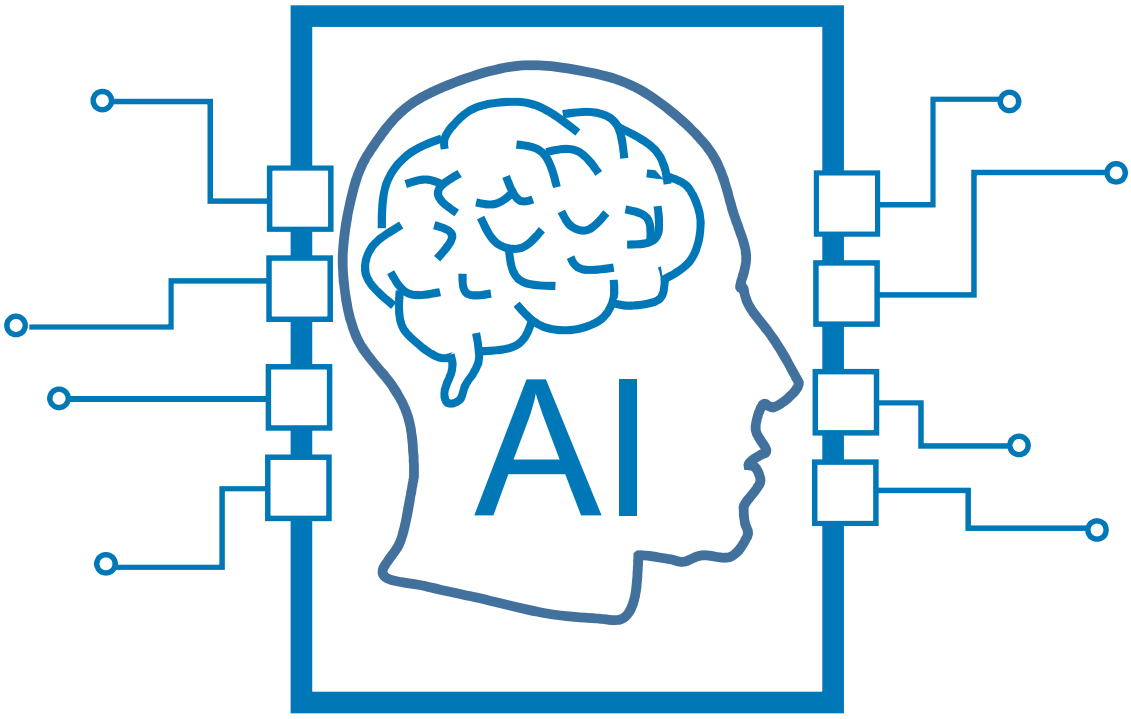
SE | RWTH AACHEN UNIVERSITY

Evgeny Kusmenko

Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems



EMADL

Aachener Informatik-Berichte,
Software Engineering
Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Band 49

Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems

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

vorgelegt von

Dipl.-Ing.
Evgeny Kusmenko
aus Odessa, Ukraine

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Uwe Aßmann

Tag der mündlichen Prüfung: 19. Juli 2021

[Kus21] E. Kusmenko:
Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems
In: Aachener Informatik-Berichte, Software Engineering, Band 49, ISBN 978-3-8440-8286-9
Shaker Verlag, November 2021.
www.se-rwth.de/publications/

D 82 (Diss. RWTH Aachen University, 2021)

Eidesstattliche Erklärung

Ich, Evgeny Kusmenko, erkläre 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:

   - [KPRS19] E. Kusmenko, S. Pavlitskaya, B. Rumpe, S. Stüber: On the Engineering of AI-Powered Systems. In: ASE'19. Software Engineering Intelligence Workshop (SEI'19), San Diego, pp. 126-133, IEEE, Nov. 2019.
   - [KNP+19] E. Kusmenko, S. Nickels, S. Pavlitskaya, B. Rumpe, T. Timmermanns: Modeling and Training of Neural Processing Systems. In: Conference on Model Driven Engineering Languages and Systems (MODELS'19), Munich, pp. 283-293, IEEE, Sep. 2019.
   - [GKR19] N. Gatto, E. Kusmenko, B. Rumpe: Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In: Proceedings of MODELS 2019. Workshop MDE Intelligence, Munich, pp. 196-202, IEEE, Sep. 2019.
   - [KKRZ19] J. C. Kirchhof, E. Kusmenko, B. Rumpe, H. Zhang: Simulation as a Service for Cooperative Vehicles. In: Proceedings of MODELS 2019. Workshop MASE, Munich, pp. 28-37, IEEE, Sep. 2019.

- [KKMR19] J. C. Kirchhof, E. Kusmenko, J. Meurice, B. Rumpe: Simulation of Model Execution for Embedded Systems. In: Proceedings of MODELS 2019. Workshop MLE, Munich, pp. 331-338, IEEE, Sep. 2019.
- [KKR19] N. Kaminski, E. Kusmenko, B. Rumpe: Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. In: The Journal of Object Technology, 18(2), pp. 1-20, AITO, July 2019.
- [HKKR19] A. Hellwig, S. Kriebel, E. Kusmenko, B. Rumpe: Component-based Integration of Interconnected Vehicle Architectures. In: 30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles, Paris, pp. 146-151, IEEE, June 2019.
- [KKRS19] S. Kriebel, E. Kusmenko, B. Rumpe, I. Shumeiko: Learning Error Patterns from Diagnosis Trouble Codes. In: 30th Intelligent Vehicles Symposium (IV'19). Workshop on Unsupervised Learning for Automated Driving, Paris, pp. 246-251, IEEE, June 2019.
- [DGH+19] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, A. Wortmann: SMArDT Modeling for Automotive Software Testing. In: Software: Practice and Experience, 49(2):301-328, Wiley Online Library, Feb. 2019.
- [FIK+18] C. Frohn, P. Ilov, S. Kriebel, E. Kusmenko, B. Rumpe, A. Ryndin: Distributed Simulation of Cooperatively Interacting Vehicles. In: International Conference on Intelligent Transportation Systems (ITSC'18), pp. 596-601. IEEE, Hawaii, 2018.
- [KRSvW18a] E. Kusmenko, B. Rumpe, S. Schneiders, M. von Wenckstern: Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In: Conference on Model Driven Engineering Languages and Systems (MODELS'18), pp. 447-457, Copenhagen, ACM, Oct. 2018.
- [KRSvW18b] E. Kusmenko, B. Rumpe, I. Strepkov, M. von Wenckstern: Teaching Playground for C&C Language EmbeddedMontiArc. In: Proceedings of MODELS 2018. Workshop ModComp, Copenhagen, Oct. 2018.
- [KRRvW18] E. Kusmenko, J. Ronck, B. Rumpe, M. von Wenckstern: EmbeddedMontiArc: Textual Modeling Alternative to Simulink. In: Proceedings of MODELS 2018. Workshop EXE, Copenhagen, Oct. 2018.
- [BKL+18] C. Brecher, E. Kusmenko, A. Lindt, B. Rumpe, S. Storms, S. Wein, M. von Wenckstern, A. Wortmann: Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models.

International Symposium on Computer Science and Intelligent Control (ISCSIC'18). Stockholm, ACM, Sep. 2018.

- [KKRvW18] S. Kriebel, E. Kusmenko, B. Rumpe, M. von Wenckstern: Finding Inconsistencies in Design Models and Requirements by Applying the SMARDT Process. In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18). Univ. Hamburg, Apr. 2018.
- [KSRvW18] E. Kusmenko, I. Shumeiko, B. Rumpe, M. von Wenckstern: Fast Simulation Preorder Algorithm. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pp. 256-267. Funchal, Portugal, SciTePress, Jan. 2018.
- [HKK+18] S. Hillemacher, S. Kriebel, E. Kusmenko, M. Lorang, B. Rumpe, A. Sema, G. Strobl, M. von Wenckstern: Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pp. 163-178. Funchal, Portugal, SciTePress, Jan. 2018.
- [GKR+17] F. Grazioli, E. Kusmenko, A. Roth, B. Rumpe, M. von Wenckstern: Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In: Proceedings of MODELS 2017. Workshop EXE, Austin, CEUR 2019, Sept. 2017.
- [KRRvW17] E. Kusmenko, A. Roth, B. Rumpe, M. von Wenckstern: Modeling Architectures of Cyber Physical Systems. In: European Conference on Modelling Foundations and Applications (ECMFA'17), Marburg, pp. 34-50. LNCS 10376, Springer, July 2017.
- [DDE+17] J. Dankert, C. Dernehl, L. Eckstein, S. Kowalewski, E. Kusmenko, B. Rumpe: RapidCoop - Robuste Architektur durch geeignete Paradigmen für Kooperativ Interagierende Automobile. In: Automatisiertes und Vernetztes Fahren (AAET'17), Feb. 2017.

Aachen, den 27.04.2021

Evgeny Kusmenko

Abstract

The development of cyber-physical systems poses a multitude of challenges requiring experts from different fields. Such systems cannot be developed successfully without the support of appropriate processes, languages, and tools. Model-driven software engineering is an important approach which helps development teams to cope with the increasing complexity of today's cyber-physical systems. The aim of this thesis is to develop a model-driven engineering methodology with a particular focus on interconnected intelligent cyber-physical systems such as cooperative vehicles.

The basis of the proposed methodology is a component-and-connector architecture description language focusing on the decomposition and integration of cyber-physical system software. It features a strong, math-oriented type system abstracting away from the technical realization and incorporating physical units. To facilitate the development of highly-interconnected self-adaptive systems, the language enables its users to model component and connector arrays and supports architectural runtime-reconfiguration. Architectural elements can be altered, added, and removed dynamically upon the occurrence of trigger events.

In order to fully cover the development process, the proposed methodology, in addition to structural modeling, provides means for behavior specification and its seamless integration into the components of the architecture. A matrix-oriented scripting language enables the developer to specify algorithms using a syntax close to the mathematical domain. What is more, a dedicated deep learning modeling language is provided for the development and training of neural networks as directed acyclic graphs of neuron layers. The framework supports different learning methods including supervised, reinforcement, and generative adversarial learning, covering a broad range of applications from image and natural language processing to decision making and test data generation.

The presented toolchain enables an automated generation of fully functional C++ code together with the corresponding build and training scripts based on the architectural models and behavior specifications. Finally, to facilitate the integration and deployment of the modeled software in distributed environments, we use a tagging approach to model the middleware and to control a middleware generation toolchain.

Kurzfassung

Die Entwicklung cyber-physischer Systeme stellt eine Vielzahl von Herausforderungen und erfordert Experten aus unterschiedlichen Bereichen. Solche Systeme können nicht ohne die Unterstützung durch geeignete Prozesse, Sprachen und Tools erfolgreich entwickelt werden. Modellgetriebenes Software Engineering stellt einen wichtigen Ansatz dar, der Entwicklungsteams hilft, die zunehmende Komplexität heutiger cyber-physischer Systeme zu bewältigen. Das Ziel dieser Arbeit besteht darin, eine modellgetriebene Engineering-Methodik mit besonderem Fokus auf vernetzte intelligente cyber-physische Systeme wie kooperative Fahrzeuge zu entwickeln.

Die Grundlage der vorgestellten Methodik bildet eine Komponenten- und Konnektoren-basierte Architekturbeschreibungssprache zur Dekomposition und Integration von Software für cyber-physische Systeme. Diese verfügt über ein starkes statisches, mathematisch orientiertes Typsystem, welches physikalische Einheiten unterstützt und von der technischen Realisierung abstrahiert. Um die Entwicklung hochvernetzter selbstadaptiver Systeme zu erleichtern, ermöglicht die Sprache die Modellierung von Komponenten- und Konnektorarrays und unterstützt Laufzeit-Rekonfigurationen der Architektur. Architekturelemente können dabei ereignisbasiert dynamisch geändert, hinzugefügt und entfernt werden.

Um den Entwicklungsprozess vollständig abzudecken, bietet die vorgestellte Methodik neben der strukturellen Modellierung Mittel zur Verhaltensspezifikation und deren nahtlose Integration in die Komponenten der Architektur. Eine matrixorientierte Skriptsprache ermöglicht es dem Entwickler, Algorithmen in einer Syntax zu spezifizieren, die der mathematischen Domäne sehr nahe kommt. Darüber hinaus wird eine dedizierte Deep-Learning-Modellierungssprache für die Entwicklung und das Training von neuronalen Netzen in Form von azyklischen, aus Neuronenschichten bestehenden Graphen bereitgestellt. Das Framework unterstützt verschiedene Lernmethoden wie überwachtes, verstärkendes sowie GAN-basiertes Lernen und deckt damit ein breites Anwendungsspektrum von der Bild- und natürlichen Sprachverarbeitung bis hin zur Entscheidungsfindung und Testdatengenerierung ab.

Auf Basis der Architekturmodelle und Verhaltensspezifikationen erlaubt die vorgestellte Toolchain eine automatisierte Generierung von voll funktionsfähigem C++-Code zusammen mit den entsprechenden Build- und Trainingsskripten. Um die Integration und Bereitstellung der modellierten Software für verteilte Umgebungen zu erleichtern, verwenden wir schließlich einen Tagging-Ansatz zur Modellierung und Generierung von Middleware.

Acknowledgements

Ich möchte mich ganz herzlich bei meinem Doktorvater, Prof. Dr. Bernhard Rumpe, bedanken, der mir die Möglichkeit gegeben hat, am Lehrstuhl für Software Engineering der RWTH Aachen University zu promovieren und mich wissenschaftlich, aber auch persönlich zu entfalten. Für die zahlreichen Diskussionen, Ideen und Sichtweisen. Und für die interessanten Aufgaben, spannenden Projekte und Herausforderungen, an denen ich arbeiten durfte und die immer Lust auf mehr gemacht haben.

Ich danke Prof. Dr. Aßmann für die Zweitbegutachtung meiner Dissertation, aber auch dafür, während meiner Studienzeit an der TU Dresden als Dozent mein besonderes Interesse für das Gebiet Software Engineering geweckt zu haben. Des Weiteren bedanke ich mich bei Prof. Dr. Leibe für die Leitung meines Prüfungskomittees sowie bei Prof. Dr. Erika Ábrahám für die Abnahme der mündlichen Prüfung im Bereich der theoretischen Informatik.

Ganz besonders möchte ich mich auch bei meinen Kolleginnen und Kollegen vom Lehrstuhl für Software Engineering für eine produktive und harmonische Zusammenarbeit, interessante Diskussionen und eine schöne Zeit am Lehrstuhl bedanken. Mein Dank gilt Dr. Michael von Wenckstern für die enge Zusammenarbeit an EmbeddedMontiArc und vielen gemeinsamen Papieren, vor allem aber auch für die zahlreichen tollen Erlebnisse und Erinnerungen aus unserer gemeinsamen Promotionszeit; Jun.-Prof. Dr. Andreas Wortmann, der mich während meiner Promotionszeit von seiner Erfahrung profitieren ließ, die gemeinsame spannende Zeit in Schweden zu Beginn meiner Promotion sowie eine gute Zeit bei Konferenzen, die wir gemeinsam besucht haben; Dr. Judith Michael für die vielen Diskussionen, Anregungen und Ideen und ihr immer offenes Ohr; Dr. Katrin Hölldobler für ihre Unterstützung rund um MontiCore und die Gradle-Integration; Imke Drave für die produktive Zusammenarbeit im Bereich Automotive, insbesondere an unseren KI-lastigen Projekten und die KI-Diskussionen, aber auch dafür, die SE Runners wiederbelebt zu haben; Christian Kirchhof für seine Beiträge zu MontiSim, sein Engagement im Bereich Deep Learning für IoT und die gemeinsamen Papiere; Steffen Hillemacher für seine Unterstützung im Bereich Lehre und die Diskussionen rund um das Artefaktenmodell; Malte Heithoff für die spannende Zusammenarbeit an EmbeddedMontiArc und die gemeinsame Veröffentlichung in den letzten Zügen meiner Promotion, aber auch für seine tatkräftige Unterstützung bei der Umsetzung der SI Units; Matthias Markthaler für die interessanten Einblicke in die BMW-Welt und seine Unterstützung während meiner Projektaufenthalte in München. Deni Raco für die kurzweiligen Pokerturniere und Kickerpartien sowie die produktive Zusammenarbeit, vor allem an Avionics-Themen. Marita Breuer und Galina Volkova für die technische Unterstützung rund um MontiCore, Nexus, und eine Vielzahl anderer Themen. Sylvia Gunder und Sonja Müßigbrodt, die bei jeder Frage und Problemstellung immer wussten, was zu tun ist. Mein Dank gilt insbesondere auch Kai Adam, Vincent Bertram, Arvid

Butting, Joel Charles, Manuela Dalibor, Arkadii Gerasimov, Dr. Timo Greifenberg, Nico Jansen, Dr. Oliver Kautz, Achim Lindt, Dr. Markus Look, Dr. Klaus Müller, Pedram Nazari, Lukas Netz, Alexander Roth, David Schmalzing, Steffi Schrader, Dr. Christoph Schulze, Igor Shumeiko, Sebastian Stüber, Simon Varga, Louis Wachtmeister, Vassily Aliseyko, Lennart Bucher, Niklas Dienstknecht, Annika Donath, Christoph Engels, Joshua Mingers, Jerome Pfeiffer, Nina Pichler, Manuel Pützer, Brian Sinkovec und Max Voß.

Ferner bedanke ich mich bei den laufaffinen Kollegen Michael von Wenckstern, Alexander Roth, Pedram Nazari, Imke Drave, Malte Heithoff, Steffen Hillemacher, Matthias Markthaler und Lukas Netz für die zahlreichen zusammen zurückgelegten Kilometer, wenn es mal wieder Zeit war abzuschalten.

Für das Korrekturlesen und sehr hilfreiches Feedback zu meiner Dissertation bedanke ich mich bei Arvid Butting, Arkadii Gerasimov, Manuela Dalibor, Imke Drave, Malte Heithoff, Steffen Hillemacher, Nico Jansen, Oliver Kautz, Christian Kirchhof, Lukas Netz, David Schmalzing, Sebastian Stüber, Marcos Sullivan und Simon Varga.

Ferner bedanke ich mich bei meinen ehemaligen Bacheloranden und Masteranden, die mich tatkräftig bei der Erforschung spannender Themen unterstützt haben und ohne die an eine so komplexe Toolchain wie EmbeddedMontiArc mitsamt Simulator und Deep Learning Framework nicht zu denken wäre.

Mein ganz besonderer Dank gilt meiner Mutter Janna. Danke, dass du mir diesen Weg ermöglicht, mich ausnahmslos bei allen meinen Vorhaben unterstützt und mich stets motiviert hast nach den Sternen zu greifen.

Von ganzem Herzen bedanke ich mich auch bei meiner Freundin Daria für ihre Unterstützung und Motivation während meiner Promotionszeit. Danke für die Geduld, die du während der letzten Jahre aufbringen musstest, wenn die Dissertation alles um mich herum zu verdrängen schien. Danke auch für das Korrekturlesen zahlreicher meiner Papiere und meiner Dissertation.

