");
42    SpectreMight implements SpectreModalVerb
43      = ("Might"| "might");
44    SpectreMust implements SpectreModalVerb
45      = ("Must" | "must" | "Have to" | "have to" | "Has to" | "has
           to");
46    SpectreShall implements SpectreModalVerb
47      = ("Shall" | "shall");
48    SpectreShould implements SpectreModalVerb
49      = ("Should"| "should");
50    SpectreWill implements SpectreModalVerb
51      = ("Will"| "will");
52    SpectreWould implements SpectreModalVerb
53      = ("Would"| "would");
54
55    // Interface for defining quantifiers
56    interface SpectreQuantifier;
57    SpectreAll implements SpectreQuantifier
58      = (key("All") | key("all"));
59    SpectreExactlyOne implements SpectreQuantifier
60      = ((key("Exactly")| key("exactly")) key("one"));
```

```
61  SpectreOne implements SpectreQuantifier
62    = (key("One") | key("one"));
63  SpectreThereExistsOne implements SpectreQuantifier
64    = (key("There") | key("there")) (key("is")|key("exists"));
65  SpectreThereExistsExactlyOne implements SpectreQuantifier
66    = ((key("There") | key("there")) (key("is")|key("exists")) key
        ("exactly") key("one"));
67
68  // Terminal for "Not"
69  SpectreNot
70    = (key("Not") | key("not"));
71
72  // Terminals for condition indicators
73  SpectreIf
74    = ("If" | "if" | "When" | "when");
75  SpectreAsSoonAs
76    = ("As" | "as") "soon" "as";
77  SpectreAsLongAs
78    = ("As" | "as") "long" "as";
79  SpectreDuring
80    = ("During"| "during");
81
82  // Terminals for logical expressions
83  SpectreAbove
84    = ("Above" | "above");
85  SpectreBelow
86    = ("Below" | "below");
87  SpectreGreater
88    = ("Greater" | "greater");
89  SpectreBetween
90    = ("Between" | "between");
91
92  // Terminals for constraints
93  SpectreWithAMinimumOf
94    = (key("with") | key("With")) "a" key("minimum") "of";
95  SpectreWithAMaximumOf
96    = (key("with") | key("With")) "a" key("maximum") "of";
97  SpectreAtLeast
98    = key("at") key("least");
99
100 // Terminal for "Of"
101 SpectreOf
102   = ("Of" | "of");
103
104 // Terminals for actions
105 SpectreProvide
```

```
106     = (key("Provide") | key("provide"));
107   SpectreEnable
108       = (key("Enable") | key("enable"));
109   SpectreWithTheAbilityTo
110     = (key("With") | key("with")) key("the") key("ability") key("
          to");
111   SpectreTo
112       = (key("To") | key("to"));
113 }
```

Listing B.10: The WordBasis grammar.

Appendix C

Use Case Specification Template, Language and Examples

C.1 Use Case Specification Template

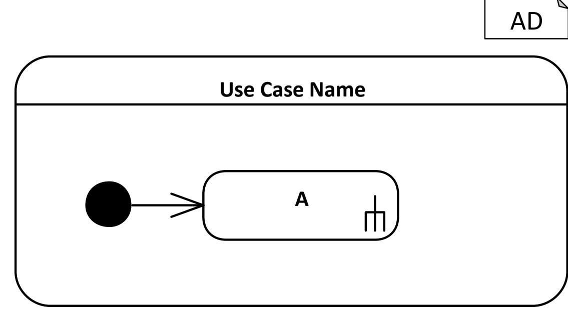
Table C.1: A template for use case behavior specification elicitation.

<table>
<tr><td>UC Name</td><td colspan="8">Scenario Name</td></tr>
<tr><td>Trigger</td><td colspan="8">Event that triggers the Use Case</td></tr>
<tr><td>Pre-condition</td><td colspan="8">Pre-condition that shall be fulfilled before the execution of the Use Case</td></tr>
<tr><td rowspan="4">Desired Behavior</td><td colspan="8">Description of the desired behavior of the system in one sentence.</td></tr>
<tr><td>#</td><td>Actor</td><td>Action</td><td>Input</td><td>Output</td><td>Condition</td><td>Execution</td><td>Next</td></tr>
<tr><td>1</td><td>Actor executing the action</td><td>Action this step executes</td><td>name: Type</td><td>name: Type</td><td>Condition</td><td>Sequential/ Parallel/ Alternative/ Exception</td><td>2</td></tr>
<tr><td>2</td><td>...</td><td>...</td><td>...</td><td>...</td><td>...</td><td>...</td><td>End</td></tr>
<tr><td>Post-conditions</td><td colspan="8">Post-condition that shall be fulfilled after the successful execution of the Use Case</td></tr>
</table>

C.2 Transformation Rules to Transform Use Case Specifications to Activity Diagrams and Back

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
	1		A					

(a) The desired behavior pattern for the initial action.

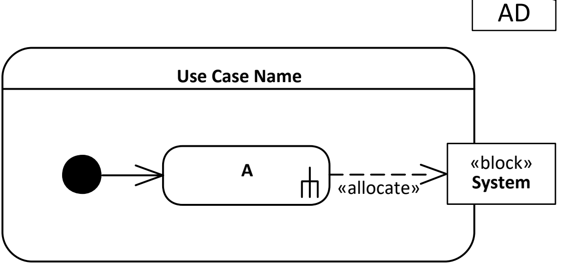


(b) The activity diagram pattern for the initial action.

Figure C.1: The transformation rule for the initial action (Initial Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
		System	A					

(a) The desired behavior pattern for action execution.

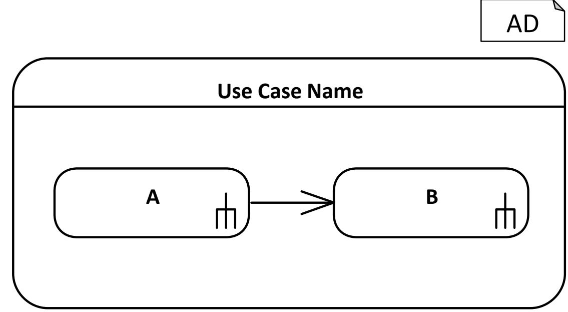


(b) The activity diagram pattern for action execution.

Figure C.2: The transformation rule for action execution (Execution Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
	m		A					n
	n		B					

(a) The desired behavior pattern for sequential actions.

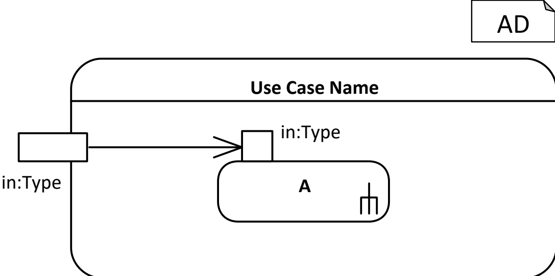


(b) The activity diagram pattern for sequential actions.

Figure C.3: The transformation rule for sequential actions (Sequential Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			A	in:Type				

(a) The desired behavior pattern for external inputs.

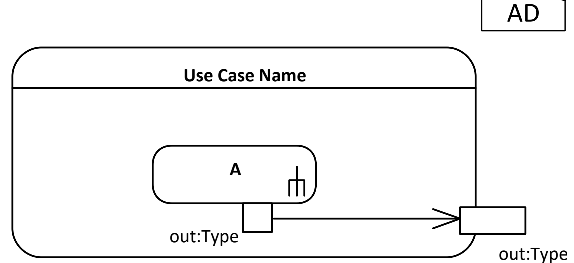


(b) The activity diagram pattern for external inputs.

Figure C.4: The transformation rule for external inputs (External Input Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			A		out:Type			

(a) The desired behavior pattern for external outputs.

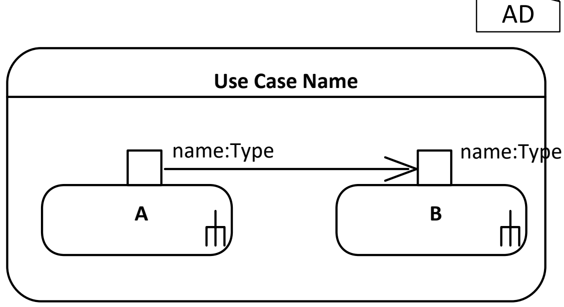


(b) The activity diagram pattern for external outputs.

Figure C.5: The transformation rule for external outputs (External Output Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			A		name:Type			
			B	name:Type				

(a) The desired behavior pattern for internal flows.

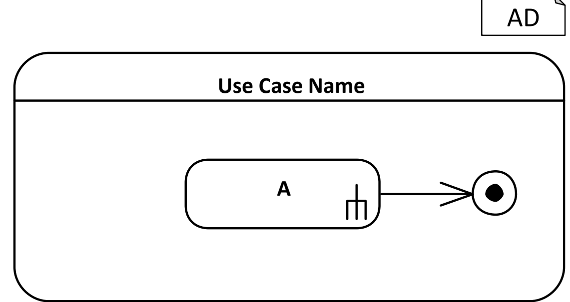


(b) The activity diagram pattern for internal flows.

Figure C.6: The transformation rule for internal flows (Internal Flow Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			A					Final

(a) The desired behavior pattern for the final actions.

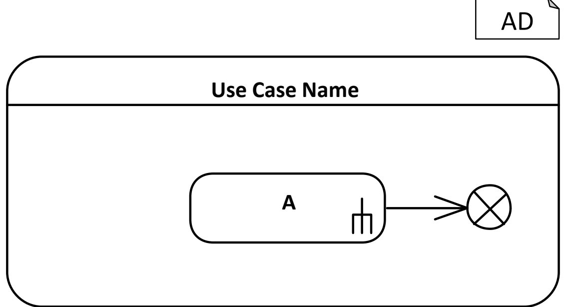


(b) The activity diagram pattern for the final actions.

Figure C.7: The transformation rule for the final actions (Final Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			A					FlowFinal

(a) The desired behavior pattern for the flow final actions.

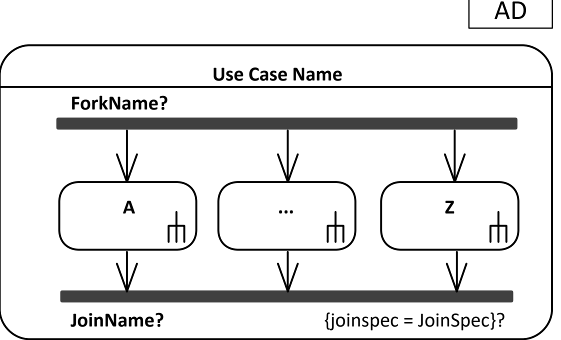


(b) The activity diagram pattern for the flow final actions.

Figure C.8: The transformation rule for the final actions (Flow Final Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
	m?		ForkName?				Parallel	-
	m a		A				Parallel	
	m ...		...				Parallel	
	m z		Z				Parallel	
	m?		JoinName?			JoinSpec?	Parallel	

(a) The desired behavior pattern for parallel actions.

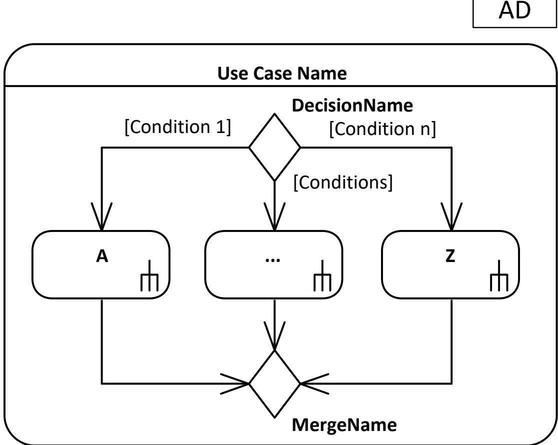


(b) The activity diagram pattern for parallel actions.

Figure C.9: The transformation rule for parallel actions (Parallel Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Out	Condition	Execution	Next
	m?		DecisionName?				Alternative	
	m a		A			Condition1	Alternative	
	m ...		...			Conditions	Alternative	
	m z		Z			Condition3	Alternative	
	m?		MergeName?				Alternative	

(a) The desired behavior pattern for alternative actions.

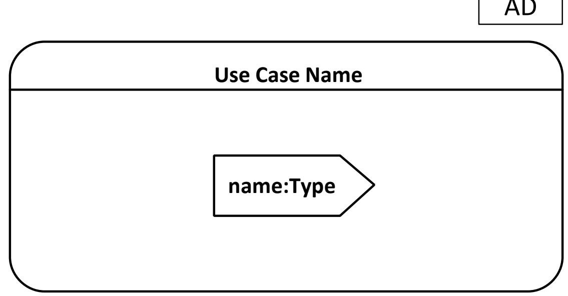


(b) The activity diagram pattern for alternative actions.

Figure C.10: The transformation rule for alternative actions (Alternative Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			SendSignal		name: Type			

(a) The desired behavior pattern for send signal actions.

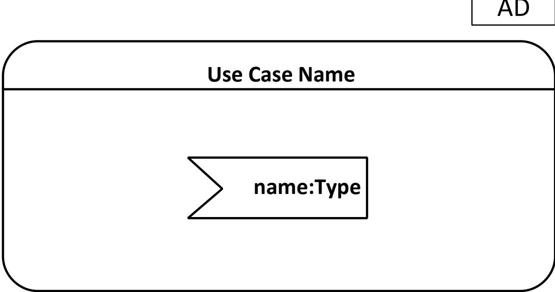


(b) The activity diagram pattern for send signal actions.

Figure C.11: The transformation rule for send signal actions (Send Signal Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			AcceptEvent	name: Type				

(a) The desired behavior pattern for accept event actions.

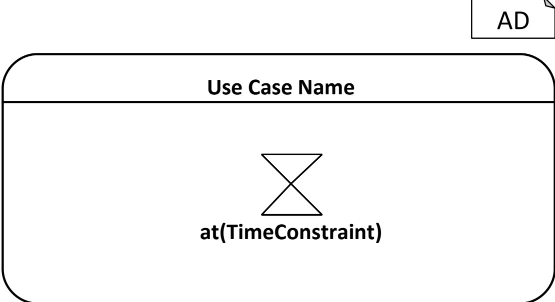


(b) The activity diagram pattern for accept event actions.

Figure C.12: The transformation rule for accept event actions (Accept Event Rule).

UC	Use Case Name ...						
⋮	...						
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
			Accept Event Timer			Time Constraint		

(a) The desired behavior pattern for accept event timer actions.

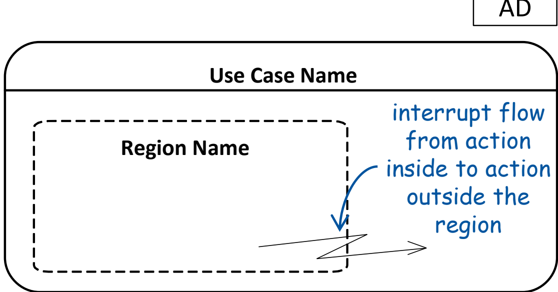


(b) The activity diagram pattern for accept event timer actions.

Figure C.13: The transformation rule for accept event timer actions (Accept Event Timer Rule).

UC	Use Case Name ...							
⋮	...							
⋮	#	Actor	Action	Input	Output	Condition	Execution	Next
	m		RegionName					
	m a		A					
	m ...		...				Exception	
	m z		Z					
	m		RegionName					

(a) The desired behavior pattern for interruptable action regions.



(b) The activity diagram pattern for interruptable action regions.

Figure C.14: The transformation rule for interruptable action regions (Interruptable Action Rule).

C.3 Use Case Specification Language

C.3.1 ActivityDiagram

MCG

```
1 grammar ActivityDiagram extends UCDSys {
2
3   ActivityDiagramArtifact = ActivityDiagram;
4
5   scope ActivityDiagram =
6     "activitydiagram" "{"
7       ADElement*
8     "}";
9
10  symbol scope NamedActivityDiagram extends ActivityDiagram =
11    "activitydiagram" Name "{"
12      ADElement*
13    "}";
14
15  /* ============= Activity Diagram Elements ============= */
16  interface ADElement;
17
18  symbol ADSwimLane implements ADElement =
19    "allocate" Name@UCDSystem "{"
20      ADNode*
21    "}";
22
23  /* ============= Nodes ============= */
24  interface symbol scope ADNode extends ADElement;
25
26  ADActionNode implements ADNode =
27    Name
28      (("{"
29        ADActionElement*
30      "}")
31    | ";");
32
33  ADSendSignalActionNode implements ADNode =
34    "sendSignal" MCType Name;
35
36  ADAcceptEventActionNode implements ADNode =
37    "recieveSignal" MCType Name;
38
39  interface ADControlNode extends ADNode;
40
```

```
41  ADDecisionNode implements ADControlNode =
42    "decision" Name
43      (("{"
44        ADPin*
45      "}")
46    | ";");
47
48  ADMergeNode implements ADControlNode =
49    "merge" Name ";";
50
51  ADForkNode implements ADControlNode =
52    "fork" Name ";";
53
54  ADJoinNode implements ADControlNode =
55    "join" Name ";";
56
57
58  /* ============= Control Flow ============= */
59  ADEdge implements ADElement =
60    ADEdgeStart "->" ADEdgeEnd ";";
61
62  ADEdgeStart =
63    ["initial"] | Name@ADNode | ADMergeEdges | ADJoinEdges;
64
65  ADEdgeEnd =
66    ["final"] | ["flowfinal"] | Name@ADNode | ADDecisionEdges |
        ADForkEdges;
67
68  ADMergeEdges =
69    (Name@ADNode || "|")+;
70
71  ADJoinEdges =
72    (Name@ADNode || "&")+;
73
74  ADDecisionEdges =
75    (ADConditionalEdge || "|")+;
76
77  ADConditionalEdge =
78    "[" Expression "]" (Name@ADNode | ["final"] | ["flowfinal"]);
79
80  ADForkEdges =
81    (Name@ADNode || "&")+;
82
83  ADInterruptableRegion implements ADElement =
84    "interruptable" Name@UCDSystem "{"
85      (ADNode | ADInterruptFlow)+;
```

```
86     "}";
87
88   ADInterruptFlow implements ADElement =
89     ADEdgeStart "/->" ADEdgeEnd ";";
90
91
92   /* ============= Object Flow ============= */
93   ADObjectEdge implements ADElement =
94     start:ADActionPin "->" end:(ADActionPin || ",")+ ";";
95
96   ADActionPin =
97     node:Name@ADNode "." pin:Name@ADPin;
98
99   /* ============= Action Elements ============= */
100  interface ADActionElement;
101
102  ADActionDescription implements ADActionElement =
103    String ";";
104
105  ADHierarchicalAD implements ADActionElement =
106    ActivityDiagram | ("activitydiagram" Name@NamedActivityDiagram
         ";");
107
108  /* ============= Object Pins ============= */
109  symbol ADPin implements ADActionElement =
110    (in:[key("in")] | out:[key("out")]) MCType Name ("="
        defaultValue:PinValue)? ";";
111
112  PinValue = Expression;
113
114 }
```

Listing C.1: ActivityDiagram grammar based on [Bla23].

C.3.2 UCST

MCG

```
1 grammar UCST extends ActivityDiagram {
2
3   UCSTArtifact = UCSTElement+;
4
5   interface UCSTElement;
6
7   UCD implements UCSTElement =
8     UseCaseDiagram;
```

```
9
10   AD implements UCSTElement =
11     NamedActivityDiagram;
12
13   /* ============= Scenario Template =============*/
14
15   symbol scope NamedScenarioTemplate implements UCDUseCase =
16     "scenariotemplate" Name@UCDUseCase "{"
17       STElement*
18     "}";
19
20   /* ============= Scenario Template Elements =============*/
21
22   interface STElement;
23
24   Comment implements STElement =
25     SL_COMMENT | ML_COMMENT;
26
27   STTrigger implements STElement =
28     "trigger" String ";";
29
30   STConditions =
31     Expression*;
32
33   STPreConditions extends STConditions implements STElement =
34     "pre" ((Expression | String) || ",")+ ";";
35   STPostConditions extends STConditions implements STElement =
36     "post" ((Expression | String) || ",")+ ";";
37
38   STGoal implements STElement =
39     "goal" String ";";
40
41   STPrimaryActor implements STElement =
42     "primary" (STActor || ",")+ ";";
43
44   STSecondaryActors implements STElement =
45     "secondary" (STActor || ",")+ ";";
46
47   STActor =
48     Name@UCDActor | Name@UCDSystem;
49
50   STActivityDiagram implements STElement =
51     "behavior" ActivityDiagram | ("activitydiagram"
        Name@NamedActivityDiagram ";");
52
53 }
```

Listing C.2: Use case scenario template (UCST) grammar based on [Bla23].

C.4 Use Case Specification Examples

Table C.2: A tabular use case specification of the use case 'Enter the Vehicle' form the stakeholder value in Figure 6.1.

UC Name	Enter the Vehicle							
Trigger	A person requests to enter the vehicle.							
Pre-condition	The driver is in a safe position for passengers to enter.							
Desired Behavior	A person enters the vehicle.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1a	Vehicle	Unlock the Vehicle	key: Information	-	Locked && valid(key)	Alternative	2
	1b	Vehicle	-	-	-	Unlocked	Alternative	2
	1c	Vehicle	-	-	-	not valid(key)	Alternative	Final
	2	Structure	Open the Door	doorOpEng: Energy	-	Unlocked	Sequential	3
	3	Structure	Pick up person	person: Material	-	-	Sequential	4
	4	Structure	Close the Door	doorOpEng: Energy	-	-	Sequential	Final
Post-conditions	The vehicle reached its destination.							

Table C.3: A tabular use case specification of the use case ‘Exit the Vehicle’ form the stakeholder value in Figure 6.1.

UC Name	Exit the Vehicle							
Trigger	A person requests to exit the vehicle.							
Pre-condition	A person requests to exit the vehicle and the vehicle is in a safe position for entering.							
Desired Behavior	A person exits the vehicle.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1a	Vehicle	Unlock the Vehicle	key: Information	-	Locked	Alternative	2
	1b	Vehicle	-	-	-	Unlocked	Alternative	2
	2	Structure	Open the Door	doorOpEng: Energy	-	Unlocked	Sequential	3
	3	Structure	Get off person	-	person: Material	-	Sequential	4
	4	Structure	Close the Door	doorOpEng: Energy	-	-	Sequential	5
	5a	Structure	Lock the Door	key: Energy	-	valid(key)	Alternative	Final
	5b	Structure	-	key: Energy	-	not valid(key)	Alternative	Final
	5c	Structure	-	-	-	else	Alternative	Final
Post-conditions	A person got off the vehicle.							

Table C.4: A tabular use case specification of the use case 'Unlock the Vehicle' form the stakeholder value in Figure 6.1.

UC Name	Unlock the Vehicle							
Trigger	A person requests to unlock the vehicle.							
Pre-condition	The vehicle is locked and the key is valid.							
Desired Behavior	The vehicles unlocks.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1a	Structure	Unlock the Door	key: Information	-	Locked	Alternative	2
	2	Others	Other elements required to unlock	-	-	Unlocked	Alternative	Final
Post-conditions	The vehicle is unlocked.							

Table C.5: A tabular use case specification of the use case 'Lock the Vehicle' form the stakeholder value in Figure 6.6.

UC Name	Unlock the Vehicle							
Trigger	A person requests to lock the vehicle.							
Pre-condition	The vehicle is unlocked and the key is valid.							
Desired Behavior	The vehicles locks.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1	Structure	Lock the Door	key: Information	-	Unlocked && valid(key)	Sequential	2
	2	Others	Other elements required to lock	-	-	Unlocked && valid(key)	Sequential	Final
Post-conditions	The vehicle is unlocked.							

Table C.6: A tabular use case specification of the feature ‘Open or Close Doors’ form the stakeholder value in Figure 6.6.

UC Name	«feature» Open or Close Doors							
Trigger	A person requests to open or close a door.							
Pre-condition	The doors are not locked.							
Desired Behavior	The vehicles opens or close the door.							
	#	Actor	Action	Input	Output	Condition	Execution	Next
	1a	Structure	Open the Door	doorOpEng: Energy	-	Closed && Unlocked	Alternative	Final
	1b	Structure	Close the Door	doorOpEng: Energy	-	Open && Unlocked	Alternative	Final
Post-conditions	-							

Table C.7: A tabular use case specification of the feature 'Lock or Unlock Doors' form the stakeholder value in Figure 6.6.

<table>
<tr><td>UC Name</td><td colspan="8">«feature» Lock or Unlock Doors</td></tr>
<tr><td>Trigger</td><td colspan="8">A person requests to lock or unlock a door.</td></tr>
<tr><td>Pre-condition</td><td colspan="8">The provided key is valid.</td></tr>
<tr><td rowspan="5">Desired Behavior</td><td colspan="8">The vehicles lock or unlocks the door.</td></tr>
<tr><td>#</td><td>Actor</td><td>Action</td><td>Input</td><td>Output</td><td>Condition</td><td>Execution</td><td>Next</td></tr>
<tr><td>1a</td><td>Structure</td><td>Unlock the Door</td><td>key: Information</td><td>-</td><td>Locked && valid(key)</td><td>Alternative</td><td>Final</td></tr>
<tr><td>1b</td><td>Structure</td><td>Close the Door</td><td>key: Information</td><td>-</td><td>Unlocked && Closed && valid(key)</td><td>Alternative</td><td>Final</td></tr>
<tr><td>1c</td><td>Structure</td><td>-</td><td>key: Information</td><td>-</td><td>Open && valid(key)</td><td>Alternative</td><td>Final</td></tr>
<tr><td>Post-conditions</td><td colspan="8">-</td></tr>
</table>

Appendix D

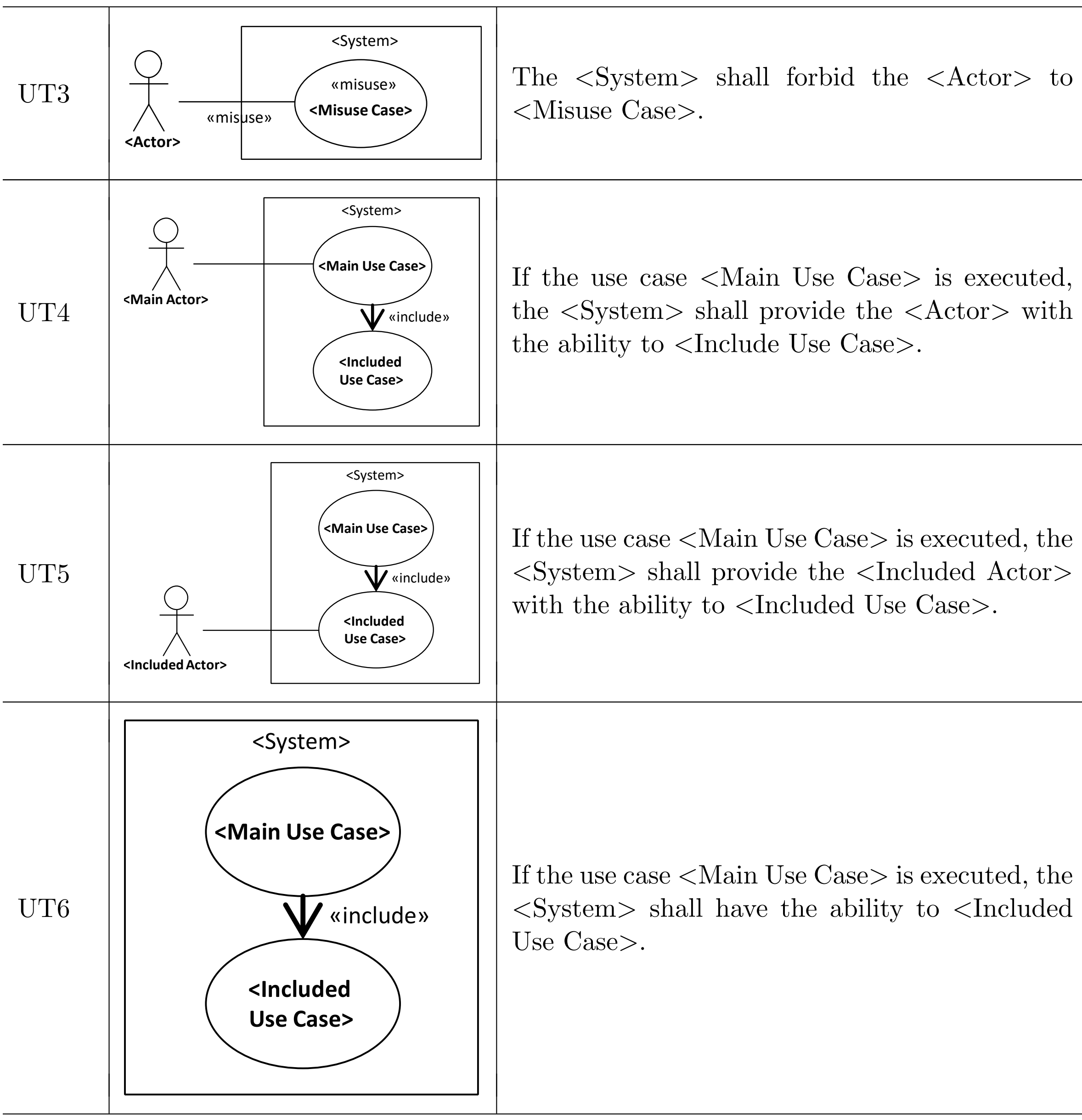
Transformation Rules and Guidelines to Generate Natural Language Requirements from SysML Diagrams

D.1 Transformation Rules and Guidelines to Generate Natural Language Requirements from Use Case Diagrams

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.