Aachen, Oktober 2021
Evgeny Kusmenko

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Research Questions . . . 2
- 1.3 CPS Basics . . . 2
  - 1.3.1 Autonomous Driving Architectures . . . 2
  - 1.3.2 Control . . . 4
  - 1.3.3 Machine Learning in Autonomous Driving . . . 4
- 1.4 Requirements . . . 5
- 1.5 Thesis Structure . . . 6
- 1.6 Publications . . . 7
- 1.7 Preliminaries . . . 10
  - 1.7.1 Notation . . . 10
  - 1.7.2 Model-Driven Engineering and Domain Specific Languages . . . 12
  - 1.7.3 The MontiCore Language Workbench . . . 13
  - 1.7.4 SI Units . . . 15
  - 1.7.5 Model-Driven Engineering Processes in the Automotive Domain . . . 17
  - 1.7.6 Component & Connector Modeling . . . 19

2 EmbeddedMontiArc **23**
- 2.1 Requirements . . . 23
- 2.2 The Data Type System . . . 24
  - 2.2.1 Primitive Data Types . . . 24
  - 2.2.2 Vectors, Matrices and Cubes . . . 26
  - 2.2.3 Matrix Properties . . . 26
  - 2.2.4 SI Units in EmbeddedMontiArc . . . 29
  - 2.2.5 Structs and Enums . . . 29
- 2.3 Static Architecture Description . . . 31
  - 2.3.1 Components, Ports and Connectors . . . 31
  - 2.3.2 Arrays and Connection Patterns . . . 33
  - 2.3.3 Execution Semantics . . . 35
- 2.4 MontiMath . . . 40
  - 2.4.1 Basic Syntax . . . 40
  - 2.4.2 Deriving Matrix Properties for Concrete Matrices . . . 42
  - 2.4.3 Deriving Matrix Properties for Operations . . . 45

2.4.4 EmbeddedMontiArcMath . . . 46
2.4.5 Optimization in MontiMath . . . 47
2.4.6 Component Variabilty for Self-Adaptable Systems . . . 49
2.5 Code Generator . . . 53
2.6 Model-Driven Unit-Testing . . . 56
2.6.1 The Stream Language . . . 56
2.6.2 The Maven Streamtest Plugin . . . 58
2.6.3 Simulation . . . 59

3 Dynamics Aspects of EmbeddedMontiArc 63
3.1 Cooperative Agents . . . 63
3.2 Background & Requirements . . . 65
3.3 Dynamic ADLs . . . 68
3.3.1 Brief Overview . . . 68
3.3.2 Requirements Assessment . . . 72
3.4 EmbeddedMontiArc Dynamics . . . 73
3.4.1 EMAD Execution Semantics . . . 74
3.4.2 Data-Triggered Internal Reconfiguration . . . 74
3.4.3 Service-Based External Reconfiguration . . . 79
3.4.4 Modeling Component Pipelines . . . 86
3.4.5 Reconfiguration Views and Graphical Notation . . . 89
3.4.6 Remarks on Architectural Consistency . . . 90
3.5 Conclusion . . . 93

4 Modeling Artificial Neural Networks with MontiAnna 95
4.1 Deep Learning for Autonomous Systems . . . 95
4.1.1 Supervised Machine Learning Foundations . . . 95
4.1.2 Neural Networks . . . 96
4.1.3 Training of Layered Neural Networks . . . 99
4.1.4 Deep Network Architectures . . . 101
4.2 Requirements of a Deep Learning Modeling Framework for Cyber-Physical Systems . . . 106
4.3 Overview of Deep Learning Frameworks . . . 108
4.4 Machine Learning Modeling Frameworks . . . 112
4.5 The MontiAnna Framework . . . 116
4.6 An Overview of Modeling Languages . . . 117
4.6.1 The Compiler Toolchain . . . 119
4.6.2 The Generated Artifacts . . . 120
4.7 Modeling Feedforward Neural Architectures with CNNArc . . . 121
4.7.1 Defining a Stand-Alone Network . . . 121
4.7.2 Modeling Layers and Networks . . . 122

4.7.3 Code Reuse in CNNArc . . . 128
4.8 Modeling Recurrent Neural Networks . . . 134
4.8.1 Basic Concepts . . . 134
4.8.2 Modeling an Encoder-Decoder Network . . . 135
4.9 Modeling Training . . . 140
4.9.1 The Composed Model . . . 143
4.10 EmbeddedMontiArcDL . . . 144
4.10.1 CNNArc as Implementation Language for EmbeddedMontiArc Components . . . 145
4.10.2 Modeling the Dataset . . . 146
4.10.3 The MNISTCalculator Example . . . 147
4.10.4 Modeling a Direct Perception Autonomous Vehicle Controller . . . 151

5 Modeling Deep Reinforcement Learning Architectures 157
5.1 Foundations of Deep Reinforcement Learning . . . 157
5.2 Requirements . . . 159
5.3 Related Work . . . 160
5.4 Modeling Reinforcement Learning . . . 164
5.5 Modeling the Function Approximators . . . 164
5.6 Modeling the Training . . . 168
5.6.1 General Reinforcement Learning Parameters . . . 168
5.6.2 The TORCS Training Model . . . 171
5.6.3 Reward Function . . . 173
5.7 Environment . . . 175
5.8 Code Generation . . . 177
5.9 Modeling Generative Adversarial Networks . . . 179
5.10 Evaluation . . . 180
5.10.1 TORCS and Open AI Gym . . . 180
5.10.2 Decision Making in Forestry 5.0 . . . 183
5.11 Conclusion and Future work . . . 188

6 Modeling Distributed Architectures 191
6.1 The Need for Distributed Systems . . . 191
6.2 Existing Approaches for Middleware Integration . . . 192
6.3 Running Example and Use Cases. . . . 194
6.4 Requirements . . . 196
6.5 Tagging-Based Middleware Modeling . . . 197
6.6 Code Generator Composition . . . 202
6.7 Evaluation . . . 210
6.8 Automating Model Slicing for Distributed Deployment . . . 211
6.8.1 Motivation . . . 211

6.8.2 Component Clustering . . . 212
6.8.3 Weights . . . 215
6.8.4 Encorporating Structural Constraints . . . 215
6.8.5 Related Work on Automated Deployment . . . 216
6.9 Conclusion and Future Work . . . 218

7 Conclusion **219**

Bibliography **223**

A Diagrams and Listings **251**

B Further Documentation **261**
B.1 CNNArc Layer Classes . . . 261
B.2 CNNTrain Evaluation Metrics . . . 265
B.3 CNNTrain for Reinforcement Learning . . . 266
B.3.1 General Reinforcement Learning Parameters . . . 266
B.3.2 DQN Exclusive Parameters . . . 268
B.3.3 TD3 Exclusive Parameters . . . 269
B.3.4 Training Results . . . 270
B.4 MontiCore 5 Grammars . . . 271

List of Figures **291**

List of Tables **297**

Chapter 1

Introduction

1.1 Motivation

In cyber-physical systems (CPSs) physical processes are highly intertwined with software components [Lee08]. CPSs such as automated vehicles, surgery assisting robots, and intelligent production plants are becoming more and more pervasive in a multitude of areas in industry and daily life. The development of such systems poses challenges which we do not encounter in pure software development or classical mechanical engineering [LAB+11, YCC+17, PWH+18]. This leads to the necessity of appropriate development processes, languages and tools helping us to cope with the increasing system complexity. The heterogeneity and complexity of CPSs requires a multi-paradigm approach allowing experts from different domains to focus on specific modules while ensuring a seamless integration thereof. Typical modules indispensable in CPSs include but are not limited to: sensors and actuators interacting with the environment, detection and situation understanding components, planners, controllers, and the like. Usually these modules are part of a feedback loop where software components affect the mechanical system and the environment, and vice versa. The (de)composition of such systems and the analysis of their modules' interaction is an important basis for a successful CPS design [RW18]. For this reason architecture description languages (ADLs) have been particularly popular in domains like automotive and avionics [WM95, FLV06].

Furthermore, complex behavioral schemes require appropriate means for development and testing. CPS components need to be able to process large amounts of data and make intelligent decisions in real-time. Physical processes must be modeled along with high-performance tensor computations. More and more tasks are delegated to machine learning algorithms such as deep neural networks [GLSU13, CSKX15], which fundamentally changes the way software is written and thought about.

Obviously, a CPS development process must incorporate ideas and process approaches from different disciplines, including mechanical, control, and software engineering. The development approaches of these different worlds are by no means compatible. For instance, while mechanical engineers often deal with geometric models [VR77], e.g. using Computer-Aided Design (CAD) software such as AutoCAD or CATIA [DJM+19], computer scientists think in abstractions and prefer a logical decomposition, e.g. using the

unified modeling language (UML) [Rum11b]. While agile development methods have been successfully adopted in business and web software development, they can be difficult to implement in other engineering disciplines. A CPS development methodology must, hence, be able to link the different engineering disciplines while taking the best from all worlds. The aim of this thesis is to develop a model-driven methodology for the design of *cooperative* and *intelligent* cyber-physical systems.

1.2 Research Questions

The main research question to be answered in this thesis is:

Main Research Question. *How can the agile development of artificial intelligence for cooperating cyber-physical systems be supported by a model-driven engineering methodology?*

Moreover, the main research question can be subdivided into the following partial research questions:

RQ1 How can the complexity of cooperating cyber-physical systems be handled using an architecture-centric modeling approach?

RQ2 How can the heterogeneity of cyber-physical system software architectures be handled by a multi-paradigm modeling language family?

RQ3 How can machine learning-based methods be incorporated into cyber-physical software architectures within a model-driven engineering life-cycle?

1.3 CPS Basics

1.3.1 Autonomous Driving Architectures

To deliver a well-suited modeling methodology and the required domain-specific languages (DSLs), we first need to understand the nature of CPS architectures. For this purpose, we are going to revise architectures from the literature starting with four examples stemming from the DARPA Urban Challenge 2007. The Tartan Racing team placed first in the competition with their Boss robotic vehicle architecture [UBB+07]. By abstracting from the technical details, we can reduce the architecture to the following main building blocks: the mechanics (including a variety of sensors, actuators and a controller area network (CAN) bridge), perception and world modeling, mission planning, motion planning, and behavior generation. The perception and world modeling part uses signals sampled by the sensors to create a model of the vehicle's environment

and to understand what is happening. Having an understanding of the situation, the mission planner computes or updates a high level route, e.g. a route to the destination. The behavior generator breaks down this plan into a sequence of driving scenarios. The motion planning block realizes the driving scenarios by computing concrete trajectories, which in turn are fed into the vehicle controller. The latter drives the actuators closing the loop.

Junior, an autonomous vehicle by the University of Standford, placed second in the DARPA challenge [MBB+08]. The two core component groups are perception and navigation. Perception modules get their inputs from a sensor interface with a multitude of sensors and are responsible for localization, obstacle tracking, and pose estimation which resembles the perception module of Tartan. The navigation components unify top level control, path planning, and actuation control. The results of the latter are fed into the vehicle interface. A global services block contains further functionality for process control, logging, and interprocess communication. In case of malfunctioning modules, the health monitor can emit an emergency stop command.

The Odin robotic vehicle by VictorTango won the third place in the competition [BBF+08]. Again, the blocks of the architecture can be grouped according to their responsibilities. The first group contains a series of sensors. Their data is fed into blocks of the perception group, including the dynamic obstacle predictor, object classification, localization and road detection. The planning blocks cover several abstractions of granularity from route to motion planning. Finally, system blocks cover a user interface and a health monitor. Many blocks are similar to the two other architectures described above, e.g. the route planner, driving behavior and motion planner of Odin have similar responsibilities to the blocks mission planning, behavior generation and motion planning of the Boss vehicle.

Caroline was the best placed team developed by a non-US team [RBL+08, BBB+08]. Its architecture can be broken down into the component groups sensors and data acquisition, sensor data processing, decision making, control, emergency logic, and utilities, thereby resembling the other architectures introduced above.

Despite the age of the DARPA architectures, the key principles remain mostly unchanged. The complexity of the architectures can be coped with by breaking down the system into abstract functional blocks with clear and explicit dataflows. Hierarchical decomposition can help us to grasp complex modules. The processing relies on inputs from different, often redundant sensors complementing each other to draw a precise picture of the environment. The components can be classified into four important functional groups, which need to be present in an autonomous vehicle: perception, planning, control, and emergency handling. In addition, cooperative vehicles require a communication module as well as cooperation-specific components such as a platoon manager [KAE+12]. Modules are kept independent and use the publish/subscribe pattern for anonymous communication enforcing the low coupling principle: data providers post messages to uniquely identifiable topics, which can then be subscribed to by other com-

ponents interested in their content.

1.3.2 Control

An important part of an autonomous system is its control facility. The task of a controller is to execute a given plan in a physical, non-ideal environment with the help of the system's actuators by minimizing the error, i.e. the deviation of the measured from the desired state. In the context of autonomous driving we need to control the trajectory including the vehicle's speed and acceleration using steering, throttle, breaking, and optionally gear switching. Classical controller systems are based on proportional–integral–derivative (PID) blocks taking the error as the input and computing a weighted sum of the actual error, its first order derivative, and its integral. This approach has several drawbacks [Koz16]. For instance, a PID controller only takes the error signal into account, while ignoring the nature of the controlled system and its environment, i.e. the actual source of the error. Furthermore, PID controllers can only react to measured errors and are not able to prevent the occurrence of errors proactively. To control real systems, elaborate controller structures consisting of multiple intertwined PIDs are required, which results in a rising tuning complexity.

Model predictive control (MPC) is a control approach modeling the process explicitly to predict a system's behavior depending on a series of actions [CA13]. The control problem can be formulated as a minimization of the error over the action vector. This minimization program needs to be solved at runtime in each control step in order to obtain the next actions.

There are multiple advantages when using the MPC approach. The process is modeled explicitly. A single MPC controller can handle multivariate control variables including their interactions. The approach can incorporate constraints such as limits on speed and acceleration. Furthermore, it takes future steps into account when making a decision. For instance, a vehicle can start decelerating early if it knows that it is approaching a sharp turn. MPC is widely used in autonomous systems including self-driving and cooperative vehicles [KSM+19, KSB+19, KKM+19]. For this reason, a methodology for the design of cooperative CPSs needs to provide a means for the specification of optimization problems in a natural way.

1.3.3 Machine Learning in Autonomous Driving

More recent works have shown that functionality needed by autonomous vehicles can be implemented by means of machine learning techniques. The end-to-end approach transfers all tasks of the autonomous vehicle to a single deep neural network. The approach dates back to the 1980s, where the ALVINN architecture used a simple neural network to map sensor inputs to decisions [Pom89]. A more recent convolutional neural network (CNN)-based approach was demonstrated by NVIDIA [BDTD+16]. It uses camera im-

ages as the input for a deep CNN consisting of a normalization layer, 5 convolutional layers, and 3 fully connected layers to compute the steering commands. End-to-end driving has the drawback of being inflexible as the whole system is a single black box and does not provide interfaces to embed further code.

Some approaches use deep learning for perception, e.g. object detection in images. For instance, the KITTI-dataset [GLSU13] is a collection of annotated images captured by a vehicle camera in different traffic situations. Annotations include: bounding boxes for objects, the objects' types, e.g. car, truck, pedestrian, etc., as well as geometrical data. Further processing of the detected objects to prepare an actuation decision, however, can still be a complex task. Direct perception tries to tackle this problem by using CNNs to extract *affordance indicators* from a given image, which are features actually relevant for and directly usable by the controller. These indicators include the distance to the vehicle in front, the distance to the street boundary, the orientation of the vehicle relative to the track, etc. [CSKX15]. The decision making controller can then be implemented as conventional code using the affordance indicators extracted by the neural network as inputs.

1.4 Requirements

The examples given above are just a small excerpt of autonomous system approaches available in the literature. However, they are representative of some key ideas used in current practice and research. Based on these ideas we formulate the following *high-level* requirements for the model-driven CPS design methodology to be developed in this thesis. A refinement will be given throughout the following chapters.

(R1) Decomposition and communication: the methodology must enable the design of CPS architectures. In particular, it must ensure a clear separation of concerns and allow a hierarchical decomposition. It must make dataflows between components explicit and allow processing and communication of sensor signals. Furthermore, it must allow the integration of different technologies used in CPS design such as computer vision, deep learning, and optimization.

(R2) Architecture dynamics: the methodology must enable the designer to model systems, which are able to communicate with their environment, react to unforeseen changes and adapt their architecture if necessary, e.g. to switch between operational modes or to process data from unknown sources.

(R3) Training architectures: the methodology must enable the developer to integrate machine learning models as components into the software architecture and provide means to train such components based on available data or in simulators.

(R4) Distributed architectures: the methodology must enable the design of distributed architectures with low coupling between the components, e.g. to deploy software components on dedicated electronic control units (ECUs). Moreover, it should support the developer to find viable distribution schemes.

1.5 Thesis Structure

In order to answer the research questions posed above, each of the following chapters will discuss a part of the developed methodology while keeping the requirements in mind. Therefore, each chapter defines its own specialized research question and a set of requirements refining those introduced above. Chapter 2 covers static architecture design with EmbeddedMontiArc (EMA), a component-and-connector (C&C)-based ADL for embedded and cyber-physical systems. EMA is the core language of this work allowing us to decompose CPS models into manageable components. Together with its type system it serves as a basis for most other concepts of the thesis. In the second part of the chapter, we discuss MontiMath, a matrix-oriented scripting language, which is used to describe the behavior of EMA components procedurally. Generation aspects of the language family are covered at the end of the chapter.

Chapter 3 presents EmbeddedMontiArc Dynamics (EMAD), a conservative language extension for EMA allowing us to define software architectures with the ability to adapt themselves at runtime. This is achieved using an event-driven approach, where particular data inputs or architectural changes can trigger the instantiation or removal of architectural elements such as ports, components, and connectors.

To create intelligent architectures, which are able to learn from experience, we need to incorporate a means to make use of machine learning components in our methodology. We start Chapter 4 with a brief introduction of the main machine learning topics relevant for the thesis and present a DSL family for the design of neural networks afterwards. While Chapter 4 focuses on a supervised training pipeline, Chapter 5 extends the framework for reinforcement learning and generative adversarial networks (GANs).

Chapter 6 addresses the need for a modeling technique allowing us to define distributed architectures. We tackle the problem by using the tagging approach to attach middleware information to components and ports. The second part of the chapter presents an unsupervised learning approach deriving component distribution schemes automatically. The thesis is concluded in Chapter 7.

Each chapter defines a set of requirements to be addressed. Each requirement is identified by a unique name starting with a capital R followed by another capital letter related to the chapter and a number, e.g. RE1, RE1.1 being requirements related to EmbeddedMontiArc, RD1, RD2 being requirements related to dynamics etc.

1.6 Publications

Evgeny Kusmenko, the author of this dissertation, is author or co-author of the following peer reviewed research publications on which the contributions of this thesis are partially based:

- [KPRS19] E. Kusmenko, S. Pavlitskaya, B. Rumpe, S. Stüber: On the Engineering of AI-Powered Systems. In: ASE'19. Software Engineering Intelligence Workshop (SEI'19), San Diego, pp. 126-133, IEEE, Nov. 2019.
- [KNP+19] E. Kusmenko, S. Nickels, S. Pavlitskaya, B. Rumpe, T. Timmermanns: Modeling and Training of Neural Processing Systems. In: Conference on Model Driven Engineering Languages and Systems (MODELS'19), Munich, pp. 283-293, IEEE, Sep. 2019.
- [GKR19] N. Gatto, E. Kusmenko, B. Rumpe: Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In: Proceedings of MODELS 2019. Workshop MDE Intelligence, Munich, pp. 196-202, IEEE, Sep. 2019.
- [KKRZ19] J. C. Kirchhof, E. Kusmenko, B. Rumpe, H. Zhang: Simulation as a Service for Cooperative Vehicles. In: Proceedings of MODELS 2019. Workshop MASE, Munich, pp. 28-37, IEEE, Sep. 2019.
- [KKMR19] J. C. Kirchhof, E. Kusmenko, J. Meurice, B. Rumpe: Simulation of Model Execution for Embedded Systems. In: Proceedings of MODELS 2019. Workshop MLE, Munich, pp. 331-338, IEEE, Sep. 2019.
- [KKR19] N. Kaminski, E. Kusmenko, B. Rumpe: Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. In: The Journal of Object Technology, 18(2), pp. 1-20, AITO, July 2019.
- [HKKR19] A. Hellwig, S. Kriebel, E. Kusmenko, B. Rumpe: Component-based Integration of Interconnected Vehicle Architectures. In: 30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles, Paris, pp. 146-151, IEEE, June 2019.
- [KKRS19] S. Kriebel, E. Kusmenko, B. Rumpe, I. Shumeiko: Learning Error Patterns from Diagnosis Trouble Codes. In: 30th Intelligent Vehicles Symposium (IV'19). Workshop on Unsupervised Learning for Automated Driving, Paris, pp. 246-251, IEEE, June 2019.
- [DGH+19] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von

Wenckstern, A. Wortmann: SMArDT Modeling for Automotive Software Testing. In: Software: Practice and Experience, 49(2):301-328, Wiley Online Library, Feb. 2019.

- [FIK+18] C. Frohn, P. Ilov, S. Kriebel, E. Kusmenko, B. Rumpe, A. Ryndin: Distributed Simulation of Cooperatively Interacting Vehicles. In: International Conference on Intelligent Transportation Systems (ITSC'18), pp. 596-601. IEEE, Hawaii, 2018.

- [KRSvW18a] E. Kusmenko, B. Rumpe, S. Schneiders, M. von Wenckstern: Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In: Conference on Model Driven Engineering Languages and Systems (MODELS'18), pp. 447-457, Copenhagen, ACM, Oct. 2018.

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

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

- [BKL+18] C. Brecher, E. Kusmenko, A. Lindt, B. Rumpe, S. Storms, S. Wein, M. von Wenckstern, A. Wortmann: Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. International Symposium on Computer Science and Intelligent Control (ISCSIC'18). Stockholm, ACM, Sep. 2018.

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

- [KSRvW18] E. Kusmenko, I. Shumeiko, B. Rumpe, M. von Wenckstern: Fast Simulation Preorder Algorithm. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pp. 256-267. Funchal, Portugal, SciTePress, Jan. 2018.

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

- [GKR+17] F. Grazioli, E. Kusmenko, A. Roth, B. Rumpe, M. von Wenckstern: Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In: Proceedings of MODELS 2017. Workshop EXE, Austin, CEUR 2019, Sept. 2017.

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

- [DDE+17] J. Dankert, C. Dernehl, L. Eckstein, S. Kowalewski, E. Kusmenko, B. Rumpe: RapidCoop - Robuste Architektur durch geeignete Paradigmen für Kooperativ Interagierende Automobile. In: Automatisiertes und Vernetztes Fahren (AAET'17), Feb. 2017.

The following paragraphs give an overview of how the publications listed above were used in the context of this dissertation. In [HKK+18, KKRvW18, DGH+19, KSRvW18], we presented a variant of the model-driven automotive development process SMArDT as well as related testing and verification methods. An overview of SMArDT will be given in Section 1.7.5 later in this chapter. While it is not a direct research subject of this thesis, the methods, languages, and tools discussed throughout the dissertation are put in context of this process. In particular, the research questions of each chapter are posed in the context of SMArDT.