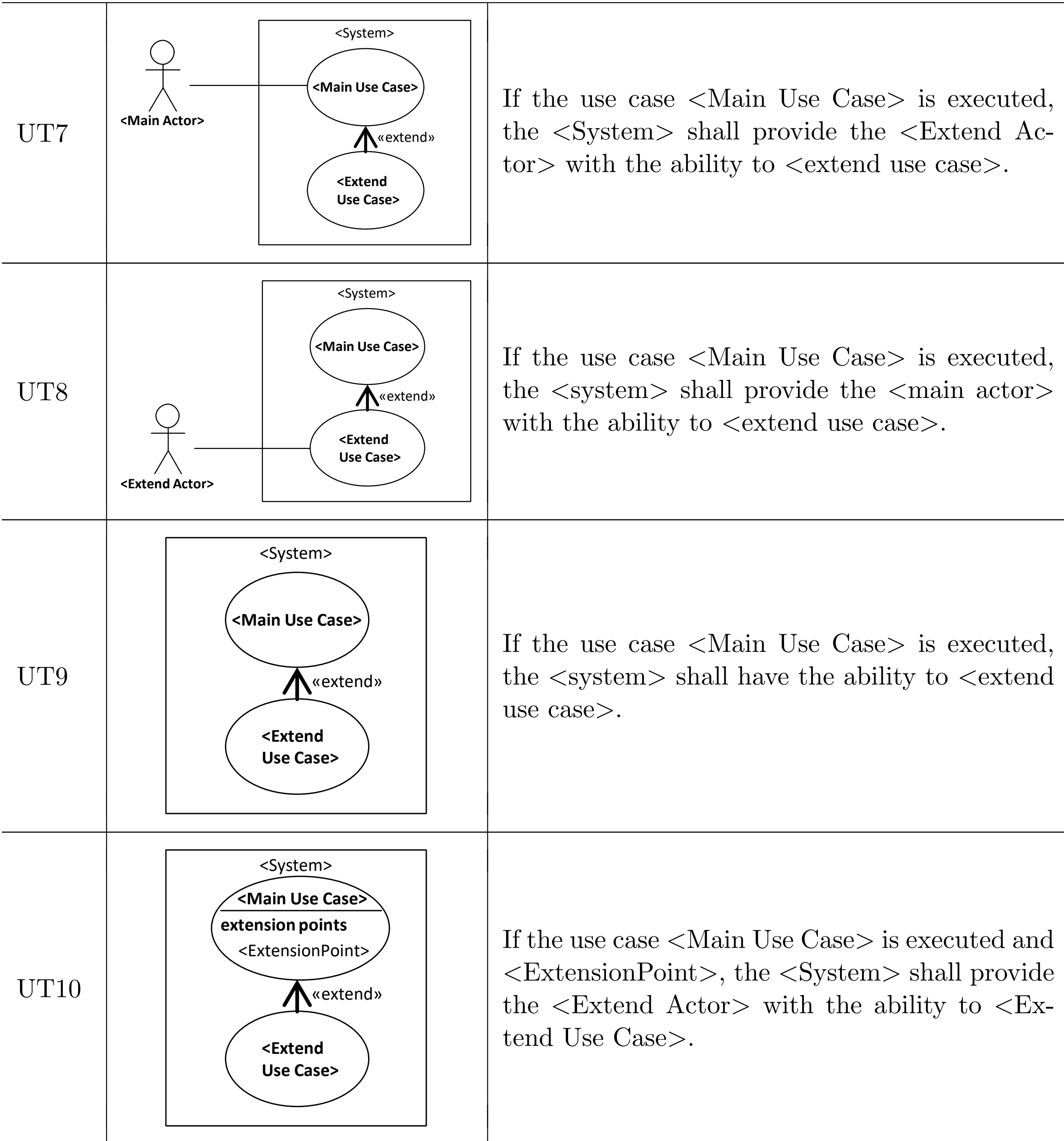
Requirement Generation Templates		
ID	Graphical Notation	Requirement Template
UT1	<System> [«feature»]? <Use Case> <Actor>	The <System> shall provide the <Actor> with the ability to <Use Case>.
UT2	<System> <Use Case>	The <System> shall have the ability to <Use Case>.

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.



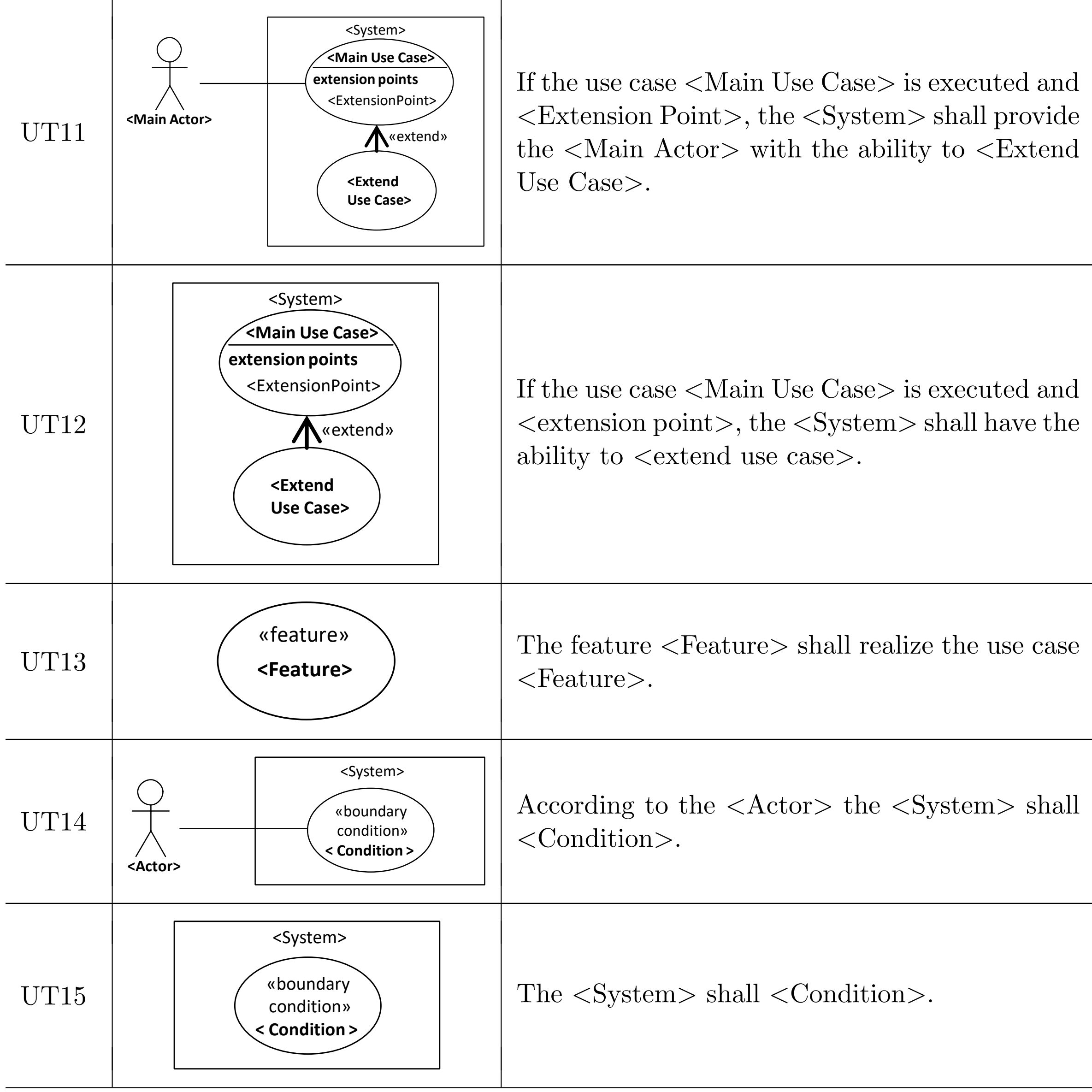
UT3		The <System> shall forbid the <Actor> to <Misuse Case>.
UT4		If the use case <Main Use Case> is executed, the <System> shall provide the <Actor> with the ability to <Include Use Case>.
UT5		If the use case <Main Use Case> is executed, the <System> shall provide the <Included Actor> with the ability to <Included Use Case>.
UT6		If the use case <Main Use Case> is executed, the <System> shall have the ability to <Included Use Case>.

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.



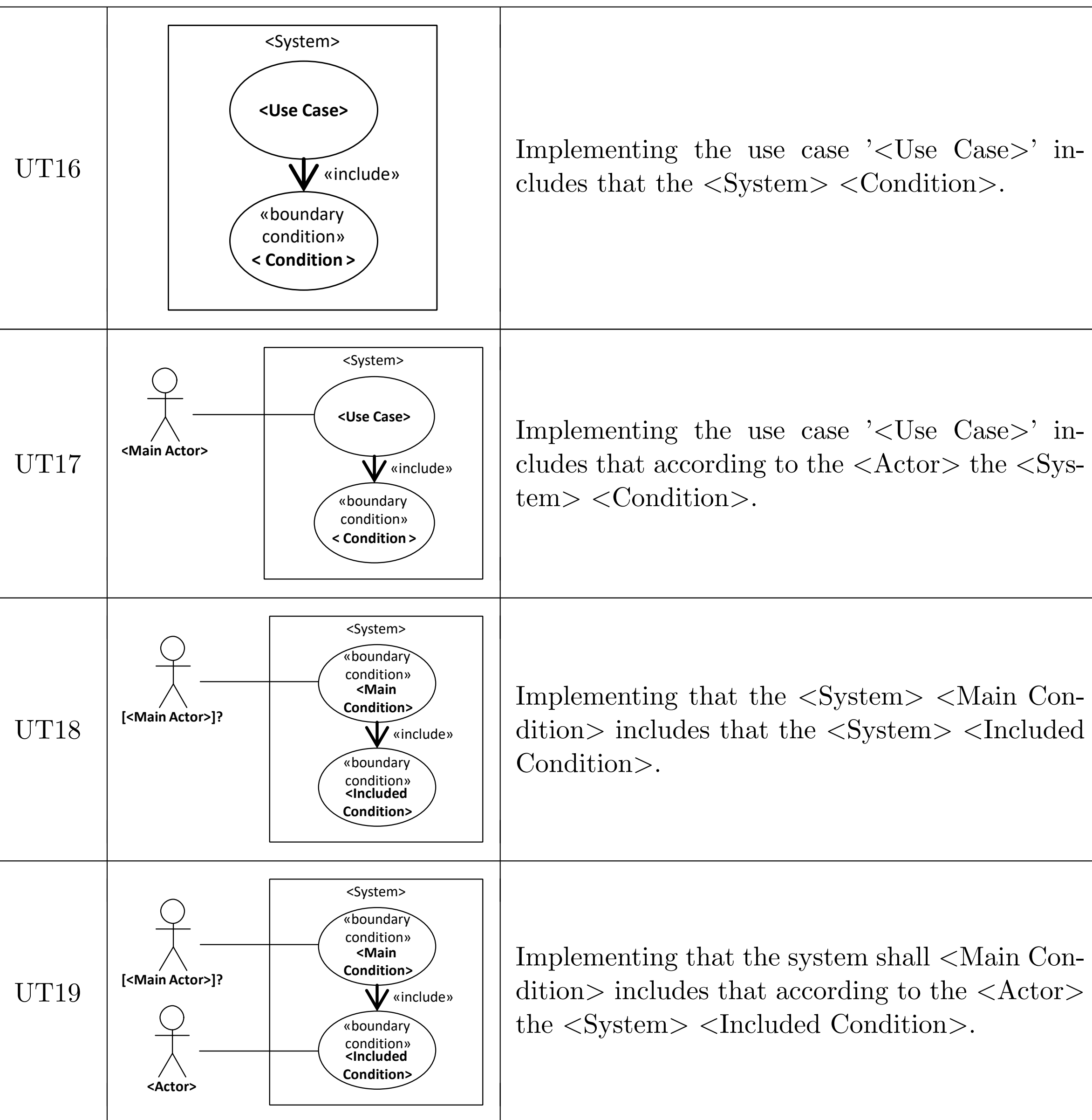
UT7		If the use case <Main Use Case> is executed, the <System> shall provide the <Extend Actor> with the ability to <extend use case>.
UT8		If the use case <Main Use Case> is executed, the <system> shall provide the <main actor> with the ability to <extend use case>.
UT9		If the use case <Main Use Case> is executed, the <system> shall have the ability to <extend use case>.
UT10		If the use case <Main Use Case> is executed and <ExtensionPoint>, the <System> shall provide the <Extend Actor> with the ability to <Extend Use Case>.

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.



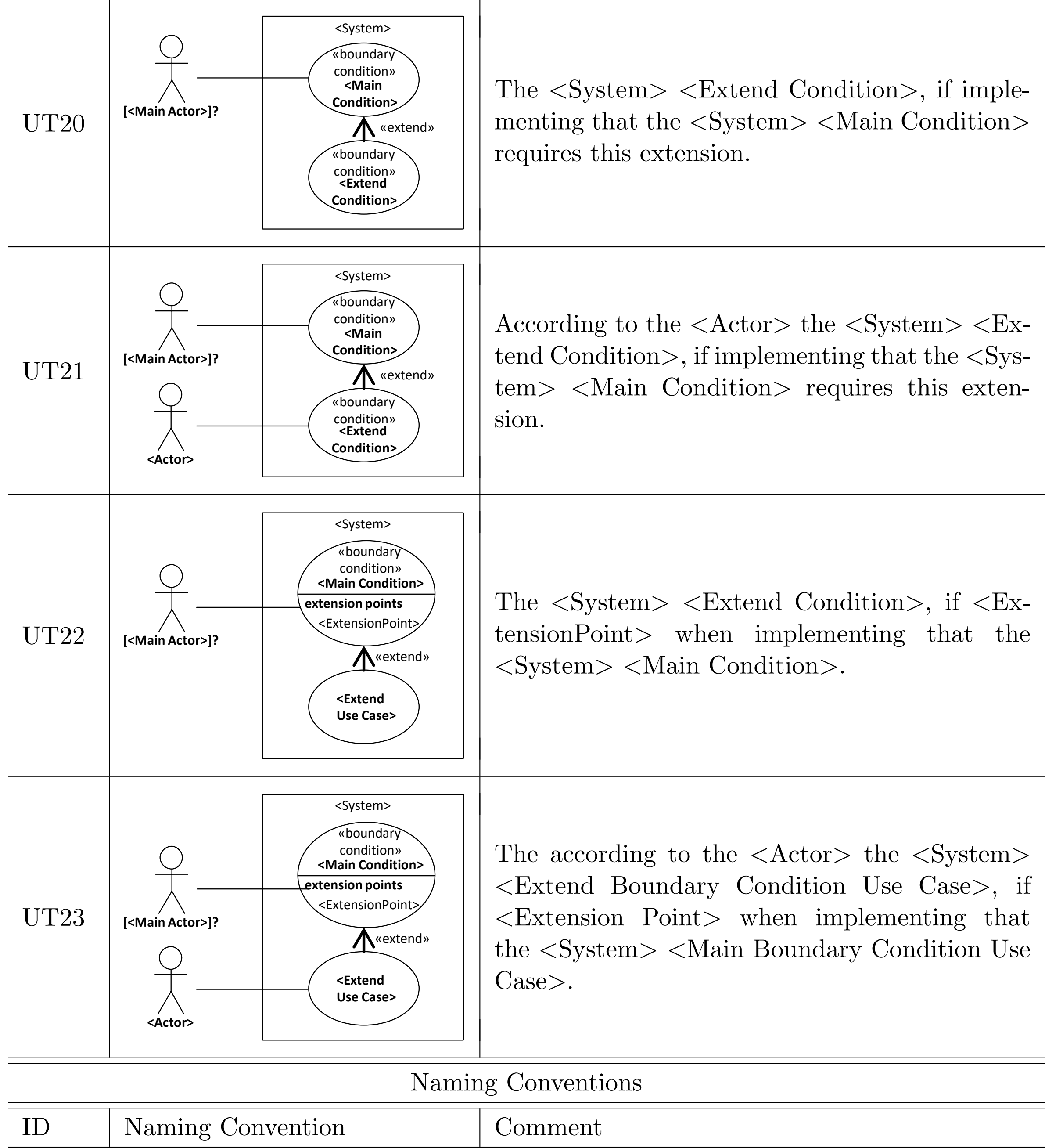
UT11	<System> <Main Actor> <Main Use Case> extension points <ExtensionPoint> «extend» <Extend Use Case>	If the use case <Main Use Case> is executed and <Extension Point>, the <System> shall provide the <Main Actor> with the ability to <Extend Use Case>.
UT12	<System> <Main Use Case> extension points <ExtensionPoint> «extend» <Extend Use Case>	If the use case <Main Use Case> is executed and <extension point>, the <System> shall have the ability to <extend use case>.
UT13	«feature» <Feature>	The feature <Feature> shall realize the use case <Feature>.
UT14	<System> <Actor> «boundary condition» < Condition >	According to the <Actor> the <System> shall <Condition>.
UT15	<System> «boundary condition» < Condition >	The <System> shall <Condition>.

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.



ID	Diagram	Requirement template
UT16		Implementing the use case '<Use Case>' includes that the <System> <Condition>.
UT17		Implementing the use case '<Use Case>' includes that according to the <Actor> the <System> <Condition>.
UT18		Implementing that the <System> <Main Condition> includes that the <System> <Included Condition>.
UT19		Implementing that the system shall <Main Condition> includes that according to the <Actor> the <System> <Included Condition>.

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.



UT20		The <System> <Extend Condition>, if implementing that the <System> <Main Condition> requires this extension.
UT21		According to the <Actor> the <System> <Extend Condition>, if implementing that the <System> <Main Condition> requires this extension.
UT22		The <System> <Extend Condition>, if <ExtensionPoint> when implementing that the <System> <Main Condition>.
UT23		The according to the <Actor> the <System> <Extend Boundary Condition Use Case>, if <Extension Point> when implementing that the <System> <Main Boundary Condition Use Case>.

Naming Conventions

ID	Naming Convention	Comment

Table D.1: UCD requirement generation templates to derive natural language requirements from use case models.

UN1	<Process verb>+ [details on the verb or the object]? <object>+ [details on the object]?	This naming convention for use cases and feature use cases aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (*cf.* section 5.2)
UN2	<Main Use Case> **extension points** <ExtensionPoint>	As the formulation of extension points affects the correctness of the requirements according to UT10, UT11, and UT12, this naming convention for the extension points aims at ensuring the generation of grammatically correct requirements.
UN3	<Actor> «secondary»/ «tertiary» <Process verb>+ [details on the verb or the object with respect to the actor] + <object>+ [details on the object with respect to the actor]+	As secondary and tertiary actors often have a complicated interaction with the system by describing details on the verb or subject, this naming convention ensures that the modeler expresses them in the <details on subject or verb defining the interaction with the secondary or tertiary actor>.
UN4	«boundary condition» [system attribute]? shall <Process verb>+ [details on the verb or the object]? <object>+ [details on the object]?	To derive grammatically correct natural language requirements from boundary condition use cases, the boundary conditions must contain an optional system attribute the system modifies, the modal verb shall, a process verb and optional objects and details on the process verb.

Additional notes on the application of the templates:

- General Note: All generated SysML requirement elements should be linked (*e.g.,* via a «derive»relationship) to the element it was generated from to achieve traceability according to [SPI15].
- Note on UT4: The formulation of this template might seem misleading at the first sight, as the actor connected to the main use case is not involved in the execution of the include use case according to the use case semantic definition. This template was added on request of the stakeholders in the industry projects Zeta and Delta to highlight that the actor involved in the main use case, gets the

included functionality as surprise. Principally, the formulation from UT6 would suffice.

- Note on UT13: is executed in addition to UT1-2 to link features with their implementing use cases.