[KRRvW17] introduces the basic elements of the EmbeddedMontiArc ADL and the scripting language MontiMath. [KRSvW18a] presents an efficient generator toolchain for these languages using algebraic optimizations. An integrated development environment (IDE) for model-driven development with EmbeddedMontiArc was presented in [KRRvW18]. EmbeddedMontiArc and MontiMath are discussed and extended in Chapter 2. Furthermore, EmbeddedMontiArc provides the foundations for all other chapters of this dissertation. For instance, in Chapter 3 a modeling language for architectural runtime dynamics is developed as a conservative extension of EmbeddedMontiArc. This extension is based on [KKR19]. In [DDE+17] we introduced the notion of local traffic systems, which serves as a motivation and a running example in Chapter 3.

A simulative approach for integration testing of autonomous driving models was presented in [GKR+17]. This approach was further elaborated for spatial simulation subdivision and hardware emulation in [FIK+18, KKMR19, KKRZ19]. These publications are the basis of Section 2.6.3. A lightweight simulative approach for EmbeddedMontiArc teaching purposes was presented in [KRSvW18b].

One of the main contributions of this dissertation is the deep learning framework MontiAnna discussed in Chapter 4, the foundations of which were presented in [KNP+19]. The integration into the EmbeddedMontiArc language family and the TORCS example, which are introduced in Section 4.10, are based on [KPRS19]. In [GKR19], we presented

an extension of MontiAnna for the reinforcement learning domain, which is used as a basis for Chapter 5. In [BKL+18], we developed a multi-level modeling framework for agile manufacturing. This publication serves as the preliminary work for the evaluation of MontiAnna for reinforcement learning based on a problem from the forestry 5.0 domain in cooperation with WZL in Section 5.10.2. In [KKRS19] we presented an unsupervised learning approach for DTC pattern identification supporting the automation of quality assurance in the automotive domain. This example is used as a motivation for unsupervised learning in MontiAnna in Section 5.11.

In [HKKR19], we presented a tag-based approach to enrich component models with middleware information as well as a code generator composition approach for the generation of target code for multiple middleware platforms. These concepts are the basis of the first part of Chapter 6.

1.7 Preliminaries

1.7.1 Notation

This section covers the notation used throughout this thesis. $\mathbb{N}$ is the set of natural numbers (including zero). $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ denote the sets of integers, rational, real, and complex numbers, respectively. $\mathbb{Q}^+$ and $\mathbb{R}^+$ denote the sets of positive rational and positive real numbers, respectively. A zero in the subscript adds zero to the respective set, e.g. $\mathbb{R}_0^+ = \mathbb{R}^+ \cup \{0\}$ is the set of non-negative real numbers. The symbol used for the imaginary unit is j, with

$$j := \sqrt{-1}. \tag{1.1}$$

The real and imaginary part of a complex number $a \in \mathbb{C}$ are denoted by $\Re\{a\}$ and $\Im\{a\}$, respectively, i.e. $a = \Re\{a\} + j\Im\{a\}$, where $\Re\{a\}, \Im\{a\} \in \mathbb{R}$. Custom sets are denoted by calligraphic capital letters, e.g. $\mathcal{A}$, $\mathcal{B}$. Matrices and random variables are expressed by capital letters, e.g. A, B. The element in i-th row and j-th column of a matrix A is denoted by A_{ij}. A separating comma can be used to avoid ambiguities. For instance, $A_{4,5}$ is used to refer to the element in the fourth row and fifth column of A. Scalars and vectors are denoted by lower case letters, e.g. a, b. The i-th element of vector v is denoted by v_i. Vectors are interpreted as column vectors. A row vector can be obtained by transposing a column vector. Transpose and conjugate transpose of a matrix are denoted by a superscript T and $*$, respectively, i.e.

$$A = B^T \Leftrightarrow \forall i, j : a_{ij} = b_{ji} \tag{1.2}$$

and

$$A = B^* \Leftrightarrow \forall i, j : a_{ij} = \Re(b_{ji}) - j\Im(b_{ji}). \tag{1.3}$$

The set of all $m \times n$ matrices is denoted as $\mathcal{M}_{m,n}$. If m is equal to n, the abbreviation $\mathcal{M}_n$ is used. $\mathcal{H}_n \subset \mathcal{M}_n$ denotes the set of all positive semi-definite Hermitian $n \times n$ matrices.

For an n-dimensional vector x the function $\text{diag} : \mathbb{C}^n \to \mathbb{C}^{n \times n}$ is defined as

$$\text{diag}(x) = D \text{ with } D_{ij} = \begin{cases} x_i & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{1.4}$$

$\mathbb{1}_n$ denotes an n-dimensional vector full of ones. $I_n = \text{diag}(\mathbb{1}_n)$ denotes the identity matrix of size $n \times n$.

The expectation value of a random variable X is denoted by $\mathbb{E}[X]$, i.e. $\mathbb{E}[X] := \int_{\mathcal{X}} x p_X(x)\,\mathrm{d}x$, where $\mathcal{X}$ is the support of X and p_X is the probability density function (PDF) of X. If $\mathcal{X}$ is a discrete set, i.e. X is a *discrete* random variable, the integral reduces to a sum and hence, $\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x p_X(x)$. The expectation value of $g(X)$ is given as $\mathbb{E}[g(X)] = \int_{\mathcal{X}} g(x) p_X(x)\,\mathrm{d}x$ for continuous and as $\mathbb{E}[g(X)] = \sum_{x \in \mathcal{X}} g(x) p_X(x)$ for discrete supports of X.

Variables, types, and keywords are typeset in typewriter font, e.g. `int`, `component`, `foo`. The abbreviation L.x is used to refer to the x-th line in a listing. L.x-y is used to refer to a code block delimited by and including the lines x and y. Listings and diagrams contain a flag in the upper right corner indicating the language or diagram type used. An overview of the tags used in this thesis is given in Table A.1.

The definition of a signal for this thesis is given as follows.

Definition 1. *A signal is a function depending on time. Both the time domain as well as the co-domain can be continuous or discrete.*

In this thesis we only deal with signals, where both time and the co-domain are discrete.

A graph $\mathcal{G}$ is defined as a the tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. The adjacency matrix of a graph $\mathcal{G}$ is defined as

$$A \in \mathcal{M}_{|\mathcal{V}|} \text{ with } A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{otherwise.} \end{cases} \tag{1.5}$$

We distinguish between directed and undirected graphs. An undirected graph has a symmetric adjacency matrix, i.e. $\forall i, j \in \mathcal{V} : A_{ij} = A_{ji}$ and (i, j) is the same edge as (j, i). If the graph is directed, it can have an arbitrary adjacency matrix and (i, j) and (j, i) denote two distinct edges. In a weighted graph we assign a numeric weight to each edge using a graph-specific function $w : \mathcal{E} \to \mathbb{R}$. A weight of zero is equivalent to an absence of the edge in the graph. The similarity or affinity matrix $S \in \mathcal{M}_{|\mathcal{V}|}$ of a graph $\mathcal{G}$ is obtained by setting its entries S_{ij} to the respective edge weights (or to zero if nodes i and j are not connected). If the weights denote distances, i.e. higher values denote a smaller similarity, we refer to S as a distance matrix.

An important class of graphs used in this thesis are directed acyclic graphs (DAGs), defined as follows:

Definition 2. *A DAG is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ without cycles, i.e. for all paths $(i_1, i_2), (i_2, i_3), ..., (i_{k-2}, i_{k-1}), (i_{k-1}, i_k)$, where $i_1, ..., i_k \in \mathcal{V}$ and each tuple in the path is an edge in $\mathcal{E}$, $i_1 \neq i_k$.*

1.7.2 Model-Driven Engineering and Domain Specific Languages

A model is an abstraction of an original. It provides some information about the latter while leaving out unnecessary details. Understanding and creating complex systems is only possible with an appropriate model set. This applies to most sciences and fields including physics, biology, politics, economy, engineering disciplines, etc. Well-established modeling tools in computer science include: graphs, finite state machines (FSMs), Petri nets, and more, enabling a concise description of structure and behavior. More specific modeling notations have arisen from the field of software engineering, starting from simple flowcharts in imperative and procedural programming, over the UML for object-oriented design [Rum11b] to DSLs tailored to particular application domains.

General modeling languages such as the UML have manifested themselves as an indispensable toolset in widely used development processes such as the Rational Unified Process (RUP) [KK03], Open Unified Process (OpenUP)[1], the V-model [BD95] and others [BBR07]. While their use was first limited to conceptual phases and documentation purposes, generative approaches have helped modeling techniques to become an important, sometimes even the most important development means in various projects and disciplines. For instance, class diagrams can be used to generate websites and whole enterprise information systems [Rot17, GHK+20, GMN+20].

However, DSLs have gained importance in many fields, e.g. in linear TV program scheduling [DHH+20], as they exhibit multiple advantages when compared with general purpose programming languages (GPLs) or general purpose modeling languages such as the UML. First, they allow a concise problem description while hiding a large part of the complexity. Second, DSL programs often can be read and written by domain experts without a technical background as the syntax is domain-oriented. Third, DSLs can evolve much faster as they are used by much smaller groups of people than GPLs.

Many problems require very specific development concepts. For instance, graphical user interfaces (GUIs) can be described using a descriptive language made just for that purpose [GMN+20]. In the field of compiler construction, DSLs are used by tools like ANTLR [Par13] to describe context-free grammars of languages in a syntax resembling the theoretical notation. In particular, domain-specific modeling techniques are often applied to the CPS and robotics domain [NHW14] aiming to tackle specific requirements

[1] https://www.eclipse.org/epf/general/OpenUP.pdf, accessed August 24, 2020

of this field and accelerating the development while enhancing the reliability and safety of the product. The availability of parser generators [Par13] and ecosystems for language development, e.g. the Eclipse Modeling Framework (EMF) [SBMP08], enable a fast development and adaptation of DSLs and their integration into the targeted development processes.

Each computer language including DSLs consists of the following components: concrete syntax, abstract syntax, context conditions, and semantics [HR00, HR04].

Concrete syntax: the concrete syntax of a language is the textual or graphical syntax used by a developer to write down a model. It is often defined using a context-free grammar according to the Chomsky hierarchy [Cho59]. A well-designed concrete syntax should represent the domain as exactly as possible, be easy to read for the target audience, and avoid unnecessary generality [KKP+09].

Abstract syntax: the abstract syntax is a tree syntax reducing a model to its essential content which is used internally, e.g. by the compiler. In particular, the abstract syntax should be free of irrelevant concrete syntax elements.

Context conditions: context conditions, also referred to as static semantics, are Boolean predicates constraining the context-free syntax of a language by context-sensitive rules. Typical context conditions ensure that identifiers are not used before declaration and check the program for type errors. A model is considered well-formed if the context condition checks were successful.

Semantics: the semantics defines the meaning of a language. There are various approaches to define the semantics of a language. For instance, operational semantics maps a valid program to a sequence of computation steps [Plo81]. On the other hand, axiomatic semantics focuses on theorems and assertions: the meaning of a command is defined as a sequence of its pre- and post-conditions [Hoa69].

1.7.3 The MontiCore Language Workbench

Textual language engineering is a complex process which requires strong tool support to be fast, agile, and reliable. The required toolset can be found in a *language workbench*, a framework providing functionality needed for language definition, evolution, and composition [Fow10]. At the core of such a framework we can often find a lexer and a parser. The lexer (also referred to as scanner or tokenizer) converts a textual input model into a token stream. This stream is then processed by the parser according to a context-free grammar to verify that the model is valid and to derive an internal traversable tree representation, e.g. a parse tree or the abstract syntax tree (AST). In addition to parsers, language workbenches often provide editors, IDE plugins, as well as an infrastructure for abstract syntax, symbol tables, and context conditions (CoCos). Some examples of language workbenches are given in the following.

Xtext [EB10]. Xtext is a textual language workbench, which uses ANTLR [Par13] as its parser generator in the backend, but provides a lot of additional functionality including an EMF [SBMP08] based AST as well as Eclipse editors. Consequently, Xtext is heavily interwoven with the Eclipse ecosystem[2]. This has the advantage that language developers can reuse analysis and visualization tools operating on EMF-models. However, restricting the choice to a single IDE can be a disadvantage.

MPS. Meta Programming System (MPS)[3] follows the projectional editing approach where the user interacts with the AST directly. This has the advantage that one does not need to learn the concrete syntax of a language. The editor shows the user where values belong. The model can be projected to any appropriate form, e.g. tabular or graphical. An MPS language consists of several aspects like structure, editor, constraints, etc.

MontiCore [KRV10, HR17]. MontiCore is the language workbench we are going to work with in this thesis. It is being developed by the Software Engineering chair of the RWTH Aachen University and comes with a particular focus on language modularity, extensibility, and composition. These aspects are of high importance, since DSLs need to be tailored to specific requirements quickly and hence cannot be developed from scratch every time.

Similar to Xtext, MontiCore uses ANTLR in its backend for lexing and parsing. MontiCore uses a grammar description language based on the Extended Backus Naur Form (EBNF) and ANTLR to define the context-free (concrete *and* abstract) syntax of a language. The grammar is used to generate a variety of mostly Java-based artifacts including the parser and the data structures for the abstract syntax, i.e. the AST classes, transformations, and the symbol management infrastructure (SMI). The abstract syntax of a MontiCore language is represented by its AST together with the symbol table [MSN17]. In contrast to the parse tree provided by ANTLR, the MontiCore AST does not contain concrete syntax, but truly represents the model structure.

Context-sensitivity can be added to a MontiCore language by defining CoCos. Also, some context-free restrictions can be expressed more elegantly using CoCos. In MontiCore CoCos are encapsulated in Java classes. MontiCore generates CoCo interfaces for the AST node types of a language. To implement a concrete CoCo one has to implement these interfaces. Finally, the CoCo classes need to be instantiated and added to the CoCo checker of the language. The CoCo checker implements an AST traversal mechanism based on the visitor design pattern [GHJV95].

The language developer can decide when exactly to run the CoCo checker, e.g. right after parsing, after the creation of the symbol table (if it is needed by the CoCos), or after some transformation steps.

[2] https://www.eclipse.org/, accessed August 20, 2020
[3] https://www.jetbrains.com/mps/, accessed August 20, 2020

MontiCore supports different ways of language composition, which we are going to use heavily in the course of this thesis:

Language aggregation can be used when a concept of the target domain needs to be described using two or more DSLs. Each DSL has its own syntax and is used to define independent models in separate artifacts. There is no language composition on syntactic level. The models of all DSLs are processed independently. However, elements of the different models can be linked with each other after parsing. Consequently, inter-model context conditions can ensure the consistency of models written in different languages. Furthermore, code generators can look up information in different models to generate the target code. For instance, we are going to use language aggregation to use structs, defined in a dedicated struct language, as types in EmbeddedMontiArc models in Chapter 2. A further application is the composition of a neural network architecture with a training model in Chapter 4.

Language embedding is similar to language aggregation in the backend, but combines the concrete syntax of the composed languages instead of leaving each model independent. Consequently, a single artifact can contain a model written in different languages. An important example given in this thesis is the embedding of the behavior languages MontiMath and CNNArc into EmbeddedMontiArc in Chapters 2 and 4, respectively.

Language inheritance is similar to the object-oriented inheritance concept. The derived language reuses the elements of the base language, can modify them, or add new ones. Modifications which do not render base language elements invalid, are referred to as conservative. If the derived language only contains conservative modifications, it is referred to as **language extension**.

MontiCore provides a rich library of component grammars meant to be used as building blocks when designing new DSLs and GPLs. For instance, MontiCore literals, expressions, système international d'unités (SI) units, and types can be used as components in new languages out of the box. Some important languages developed in MontiCore are the UML/P [Rum11a], OCL [MMR+17] and MontiArc [Hab16].

1.7.4 SI Units

To give the reader a better idea of MontiCore's language components, we are going to introduce the SI unit language. In various domains including CPSs, we often need to model physical processes and hence, have to take care of the types of quantities the variables represent. Checking physical compatibility of variables in computations and assignments is a cumbersome but important manual task, as most languages and type systems do not provide automated checks. A common programming pattern is to append the physical unit as a suffix to a variable's name to ensure a correct usage

and interpretation, e.g. `double v_kmh = 10;` is probably supposed to mean that the value stored in this variable is to be interpreted as km/h. This can be seen as a manual version of strict and static typing: the variable's unit is known at compile time and cannot be overridden at runtime. However, the compiler does not know anything about this name-based type system and cannot help developers to avoid logical errors.

MontiCore offers syntax and a type checking facility for physical units. It can be used optionally when developing a language to build a unit-sensitive type system. The syntactic part consists of four grammars available in the package `de.monticore`:

SIUnits.mc4. Contains rules to parse units with optional prefixes as well as compound units. This includes unit products such as `VA`, quotients such as `km/s`, and powers such as `m/s^2`. The supported SI units are given in Table A.3. Further provided non-SI units are listed in Table A.2. The supported prefixes are given in Table A.4. Units and prefixes are fully compliant with the SI Brochure [dPeM19]. Additionally, the prefix μ for milli and the symbol Ω for ohm can be written as `u` and `Ohm`, respectively, if the developer does not want to or cannot use Greek symbols. Note that non-SI units defined in Table A.2 as well as °C, rad and sr cannot be combined with prefixes. An exception to this rule are liters, denoted by `l` or `L`.

SIUnitLiterals.mc4. Extends numeric MontiCore literals to be used in conjunction with SI units, e.g. `5.0 km`. Note that a numeric MontiCore literal can optionally end with an `f` or `F` denoting a floating point literal. Furthermore, `l` or `L` can be used to denote a long literal. This clashes with the SI units `l` for liter and `F` for farad. For this reason, a whitespace is obligatory between the number and the unit if the unit is `l` or `F`. Otherwise `l` and `F` are interpreted as long and float, respectively.

SIUnitTypes4Math.mc4. An SI unit can be used as a type. Therefore, the grammar extends `MCBasicTypes`, a central MontiCore grammar situated in the package `de.monticore.types` and provides an implementation of the `MCType` interface. Since such a unit type does not state explicitly what kind of number representation (int, float, etc.) is to be used, double is assumed implicitly.

SIUnitTypes4Computing.mc4. An extension of `SIUnitTypes4Math` providing a syntax to define types composed of a numeric type and a unit. The syntax for such a composed type is defined as `SIUnitType4Math "<" MCPrimitiveType ">"`. Non-numeric primitive types are excluded by a context condition.

MontiCore's type checking facility checks the compatibility of units along with the usual types. A sum of two or more variables is only allowed if the types of all summands are compatible. The same holds for assignments. Strong typing applies to units, as well. This means that an assignment cannot change the unit of the target variable, but rather involves a conversion of the source unit type into the target unit type. Consider the assignment `km<double> length = 10m;`. The variable `length` is statically and

strongly typed as a double with the unit km. The right hand side (rhs) however is given in meters. After the assignment, the numerical value of `length` is 0.01 and the variable must still be interpreted as km. For statically and strongly typed languages such conversions can be identified by the code generator. Consequently, information concerning units can be fully avoided in the generated code. All the generator needs to do is to look up and insert the necessary conversions. For the example given above, the (unoptimized) generated C/C++ code would be `double length = 0.001*10;`. Here, 0.001 is the conversion factor and 10 is the numeric value of the SI literal.

Sometimes it is necessary to extract the numeric value of a physical quantity expression. This can be achieved using the function `basevalue(.)` which takes a unit-numeric expression as input, converts it to base units, i.e. s, m, kg, A, K, mol, and cd, and returns the numeric part of the resulting expression. For instance, in `basevalue(2dm + 5cm)`, the argument summands are mapped to the base unit meters first, which results in 0.2 m + 0.05 m = 0.25 m. Hence, the returned value is 0.25. The conversion to base units does not always make sense, e.g. if the prefix is very big or very small. For such cases, the `value(.)` function can be used instead to convert a given unit-numeric expression to the value corresponding to the smallest contained prefix. Obviously, this operation always results in an integer value. For instance, `value(2dm + 5cm)` results in 25, since centi is the smallest prefix of the expression.

1.7.5 Model-Driven Engineering Processes in the Automotive Domain

A successful application of modeling techniques requires a clear underlying methodology. We distinguish between model-based and model-driven engineering processes. The former use models in an informal manner and are meant to support the developers, e.g. in planning tasks or when writing code manually. In model-driven engineering the role of models is more crucial. Models are used for all aspects of the engineering process including code or further model generation, evolution, analyses, visualization, tracing, etc. The goal of model-driven software engineering (MDSE) is to automate the software development process and to make it as robust as possible. In this thesis our focus is on MDSE.

In the automotive industry the development processes need to be compliant with the ISO26262 norm for functional safety. Therefore, they are often based on the ISO26262 V-model which prescribes systematic testing and validation. To grasp the complexity of automotive systems, models of appropriate abstraction, e.g. SysML [FMS14, Gro17], Simulink [Mat16], and LabView [Ins98], are employed to render the system design comprehensive and maintainable. However, maintaining a large zoo of models, updating and adapting them manually is error-prone and time-consuming. An automation or semi-automation of the engineering process featuring automated derivation of models, code, and tests is highly desirable. Navigation between models and tracing should prevent inconsistencies [KKRvW18] between different models as well as between models and code.