- Note on UT15 and UT16: For boundary condition use cases only these to
- Note on UN3: As secondary and tertiary actors often have a complicated interaction with the system by describing details on the verb or subject, the easiest way to express them in the model are to mention then in the <details on subject or verb>. As a straightforward conformance check, the appearance of the actor in this string is automatically testable.

D.2 Transformation Rules and Guidelines to Generate Natural Language Requirements from Activity Diagrams

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

Main Requirement Generation Templates		
ID	Graphical Notation	Requirement Template
AT1.1	<Feature> AD <in1>: <In1Type> <inN>: <InNType> <action> <out1>: <Out1Type> <outN>: <OutNType>	The <Feature> shall <action> using <in1> of type <In1Type>, ..., and <inN> of type <InNType>, providing <out1> of type <Out1Type>, ..., and <OutN> of type <OutNType>.

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

AT1.2	<Feature> AD «allocate» :<System> <in1>: <In1Type> <inN>: <InNType> <action> <out1>: <Out1Type> <outN>: <OutNType>	The <System> shall <action> using <in1> of type <In1Type>, ..., and <inN> of type <InNType>, providing <out1> of type <Out1Type>, ..., and <OutN> of type <OutNType>.
AT5.1	<Feature> AD <send action> : <Signal>	The <Feature> shall <send action> providing the <Signal> signal.
AT5.2	<Feature> AD «allocate» :<System> <send action> : <Signal>	The <System> shall <send action> providing the <Signal> signal.
AT6.1	<Feature> AD <receive action> : <Signal>	The <Feature> shall <receive action> receiving the <Signal> signal.
AT6.2	<Feature> AD «allocate» :<System> <receive action> : <Signal>	The <System> shall <receive action> receiving the <Signal> signal.

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

AT7.1	<Feature> AD <accept time event action>	The <Feature> shall <accept time event action>.
AT7.2	<Feature> AD «allocate» :<System> <accept time event action>	The <System> shall <accept time event action> receiving the <Signal> signal.
AT8.1	<Feature> AD	The <Feature> shall terminate its execution.
AT8.2	<Feature> AD «allocate» :<System>	The <System> shall terminate the execution of the feature ‘<Feature>’.

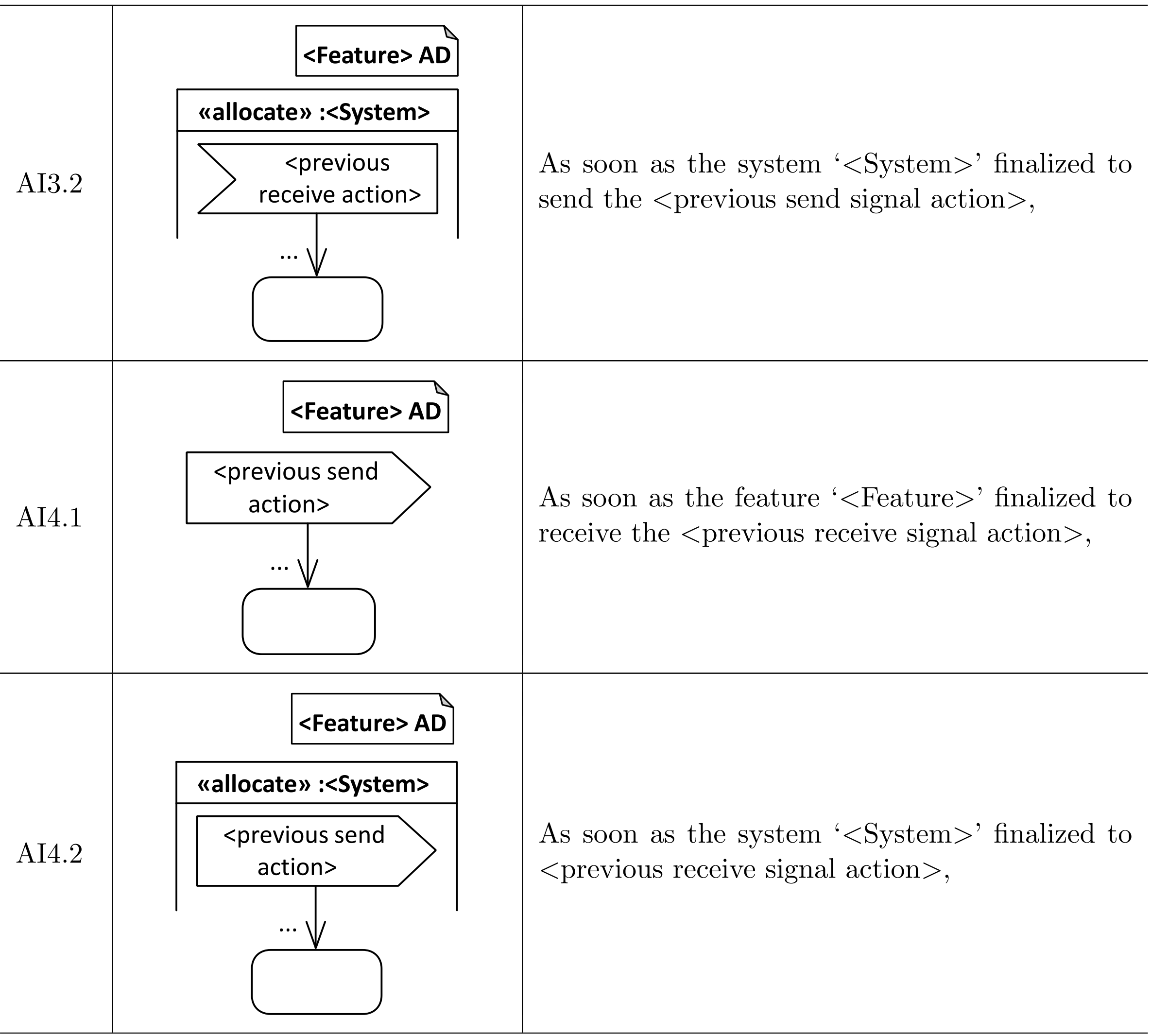
Execution Trigger Template

ID	Graphical Notation	Requirement Template
AI1.1	<Feature> AD ...	As soon as the feature ‘<Feature>’ is executed,

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

AI1.2	<Feature> AD «allocate» :<System> ...	As soon as the <System under Development> executes the feature ‘<Feature>’,
AI2.1	<Feature> AD <previous action> ...	As soon as the feature ‘<Feature>’ finalized to <previous action>,
AI2.2	<Feature> AD «allocate» :<System> <previous action> ...	As soon as the system ‘<System>’ finalized to <previous action>,
AI3.1	<Feature> AD <previous receive action> ...	As soon as the feature ‘<Feature>’ finalized to send the <previous send action>,

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

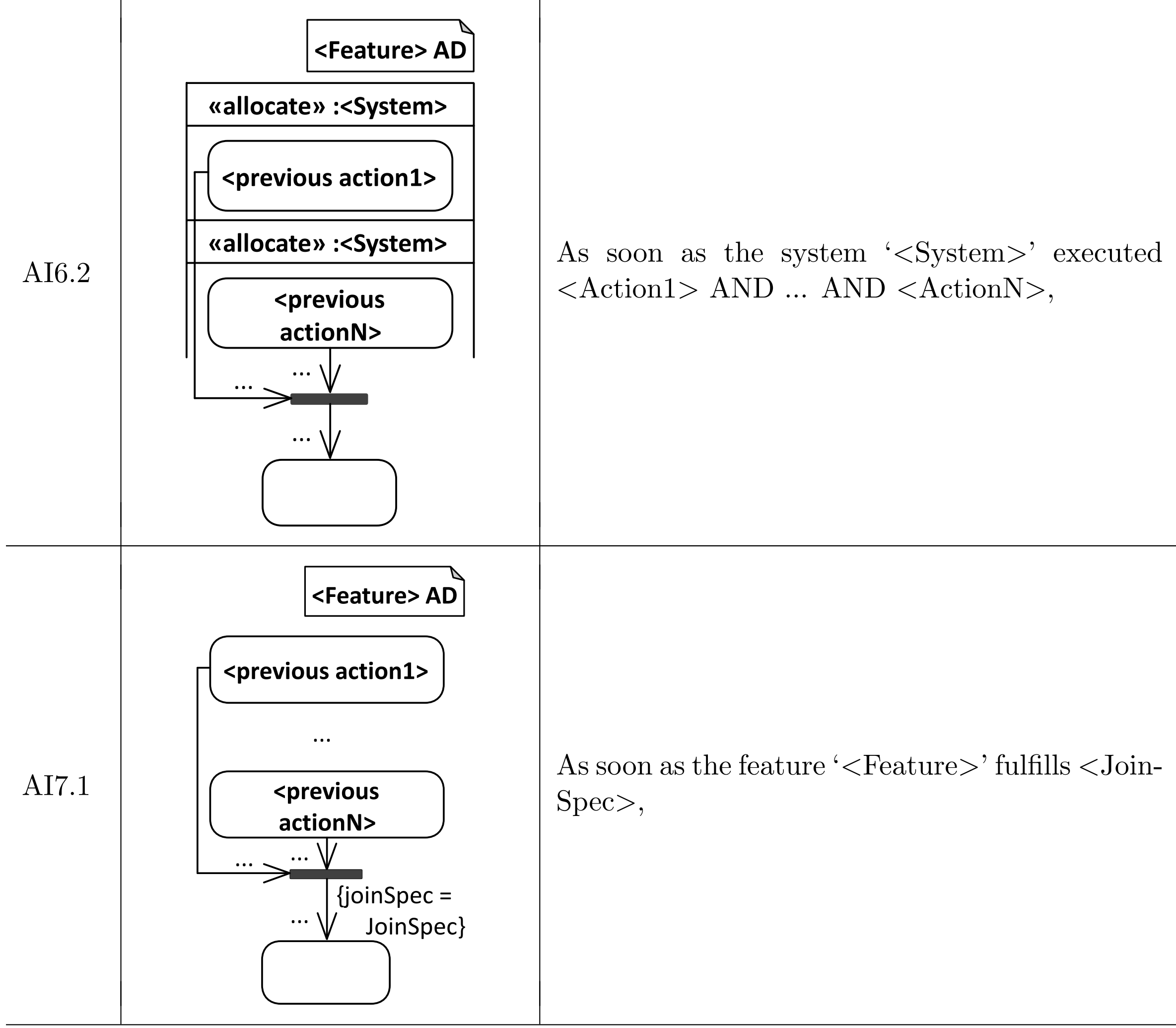


AI3.2		As soon as the system ‘<System>’ finalized to send the <previous send signal action>,
AI4.1		As soon as the feature ‘<Feature>’ finalized to receive the <previous receive signal action>,
AI4.2		As soon as the system ‘<System>’ finalized to <previous receive signal action>,

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

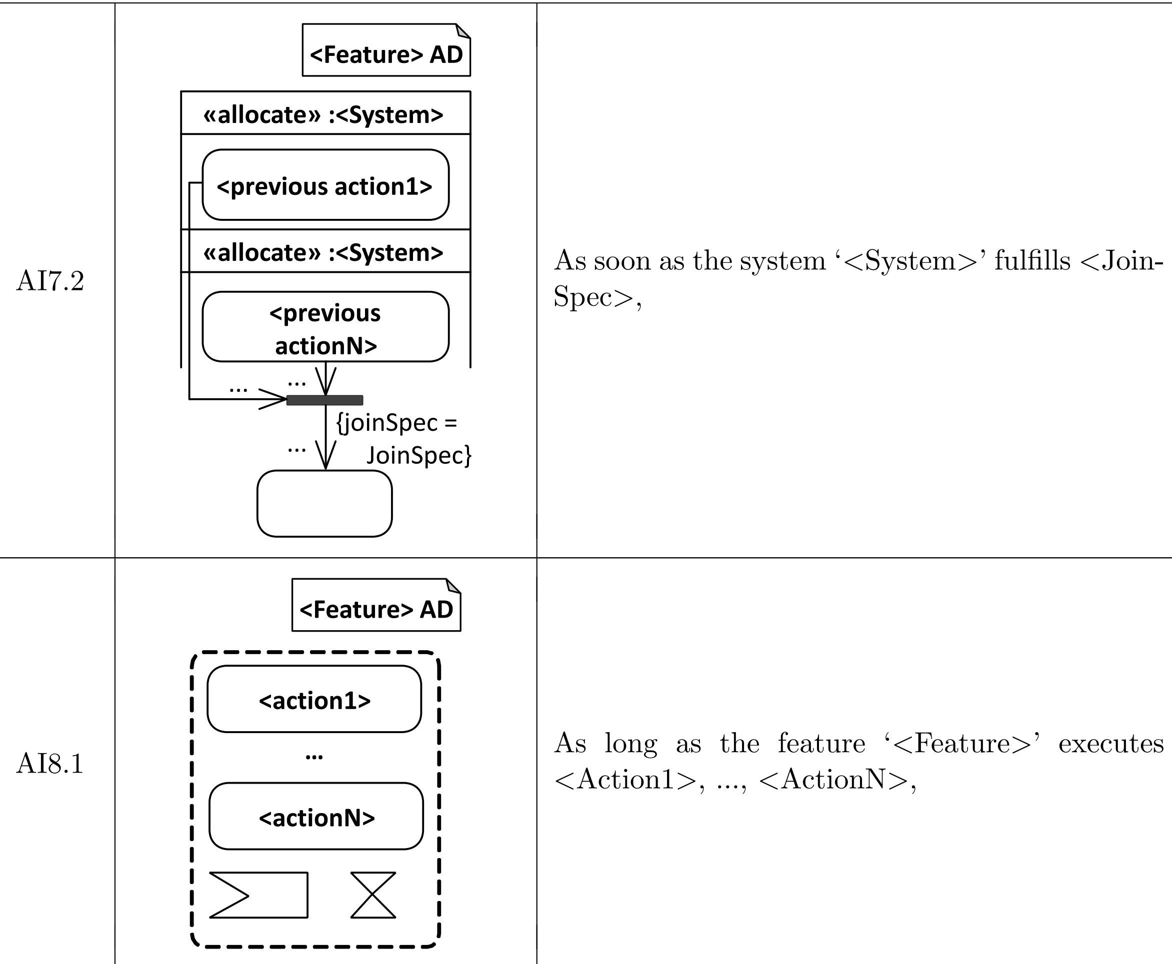
AI5.1	<Feature> AD <previous time event> ...	As soon as the feature '<Feature>' detected <previous time event> as time event,
AI5.2	<Feature> AD «allocate» :<System> <previous time event> ...	As soon as the system '<System>' detected <previous time event> as time event,
AI6.1	<Feature> AD <previous action1> ... <previous actionN>	As soon as the feature '<Feature>' executed <Action1> AND ... AND <ActionN>,

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.



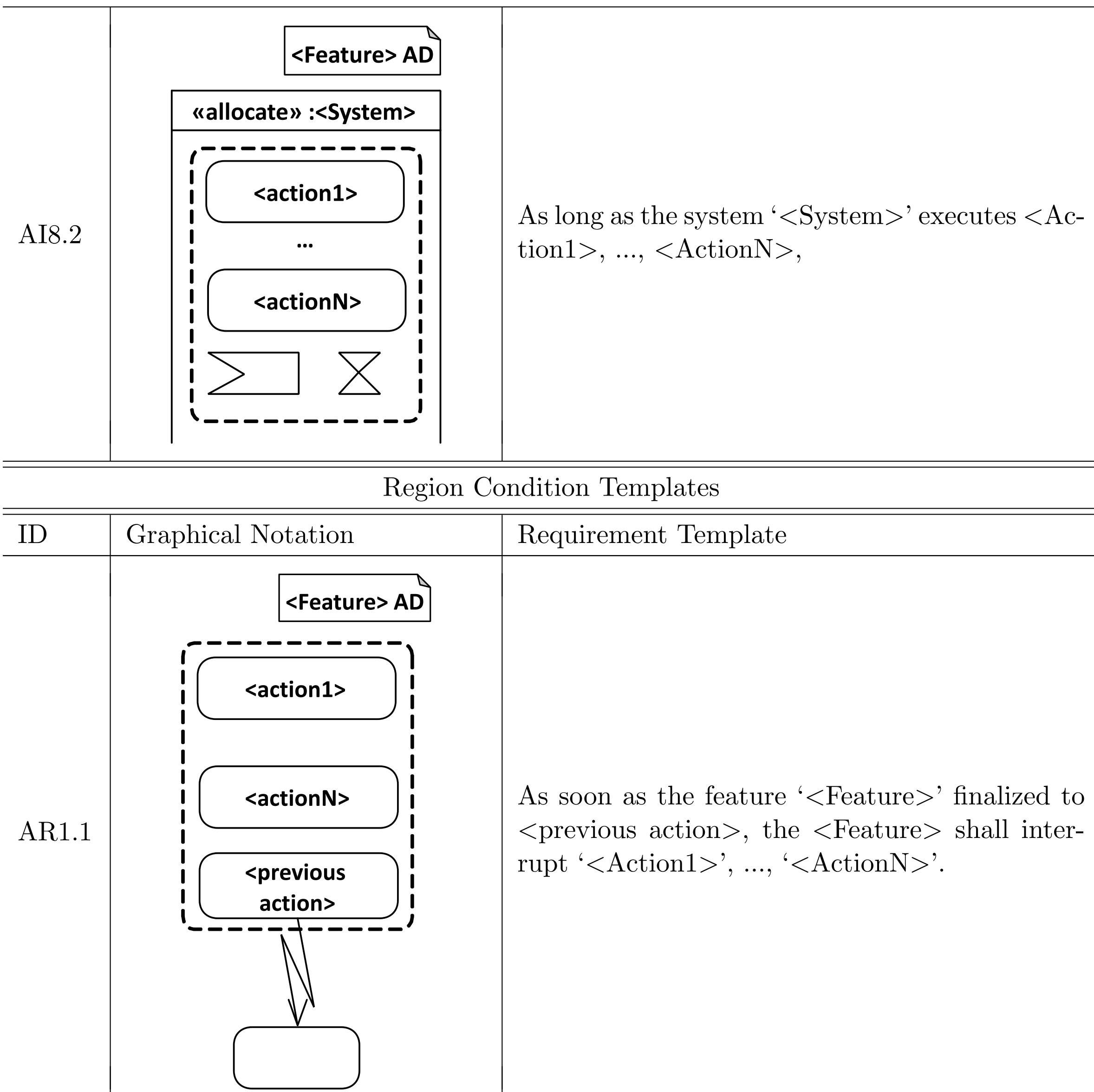
AI6.2		As soon as the system '<System>' executed <Action1> AND ... AND <ActionN>,
AI7.1		As soon as the feature '<Feature>' fulfills <JoinSpec>,

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.



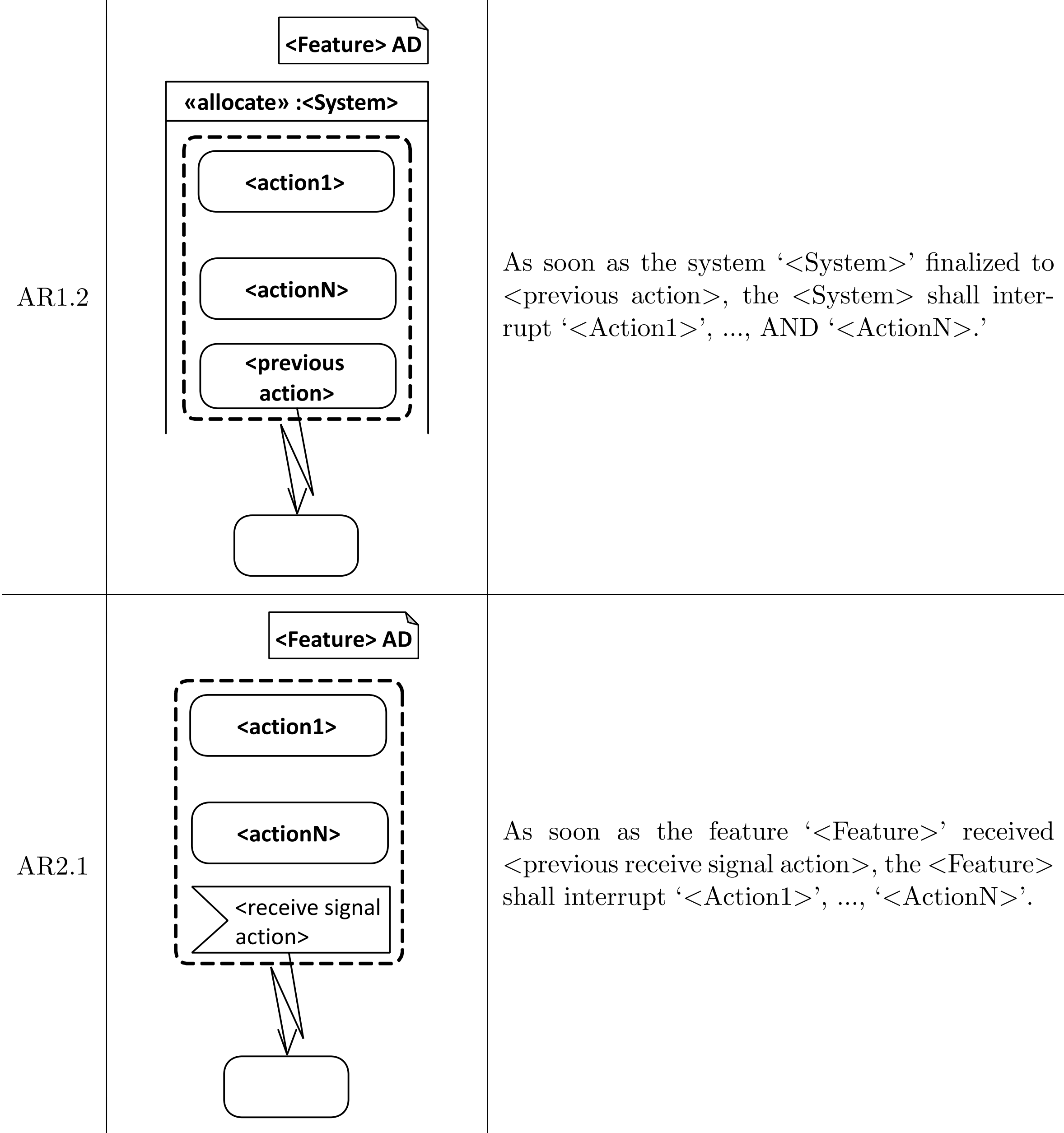
AI7.2		As soon as the system '<System>' fulfills <JoinSpec>,
AI8.1		As long as the feature '<Feature>' executes <Action1>, ..., <ActionN>,

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.



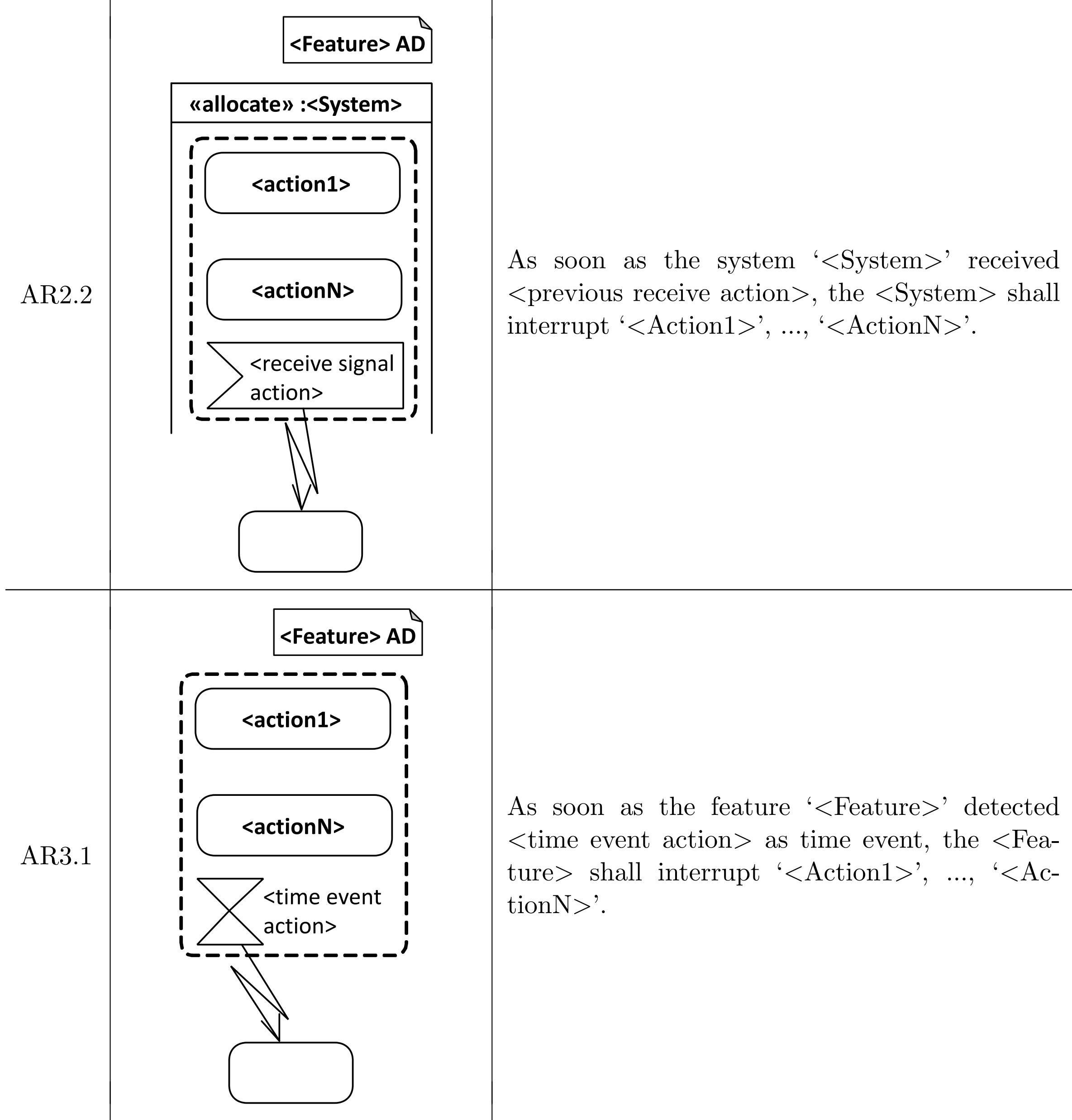
ID	Graphical Notation	Requirement Template
AI8.2		As long as the system '<System>' executes <Action1>, ..., <ActionN>,
Region Condition Templates		
ID	Graphical Notation	Requirement Template
AR1.1		As soon as the feature '<Feature>' finalized to <previous action>, the <Feature> shall interrupt '<Action1>', ..., '<ActionN>'.

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.



AR1.2		As soon as the system ‘<System>’ finalized to <previous action>, the <System> shall interrupt ‘<Action1>’, ..., AND ‘<ActionN>.’
AR2.1		As soon as the feature ‘<Feature>’ received <previous receive signal action>, the <Feature> shall interrupt ‘<Action1>’, ..., ‘<ActionN>’.

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.



AR2.2		As soon as the system ‘<System>’ received <previous receive action>, the <System> shall interrupt ‘<Action1>’, ..., ‘<ActionN>’.
AR3.1		As soon as the feature ‘<Feature>’ detected <time event action> as time event, the <Feature> shall interrupt ‘<Action1>’, ..., ‘<ActionN>’.

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

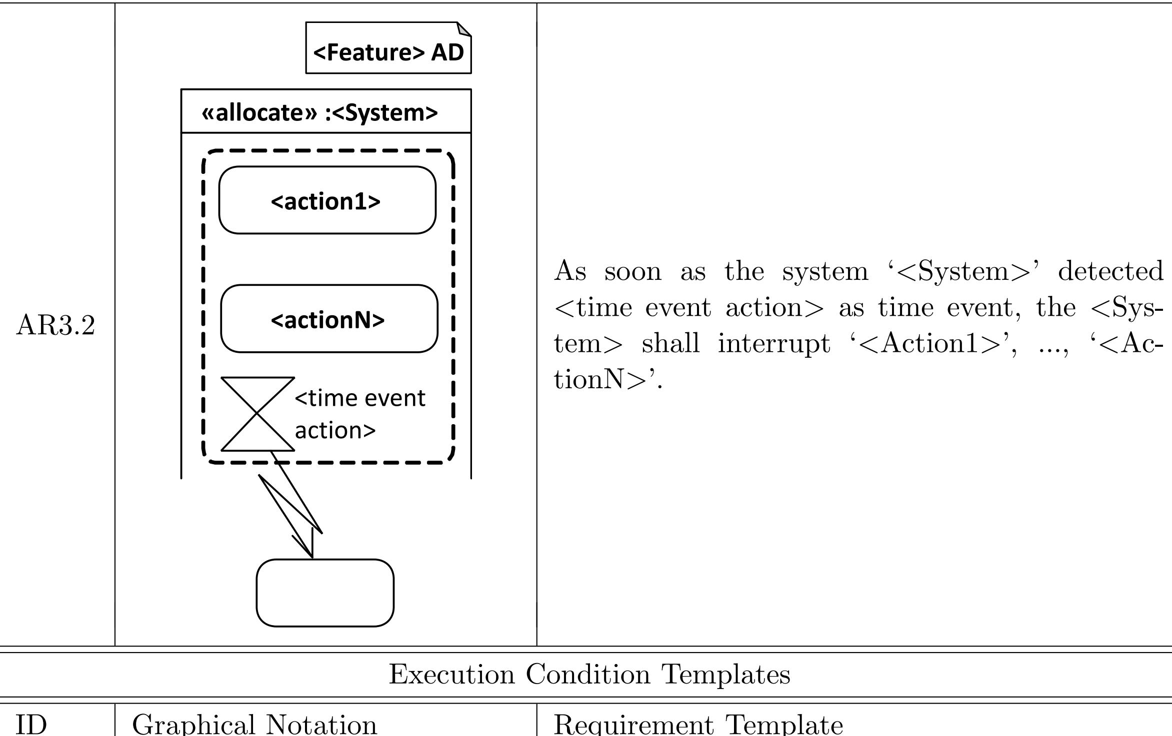
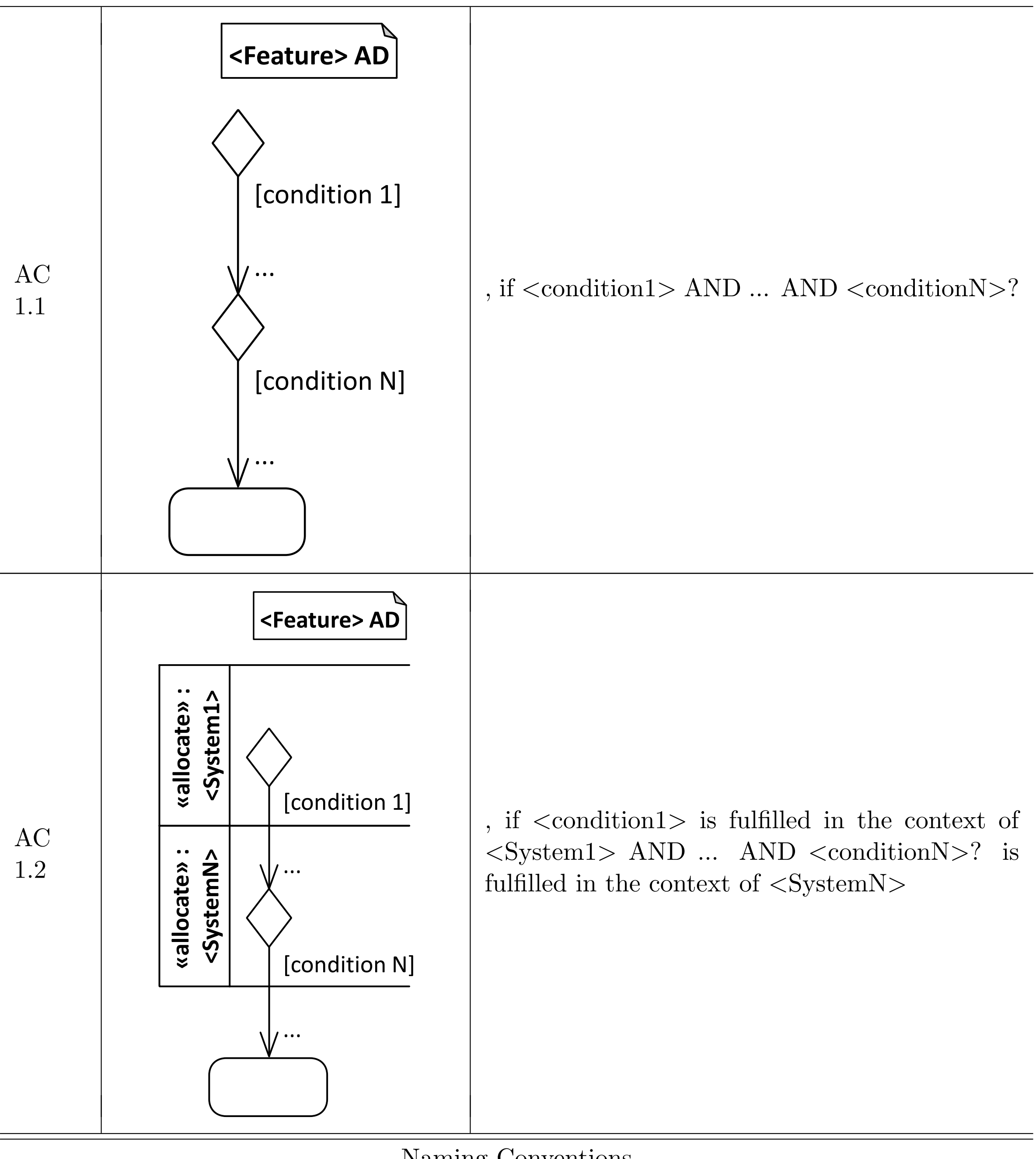
AR3.2		As soon as the system ‘<System>’ detected <time event action> as time event, the <System> shall interrupt ‘<Action1>’, ..., ‘<ActionN>’.
Execution Condition Templates		
ID	Graphical Notation	Requirement Template

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.



AC 1.1		, if <condition1> AND ... AND <conditionN>?
AC 1.2		, if <condition1> is fulfilled in the context of <System1> AND ... AND <conditionN>? is fulfilled in the context of <SystemN>

Naming Conventions

Table D.2: AD requirement generation templates to derive natural language requirements from use case models.

<table>
<tr><th>ID</th><th>Naming Convention</th><th>Comment</th></tr>
<tr><td>AN1</td><td><Process verb>+
[details on the verb or the object]?
<object>+
[details on the object]?</td><td>This naming convention for actions aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (cf. section 5.2)</td></tr>
<tr><td>AN2</td><td>a z
{joinSpec =
([!]?[a..z]
(['||' | '&&']
[a..z])?)+ }</td><td>This naming convention for join specifications at join nodes aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (cf. section 5.2)</td></tr>
<tr><td>AN3</td><td>[conditional
sencence
without if]
...</td><td>This naming convention for conditions after decision nodes aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (cf. section 5.2)</td></tr>
</table>

- General Note: All generated SysML requirement elements should be linked (*e.g.,* via a «derive»relationship) to the element it was generated from to achieve traceability according to [SPI15].
- General Note: Depending on the contents of the templates the capitalization of AT1.1 and AT1.2 must be adapted.
- General Note: The ... on the control flow indicates that there exists a direct control flow between the element with potential fork, joins and merges in-between, but no additional action.
- AI8.1/AI8.2: The requirement prefix shall be generated for each accept event (accept event action and accept time event action) requirement in the region without in-going control flow.

D.3 Transformation Rules and Guidelines to Generate Natural Language Requirements from Internal Block Diagrams

Table D.3: Requirement generation templates to derive natural language requirements for functionality and interface requirements from activity diagram models based on [Zab23].

Requirement Generation Rules		
ID	Graphical Notation	Requirement Template
LT1	AD «allocate» :<System> <ActionExecution PreCondition>? [<Action Execution Condition>?] <in1>: <In1Type> <inN>: <InNType> <action> <out1>: <Out1Type> <outN>: <OutNType>	<ActionExecutionPreCondition>?, The <Systemy> shall <action> using '<in1>' of type '<In1Type>', ..., and '<inN>' of type '<InNType>', providing '<out1>' of type '<Out1Type>', ..., and '<outN>' of type '<OutNType>', <ActionExecutionCondition>?. (According to the transformation rules in Table D.2)
LT2	IBD : <Providing System> <out>:<Type> : <Receiving System>	The <Providing System> shall provide the <Receiving System> with the '<out>' of type '<Type>'.