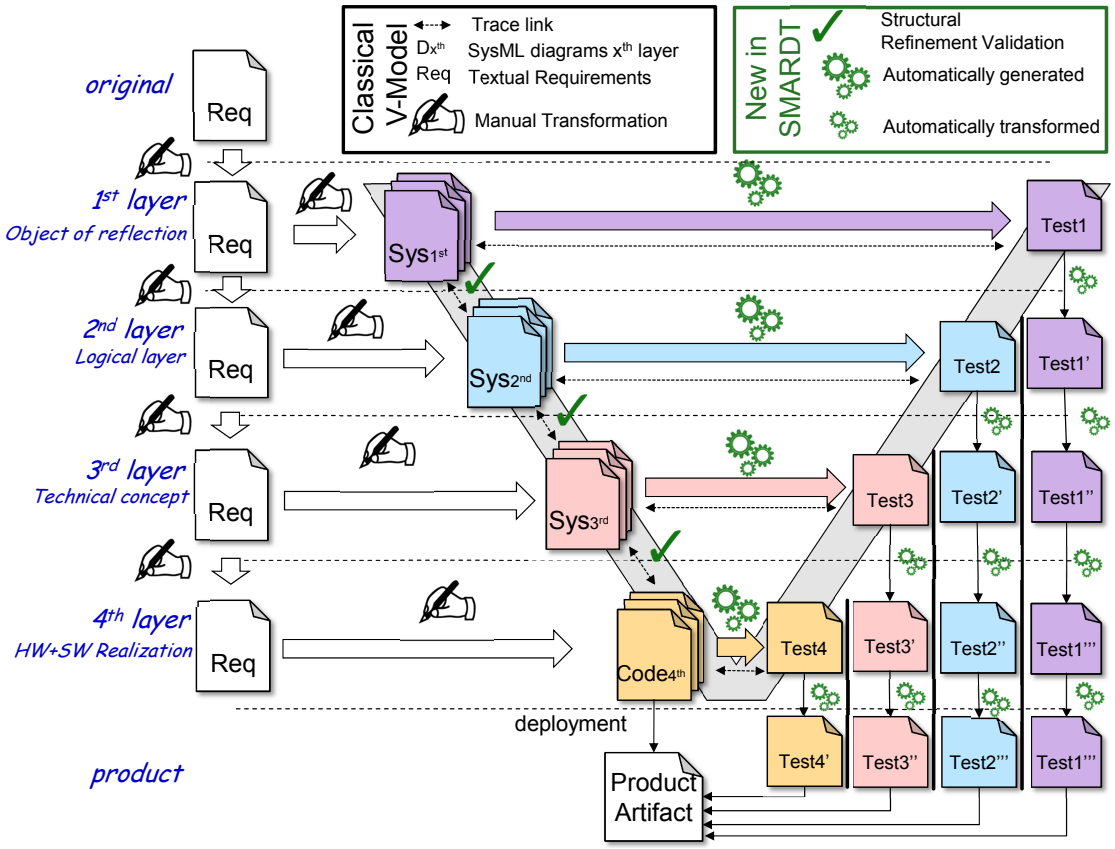


Figure 1.1: An overview of the SMArDT process [HKK+18].

Inconsistencies in model-driven software and systems engineering can arise quickly. For instance, if a bug is found or a requirement needs to be changed urgently, it is often fixed in code without updating the models the code is based on.

The specification method for requirements, design, and test (SMArDT) is a model-driven automotive development process developed by BMW AG, which aims to tackle the aforementioned challenges. We present a variant of this process [HKK+18, DGH+19] in the following. A graphical overview is given in Figure 1.1. Similar to the V-model, the SMArDT process is composed of several consecutive phases, also referred to as levels or layers. Each layer defines a development scope and is accompanied by a set of requirements and test cases. The artifacts of each level are linked to each other enabling automated validation and transformations which enforces a high level of consistency and enables agility.

Level 1 of the SMArDT process covers a first description of the product under development and shows its properties from a customer's point of view. The artifacts at this level contain natural language requirements and the like.

Level 2 deals with the functional specification while omitting technical realization details. Models at this layer include: activity diagrams (ADs), feature diagrams (FDs), C&C architectures, etc.

Level 3 develops a technical concept of the system including algorithms and protocols.

Level 4 is the final level which represents the concrete implementation of the software and the hardware system.

This thesis supports SMArDT using the C&C-based EMA language family together with integrated behavior description languages, model tagging, and a fully generative toolchain.

1.7.6 Component & Connector Modeling

Component-based software engineering (CBSE) is a design approach unifying concepts from different disciplines aiming to build large systems out of self-contained building blocks, which are often referred to as components [Nin97, KB98]. A central concern of this thesis is the C&C paradigm, an architecture-centric modeling approach used in CBSE. The aim of ADLs is to grasp the abstract top level structure of a software or a system preventing the implementation to diverge from the original specification. The aim of C&C models is the description of the logical software architecture focusing on functionality and logical communication [Rin14]. **Components** are first-level citizens of a C&C model. Each component encapsulates a piece of functionality. A component's interface is defined by a set of typed *input* and *output* **ports**. While input ports receive data from the outside world to work on, the result of a component's computation is

provided through the output ports, making the component reusable as a black box. A C&C architecture is usually composed of multiple components communicating with each other. Communication between components is made explicit by **connectors**, which are logical communication channels connecting an output port with an input port of another component. Communication over shared memory and side-effects are generally considered bad practice. Implicit communication makes C&C models difficult to understand and undermines testability.

The functionality encapsulated by a component, i.e. its behavior, can be described in two ways. First, an implementation language can be employed to declare the behavior explicitly, e.g. a GPL such as C++ or Java or, alternatively, a behavior describing DSL such as an FSM language. Second, a component can be composed of multiple subcomponents. Thereby, the ports of the parent component forward and receive their data to and from their subcomponents.

The C&C paradigm is widely used in engineering domains such as control, automotive, and avionics. It enables a divide-and-conquer approach to systems design by a hierarchical decomposition of a component into more concrete sub-tasks. This makes the paradigm particularly suited for the design of complex software in large teams: while a systems engineer or an architect can design the high-level architecture, expert teams can independently work on their respective submodules. Components can then be integrated based on the contracts manifested in their interfaces.

C&C models are often executed repeatedly in cycles, e.g. to process the sensor input of a CPS. The execution semantics of a C&C language can be characterized with respect to two aspects, the first of them being *synchronization*. In time-synchronous models there is a common notion of time. Execution frequencies of a model's components need not be equal, but they are in sync. Asynchronous components, on the other hand, are executed independently without a common clock.

The second important aspects of the semantics of a C&C language is *causality*. Causality means that a component can only operate on current and past but not on future outputs of other components, which is an appropriate assumption for practical systems. In this context, we speak of strong causality if only past outputs of other components can be used by a component for its computation. Weak causality allows the usage of the current outputs of other components, as well [BS12].

While a strongly causal semantics is more appropriate when modeling distributed systems, since communication between components introduces non-negligible delays, weak causality is more natural for the development of self-contained systems, where a model is mapped to a single process.

To give a reader a more tangible idea of C&C languages, we are now going to discuss a selection of practical representatives of the paradigm. For an extensive overview of C&C languages, see [KRRvW17, vW20, MT00].

Simulink: Simulink [Mat16] is a graphical C&C language. The developer can assemble a software from building blocks from a large library. Custom blocks can be written in MATLAB, a matrix-based scripting language with a focus on numerical computing. During an execution cycle, each component is executed exactly once, hence Simulink is a synchronized language. Having been designed with control engineering in mind, Simulink components are weakly causal. However delays can be modeled explicitly using delay components. The order in which the blocks are executed is determined by the *execution order* list assigning a priority to each component. For the creation of this list, Simulink classifies the ports of a component as *direct-feedthrough* and *non-direct-feedthrough* ports. An input port is considered to be direct-feedthrough if its current value determines at least one of the outputs directly, i.e. the respective output does not depend on past values of this input port. Components such as adders, multipliers, and gains use direct-feedthrough ports. Non-direct-feedthrough ports are used in components such as integrators, delays, and the like.

Simulink arranges the execution priorities such that a component A providing inputs to a direct-feedthrough port of a component B is executed before B. Components without direct-feedthrough ports can be placed anywhere in the execution order as long as they precede the direct-feedthrough ports which they provide inputs to. Therefore, Simulink schedules the execution of non-feedthrough components at the beginning of the execution list. *EmbeddedMontiArc*, the central modeling language family of this thesis is meant to be compatible with Simulink and hence, adopts parts of its semantics including the execution order algorithm.

AutoFocus 3: AutoFocus is a specification and development methodology for distributed embedded systems [AVT+15]. It covers the whole development process from the requirement definition to the integration phase. The foundations for formal verification and analysis are based on the FOCUS theory [BS12]. AutoFocus 3 supports runtime component reconfiguration governed by *mode FSMs*.

MontiArc: MontiArc is a *textual* C&C ADL for the design of distributed software systems [HRR12]. The ADL offers different modes of operation, covering both synchronous and asynchronous semantics. Due to its distributed systems focus, it assumes strong causality. However, variants with weak causality also exist [Hab16]. Custom component implementations can be realized as integrated hand-written Java code or, alternatively, as an FSM using the automaton DSL in a dedicated implementation block [Wor16]. Similar to AutoFocus 3, runtime reconfiguration can be defined using mode FSMs [HKR+16]. MontiArc is a MontiCore-based language. As such it can be extended and composed with other languages using MontiCore's language extension and composition mechanisms. For this reason, we use MontiArc as the basis for EmbeddedMontiArc, one of the central languages of this thesis, which we are going to discuss in the next chapter. Thereby,

we adopt a large portion of the MontiArc syntax in EmbeddedMontiArc, extend it, and compose it with other domain-specific, MontiCore-based languages.

Chapter 2

EmbeddedMontiArc

In this chapter we are going to introduce the fundamental concepts of the EMA language family for the design of CPSs [KRRvW17, KRSvW18a]. The concepts presented in this chapter have been developed in collaboration with Michael von Wenckstern; his contributions, e.g. parts of the formal grammar of the EMA language, have been presented in his dissertation [vW20]. The research question to be answered in this chapter is the following:

Research Question 1. *How can a C&C-based development methodology support the design of CPSs at SMArDT levels 2 and 3?*

2.1 Requirements

As was outlined in the introduction, CPS architectures are often modeled using architecture-centric notations and languages making dataflows explicit. To grasp the essence of such systems enabling separation of concerns, maintainability, the development in large teams, and composability, we employ the C&C paradigm as introduced in Section 1.7.6. To tailor a C&C-based methodology to the target domain of cooperative CPSs in the best possible way, we introduce the following set of requirements:

(RE1) Type system: a DSL for the design of CPS architectures needs a type system, which is appropriate for the description of different kinds of signals to be processed.

- **(RE1.1) Abstraction:** a truly model-driven methodology needs to represent the domain as exactly as possible without exhibiting technical realization details. The type system therefore needs to be abstract and refrain from highly technical types such as floats and doubles. Instead, it has to provide types used in mathematics and engineering disciplines.
- **(RE1.2) Units:** the type system must support the developer when dealing with physical quantities. In particular, the type system must be able to recognize incompatible unit types at compile-time and automatically convert compatible ones.

(RE1.3) Matrices: the type system must provide the means to deal with matrices and vectors as these are often needed in CPS, e.g. to represent complex sensor data such as images and to perform algebraic operations on them. Matrix and vector dimensions should be fixed at compile-time and must not be changed at runtime.

(RE1.4) Algebraic requirements: many operations impose algebraic requirements on their operands. For instance, matrix addition and multiplication only work if the operands have compatible shapes; a matrix can only be inverted if none of its eigenvalues is equal to zero. The static type system must be able to recognize incompatible operands at compile-time.

(RE2) Architectural modeling: the methodology must enable a dataflow-centric hierarchical modeling of CPSs.

(RE2.1) Logical modeling: the language family must enable the developer to model the business logic while hiding technical details.

(RE2.2) Clean components: the language must prohibit dirty components, i.e. components with side-effects, in order to render the models maintainable and testable.

(RE3) Behavior specification: the modeling language family must provide domain-oriented means for behavior specification of components. For instance, it should be possible to write down optimization problems in order to be able to develop MPC controllers.

(RE4) Code generation: the methodology must provide a generative concept. The generation toolchain should deliver complete and fully functional executable target code.

(RE4.1) No mix of generated and manual code: the generated code should be hidden from the developer. In particular, adding manual code to generated files should be discouraged.

(RE4.2) Efficiency: the performance of the generated code must be comparable to a GPL implementation and alternative tools.

2.2 The Data Type System

2.2.1 Primitive Data Types

A central concept of EMA is its abstract data type system. It is based on primitive types, which can be refined or grouped together, enabling the developer to create new types tailored to the application. The primitive types are abstract in the sense that they

are not bound to a specific realization. This is in contrast to most typical type systems, which prescribe the implementation of their primitive types up to some parameters (for instance, many languages support floating point types with their representation and operations based exactly on IEEE754 [IEE08]; however, the length of integer types and hence the available range can depend on the executing hardware platform). Instead, EMA types resemble mathematical sets they aim to represent. The basic types are given in the following:

- `N` represents the set of positive integers including 0, i.e. $\mathbb{N}$,
- `N1` represents the set of positive integers not including 0, i.e. $\mathbb{N} \setminus \{0\}$,
- `Z` represents the set of signed integers $\mathbb{Z}$,
- `Q` represents the set of signed rational numbers $\mathbb{Q}$,
- `C` represents the set of Gaussian rationals $\mathbb{Q}[j] = \{a + jb : a, b \in \mathbb{Q}\} \subset \mathbb{C}$,
- `B` represents the set of Booleans (`true` and `false`). For the sake of convenience the alias `Boolean` can be used interchangeably.

The types `N1`, `N`, `Z`, `Q`, and `C` form a directed compatibility relation, where a type is compatible with another type if the latter can represent all the elements of the former. For instance, `N` is compatible with `Z`, `Q`, and `C`, but not with `N1`, since the latter does not include zero. A variable of type `N` can hence be assigned to variables of types `Z`, `Q`, and `C`, but not to variables of type `N1`. Note that these types represent infinite sets of numbers. Since no technical system can represent arbitrarily large numbers, using primitive EMA types leads to a model that can only be implemented partially by definition. Obviously, this does not hold for Booleans (`B`). The decision how to implement such types is delegated to the compiler and can depend on the application.

EMA types can be refined by adding a range consisting of a lower and an upper bound to the primitive type. A bounded type is defined as `T(minValue : maxValue)`, where `T` can be any primitive type except `B`. The bounded type covers a subset of the primitive type `T` bounded by `minValue` and `maxValue`. `minValue` and `maxValue` must be of type `T` themselves and their values are included in the bounded type. For instance, the bounded type `N(5:7)` represents the set $\{5, 6, 7\}$. A type can be bounded only from one side by using the infinity operator `oo` as the other bound. For instance, `N(5:oo)` is a type covering all integers in $\{n \in \mathbb{N} | n \geq 5\}$.

Bounded types are still not completely implementable if the base type is `Q` or `C`, as a technical system cannot handle arbitrarily high resolutions for non-integers. To obtain a completely realizable type, a bounded type needs to be refined by a resolution or step size. This parameter is written between the minimum and maximum value of a bounded type, i.e. `T(minValue : resolution : maxValue)`. The refined

type only contains values of the form `minValue`+k×`resolution` satisfying `minValue` ≤ `minValue`+k×`resolution` ≤ `maxValue`, where $k \in \mathbb{N}$. For instance, the type `Q(5:0.5:6.5)` represents the set $\{5.0, 5.5, 6.0, 6.5\}$ Similarly to the lower and the upper bounds, the step size needs to be of the basic type it is restricting.

Different levels of type refinements can be employed in different phases of a systems engineering process such as SMArDT during the development of a CPS. While unbounded types can be used at SMArDT layer 2 to develop the logical models, a concretization is required to proceed to the technical concept in SMArDT layer 3.

2.2.2 Vectors, Matrices and Cubes

In complex technical systems, data is often multidimensional. For this reason, primitive types of EMA can be organized as one-, two- or multidimensional arrays. The syntax to do so is based on the LaTeX syntax for raising a base to a power. To specify the dimensionality of an array type, we need to append a circumflex followed by a list of comma-separated integer-valued dimension sizes in curly brackets to the primitive type's name: `T^{a,b,...}`. Each argument initializes the size of the respective array dimension. For instance, `Q^{5}` represents the set of all five-dimensional rational vectors $\mathbb{Q}^5$, `Z^{2,3}` represents the set of all integer-valued 2×3 matrices, and so on. We refer to one-dimensional arrays as vectors, to two-dimensional arrays as matrices, to three-dimensional arrays as cubes, and to multidimensional arrays as (n-dimensional) hypercubes. Thereby, `Q^{a_1,...,a_n}` and `Q^{a_1,...,a_n,1}` are compatible types as they represent isomorphic vector spaces. Hence, `Q^{a_1,...,a_n}` and `Q^{a_1,...,a_n,1,1,...,1}` are compatible, as well, for arbitrarily many array dimensions of size 1 after the n-th array dimension. In particular, vectors are interpreted as column vectors according to this scheme and can be multiplied with matrices of compatible sizes, which is in line with MATLAB.

Of course, we can also create arrays, vectors, and hypercubes of bounded types. For instance, the type `N(0:255)^{3,w,h}`, is often used to represent images with three channels, a size of `w`×`h`, and a color depth of 8 bit. In contrast to MATLAB, dimensions are set at compile-time and cannot be changed at runtime. Variables of the aforementioned matrix type `Z^{2,3}` can only be assigned 2×3 matrices.

2.2.3 Matrix Properties

In matrix-based signal processing, operations often require the operand to exhibit particular properties. EMA allows the developer to specify an arbitrarily long list of properties preceding a matrix type, e.g. `prop1, prop2,..., propn Q^{n,n}`. These properties can be seen as pre- and post-conditions if applied as predicates to component ports or variables. The properties supported by the EMA type system are based on the matrix taxonomy by Horn et al. [HJ90] and are only applicable to square matrices. The

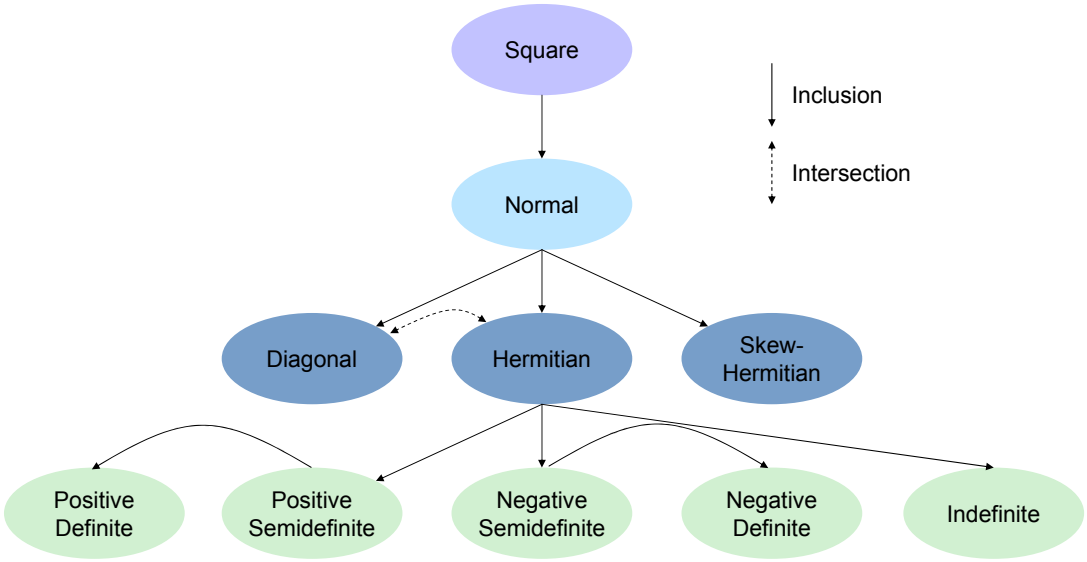


Figure 2.1: A simplified taxonomy of matrices containing the matrix properties relevant for the EMA type system based on [HJ90].

following properties are supported:

inv: states that an $n \times n$ matrix $M \in \mathcal{M}_n$ is invertible (also referred to as nonsingular), i.e. $\exists N \in \mathcal{M}_n . NM = MN = I_n$. Equivalent statements are: all eigenvalues of M are non-zero, $\det(M) \neq 0$.

norm: states that an $n \times n$ matrix $M \in \mathcal{M}_n$ is normal, i.e. $MM^* = M^*M$. Normal matrices are closed under raising to an integer power and, if nonsingular, inversion.

sym: states that an $n \times n$ matrix $M \in \mathcal{M}_n$ is symmetric, i.e. $\forall i, j \in \{1, ..., n\} : M_{ij} = M_{ji}$.

herm: states that an $n \times n$ matrix $M \in \mathcal{M}_n$ is Hermitian, i.e. $\forall i, j \in \{1, ..., n\} : M_{ij} = M^*_{ji}$. Consequently the elements on the main diagonal are real-valued. Furthermore, all eigenvalues of M are real. Hermitian matrices are closed under addition, multiplication by a scalar, raising to an integer power and, if nonsingular, inversion. Hermitian matrices in the real domain are also symmetric.

skew: states that an $n \times n$ matrix $M \in \mathcal{M}_n$ is skew-hermitian, i.e. $\forall i, j \in \{1, ..., n\} : M_{ij} = -M^*_{ji}$. Consequently, the real part of the elements on the main diagonal is zero, i.e. $\forall i \in \{1, ..., n\} : \Re\{M_{ii}\} = 0$.

diag: states that an $n \times n$ matrix $M \in \mathcal{M}_n$ is diagonal. Only the entries on the main diagonal are allowed to be non-zero, i.e. $\forall i, j \in \{1, ..., n\} : M_{ij} \neq 0 \implies i = j$.

Furthermore, if $\forall i, j \in \{1, ..., n\} : M_{ij} \neq 0 \Leftrightarrow i = j$, then M is non-singular. Diagonal matrices are closed under addition, multiplication by another diagonal matrix or a scalar, raising to an integer power and, if nonsingular, inversion.

psd: states that an $n \times n$ Hermitian matrix $M \in \mathcal{H}_n$ is positive semidefinite (PSD), i.e. $\forall x \in \mathbb{C}^n : x^* M x \geq 0$.

pd: states that an $n \times n$ Hermitian matrix $M \in \mathcal{H}_n$ is positive definite (PD), i.e. $\forall x \in \mathbb{C}^n \setminus \{0\} : x^* M x > 0$.

nsd: states that an $n \times n$ Hermitian matrix $M \in \mathcal{H}_n$ is negative semidefinite (NSD), i.e. $\forall x \in \mathbb{C}^n : x^* M x \leq 0$.

nd: states that an $n \times n$ Hermitian matrix $M \in \mathcal{H}_n$ is negative definite (ND), i.e. $\forall x \in \mathbb{C}^n \setminus \{0\} : x^* M x < 0$.

indef: states that an $n \times n$ Hermitian matrix $M \in \mathcal{H}_n$ is indefinite, i.e. neither PSD nor NSD.

Note that all properties are applicable to square matrices only. The square matrix property itself is not represented by a keyword, as it is implicitly encoded in the matrix dimensions definition. Furthermore, some matrix properties imply or exclude others, cf. Figure 2.1. We will discuss how matrix properties are used and derived in algebraic operations of component behavior implementation in Section 2.4.

The EMA type system is a static and strong type system. This means that the type of a port or a variable must be explicitly declared and known at compile-time. Furthermore, the type of a variable is fixed and cannot be changed nor cast throughout the course of the program. This applies to the primitive base type of a variable, i.e. `B`, `N`, `Z`, `Q`, and `C`, but also to the range and the dimensions of a type. In contrast to other matrix-oriented languages like MATLAB, the matrix dimensions are known at compile-time and cannot be changed dynamically. This enables us to perform matrix property checks such as whether a matrix is square or not at compile-time.

However, in an assignment the target variable or port type can have a wider range than the source type, i.e. a variable of the type `T(minValue1:maxValue1)` can be assigned to another variable of the type `T(minValue2:maxValue2)` iff `minValue1`$\geq$ `minValue2` and `maxValue1`$\leq$`maxValue2`. Defining compatibility for resolutions is less straightforward and might require different strategies depending on the problem domain. Specifying a resolution provides a guarantee that the desired precision is delivered. However, the internal representation and computations might be of higher precision. A lower resolution `res1` can always be mapped to a higher resolution `res2` unambiguously if `res1`=$n\times$`res2` and if the range boundaries of the higher resolution variable are compatible with the resolution of the lower resolution variable, i.e. `minValue2`=$n\times$`res1` and `maxValue2`=$m\times$`res1` for $n, m \in \mathbb{N}$.

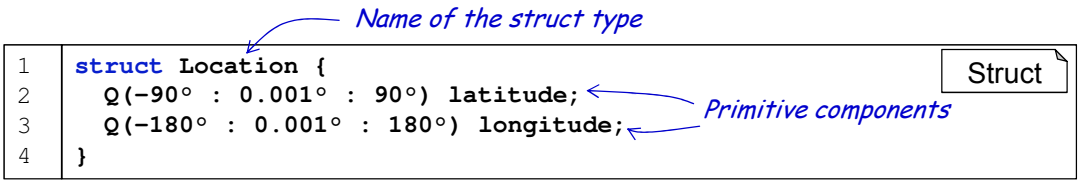


Figure 2.2: A struct example encapsulating the primitive variables longitude and latitude into a type named Location.

Assignments of higher resolution variables to lower resolution ones can be tackled using different strategies, the main two of which are: handling the types as incompatible at compile-time or rounding combined with a type cast warning. For the rest of this work we are going to stick with the former variant as it provides the highest degree of type safety.

2.2.4 SI Units in EmbeddedMontiArc

Since EMA has its own type system and does not use MontiCore's primitive types, we need to adapt the syntax of unit types presented in Section 1.7.4. In EMA the unit of a variable is written as part of the range and resolution definition. For instance, `Q(0m:1dm:1km)` is a rational variable representing a length between 0 m and 1 km with a resolution of 1 dm. If the type has no range, only the unit is given in brackets. For instance, `Q(m)` denotes the rangeless rational number type to be interpreted as meters. Except this syntactic variation, all other aspects of MontiCore's unit type system continue to hold. Two variables are only compatible if they represent the same physical quantity or do not have a unit. Conversions are carried out automatically in assignments featuring compatible but different units. Tables A.2 to A.4 list the available non-SI units, SI units, and the prefixes together with their syntax in the appendix.

2.2.5 Structs and Enums

Often, we need to group different pieces of data to create a logical data type. For instance, a geographical location consists of a longitude and a latitude component. Such a location data type can be modeled as a two-dimensional vector, e.g. `Q^2`. However, the readability and maintainability of the resulting models would suffer heavily if we sticked to this pattern. Given a vector like `Q^2 location`, it is not clear what variables it consists of. Furthermore, the order of the components in the vector must be known by the user.

To tackle this challenge, we introduce struct types. Similar to other languages like C/C++ and MATLAB, a struct bundles named variables into a new data type. An EMA struct is defined in a separate struct artifact, the MontiCore grammar is provided

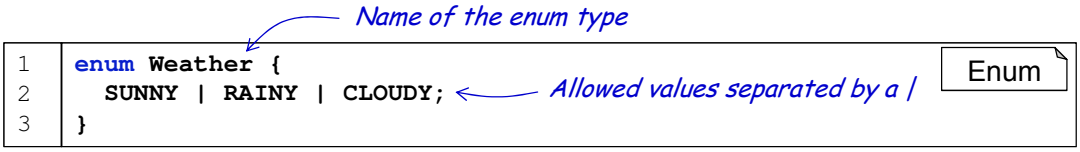


Figure 2.3: An enum example representing different weather conditions.

in the appendix in Listing B.5. A concrete example model representing a geographical location is given in Figure 2.2. The keyword `struct` initiates a struct definition followed by the type name, in this case `Location`. The body contains a list of typed variables. These variables can be of a standard EMA type or structs themselves. However, circular dependencies are forbidden, i.e. the parent struct type must not be used as a field type in any of its child properties. This is verified using a context condition at compile-time. Struct types are made available for EMA models and resolved using MontiCore's language aggregation feature. A struct member can be accessed using the dot operator. Given a port or variable `Location loc`, the latitude component can be accessed as `loc.latitude`. In contrast to C structs, an EMA struct type is only compatible with itself, i.e. two different struct types are incompatible, even if they have the same structure. The rationale behind this is that EMA, in contrast to C, is a high-level modeling language. The aim of the type system is to capture the domain and prevent logical errors by prohibiting implicit type conversions.

Similar to structs, enums are a type extension mechanism, which helps improve maintainability and reduce logical errors. An enum is used to represent a categorical type. Again, enums are defined in separate files and composed with EMA models and structs using the language aggregation mechanism. The MontiCore grammar of the enum definition language is given in Listing B.6 in the appendix. A concrete example modeling an enum data type to describe the weather can be found in Figure 2.3. The keyword `enum` is followed by the name of the enum type being modeled, which, in this case, is `Weather`. The list of possible values is given in the enum body with the elements separated by a pipe character. In this example, only three types of weather are allowed, namely, `SUNNY`, `RAINY`, and `CLOUDY`. A variable or a port of type `Weather` may, thus, only take one of these three values. Again, an enum type is only compatible with itself. Variables of different enum types are always incompatible regardless of their structure and entries' names. In an EMA model, a concrete value can be specified using the enum name followed by the dot operator and the desired category, e.g. `Weather.SUNNY`.

```
1  component Main<T, N(0:10) n> (Q param1, N param2,…) {
2    ports in T A,
3          in T B,
4          out T C;
5
6    instance Add<T, n> adder(0);
7    instance Mult<T, n> multiplier(1);
8
9    connect A -> adder.A;
10   connect B -> adder.B;
11   connect adder.C -> multiplier.A;
12   connect B -> multiplier.B;
13   connect multiplier.C -> C;
14 }
```

EMA

Figure 2.4: A basic example of an EMA architecture. The component `Main` contains two subcomponents `Add adder` and `Mult multiplier`.

2.3 Static Architecture Description

2.3.1 Components, Ports and Connectors

EmbeddedMontiArc is a textual ADL implementing the C&C paradigm introduced in Section 1.7.6. Its syntax is mostly based on the MontiArc ADL [HRR12]. However, there are several deviations and extensions to consider. We are going to introduce the concrete syntax based on Figure 2.4. The base grammar of EMA is given in Listing B.7.

Components are first-level citizens in EMA. A component type is defined using the keyword `component` followed by a name which can later be used to create instances of this component type[1]. For instance, we declare the component type `Main` in L.1 of Figure 2.4. Optionally, a component type declaration can include a list of generic parameters in angle brackets and another list of component parameters in round brackets. While generic parameters are allowed to change a component's interface, component parameters can only be used to parameterize a component's implementation. Depending on the use case, a generic parameter can be set to a component type, a data type, or a concrete value. We will see, how the latter can change a component's interface in Section 2.3.2.

The syntax for declaring a generic component or data type in a component header definition is just the parameter name, cf. parameter `T` in L.1. If the generic parameter is a concrete value, its name needs to be preceded by its data type, cf. generic parameter `n`, which is of type `N(2:10)` in this example. Component parameters, in contrast to generic parameters, can only be of a data type. The syntax resembles the definition of

[1]The component type system is not to be confused with the data type system introduced in Section 2.2.

function parameters in many languages, where a type is followed by a unique name, cf. parameters `Q param1` and `N param2` in L.1.

The body of a component definition is enclosed in curly brackets and contains an interface and a structure definition. The interface definition is initiated with the keyword `ports` and is followed by a port list. A port definition consists of the port kind, which can be either `in` or `out` (EMA ports are strictly unidirectional), a data type, and a unique port name, cf. L.2-4 in Figure 2.4. A component must have at least one input and one output port, since a major assumption of EMA is the absolute absence of side effects. Clean side-effect-free models are crucial for testability, maintainability, and extensibility. An exception are components outputting a constant or a (possibly parameterizable) constant sequence. Such components obviously do not need an input port, but can require a component parameter, which alone defines the output behavior in every execution step.

Subcomponents are created using the keyword `instance` followed by the component type to instantiate and a component instance name, which is unique in the scope. If the component type to be instantiated has generic and/or component parameters, these have to be set by providing appropriate arguments in (generic parameters in angle brackets and component parameters in round brackets). In L.6-7 of Figure 2.4 two components are instantiated with their generic parameters being set to the type `T` and the value `n`. Furthermore, both subcomponents receive a component parameter in round brackets, which is 0 in L.6 and 1 in L.7.

To interconnect the subcomponents and to connect them to the parent component in the first place, we need to create connectors. The source of a connector must be either an output port of a sibling or subcomponent or an input port of the enclosing component. Similarly, the target of a connector must be either an input port of a sibling or subcomponent or an output port of the enclosing component. A connector is created using the `connect` keyword followed by the source port, the arrow operator `->`, and a target port. Ports of subcomponents can be referenced by using the subcomponent's name and the dot access operator. Connector examples are given in L.9-13.

The EMA ADL is used to describe the logical structure and dataflows of a software system. It is important to stress that no technical details like communication protocols, deployment schemes, or platform specifics must be covered in an EMA model. EMA is a language covering the second (logical) layer of the SMArDT process, cf. Section 1.7.5. As there is no concrete functionality attached to the components, conventional testing cannot be applied to such models. However, it is possible to verify that a model fulfills some desired structural properties, e.g. that two components are connected (omitting further details such as the connector type). Structural properties can be defined using a more abstract notation such as C&C views, e.g. EmbeddedMontiView [vW20, BMR+17].

The structure definition is optional and can be replaced by a direct component implementation, e.g. by using a behavior description language as will be described later. In standard EMA, the structure, i.e. the subcomponents as well as the connectors between

```
1  component Main<N(2:10) n> {                                    EMA
2    ports in Q A[n],
3          in Q B[n],
4          out Q C[n];
5          out Q D;
6
7    instance Add2 adder[n];
8    instance Mult2n<2*n> multiplier;
9
10   connect A[1:n] -> adder[1:n].firstSummand;
11   connect B[:] -> adder[:].secondSummand;
12   connect adder[:].sum -> C[:];
13
14   connect A[:] -> multiplier.factors[1:n];
15   connect B[:] -> multiplier.factors[n+1:2*n];
16   connect multiplier.product -> D;
17 }
```

Figure 2.5: An EMA architecture example featuring port and component arrays. The component `Main` contains `n` `Add2` components, each operating on one of `n` operand pairs coming from the port arrays `A` and `B`. The `Mult2n` component computes the product of `2n` operands passed through the port arrays `A` and `B` of the `Main` component to the port array `factors` of `Mult2n`.

them, is fixed at design-time. An extension for structural reconfigurations at runtime will be discussed in Chapter 3.

2.3.2 Arrays and Connection Patterns

Modeling cooperative systems and agent networks often requires the replication of large numbers of similar components and the interconnection thereof. EMA enables the designer to create multiple similar components and/or ports by means of arrays. Based on the array syntax of many languages, an array is created by appending the array size to the port or component name in brackets. For instance, in Figure 2.5 we define the input ports `A` and `B` as well as the output port `C` as port arrays of length `n`. Since, by changing the number of ports of the enclosing component, the parameter `n` affects the interface of `Main`, it cannot be defined as a normal component parameter, but must be a generic parameter instead.

In this example we demonstrate two interconnection patterns which are commonly used when dealing with port and component arrays. In the first one, we instantiate an array of components to deal with an array of incoming streams. Therefore, we create `n` adders of the component type `Add2` in L.7, each instance to operate on two scalar inputs. Now, we need to connect the ports of the two arrays `A` and `B` of the parent

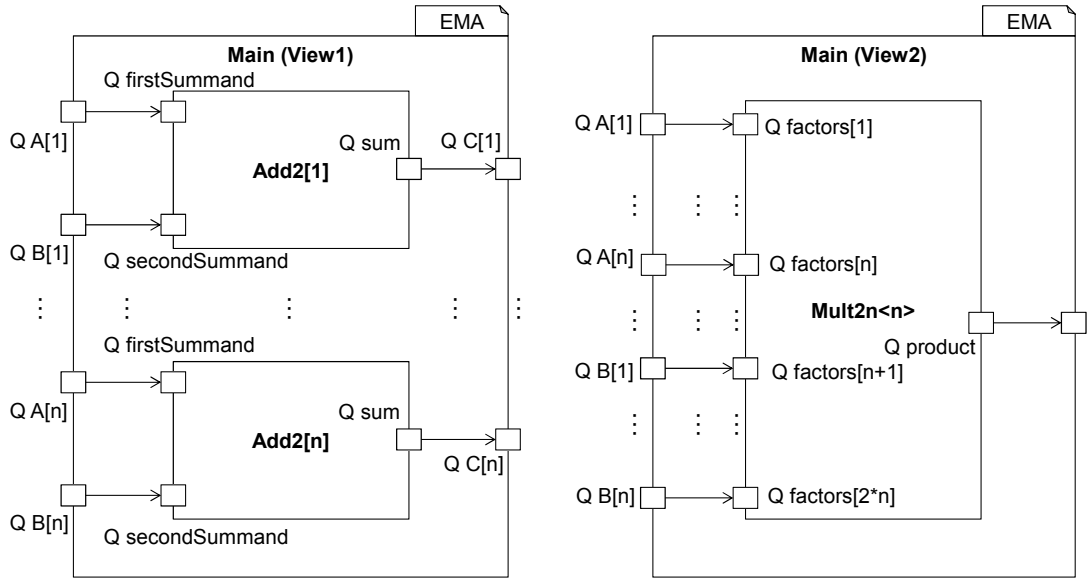


Figure 2.6: Graphical views of the component defined in Figure 2.5. On the lhs, the elements of two port arrays are connected to target ports of a component array. On the rhs, a port array is connected to another port array.

component to the respective subcomponents, i.e. `A[1]` and `B[1]` should be connected to `adder[1]` and so on. This can be done in just one line, cf. L.10, by selecting the elements `1` to `n` from the port array `A` and, similarly, the components `1` to `n` from the `adder` component array. The connect operator connects each source element to the respective target element based on the index. Since this connection pattern is often applied to *all* elements of an array, EMA offers syntactic sugar allowing the developer to leave out the indices of the first and last elements as is done in L.11. Similarly, the output of each component in the `adder` array is connected to a corresponding port in the target port array `C`. This structural pattern is depicted graphically in the view on the left side of Figure 2.6.

Furthermore, we can connect a port array to the port array of a target component, let this component aggregate the data and output a single result or a constant number of values. In our example, the port array `A` is connected element-wise to the first `n` elements of the input port array of the `multiplier` component of type `Mult2n` in L.14, while the port array `B` is connected to the remaining `n` input ports of `multiplier` in L.15. The output of the `multiplier` component is forwarded to the output port `D` of the enclosing component in L.16. This connection pattern is depicted graphically in the view on the rhs of Figure 2.6.

While the component and port array interconnection syntax is strictly declarative and

is limited to the interconnection of two layers of components, i.e. to describe a bipartite graph, more complex architectures and dataflows can be modeled using control flow constructs known from imperative languages. For instance, the Darwin ADL [MK96] allows the definition of parameterizable component *pipelines* by applying a `forall` construct to a component array. Such constructs have not been made available in EMA to maintain the declarative style of the language and to keep the syntax as simple as possible. Pipelines can be modeled using the dynamics framework of EMA which will be discussed in Chapter 3.

2.3.3 Execution Semantics

Standard EMA has a synchronous and weakly causal execution semantics, which is based on the FOCUS theory [BS12] and inspired by Simulink [Mat16]. An EMA model is executed in cycles. In each cycle, every component is executed exactly once. Once a component has finished its execution, the computation results are immediately available at its output ports. We assume that data transmission is lossless and has no delay. Connectors transmit data instantly, i.e. when a source port of a connector is updated, the data is replicated immediately to the target port. A component is only allowed to be executed, once all of its predecessors, i.e. components connected to its input ports via a connector, have finished execution. The identification of a dataflow-based execution order is, hence, crucial for a correct realization of the model semantics. A fixed execution order is established at compile-time and no re-scheduling needs to be performed at runtime. This is similar to Simulink's sorted execution order list[2]. For the computation of the sorted execution order, Simulink distinguishes between virtual and non-virtual blocks. Virtual blocks are used for the sake of clarity and are flattened internally at compile-time, i.e. replaced by their inner structure. Hence, they do not have an own execution order. Non-virtual blocks on the other hand are treated as black boxes. They have their own execution order and their subcomponents are executed according to a separate execution order list. However, defining a block as virtual or non-virtual has no impact on the semantics as both approaches lead to the same input/output behavior of a model. In EMA, for the computation of the execution order, components are considered to be virtual, i.e. the C&C model is flattened at compile-time before the execution order is computed and before code is generated. Hence, only atomic components receive an execution order. In EMA, multiple component instances can share a single execution order id if the execution order of these respective component instances can be interchanged without affecting the dataflow. This enables the parallelization of component execution in the generated code.

The flattening operation is a model transformation on the AST and the symbol table, recursively replacing composite components with their subcomponent structure and for-

[2] https://de.mathworks.com/help/simulink/ug/controlling-and-displaying-the-sorted-order.html, accessed February 3, 2020

warding external connectors directly to these subcomponents. The result is a flat EMA model with only one level of atomic component instances. Since components are scope spanning symbols with an own visibility, names of subcomponent instances are concatenated with the name of the flattened parent component. For instance, if a component `Parent` has a subcomponent `child`, the latter is renamed into `Parent_child`.

Note that the result of this model transformation does not replace the input model, but only serves as an intermediate data structure. The user remains completely unaware of this step and only operates on the hierarchical model which is much easier to grasp.

At runtime all the components are executed sequentially based on the execution order list in each cycle. A cycle is finished when all components have been executed. The next cycle is started immediately, when the preceding cycle is finished. The duration of a cycle is not fixed and only depends on the executing hardware. This duration can vary from cycle to cycle. However, the definition of a minimum execution time is conceivable to limit the workload. An upper limit on the execution time can also be necessary in the context of realtime systems. If a component has not finished its execution within a given time limit, it would output a default value or a warning instead of the real result.