LT3	IBD : <Providing System> <in>:<Type> : <Receiving System>	The <Receiving System> shall receive the '<in>' of type '<Type>' from the <Providing System>.

Notes:

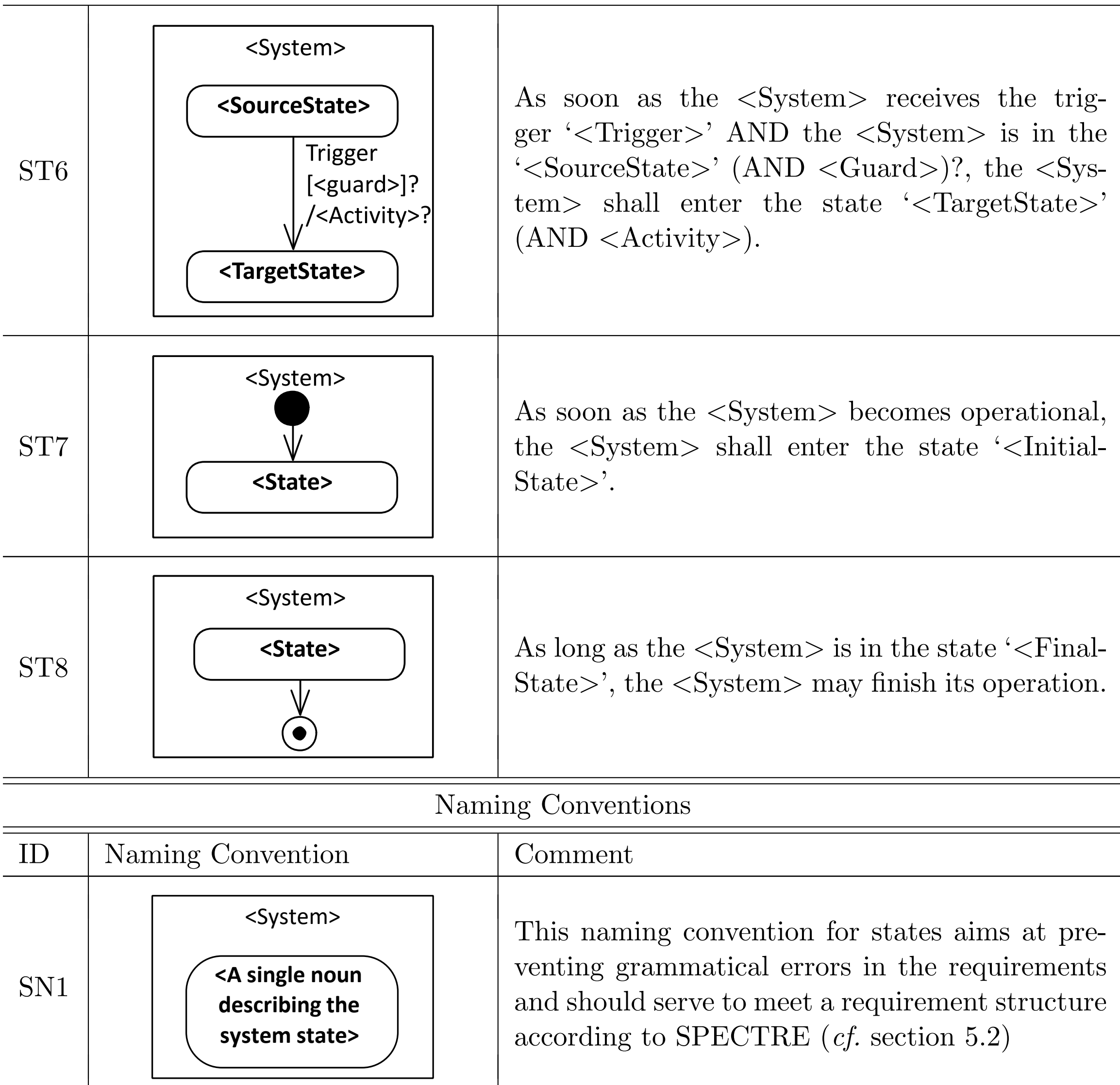
- General Note: All generated SysML requirement elements should be linked (*e.g.*, via a «derive»relationship) to the element it was generated from to achieve traceability according to [SPI15].

D.4 Transformation Rules and Guidelines to Generate Natural Language Requirements from State Machine Diagrams

Table D.4: STM requirement generation templates to derive natural language requirements from state machines.

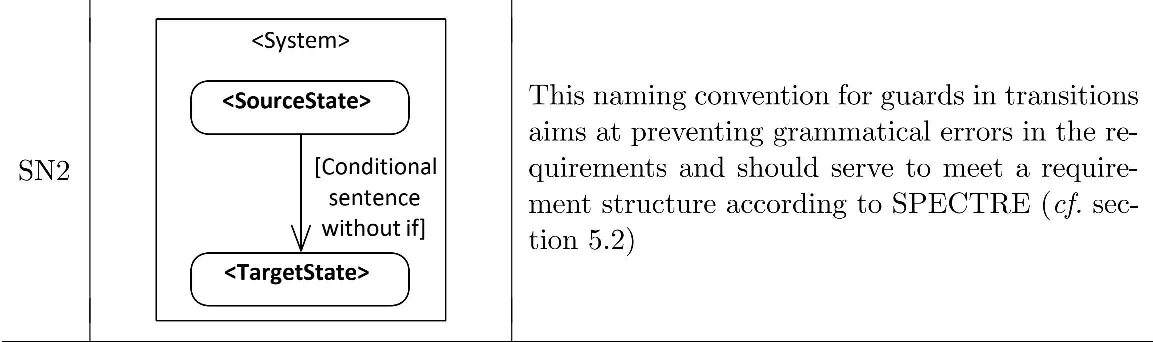
Requirement Generation Templates		
ID	Graphical Notation	Requirement Template
ST1	<System> **<State>**	The <System> shall have a <State> state.
ST2	<System> **<State>** entry /<entryActivity>	As soon as the <System> enters the state '<State>', the <System> shall <entryActivity>.
ST3	<System> **<State>** do / <doActivity>	As long as the <System> is in the state '<State>', the <System> shall have the ability to <doActivity>.
ST4	<System> **<State>** exit / <exitActivity>	As soon as the <System> exits the state '<State>', the <System> shall <exitActivity>.
ST5	<System> **<SourceState>** [<guard>?] **<TargetState>**	As long as the <System> is in the state '<SourceState>' (AND <Guard>)?, the <System> shall have the ability enter the state '<TargetState>'.

Table D.4: STM requirement generation templates to derive natural language requirements from state machines.



ST6	<System> <SourceState> Trigger [<guard>]? /<Activity>? <TargetState>	As soon as the <System> receives the trigger '<Trigger>' AND the <System> is in the '<SourceState>' (AND <Guard>)?, the <System> shall enter the state '<TargetState>' (AND <Activity>).
ST7	<System> <State>	As soon as the <System> becomes operational, the <System> shall enter the state '<Initial-State>'.
ST8	<System> <State>	As long as the <System> is in the state '<Final-State>', the <System> may finish its operation.
	Naming Conventions	
ID	Naming Convention	Comment
SN1	<System> <A single noun describing the system state>	This naming convention for states aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (*cf.* section 5.2)

Table D.4: STM requirement generation templates to derive natural language requirements from state machines.

SN2		This naming convention for guards in transitions aims at preventing grammatical errors in the requirements and should serve to meet a requirement structure according to SPECTRE (*cf.* section 5.2)

Notes:

- General Note: All generated SysML requirement elements should be linked (*e.g.*, via a «derive»relationship) to the element it was generated from to achieve traceability according to [SPI15].
- Note on ST2-6: In addition to the link, this requirement should also be linked to the implementing activity requirement in the activity in the operating principle model using the «derive»relationship.

List of Figures

2.1 The Process, Methods, Tools and Environment (PMTE) paradigm based on [Mar94]. 16
2.2 A Cyber-Physical System processes input and output flows of energy, material, and data. 23
2.3 Functional decomposition of an overall function to interconnected subfunctions on decomposition levels based on [PBFG07]. 24
2.4 The SysML structure based on [Obj19]. 33
2.5 The unique SysML diagrams. 36
2.6 The SysML structure diagrams. 37
2.7 The SysML behavior diagrams I. 38
2.8 The SysML behavior diagrams II. 39
2.9 The SysML behavior diagrams III. 40

3.1 The Compositional Unified system-Based Engineering (CUBE) model based on [GKS+21] and [MSG+22]. 55
3.2 The CUBE model without variability based on [GKS+21] and [MSG+22]. 58
3.3 Artifacts created during the CUBE process based on [GKS+21] and [MSG+22]. 59
3.4 Mapping CUBE abstraction layers to the most suited diagram for system specification based on the SysML diagram overview presented in [Obj19]. 60
3.5 Specifications are an abstract concept consisting of a model and a set of requirements that is implemented by the specification artifacts defined in Figure 3.3. 62
3.6 The meta-architecture framework as presented in [BGK+09]. 66
3.7 An adaption of the common architecture framework based on [BGK+09]. 67
3.8 Different requirement types based on a taxonomy for concern-based requirements according to [Gli07]. 69
3.9 The stakeholder values denote the stakeholder value specification by referring to stakeholder needs or external documents as specified in the specification artifacts from Figure 3.3. 71

3.10 The features denote the operating principle specification by relating the addressed use cases with the action sequence and their interfaces that define the signature of the feature to implement the specification artifacts from Figure 3.3. 74
3.11 The logical components execute the the actions the operating principle specifies to implement the specification artifacts from Figure 3.3. 76
3.12 The technical architecture denotes the technical architecture specification by allocating the runtime model to the hardware topology implementing the specification artifacts defined in Figure 3.3 based on the automotive architecture framework described in [BGK+09]. 78
3.13 Realization to implement the specification artifacts defined in Figure 3.3 extending the automotive architecture framework described in [BGK+09]. 80
3.14 A combined view of the specification artifacts and their elements from Figure 3.9, Figure 3.10, Figure 3.11, Figure 3.12, based on [GORW23]. . . 84

4.1 The automotive transportation scenario based on the simple automotive model in [FMS14a]. 94

5.1 The Specification guideline for Requirements Engineering (SPECTRE) template used in the Alpha and Beta projects based on [PR21]. 102
5.2 Additional examples for the application of the Specification guideline for Requirements Engineering (SPECTRE) template that the requirements engineers in the Alpha and Beta projects received. 102
5.3 Additional applications of the Specification guideline for Requirements Engineering (SPECTRE) template for requirements engineers to integrate variability information in the Alpha and Beta projects. 104
5.4 Basic sentence structure that formed the basis for grammar development. 106
5.5 Final grammatical structure that forms the basis for the SPECTRE-DSL. 106
5.6 Requirements excluded during the preprocessing in the Alpha and Beta projects. 112
5.7 The parsing results of the requirements in the Alpha and the Beta projects. 114
5.8 Overall evaluation results of the requirements in the Alpha and Beta projects. 115
5.9 The overall requirement evaluation results. 116
5.10 Overall evaluation results of the requirements in the Alpha subset with corrected variables. 117

6.1 Automotive driving UCD based on an initial example from [FMS14a]. . . 130
6.2 A class diagram to define the abstract syntax of the use case diagram language based on [KRW22]. 136

6.3 A stakeholder value use case diagram modeling the stakeholder value of a Highway Pilot Feature developed in the Delta project. 144
6.4 An extended version of the diagram from Figure 6.3 that integrates the syntax extensions from Table 6.3 into the diagram. 152
6.5 Automotive driving use case diagrams based on an initial example from [FMS14a]. 153
6.6 A stakeholder value use case diagram for the structure system on the next decomposition level. 156
6.7 A straightforward transformation that inserts the names of the use case diagram element in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1). 158
6.8 A modeling guideline to assure grammatically correct sentences in the SPECTRE template (*cf.* Figure 5.1). 159
6.9 The use case diagram from Figure 6.4 together with the derived requirements using the transformation rules in section D.1. 160
6.10 Correctness evaluation results of the stakeholder value requirements generated in the Zeta, Delta, VTOL, and Epsilon projects. 163
6.11 Results of the error reason classification in the Zeta, Delta, and Epsilon projects. 165
6.12 Results of the completeness evaluation comparing the model-based stakeholder value specification from the Epsilon projects with the document-based legacy requirements the project used as input. 168
6.13 Number of functionality and boundary condition use cases in the reworked use case model: Total 75 requirements. 169
6.14 The example diagram for which requirement experts should derive textual requirements. 171

7.1 Operating principles for the 'Transport Person and Goods' feature and the 'Drive Vehicle' use case from Figure 6.1. 183
7.2 A straightforward transformation that inserts the names of the actions in the diagram and the execution order in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1). 198
7.3 Correctness evaluation results of the operating principle requirements generated in the Zeta, Delta, VTOL, and Epsilon projects. 199
7.4 Results of the error reason classification in the Zeta, Delta, VTOL and Epsilon projects. 200
7.5 Results of the completeness evaluation comparing the model-based stakeholder value specification from the Epsilon projects with the document-based legacy requirements the project used as input. 202

7.6 Results from the interviews with the modelers in the project to determine the reasons for not modeling the requirement and to evaluate if the generated natural language requirement correctly covers the legacy specification. 203

8.1 Impact on organization structure. 214
8.1 Impact on organization structure (cont.). 215
8.2 An example BDD modeling the sLRA of a vehicle using a reduced selection of possible subsystems. 219
8.3 An example STM modeling the bLRA of a propulsion system. 225
8.4 An example IBD modeling the logical architecture of a simplified vehicle system. 230
8.5 Transformation rules from operating principles to logical architectures. . . 231
8.5 Transformation rules from operating principles to logical architectures (cont.). 232
8.5 Transformation rules from operating principles to logical architectures (cont.) 233
8.6 A straightforward transformation that inserts the names of the systems and ports of the internal block diagram elements in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1). 237
8.7 A straightforward transformation that inserts the names of the systems and states of a state machine diagram in the corresponding fields of the SPECTRE template (*cf.* Figure 5.1). 238
8.8 Correctness evaluation results of the logical architecture requirements generated in the Zeta, Delta, VTOL, and Epsilon projects. 240
8.9 Results of the error reason classification in logical architecture requirements in the Zeta, Delta, VTOL and Epsilon projects. 251
8.10 Correctness evaluation results of the technical architecture requirements generated in the VTOL, and Epsilon projects. 252
8.11 Results of the error reason classification in technical architecture requirements in the VTOL and Epsilon projects. 252
8.12 Results of the completeness evaluation of the technical architecture requirements comparing the model-based stakeholder value specification from the Epsilon projects with the document-based legacy requirements the project used as input. 253
8.13 Results from the interviews with the modelers in the Epsilon project to determine the reasons for not modeling the requirement and to evaluate if the generated natural language requirement correctly covers the legacy specification. 254

9.1 Evaluation results of the requirements generated in the Zeta, Delta, VTOL, and Epsilon projects. 262

9.2 Project-independent evaluation results of the generated requirements: Total 2790. 263

C.1 The transformation rule for the initial action (Initial Rule). 320
C.2 The transformation rule for action execution (Execution Rule). 320
C.3 The transformation rule for sequential actions (Sequential Rule). 321
C.4 The transformation rule for external inputs (External Input Rule). 321
C.5 The transformation rule for external outputs (External Output Rule). . . . 322
C.6 The transformation rule for internal flows (Internal Flow Rule). 322
C.7 The transformation rule for the final actions (Final Rule). 323
C.8 The transformation rule for the final actions (Flow Final Rule). 323
C.9 The transformation rule for parallel actions (Parallel Rule). 324
C.10 The transformation rule for alternative actions (Alternative Rule). 325
C.11 The transformation rule for send signal actions (Send Signal Rule). 326
C.12 The transformation rule for accept event actions (Accept Event Rule). . . . 326
C.13 The transformation rule for accept event timer actions (Accept Event Timer Rule). 327
C.14 The transformation rule for interruptable action regions (Interruptable Action Rule). 328

Listings

5.1 The SpectreText grammar based on [Kat21]. 106
5.2 SpectreSentence grammar based on [Kat21] 107
5.3 The SpectreMainSentence grammar based on [Kat21]. 107
5.4 SpectreVerbalPhrase grammar based on [Kat21] 107
5.5 SpectreName grammar based on [Kat21] 108
5.6 The SpectreSubstantives grammar. 108
5.7 The SpectreSubstantive grammar based on [Kat21]. 108
5.8 SpectreConstraint grammar based on [Kat21] 109
5.9 The SpectreCondition grammar based on [Kat21]. 109
5.10 SpectreNoWordConstraintAndSpectreOfConstraint grammar based on [Kat21] 109
5.11 SpectreComparisonOperatorConstraint grammar based on [Kat21]. 110
5.12 The SpectreWithAMinimumAndMaximumOfConstraint grammar based on [Kat21]. 110

6.1 Original MontiCore grammar for use case diagrams from [KRW22]. 141
6.2 Extenstion of the use cased diagram grammar based on a proposal of the concrete textual syntax from [Bla23]. 142
6.3 An example use case diagram representing the example from Figure 6.1 in textual notation. 142

7.1 An example use case scenario representing table from Table 7.6 in textual notation. 190

B.1 The Action grammar. 307
B.2 The Condition grammar. 308
B.3 The Constraint grammar. 309
B.4 The Logical Expression grammar. 310
B.5 The SPECTREName grammar. 311
B.6 The Sentence grammar. 312
B.7 The Substantive grammar. 313
B.8 The Unit grammar. 314
B.9 The Variable grammar. 314
B.10 The WordBasis grammar. 315

C.1 ActivityDiagram grammar based on [Bla23]. 329
C.2 Use case scenario template (UCST) grammar based on [Bla23]. 331

List of Tables

2.1 A Systems Engineering Methodology Comparison. ● = fulfilled, ◐ = partly fulfilled, ○ = not fulfilled, 🚗 = Automotive, ✈ = Avionics, 🛏 = Embedded, 🌍 = General, ⚙ = Mechatronic. 42

4.1 Selected industry projects and their decomposition method, viewpoints, and modeling paradigm. ● = followed/created, ◐ = partly followed/created, ○ = not followed/created, 🎛 = ADAS, 🚗 = Vehicle, ⚙ =Powertrain, AS = Asian OEM, EU = European Union, GE = German OEM. . 87

5.1 Requirement diagram elements required to describe natural language requirements according to the CUBE methodology. 99
5.14 Requirement preprocessing for the SPECTRE-DSL evaluation. 113
5.15 Overall evaluation results of the requirements in the Alpha and Beta projects using the SPECTRE-DSL. 113
5.16 Manual evaluation results of the requirement subsets extracted from the Alpha and Beta project requirements. 117

6.1 UCD elements required to describe the stakeholder values according to the CUBE methodology. 127
6.1 UCD elements required to describe the stakeholder values according to the CUBE methodology. 128
6.2 Further UCD elements to aid in a feature definition based on stakeholder values according to feature-driven CUBE methodology. 147
6.3 Further UCD elements to aid in a feature definition based on stakeholder values according to feature-driven CUBE methodology. 150
6.3 Further UCD elements to aid in a feature definition based on stakeholder values according to feature-driven CUBE methodology. 151
6.4 Requirement rating scheme based on the evaluation scheme use in [BBK+22b, BBK+22a]. 162
6.5 Experimentally determined parameters for calculating the process times for automated and manual creation of textual stakeholder value requirements from use case diagrams. 171

6.6 Estimation of required times to manually/automatically derive stakeholder value requirements, their review, correction, and regeneration and re-review for generated requirements. 172

7.1 AD elements required to describe the operating principles according to the CUBE methodology. 176
7.1 AD elements required to describe the operating principles according to the CUBE methodology. 177
7.1 AD elements required to describe the operating principles according to the CUBE methodology. 178
7.1 AD elements required to describe the operating principles according to the CUBE methodology. 179
7.1 AD elements required to describe the operating principles according to the CUBE methodology. 180
7.2 AD elements required to allocate actions to the logical element responsible for the execution according to the CUBE methodology. 181
7.2 AD elements required to allocate actions to the logical element responsible for the execution according to the CUBE methodology. 182
7.3 A tabular view to express the use case behavior inspired by [Coc98]. . . . 185
7.4 A tabular view to express the desired behavior as an action sequence during the use case execution. 186
7.5 A tabular use case specification of the feature operating principle ‘Transport persons and goods’ Figure 7.1a presents in the form of an activity diagram. 187
7.6 A tabular use case specification of the operating principle for the use case ‘Drive the Vehicle’ Figure 7.1b presents in the form of an activity diagram. 193
7.7 AD requirement generation templates to derive natural language requirements for functionality and interface requirements from activity diagram models based on [Zab23]. 195
7.7 AD requirement generation templates to derive natural language requirements for functionality and interface requirements from activity diagram models based on [Zab23]. 196

8.1 Different possibilities to define vehicle subsystems of a vehicles LRA using the example of vehicle propulsion. 211
8.2 BDD elements required to describe the logical architecture of a system. . 216
8.2 BDD elements required to describe the logical architecture of a system. . 217
8.3 STM elements required to describe the behavioral part of the logical architecture of a system. 222
8.3 STM elements required to describe the behavioral part of the logical architecture of a system. 223

8.3 STM elements required to describe the behavioral part of the logical architecture of a system. 224
8.4 IBD elements required to describe the logical architecture of a system. . . 228
8.4 IBD elements required to describe the logical architecture of a system. . . 229
8.5 Number of elements modeled and transformed for the 16 features modeled in the automotive development project as presented in [GKM+25] 245

9.1 Number of requirements generated on the abstraction levels of the CUBE in the analyzed projects. 260