The absence of a communication delay of one or more execution cycles means that the semantics of EMA is non-strongly causal, as the input until time t does not necessarily determine the output until time $t + 1$. However, in EMA the input until time t completely determines the output until time t rendering the semantics weakly causal [BS12]. While strong causality is indispensable in asynchronous systems such as web applications, digital integrated circuit (IC) design or high performance computing (HPC), we will refrain from using it for our rather abstract and mathematical domain. The weakly causal semantics of EMA is more natural and efficient when modeling logical architectures, mathematical algorithms, and dataflows of non-distributed software.

As an example consider the two architectures in Figure 2.7. Both systems have the same semantics in EMA and can be described mathematically using the equation

$$C_k = (A_k + B_k)\, B_k, \tag{2.1}$$

where k is a sequential index. In contrast, if the system were strongly causal under the assumption that each subcomponent required n timesteps to compute and communicate the output, the equations describing `Main1` and `Main2` would become

$$C_k = (A_{k-n} + B_{k-n})\, B_{k-n} \tag{2.2}$$

for the left and

$$C_k = (A_{k-2n} + B_{k-2n})\, B_{k-n} \tag{2.3}$$

for the right model, respectively.

Finding an execution order for linear models, i.e. models without cyclic port dependencies, is straightforward: each component instance is put on the execution list after all

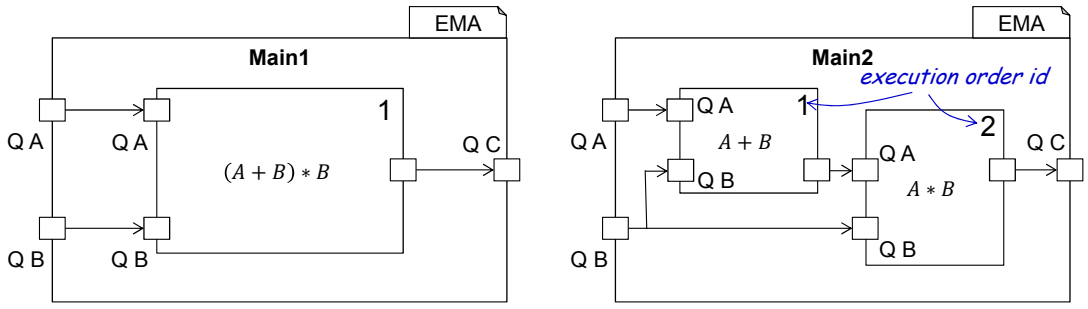


Figure 2.7: This example shows two C&C architectures `Main1` and `Main2`, which are semantically equivalent in EMA due to its synchronized and weakly causal execution model, but which might have different interpretations in a language with strongly causal semantics.

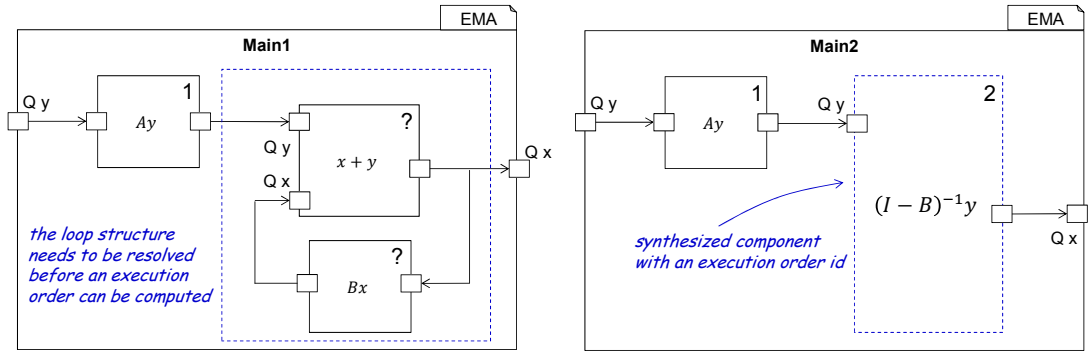


Figure 2.8: This example shows an architecture featuring an algebraic loop (`Main1`) and its loop-free equivalent synthesized automatically at compile-time (`Main2`).

component instances its input ports depend on. Consider the component `Main2` on the rhs of Figure 2.7. The subcomponent for multiplication must be executed after the one for addition because one of its input ports depends on the addition result. The execution order ids are given at the top right corner of each atomic component in this diagram.

The situation becomes more complex when cycles are present in the model, i.e. when there is a path from a subcomponent's output to its own input without a delay, cf. model on the left of Figure 2.8. Loops where all output ports depend only on the input ports of the current step (and not on previous steps) are referred to as *algebraic loops*. At compile-time, the compiler tries to replace arrangements with algebraic loops to equivalent loop-free subcomponents. Consider the block model on the left in Figure 2.8 modeling the system equation

$$x = Ay + Bx \tag{2.4}$$

with y being the input and x the output of the system. In this specific case, where the output is a linear function of itself and the input, the cycle can be removed by solving for x:

$$x = Ay + Bx \tag{2.5}$$
$$x - Bx = Ay \tag{2.6}$$
$$x = (I - B)^{-1} Ay \tag{2.7}$$

as long as $(I - B)$ is invertible. The pattern is detected by graph traversal when trying to find the dependencies of the summation component. Thereby, the compiler notices the circular dependency of its second input port from its output port. A new component is synthesized according to Equation (2.7). The new, synthesized architecture is depicted on the rhs of Figure 2.8. It has no loops and the compiler can determine the execution order based on the dataflow.

If an explicit solution cannot be found, i.e. if the loop does not correspond to a known (solvable) pattern, it can be solved at runtime using an algebraic solver. Since this must be done in each timestep and, what is even worse, there is no guarantee that a solution exists, a runtime solver would not only affect the runtime performance heavily, but might also lead to unpredictable behavior. This is obviously not a favorable solution, e.g. in safety-critical realtime systems. For this reason we only allow loops, which can be transformed into loop-free architectures at compile-time. If no such transformation can be found, the model is considered invalid.

So why are loops needed at all if they can be formulated as loop-free architectures? EMA is a specification language, which focuses on the *what* rather than on the *how*. Many technical problems can be formulated elegantly as an algebraic or differential equation system, while the explicit solution may have a cumbersome and unintuitive form. What is more, requiring that loops must be solvable at compile-time, loops can be seen as syntactic sugar for loop-free and hence, FOCUS-compatible models (cf. FOCUS theory [BS12]).

Obviously, to resolve algebraic loops, knowledge of the component behavior is required. A means to integrate appropriate behavior models into EMA components will be discussed in Section 2.4.

For non-algebraic loops, i.e. loops with stateful components, Simulink does not specify a single correct execution order. To resolve such loops, Simulink introduces the notion of direct-feedthrough ports. A direct-feedthrough port is an input port the current value of which determines the current value of at least one output port of the same component. The execution order is then created using two rules.

First, if a component instance drives a direct-feedthrough port of another component, it must precede this component in the execution order list. Second, components with non-direct-feedthrough ports can be put at any position in the execution order list.

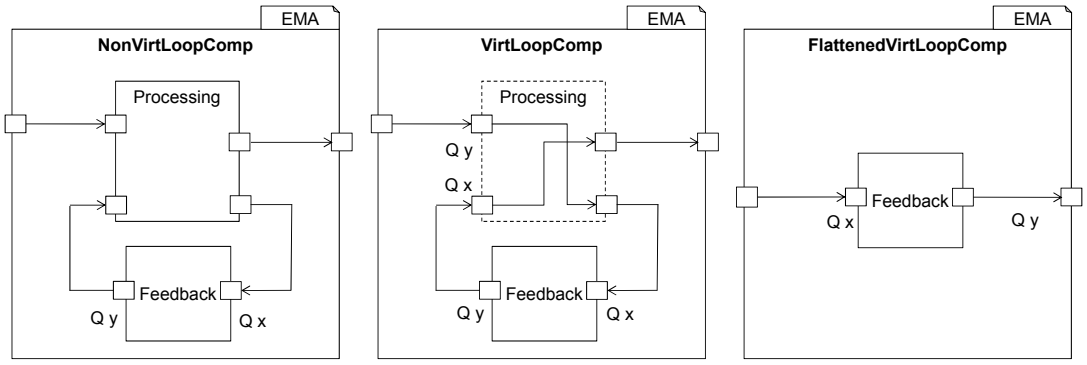


Figure 2.9: The component on the left contains a loop consisting of two non-flattened components. The model in the middle displays the contents of the `Processing` subcomponent making obvious that there are no circular dependencies on port level. The figure on the right shows how the component is seen by the compiler after the flattening step.

Therefore, such components are put in a non-specified order at the beginning of the execution list.

This ambiguity is unfavorable for EMA, as we want to keep the language understandable and thus the semantics clean, predictable, and reproducible. For this reason, non-algebraic loops are forbidden in EMA. However, while the scope of Simulink models often includes both the control algorithm as well as the plant to control, EMA is meant to design the architecture of the control software only, which excludes feedback loops. Hence, non-algebraic feedback loops inside of EMA architectures are usually not required. An exception however is when we need to access some past state of the EMA model. In such a case we need to use a `Delay<T>(Q default)` component. It takes a value of type `T` as input and outputs it in the following cycle. The `default` parameter specifies the output for the first cycle for which there is no previous input. The EMA scheduler puts all `Delay` components at the beginning of the execution list, breaking loops they are part of. Then the scheduler tries to resolve algebraic loops. If there are still circular dependencies left over after removing dependencies from `Delay` components and resolving algebraic loops, the model is not valid and the compiler delivers an error. Note that loops which are present in the original model sometimes disappear during the flattening step. This is the case if there is no pair of input/output ports so that the output port depends on the input port, and vice versa. An example is shown in Figure 2.9. The component `NonVirtLoopComp` on the left contains a loop featuring the two original components, `Processing` and `Feedback`. In `VirtLoopComp` in the middle, the contents of the `Processing` component are shown, making obvious that there is no circular dependency on port level. Flattening this model leads to

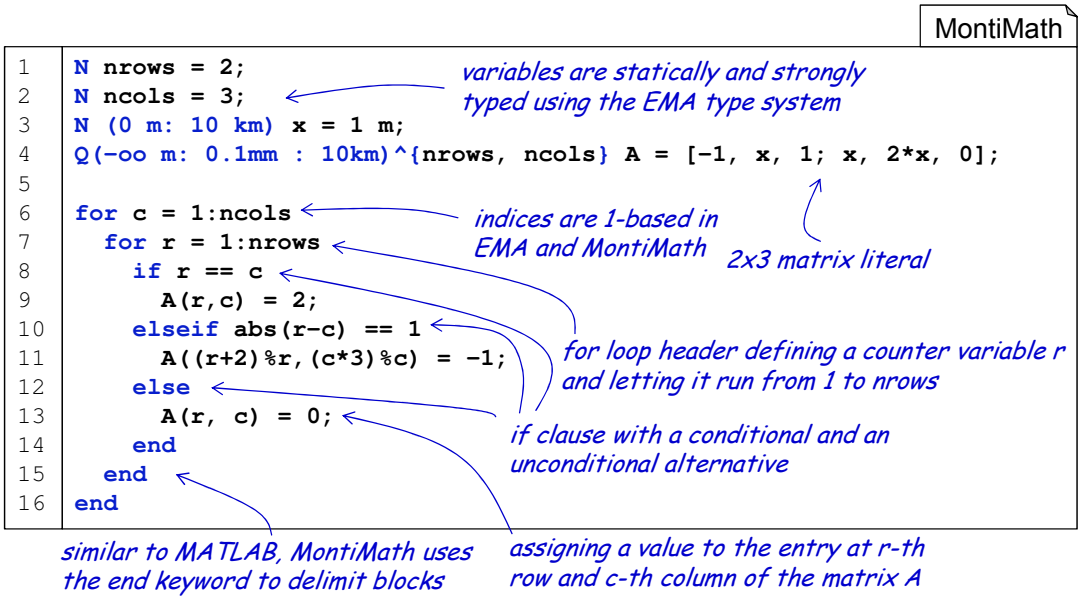


Figure 2.10: This listing shows a simple MontiMath example exhibiting the main language constructs including variable declarations, matrix literal definitions, loops and if statements.

`FlattenedVirtLoopComp` depicted on the right. This version obviously has no loops.

A future EMA extensions should enable modeling differential equation system as component loops, as well. For this case special differentiator and integrator components may be used in addition to arithmetic operations. This will enable the developer to model complex dynamic systems computing movements of objects, electric flows, and many other real-life problems. As for algebraic loops, a modeled differential equation should only be valid if it can be solved and discretized at compile-time.

2.4 MontiMath

2.4.1 Basic Syntax

MontiMath is an imperative language developed for the design and implementation of math-heavy algorithms. It has been inspired by MATLAB's matrix-oriented paradigm. The core grammar of MontiMath is given in Listing B.3. To facilitate comparisons of the two languages, translations from one language into the other, and vice versa, and to make MontiMath readable for MATLAB developers, the syntax of MontiMath, including important function names, has been taken over from MATLAB as far as possible. Built-in MontiMath functions and operators are side-effect free so that MontiMath code always

results in clean components. An example showing the basic language constructs is given in Figure 2.10. However, in contrast to MATLAB, MontiMath uses the EMA types system, which makes it a statically and strictly typed language similar to EMA itself. Declaration of a MontiMath variable requires a type definition, which is expressed by preceding the newly declared variable by an EMA type, e.g. `Q(0 Ohm : 1 nOhm : 1 MOhm)^{2,2} impedance`. The syntax to define a matrix constant is the same as in MATLAB, but the literals inside the matrix can be enriched by SI units if needed. As in MATLAB, a matrix constant is defined in square brackets. Thereby, columns and rows are separated by commas and semicolons, respectively. The initialization of the impedance matrix `impedance` modeling a two-port network can hence be written as `impedance = [10 Ohm, 5 Ohm; 6 Ohm, 8 Ohm];`. The grammars for matrix literals and matrix-specific expressions are given in Listing B.1 and Listing B.2, respectively.

To maintain compatibility to MATLAB, MontiMath indices start with 1 as opposed to most GPLs, where arrays are zero-based. Scalars are treated as 1×1 matrices, but the square brackets can be dropped when defining a scalar literal. Other than in MATLAB, statements, except conditional statements and loops, need to be terminated with a semicolon.

MontiMath supports the typical operators needed in many computations including addition (+), subtraction (-), multiplication (*), division (-), and power (^). If applied to matrices, these operators perform the corresponding algebraic matrix operation, e.g. a matrix multiplication. Division by a matrix, e.g. `A/X`, is semantically equivalent to multiplying the dividend with the inverse of `X`, i.e. `A/X` is equivalent to `A*X^-1` or `A*inv(X)`.

Furthermore, MontiMath supports the Hadamard product or element-wise multiplication (.*), inverse Hadamard product (./), and element-wise power (.^). The standard power operator (^) does not take matrices as exponents. However, the function `exp(x)` representing the mathematical function e^x can cope with square matrix-valued `x`. Then, the result is an approximation of the Taylor series of the matrix exponential

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k. \tag{2.8}$$

The transpose operation for real and the Hermitian transpose operation for complex-valued matrices can be expressed by appending the apostrophe operator (`'`) to a matrix name, e.g. `A'`. `A` can be an arbitrarily shaped matrix. The returned result of the operation is a matrix with swapped dimension sizes, i.e. if the base is of size $m \times n$, the result is of size $n \times m$. Furthermore, the entries are conjugated in the complex case.

The operators `+, -, .*, ./, .^` work on arbitrarily sized operand matrices A, B, but the operands' dimensions must coincide, i.e. $A, B \in M_{m,n}, m, n \in \mathbb{N}$. The circumflex operator `^` works only for square matrices as a basis and scalar exponents. `*` works on any operands A, B as long as the number of columns of the first operand is equal to the number of rows of the second operand, i.e. $A \in M_{m,n}, B \in M_{n,l}$.

MontiMath supports the standard control flow constructs including `for` loops and `if` clauses, enabling us to write arbitrarily complex algorithms. Again, the syntax is based on MATLAB.

The composition of a cyclically executed language such as EMA with an implementation language introduces a problem: in general, a language such as MontiMath is not aware of the repeated execution and variables are not passed from one cycle to another. On the other hand, a computation might depend on previous cycles, e.g. in a low pass filter. To solve this issue, we introduce the `static` keyword. A variable declared with this modifier, e.g. `static Q cumulativeError`, is saved in a cycle-independent scope. Its value does not get lost when an execution cycle is finished and can be reused in the next cycle. Alternatively, the problem can be modeled using delay blocks. A variable which is needed in subsequent execution cycles can be output through an output port, passed to a delay component, and input into the same component in the next cycle.

2.4.2 Deriving Matrix Properties for Concrete Matrices

Due to the strict type system of MontiMath including matrix dimensions, many errors, which only occur at runtime in dynamically typed languages, can be caught at compile-time using CoCos. Given a concrete matrix literal, we can derive its size directly by counting its rows and columns. First, this needs to be done to ensure that the matrix is valid, i.e. that each row has the same number of columns. Second, we need to ensure that the dimensions of the matrix match with the dimensions of the defined variable type the constant is assigned to. Third, when used in an expression, we need to ensure that the operands' dimensions match.

Furthermore, if a matrix or the result of an operation executed on this matrix is to be assigned to a variable defined with a matrix property list, we first check if the required properties are compatible. Two properties are compatible iff there is a directed path between them in Figure 2.1. For instance, the properties `psd` and `skew` are mutually exclusive and a variable definition containing two incompatible properties, e.g. `psd skew Q^{2,2} M;`, would result in a compile-time error.

After ensuring that the matrix is square, we check the required properties constructively one by one, starting with the most specific ones. However, the Hermitian property is checked before definiteness, since the latter is defined using the former. The matrix properties of a matrix M are checked as follows (other implementations are possible):

inv: we compute the determinant. The property holds if $\det(M) \neq 0$.

norm: we compute MM^* and M^*M. The property holds if the two are equal.

sym: we check element-wise that $\forall i \in [1,n], j \in [i,n] : M_{ij} = M_{ji}$.

herm: we check element-wise that $\forall i \in [1,n], j \in [i,n] : M_{ij} = M^*_{ji}$.

Prolog

```
%herm(m1,m2,op).
herm(X,Y,' + '):- herm(X), herm(Y).
herm(X,Y,' - '):- herm(X), herm(Y).
herm(X,Y,' * '):- herm(X), scal(Y).
herm(X,Y,' * '):- herm(X), int(Y).
herm(X,Y,' * '):- herm(Y), scal(X).
herm(X,Y,' * '):- herm(Y), int(X).
herm(X,Y,' ^ '):- herm(X), int(Y).
herm(X,'inv'):- herm(X), inv(X).
herm(X,'trans'):- herm(X).

%diag(m1,m2,op).
diag(X,Y,' + '):- diag(X), diag(Y).
diag(X,Y,' - '):- diag(X), diag(Y).
diag(X,Y,' * '):- diag(X), diag(Y).
diag(X,Y,' * '):- diag(X), scal(Y).
diag(X,Y,' * '):- diag(X), int(Y).
diag(X,Y,' * '):- diag(Y), scal(X).
diag(X,Y,' * '):- diag(Y), int(X).
diag(X,Y,' ^ '):- diag(X), int(Y).
diag(X,'inv'):- diag(X), inv(X).
diag(X,'trans'):- diag(X).

…

%conclusionssquare(X):- norm(X).
norm(X):- diag(X); herm(X); skewHerm(X).
herm(X):- pd(X); psd(X); nsd(X); nd(X); indef(X).
psd(X):- pd(X).
nsd(X):- nd(X).
```

Figure 2.11: An excerpt of the Prolog rule set for the derivation of matrix properties.

skew: we check element-wise that $\forall i \in [1,n], j \in [i,n] : M_{ij} = -M^*_{ji}$.

diag: we check element-wise that $\forall i \neq j \in [1,n] : M_{ij} = 0$.

psd: we first check if the matrix is Hermitian. Then we check if all eigenvalues are greater than or equal to zero.

pd: we first check if the matrix is Hermitian. Then we check if all eigenvalues are greater than zero.

nsd: we first check if the matrix is Hermitian. Then we check if all eigenvalues are less than or equal to zero.