9.2 Correctness evaluation results according to the evaluation scheme from Table 6.4 in the analyzed projects. 261

A.1 Explanation of the used tags in listings and figures. 305
A.2 Explanation of the used stereotypes in listings and tags. 305

C.1 A template for use case behavior specification elicitation. 319
C.2 A tabular use case specification of the use case 'Enter the Vehicle' form the stakeholder value in Figure 6.1. 334
C.3 A tabular use case specification of the use case 'Exit the Vehicle' form the stakeholder value in Figure 6.1. 335
C.4 A tabular use case specification of the use case 'Unlock the Vehicle' form the stakeholder value in Figure 6.1. 336
C.5 A tabular use case specification of the use case 'Lock the Vehicle' form the stakeholder value in Figure 6.6. 337
C.6 A tabular use case specification of the feature 'Open or Close Doors' form the stakeholder value in Figure 6.6. 338
C.7 A tabular use case specification of the feature 'Lock or Unlock Doors' form the stakeholder value in Figure 6.6. 339

D.1 UCD requirement generation templates to derive natural language requirements from use case models. 341
D.1 UCD requirement generation templates to derive natural language requirements from use case models. 342
D.1 UCD requirement generation templates to derive natural language requirements from use case models. 343
D.1 UCD requirement generation templates to derive natural language requirements from use case models. 344
D.1 UCD requirement generation templates to derive natural language requirements from use case models. 345
D.1 UCD requirement generation templates to derive natural language requirements from use case models. 346

D.1 UCD requirement generation templates to derive natural language requirements from use case models. . . . 347
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 348
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 349
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 350
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 351
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 352
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 353
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 354
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 355
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 356
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 357
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 358
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 359
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 360
D.2 AD requirement generation templates to derive natural language requirements from use case models. . . . 361
D.3 Requirement generation templates to derive natural language requirements for functionality and interface requirements from activity diagram models based on [Zab23]. . . . 362
D.4 STM requirement generation templates to derive natural language requirements from state machines. . . . 363
D.4 STM requirement generation templates to derive natural language requirements from state machines. . . . 364
D.4 STM requirement generation templates to derive natural language requirements from state machines. . . . 365

Related Work from the SE Group, RWTH Aachen, June 25

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], 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, HKR21] 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]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

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 the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able, e.g., to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was

introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

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, [Hab16] for MontiArc is used in domains such as cars or robotics [HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and, e.g., in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.

[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], 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) & the UML-P Tool

Starting with the 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 books [Rum16, Rum17] and is 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" [BCGR09], [BCGR09a], [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 [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are 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+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].

The question of 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 [EFLR99a], [FEL+98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also trans-

ferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a, HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10]. [BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented, e.g., in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a 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+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a

specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a 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+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

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 [GKR+96], and components [BR07] as well as expressive forms of composition and refinement [PR99, PR97, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to

the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15a]) 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, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing 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, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. Furthermore, [BKRW17] compares component and connector architectures similar to SysML’ block definition diagrams and [RSR+99] discusses the combination of those architectures with the UML.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] 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 to class and object diagrams in [MRR11f] 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 the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts 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 [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, PR97, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].

[Rum04c] 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] and refining pipe-and-filter architectures is explained in [PR99, PR97]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and 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, GKPR08] 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 [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] 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 to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

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, GKR+96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] 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 [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include

DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

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 the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

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 requirement management connecting requirements with features in all development phases 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]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].

[MMR10] gives an overview of the 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.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at dif-

ferent scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the 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 validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

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, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [PR13]. At 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 development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital

twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse.* Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[AHRW17] 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.

[AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM, October 2021.

[AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[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)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Journal Frontiers in Neuroinformatics*, 12, 2018.

[BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.

[BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.

[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.

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.

[BCGR09] 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.

[BCGR09a] 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.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas

Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things*, 3:1–32, February 2022.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19(3):3:1–16, October 2020.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software (JSS)*, 152:50–69, June 2019.

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BGH+97] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, LNCS 1357. Springer Verlag, 1997.

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Journal Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGH+98b] 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.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects.

In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[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.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.

[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.

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.

[BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.

[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.

[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 *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle The-*

ory for distributed, interactive systems - the untimed case. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)*, 149:437–461, March 2019.

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.

[BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12b] 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.

[BR12] 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.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.

[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+15a] 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 H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu,

Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[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.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[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. *Journal on Software: Practice and Experience*, 49(2):301–328, February 2019.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology (JOT)*, 20:1–24, November 2021.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Com-*

puter Languages (COLA), 70, June 2022.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.

[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.

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Journal 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.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] 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.

[FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.

[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.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Com-

puter and Information Science 580, pages 112–132. Springer, 2015.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects.* Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[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.

[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.

[GKR+96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State Transition Diagrams. Technical report, TU Munich, 1996.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Hei-

delberg, April 2015.

[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.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[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.

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

[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.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009.

[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.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy*. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.

[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+15] 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+15a] 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. *Journal Future Generation Computer Systems*, 56:701–718, 2015.

[HHR+15] 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)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.

[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.

[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)*, LNCS 9839, pages 175–182. Springer, December 2016.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July 2019.

[HNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 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.

[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.

[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, Informatik-Bericht 2010-01, 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.

[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.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures*, 54:386–405, 2018.

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology (JOT)*, 21:1–15, July 2022.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software Journal*, 35(6):40–47, 2018.

[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.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic

Architectures of Self-Adaptive Cooperative Systems. *Journal of Object Technology (JOT)*, 18(2):1–20, July 2019.

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Journal ACM Transactions on Internet of Things*, 3:1–30, November 2022.

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.

[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.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM, October 2021.

[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.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe,

and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[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.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[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.

[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.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings

2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRW17] 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.

[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.

[KRS+18a] 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)*, pages 447 – 457. ACM, October 2018.

[KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)*, 183:1–21, January 2022.

[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.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal*, 27:119–151, April 2020.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker

Verlag, November 2021.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[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.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer Journal*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. Model centered architecture. In *Conceptual Modeling Perspectives*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM*

International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems, pages 9–12. ACM, May 2022.

[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.

[MRR11d] 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.

[MRR11a] 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.

[MRR11e] 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.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11c] 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.

[MRR11f] 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.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 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.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *International Conference on Software Engineering (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] 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)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials.* Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[NRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[PR13] 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.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.

[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.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen.* Aachener Informatik-

Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[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.

[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.

[PR97] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97 Proceedings*, Hiroshima, Japan, 1997. IEEE CS Press.

[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.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 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.

[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.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15b] 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.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[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.

[RRW13] 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.

[RRW13c] 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.

[RRW14a] 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.

[RSR+99] Bernhard Rumpe, M. Schoenmakers, Ansgar Radermacher, and Andy Schürr. UML + ROOM as a Standard ADL? In Frances M. Titsworth, editor, *Engineering of Complex Computer Systems, ICECCS'99 Proceedings*, IEEE Computer Society, 1999.

[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.

[Rum04c] 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.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.

[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.

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert

Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021.

[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.

[Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[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.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021.

[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.