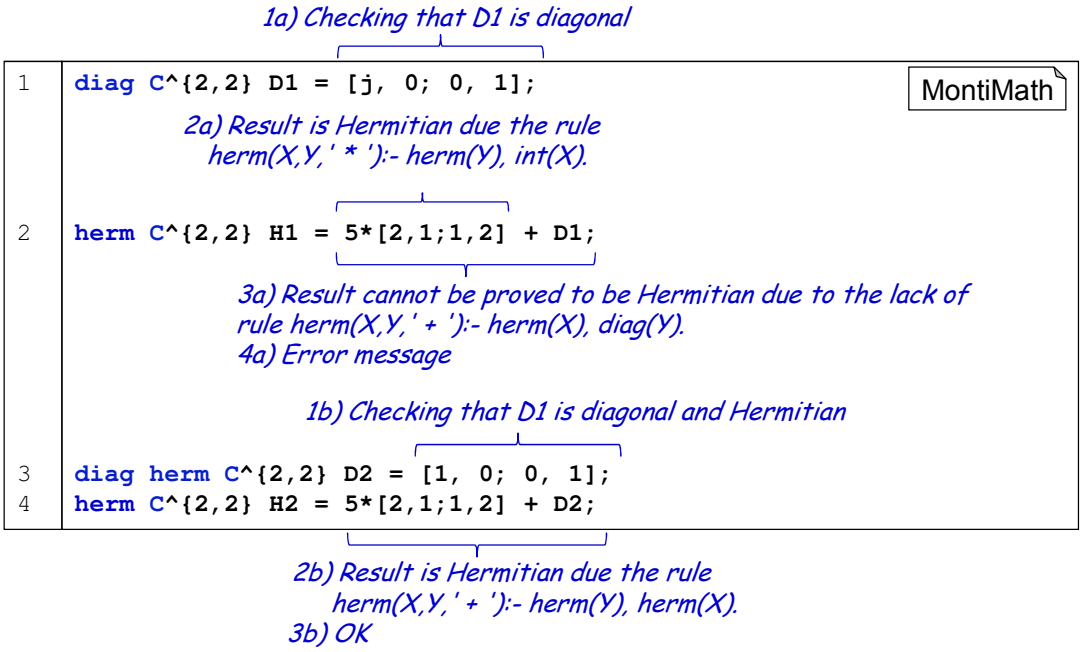


Figure 2.12: An example of matrix property derivation for operations.

nd: we first check if the matrix is Hermitian. Then we check if all eigenvalues are less than zero.

indef: we first check if the matrix is Hermitian. Then we check that at least one eigenvalue is greater and at least one is less than zero.

The properties of a matrix are saved in a list attached to the matrix symbol. Before checking computationally if a property holds, we would first look up the property in the property list.

Consider the assignment `norm diag herm indef Q^{2,2} = [0,1;1,0];`. Based on Figure 2.1, we can see that indefiniteness is the most specific property here. We hence first check whether the matrix is Hermitian. As this is the case, we can immediately confirm that the matrix is normal, as well, and don't need to check this property explicitly. Next, we check that the matrix is indefinite by verifying that it has both positive and negative eigenvalues which in this example is true with the eigenvalues being $e_1 = -1, e_2 = 1$. Last, we check for the `diag` property which does not hold, since we can find off-diagonal non-zero entries. The result is a type check error.

2.4.3 Deriving Matrix Properties for Operations

While proving the presence or absence of properties for a concrete matrix literal is straightforward as was discussed in Section 2.4.2, the task gets more involved for expressions. Operations on matrices can be property preserving, property generating, or neither. In most cases, no statement can be made. For instance, the sum and the product of two diagonal matrices is diagonal; the sum of two Hermitian matrices is still Hermitian, but the product of two Hermitian matrices does not have to be Hermitian. This leads to an extension of the algebraic type checking facility of EMA and MontiMath as described in the following.

We encode a rule set for the derivation of matrix properties based on [HJ90] as a Prolog database. An excerpt is given in Figure 2.11. Further rules are provided in Figures A.1 and A.2 in Appendix A. For each property a corresponding predicate exists. Each rule states that a property holds after an operation if the operands fulfill the required properties. For instance, L.2 in Figure 2.11 states that the predicate `herm` (Hermitian property) holds for the result of the + operator if the two operands are Hermitian, as well.

Whenever the lhs of an assignment is a matrix with properties, the rhs undergoes a sort of symbolic execution on matrix properties using these Prolog rules. Therefore, the compiler adds all known properties of the matrix variables as facts to the database and queries it for the desired property.

Consider Figure 2.12 holding a negative example in L.1-2 and a positive one in L.3-4. In L.1 the matrix variable `D1` is initialized with a concrete matrix. The variable is annotated with the `diag` property and hence, the rhs needs to be checked for being diagonal. Since the value of the rhs is known, the compiler uses the rules discussed in Section 2.4.2 to verify the property. Obviously, the matrix is diagonal and so the assignment is valid. The compiler caches the properties found for `D1` to reuse them later if needed.

L.2 is an assignment again, but now the matrix variable needs to be Hermitian. Since the concrete value of `D1` can be looked up in the previous line, the compiler can evaluate the Hermitian property using the concrete result of the sum. Obviously, it is not Hermitian as the result contains an imaginary entry on the main diagonal. However, if the concrete value of `D1` were not known, e.g. if it came from a component port, the compiler needed to execute the sum symbolically. The only rule applicable to our expression is the one in L.2 of Figure 2.11 stating that the sum of two Hermitian matrices is Hermitian. Hence, the compiler needs to check if the two summands are Hermitian. For the first summand the rule in L.6 of Figure 2.11 states that the product of a scalar with a Hermitian matrix is Hermitian. For the atomic matrix of this product the Hermitian property has been ensured explicitly and is present in the fact database.

Next, we need to prove that the second summand is Hermitian, as well. Assuming that we do not have access to the concrete value of `D1`, the compiler can only be assured

that `D1` is diagonal. Although, theoretically the sum can still be Hermitian, this cannot be guaranteed and the compiler throws a compile-time error.

The example in L.3-4 of Figure 2.12 is similar with the difference that `D2` as opposed to `D1` is declared as diagonal *and* Hermitian which, again, is verified by checking its entries. Hence, the compiler can prove that the sum is indeed Hermitian using the rule in L.2 of Figure 2.11. Note that under the assumption that the compiler had no access to the concrete value of `D2` and if it only knew that `D2` were diagonal (but there were no `herm` property attached to the symbol), it would still fire a compile-time error although the sum is actually Hermitian given the concrete value in L.3.

Besides acting as pre- and postconditions, matrix properties can be used by the compiler as hints for code optimization:

Memory consumption: due to the fixed dimensions, the compiler knows exactly how much space it needs for the matrices used in the model. There is no need to allocate space dynamically.

Dynamic programming: due to the associative property, the order of matrix multiplications in a chained multiplication expression can be changed. Unless the matrices are squared, it might be beneficial to do so in order to save operations. The optimal order can be found by solving a dynamic program. This can be done at compile-time, since the dimensions are fixed.

Diagonal matrices: knowing that a matrix is diagonal helps us save space and computation time. A diagonal `Q^{n,n}` matrix can be saved efficiently as an `n`-element vector. Multiplication with a vector has a complexity of $\mathcal{O}(n)$ instead of a standard matrix-vector multiplication complexity of $\mathcal{O}(n^2)$. Multiplication with an arbitrary `Q^{n,n}` matrix is reduced to $\mathcal{O}(n^2)$ compared to the multiplication complexity of $\mathcal{O}(n^{2.373})$ for two arbitrary square matrices using optimized Coppersmith-Winograd-based algorithms [DS13, Wil11, LG14]. Multiplication with an arbitrary `Q^{n,m}` or `Q^{m,n}` matrix has a complexity of $\mathcal{O}(nm)$ instead of standard matrix-multiplication complexity of $\mathcal{O}(mn^2)$. Furthermore, inversion complexity is reduced from $\mathcal{O}(n^{2.373})$ when using optimized Coppersmith-Winograd-based algorithms to $\mathcal{O}(n)$, since we only need to compute a single square root per diagonal entry. Similarly, the symmetric and Hermitian properties can be used by the compiler for optimization, as only one half of the matrix needs to be kept in memory.

2.4.4 EmbeddedMontiArcMath

Although MontiMath is an abstract language with a syntax close to the mathematical domain, it is used to specify a concrete behavior leaving no room for interpretations. Consequently, enriching leaf components of an EMA model with a MontiMath behavior leads to a complete level 3 (technical concept layer) SMArDT model. Model linkage

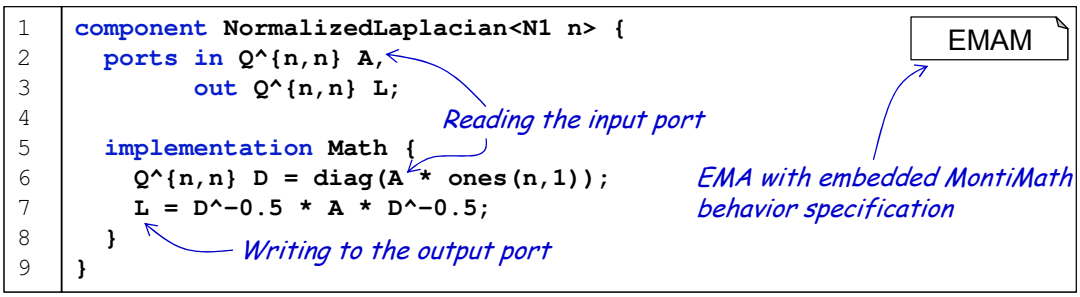
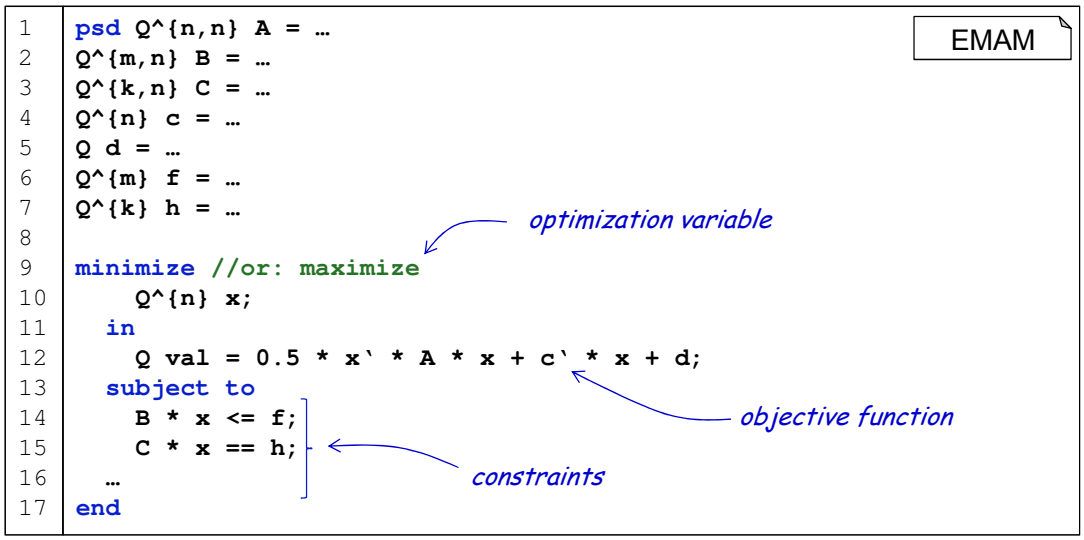


Figure 2.13: An EMAM model embeds a MontiMath script into an EMA component, thereby leveraging the latter from SMArDT level 2 to SMArDT level 3.

is implemented using the language extension and embedding framework of MontiCore. Therefore, an EMA component is extended with an implementation block. Such an implementation block is initiated by the keyword `implementation` followed by the name of the language and the actual implementation block delimited by curly brackets, cf. behavior grammar in Listing B.8. This intermediate grammar extends the EMA grammar and can be used to embed arbitrary implementation languages. In our concrete example, the implementation language is set to MontiMath (or just Math) in the grammar in Listing B.9. Components with a MontiMath implementation are referred to as EmbeddedMontiArcMath (EMAM) components. In EMAM components the `implementation` keyword is followed by the language name `Math`. The MontiMath script is put thereafter in a block delimited by curly brackets, cf. Figure 2.13. It has reading access to the input ports and writing access to the output ports of the EMA host component. All components featuring an implementation block instead of a subcomponent structure are considered atomic and are not flattened by the compiler, although, theoretically, each MontiMath command could be mapped to a dedicated component instance. Interface changes in the EMA model affect the MontiMath model directly, and vice versa. Hence, it is not possible to have diverging SMArDT level 2 and level 3 models.

2.4.5 Optimization in MontiMath

Many tasks in CPS engineering can be expressed as optimization problems, cf. Section 1.3.2. For this reason it is important that a CPS engineering methodology provides appropriate optimization tools. We tackle this requirement by introducing optimization statements in MontiMath. The syntax for optimization problems is defined as an extension to MontiMath in the grammar in Listing B.4. It is roughly based on CVXGEN [MB12] due to its closeness to the mathematical notation. Consider the quadratic




```
1  psd Q^{n,n} A = …
2  Q^{m,n} B = …
3  Q^{k,n} C = …
4  Q^{n} c = …
5  Q d = …
6  Q^{m} f = …
7  Q^{k} h = …
8
9  minimize //or: maximize
10     Q^{n} x;
11   in
12     Q val = 0.5 * x` * A * x + c` * x + d;
13   subject to
14     B * x <= f;
15     C * x == h;
16   …
17 end
```

Figure 2.14: A MontiMath optimization statement representing a quadratic problem.

problem in Equation (2.9)-Equation (2.11):

$$\min_{x} \quad \frac{1}{2}x^T Ax + c^T x + d \tag{2.9}$$

$$\text{s.t.} \quad Bx \preceq f \tag{2.10}$$

$$Cx = h, \tag{2.11}$$

where $x, c \in \mathbb{R}^n$, $d \in \mathbb{R}$, $f \in \mathbb{R}^m$, $h \in \mathbb{R}^k$, $A \in \mathcal{M}_n$, $B \in \mathcal{M}_{m,n}$, $C \in \mathcal{M}_{k,n}$, $k, m, n \in \mathbb{N} \setminus \{0\}$, and $\preceq$ denotes the element-wise less than equal operator. The corresponding MontiMath syntax to write down this problem is given in Figure 2.14.

The syntax provides dedicated keywords for optimization problems to come as close as possible to the original mathematical formulation. This is in contrast to GPL-based application programming interfaces (APIs) offered by solvers like Gurobi[3]. Such APIs can be very solver- and host language-specific leading to an unintuitive syntax.

First, we need to make sure that constants used in the optimization problem are defined beforehand, cf. L.1-7 in Figure 2.14. Although not strictly necessary, we equip `A` with the `psd` property in L.1, guaranteeing that the minimization problem has a solution. The actual optimization statement is initiated using the keyword `minimize` in L.9. Depending on the problem to be solved, the `maximize` keyword can be used instead. The keyword is followed by a declaration of the optimization variable. The type must be numeric, i.e. Booleans, structs, and enums are not allowed. However, it can be

[3]https://www.gurobi.com, accessed August 19, 2020

a vector of arbitrary dimensionality. Note that we use `Q` here as an approximation of $\mathbb{R}$ used in the original formulation.

The objective function follows the `in` keyword and is given as a single expression. In our case it is a quadratic function, cf. L.12. It is assigned to the variable `Q_val`, which contains the optimal value of the objective function after the optimization step. The argument yielding the optimum can be accessed using the optimization variable `x` when the problem is solved.

Optionally, a set of arbitrarily many constraints can be defined next. To do so, a constraint block is initiated using the `subject to` keyword. Each constraint is an equality or an inequality expression depending on the optimization variable. The optimization problem is closed using the `end` keyword, as is done in L.17.

2.4.6 Component Variabilty for Self-Adaptable Systems

CPSs rely heavily on signal processing and control components like filters, transformations, PID controllers, and others. Usage of such components, however, mostly requires a reasonable parameterization which is non-trivial in complex systems. Usually, parameters of such components are optimized with regard to specific objectives and are highly dependent on the application. We illustrate the problem by the example of the widely used PID controller. The task of such a PID block is to control a process variable, e.g. the velocity of a vehicle, based on a given target function, e.g. a concrete desired velocity by using the control function

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{\mathrm{de(t)}}{\mathrm{dt}} \tag{2.12}$$

or its discretized version, where e is the measured deviation of the current from the desired state and u is the control action reacting to this deviation, e.g. the pedal position. The behavior of the process variable might vary drastically depending on the parameters K_P, K_I, K_D. A requirement for the tuning of these parameters might be a reasonable trade-off between stability and fastness of response. Usually, the tuning takes place at

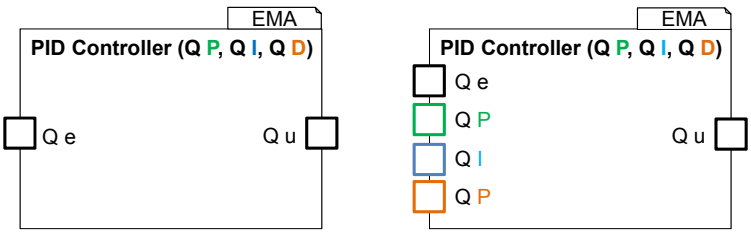


Figure 2.15: A statically parameterized and an extended, runtime-adaptive PID component interface.

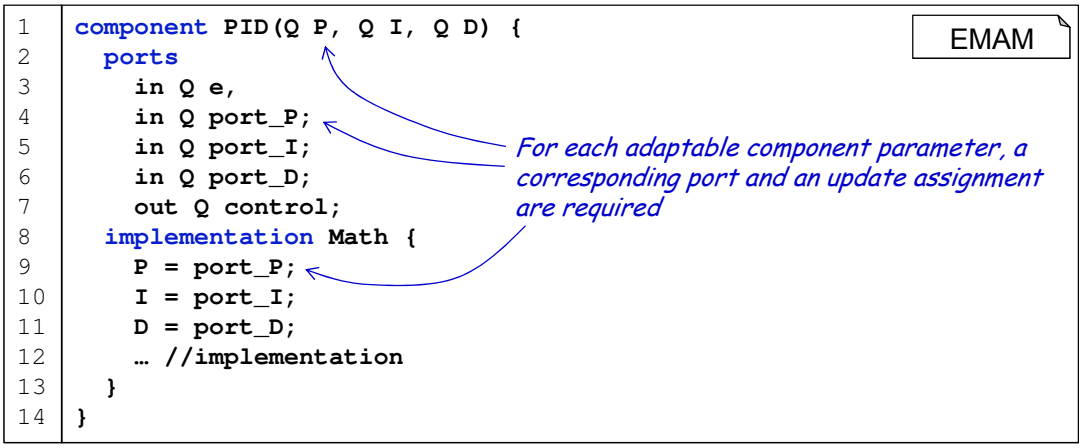


Figure 2.16: The PID component with parameters adaptable through dedicated ports.

design time and the parameters are fixed at compile time. The corresponding component interface is depicted on the left side of Figure 2.15. It receives the error e through an input and delivers the action u through an output port. The PID parameters are provided as component parameters.

However, the operating conditions of a cyber-physical system tend to change in complex environments. It is therefore necessary to re-evaluate and fine-tune the system parameters on a steady basis. In Simulink the desired parameters of a PID component can be configured using its settings menu or by calling the built-in function `set_param(Object, ParameterName, Value)`. The latter also allows one to programmatically configure the component, but is not available for code generation and can, hence, not be used at runtime. In order to enable runtime parameter reconfiguration the user has to set the parameter source to *external*, thereby adding new ports to the component instance, as depicted on the rhs of Figure 2.15. Through these ports the component parameters can be adapted at runtime from outside. However, many Simulink components such as *Communications System Toolbox* filters do not support external sources for configuration parameters. What is more, the user interface for changing the parameter source varies from toolbox to toolbox.

To model runtime reparameterization of signal processing components in EMA, we can employ the following implementation pattern, thereby obtaining the model in Figure 2.16 which is depicted graphically on the rhs of Figure 2.15:

- Identify the component parameters to be runtime-adaptable in a statically parameterizable component, e.g. the PID on the left in Figure 2.15.

- Replicate these parameters as ports.

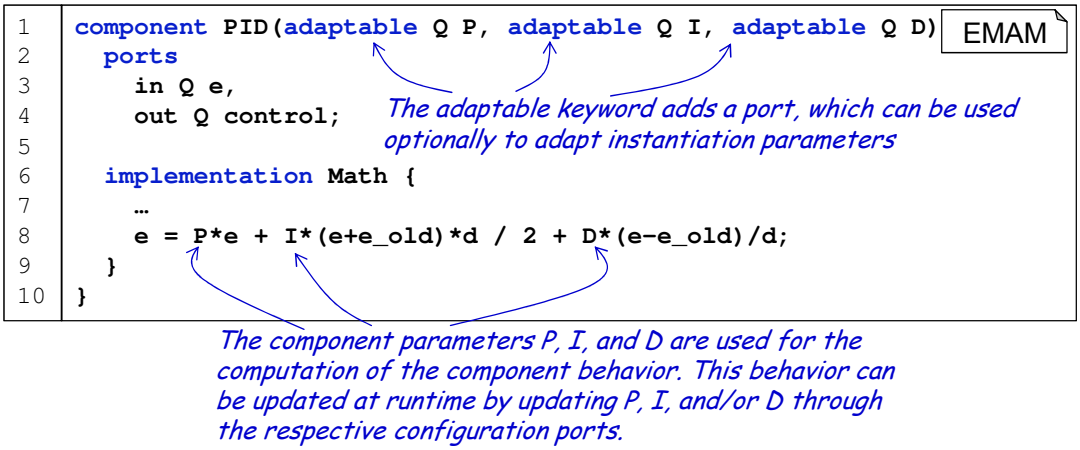


Figure 2.17: The PID component interface with adaptable parameters defined in EmbeddedMontiArc.

- Override these component parameters' values in the component implementation with the corresponding ports' inputs (an initial parameterization using component parameters is indispensable, since the big part of the parameter search needs to be done at design-time).

This implementation pattern is a repetitive task. Furthermore, if both a statically parameterized and an adaptive version of a component should be available, the component needs to be replicated. Mostly, implementation or design patterns arise from the lack of native language support for solving a specific problem. Consequently, we are able to tackle the runtime reconfiguration problem by introducing a new language feature in EMA: the `adaptable` component parameter modifier.

Component parameters which should potentially be adaptable at runtime are marked with the *adaptable* keyword, as is shown in Figure 2.17. On instantiation of a component, for each parameter marked with the `adaptable` modifier, a corresponding *configuration port* is created implicitly. Data written to this port is used to override the respective component parameter value, otherwise the parameter value remains constant. A configuration port is very similar to an ordinary data port. Nevertheless, it manifests several semantical differences, thereby increasing the reusability of the component while reducing the model complexity and improving the efficiency of the generated code.

In an EMA model, each input port has to be an endpoint of a connector which is checked at compile time using a context condition. This excludes configuration ports which can remain unconnected. In this case the respective configuration parameter will keep its original value forever; the port remains invisible for the user and no code related to this port is generated. It cannot be deduced from the target code that the port

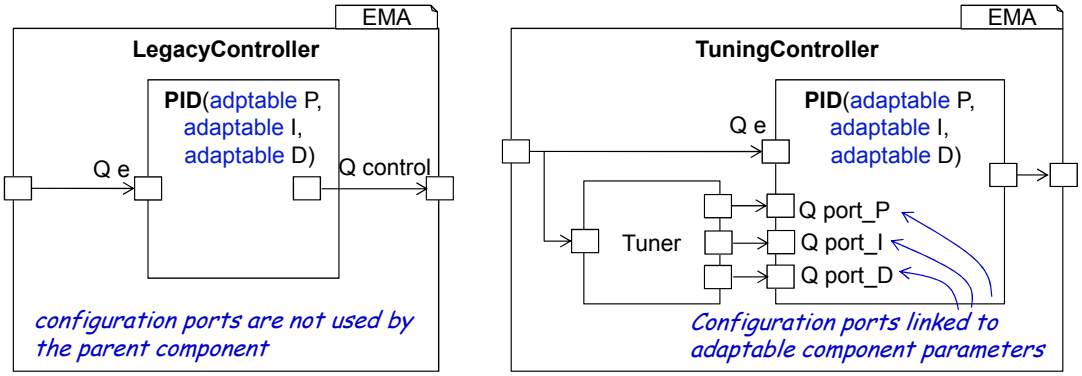


Figure 2.18: The components LegacyController and TunableController depict the static and the adaptable usage of the same PID component.

actually existed in the model. This ensures that adaptable components can be used in conventional, non-adaptive systems, where the parameters are bound at compile time, without having to remove the adaptable modifier. We hence don't need to maintain two component variants, an adaptable and a non-adaptable one. Both the adaptive and the static variant are covered by one model. As a consequence, configuration ports cannot be used in the implementation block explicitly. Otherwise, the behavior would become inconsistent if a port were not connected but used by the developer.

Second, if one or more configuration ports have incoming connectors, in each execution cycle, the *behavior phase* of the component will be preceded by a *reconfiguration* phase, where the configuration parameters will be set to the values at their respective *connected* configuration ports.

Legacy components can be upgraded easily by adding the `adaptable` modifier to the component parameters of concern. Note that parameter updates only affect future uses of the parameter of interest in the component behavior implementation, e.g. in MontiMath. Accessing an adaptive parameter as is done in the MontiMath implementation in Figure 2.17 will yield the latest update. Furthermore, if a component uses its adaptable component parameters to initialize adaptable component parameters of its subcomponents, runtime updates of the parent component's adaptable parameters will propagate automatically to the corresponding adaptable parameters of the subcomponent, cf. Figure 2.19. Updating adaptable component parameters at runtime will have no impact on the decisions taken before the update. In particular, a component's structure cannot be changed by updating an adaptable parameter at runtime. For instance, if a component parameter determines the number of subcomponents in a component array, adapting the value of this parameter will not change the number of subcomponents. Adaptable parameters serve for the tuning of the behavior computations at runtime, not to change structural properties. If the behavior computation does not depend on the

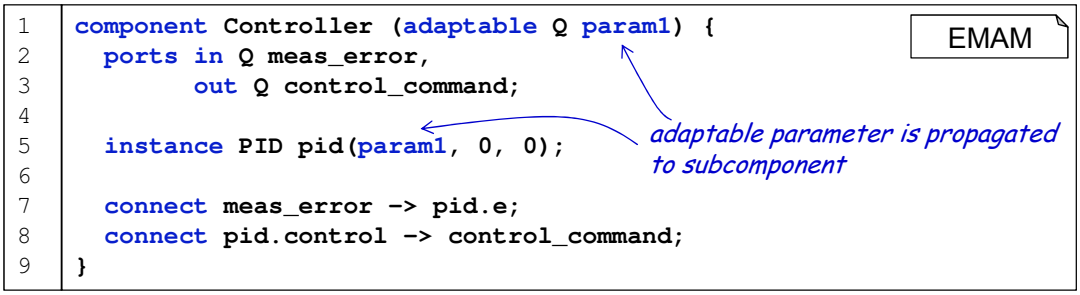


Figure 2.19: The main controller has an adaptable parameter param1. This parameter is used to initialize the (also adaptable) parameter of the PID subcomponent. Whenever param1 of the parent component gets updated at runtime, the update is propagated immediately to the PID subcomponent.

adaptable parameter, updating it will have no effect. A context condition verifies for EMAM components that adaptable parameters are used in the MontiMath implementation. Furthermore, in composite components, it is checked for each adaptable parameter whether it can be propagated to an adaptable parameter of at least one subcomponent.

In practice, a self-adaptive black box system can be realized using adaptive components combined with evolutionary or other tuning algorithms [HKK+18]. The measured error is then used as an input to the parameter tuner computing updated parameters and writing them back to the adaptive ports of the controller (which in turn can distribute these parameters to its subcomponents). If, furthermore, the controller function as well as its first derivative are known, gradient descent optimization can be used instead of an evolutionary search.

As an example of static and adaptable component usage, consider Figure 2.18: the components `LegacyController` and `TunableController` use the very same PID component in their internal structure. While the `LegacyComponent` sets the PID parameters only at instantiation and keeps them static, the `TunableController` has a `Tuner` component, which based on the error signal and, possibly, further inputs computes updates for the PID parameters, which are input into the configuration ports of the respective adaptable component parameters of the PID component. These configuration ports are available for optional usage by external components, since the corresponding component parameters have been declared adaptable.

2.5 Code Generator

Since C++ is a widely used language in the embedded and cyber-physical systems domain, we have chosen to use it as the target language for our code generation toolchain. The absence of a virtual machine or a runtime interpreter in C++ as used by Java or

Python but also its static type system are a pre-condition for high performance, energy efficiency, and realtime capability. Hence, choosing C++ supports our efficiency requirement RE4.2. Nevertheless, the choice of C++ does not exclude the applicability of EMA to web applications. The generated C++ code can be rendered executable in web browsers by transpiling it to WebAssembly (WASM)[4]. A proof-of-concept has been shown in an experiment, where EMA-based autonomous driving models were translated to WASM and executed in a web-based simulator [KRSvW18b].

To render EMA models executable without the addition of manual code as demanded by RE4, a full code generator needs to handle two major concerns: generating the structure based on the component architecture, referred to as EMA2CPP, and generating the behavior from MontiMath statements, referred to as MontiMath2CPP. The whole generator is referred to as EMAM2CPP. Further behavior generators can be added to support further behavior languages, cf. Chapter 4. This is similar to the code generator composition approach of MontiArcAutomaton which distinguishes between behavior, component, and data type generators [RRRW15].

In the generated C++ code, we waive the usage of third-party libraries as far as possible in order to keep it slim and efficient and to facilitate the distribution of binaries. There are several exceptions however: for algebraic computations, we use Armadillo, a high level linear algebra library providing a large set of matrix functions [SC16] (as an alternative GNU Octave [Eat93] was considered as a linear algebra backend, but the library turned out to be inferior in terms of speed when compared to Armadillo [KRSvW18a]). For the generation of MontiMath functions related to computer vision (CV), e.g. `dilate(.)` and `erode(.)`, we use OpenCV [Bra00]. The Interior Point Optimizer (Ipopt) solver is used to solve optimization problems at runtime [WB06]. Further third-party software is used for machine learning and middleware communication, cf. Chapters 4 and 6, but is not essential for the core toolchain described in this chapter.

The structure generation of EMA architectures is straightforward. Each atomic component type present in the model after the flattening step is generated as a C++-class. Ports are generated as variables of the corresponding type. Since EMA has a proprietary type system, a conversion concept is required which maps EMA types to C++ types. `B` represents Booleans and is mapped to the corresponding C++-type `bool`. Scalar types, i.e. `N`, `N1`, `Z`, `Q`, as well as their bounded refinements are mapped to `int` if the type contains integers only and `double` otherwise. Complex numbers are generated as `std::complex<double>`.

Dense column vectors, row vectors, matrices, and cubes are generated to the corresponding Armadillo types `arma::Col<T>`, `arma::Row<T>`, `arma::Mat<T>`, and `arma::Cube<T>`, respectively. Here, `T` is the representation of the primitive type as described above, e.g. `Q^{k,m,n}` is generated as `Cube<double>(k,m,n)`, where `k`, `m`, `n` represent the number of rows, columns, and slices, respectively. Matrices and cubes

[4] https://webassembly.org/, accessed: October 2020

defined with the `diag` predicate are generated as vectors, as well. Hypercubes are not supported by the generator as they are rarely needed in practical applications, but would introduce unnecessary complexity.

An `init()` method is used to initialize the ports with appropriate constructors if necessary. Furthermore, an `execute()`-method executes the behavior of the component. If the component consists of subcomponents, the behavior implements the execution order of the subcomponents by calling the subcomponents' `execute()`-methods and copying the results at output ports to the designated input ports according to the connectors of the model. The generated code implements the synchronous, weakly causal semantics as introduced in Section 2.3.3.

The MontiMath2CPP-generator creates C++-code from the MontiMath code inside the implementation block of the enclosing component. Control flow constructs are mapped to their C++-counterparts. Armadillo overrides the standard arithmetic operators +, -, *, and /, i.e. we can use these operators in the generator for matrices, as well. For element-wise multiplication (implemented using the .* operator in MontiMath), Armadillo offers the % operator.

MontiMath provides a library of built-in functions, many of them for matrix operations, e.g. `det(.)`, `eigval(.)`. The MontiMath generator maintains a function registry containing information how to map MontiMath function names to C++-code and which parameters the functions take. Furthermore, the registry informs the generator how the result is provided by the target function. While some functions use a return statement to return the computed result, others make use of out-parameters, i.e. the function receives a pointer as a parameter indicating where the result should be stored. However, being a DSL, MontiMath is designed to be consistent and easy-to-read. All MontiMath function signatures have a return value which can be assigned to a target variable and there are no pointers, i.e. no output parameters are possible. If a function uses output parameters in the target language, wrapper code is generated to take this into account. Whenever a built-in function is used in MontiMath, the generator consults this registry to ensure that the function is supported, used correctly, and to find out how to obtain its result in generated code. Most of the built-in functions are realized as C++ standard library, Armadillo, or OpenCV functions.

However, not all allowed functions used can deal with Armadillo types. For instance, OpenCV defines its own algebraic types which are incompatible with Armadillo. To circumvent this incompatibility, the generator needs to insert a conversion mechanisms when a matrix is passed from an Armadillo-based function to an OpenCV-function, and vice versa. Armadillo vectors, matrices, and cubes are converted to `cv:Mat`. When a `cv:Mat` structure needs to be converted back to Armadillo, an `arma::Row`, `arma::Col`, `arma::Mat`, or `arma::Cube` is used depending on the number of dimensions and their sizes. The impact on the performance due to these type conversions is yet to be analyzed.

Component arrays are often used to model the application of an algorithm to multiple

data streams, cf. Section 2.3.2. However, beyond model clarity there is often no actual reason to actually use several instances of an algorithm. Therefore, if the component replicated in a component array is stateless, which the compiler is able to check, the generated code contains only one *flyweight* instance which is reused for all input streams for the sake of efficiency. The idea of *flyweight generation* is borrowed from the object-oriented flyweight pattern [GHJV95].

The generator is a fully automated conversion of SMArDT level 3 models, i.e. EmbeddedMontiArc and MontiMath or EMAM, into level 4 code. The resulting code can then be converted into the final product as required by SMArDT using the also generated CMake build files. The resulting binary is a library offering an interface consisting of writing access to the component parameters and the input ports of the system, a reading access to its output ports, as well as an `init()` and an `execute()` method. This library can be shipped as is or integrated into an application using it.

The generation process is a black box with little room for variation. Customizing this process for different target platforms will be discussed in Chapter 6. Note that there is no other possibility to define the behavior of a leaf component besides using an implementation block. In particular, EMA does not provide a way to embed hand-written code. This is to guarantee a clean design free of side-effects and technical details in logical components and to keep the generated code always consistent with the model. While changing the generated code is technically possible, such an approach is highly discouraged.

2.6 Model-Driven Unit-Testing

2.6.1 The Stream Language

EMAM

```
1  component Abs  {
2    ports
3      in val,
4      out absVal;
5
6    implementation Math {
7      output = abs(input);
8  }}
```

Figure 2.20: The Abs component under test.

Unit testing is a black box quality assurance method applicable to self-contained basic building blocks and the foundation of test-driven engineering processes. Depending on the language used, units can be functions (functional programming), objects (object-oriented programming (OOP)), but also components (C&C). The behavior of an EMAM

Stream

```
1  stream TestAbs for Abs  {
2      val: -100 tick -0.1 tick 0.0 tick 0.01 tick 20;
3      absVal: 100 +/- 0.001 tick 0.1 +/- 0.001 tick 0.0 +/- 0.01
4                            tick 0.01 +/- 0.001 tick 20 +/-0.001;
5  }
```

Figure 2.21: A stream model testing the Abs component.

component, i.e. its output, depends exclusively on the history of its input ports. Hence, an EMAM component can be tested easily by providing streams of predefined values at its input ports and by checking whether the produced output corresponds to the expected result. The EmbeddedMontiArc *stream* language enables the test designer to do just that. Consider the `Abs` component depicted in Figure 2.20. Assume that this component is required to compute the absolute value for a given input. It therefore has `Q`-typed input and output ports named `val` and `absVal`, respectively. The implementation applies the built-in MontiMath function `abs(Q val)` to the value at the input port `val` and assigns the result to the output port `absVal`.

To test the component, a stream test is defined in Figure 2.21. L.1 starts the stream test with the keyword `stream` followed by the test's name `TestAbs`. The component under test is specified in the same line after the `for` keyword. If the component does not reside in the same package as the stream test, the fully qualified name has to be used here so that it can be resolved via the symbol table. Multiple tests might exist for the same component. The body of a stream test consists of named streams with values being separated by the `tick` keyword and an optional tolerance range (e.g. `+/-0.001`), cf. L.2-4. The `tick` keyword is used instead of a comma or a semicolon to avoid confusions with element separators in matrices. The stream names correspond to the port names of the component under test. The semantics of a stream depends on the kind of the referenced port: for input ports, the given stream values are *applied* to the respective port while for an output port, the values computed by the component are *compared* to the stream. A stream test is passed when the values at the output ports obtained during a test run are within the tolerance bounds of the corresponding reference streams.

To ensure consistency, we need to compose EMAM and the stream language using MontiCore language aggregation. Before test execution, inter-model CoCos verify that the stream names have corresponding ports in the component under test and that their types are compatible. If the checks are successful, EMAM2CPP is used to generate executable code for the component under test as well as test code including the execution of the component, inputting the given streams into the input ports, and assertions on output values. The generated code is compiled, executed, and the test result is reported.

Maven

```
<plugins>
  <plugin>
    <groupId>de.monticore.lang.monticar.utilities</groupId>
    <artifactId>maven-streamtest</artifactId>
    <version>0.0.20</version>
    <configuration>
      <pathMain>./src/main/emam</pathMain>
      <pathTest>./src/test/emam</pathTest>
      <pathTmpOut>./target/tmp</pathTmpOut>
      <generator>MinGW</generator>
    </configuration>

    <executions>
      <execution>
        <phase>test</phase>
        <goals>
          <goal>streamtest-execute</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  …
</plugin>
```

Figure 2.22: Usage of the streamtest plugin in a Maven-based EmbeddedMontiArc project.

2.6.2 The Maven Streamtest Plugin

Stream tests can be executed programmatically via the Java interface of the stream language or by using its command line interface (CLI). However, this is impractical in large projects. Instead, the build engineer should be able to integrate stream tests into a generic and automated build process directly. Build systems like Maven and Gradle can be equipped with new capabilities using *plugins*. To integrate stream testing into the Maven lifecycle we developed the `maven-streamtest` plugin providing the following three goals:

1. **streamtest-generate** generates C++ code for all stream tests and the respective components under test,
2. **streamtest-build** compiles the generated C++ code to executable stream tests,
3. **streamtest-execute** executes the compiled streams and outputs a test report; failed tests have a return value other than zero, letting Maven report a build failure.

The integration of the maven-streamtest plugin in a Maven project is accomplished by adding the plugin tag, as depicted in Figure 2.22 into the Maven pom.xml. The plugin takes several configuration parameters, cf. L.7-10:

- **pathMain** is the path where Maven will look for the models under test,
- **pathTest** is the path where the actual stream tests reside,
- **pathTmpOut** is a temporary path where the plugin can store its output,
- **generator** can be set to either `MinGW` generating MinGW makefiles or to `VS2017` creating a Visual Studio C++ project; this feature is only available under Windows.

Now, to embed the `streamtest-execute` goal into the Maven lifecycle, we need to add it to the test phase in the project's pom.xml file using the execution tag, cf. L.14-19. Note that we did not add the `streamtest-generate` and `streamtest-build` goals to the test phase explicitly. The `streamtest-execute` plugin executes them automatically in a Maven `preExecution()` step if needed. Using the maven-streamtest plugin, EmbeddedMontiArc regression tests can be executed by a continuous integration (CI) pipeline in a generic way.

2.6.3 Simulation

While stream tests can be used to test isolated components in a unit test manner, integration and system tests of a CPS cannot be carried out without an interaction with the designated target environment. However, testing a possibly immature system with real hardware in a live environment is costly, dangerous, and slow. For this reason, a test framework simulating the CPS and its environment is necessary. Such a testing framework must fulfill a set of requirements in order to enable efficient testing in an agile process:

Physics engine: a physical model of the CPS and the world it lives in must be simulatable. The degree of abstraction must fit the use case. For instance, in macroscopic large-scale traffic simulations a vehicle can be approximated by a point mass; on the other hand, crash simulations might require finite element method (FEM)-based tools to reproduce the deformations of a vehicle body as precisely as possible.

Scenario definition: it must be possible to set up simulative test cases, also referred to as *scenarios*. In particular, it must be possible to define the CPS geometry and hardware, its goals (pass and fail criteria of the scenario) and the environment.

Reproducibility: scenario results must be reproducible. In case of probabilistic set-ups, e.g. if vehicles are spawned according to a given distribution, sampling must be controllable by a seed.

Machine-usable interface: to integrate a simulator into a development process such as SMArDT, it must be possible to automate its usage, e.g. to employ it in a CI pipeline. Therefore, it must be possible to set up the simulation, execute it, and process the results automatically, i.e. without human interaction.

Visualization: it must be possible to visualize problematic parts of a simulation. An appropriate visualization is crucial for an efficient error analysis and debugging by the developer. However, it must be decoupled from the simulation process to enable automation. Visualizing the simulation results should be possible on-demand. Repeating a visual analysis of the same scenario should not require a repeated simulation.

Numerous simulation solutions are available targeting a multitude of application domains. SUMO is a microscopic traffic simulator [LBBW+18] for large scenarios. Map import is possible from OpenStreetMap [HW08], VISSIM [FV10], and other sources. The simulator API TraCI [WPR+08] makes SUMO easily extensible by allowing a programmatic simulation control and data retrieval. Multiple simulators like TraNS [PRL+08], iTETRIS [RMK+13], and Veins [SGD11] use SUMO as a basis and extend it by features like vehicle to everything (V2X) communication.

VISSIM is a commercial microscopic traffic flow simulator with a wider range of features than SUMO including connected vehicles and platooning.

The CARLA simulator focuses on autonomous driving research [DRC+17]. It provides realistic physics and a detailed 3D-visualization based on the Unreal Engine 4 [San16] enabling visual perception testing and the application of deep learning to arbitrary scenarios.

TORCS [WEG+00, LCL13] is a racing game with a 3D visualization, as well. The possibility of writing custom pilots and an API providing access to sensors and actuators makes it suitable for autonomous driving simulation, as well. However, it is limited to the domain of racing.

Gazebo is a Robot Operating System (ROS)-based solution focusing on the simulation of custom robots in indoor and outdoor scenarios with an emphasis on physics [KH04, MSK+12].

OpenDRIVE[5] and OpenCRG[6] formats are the de-facto standard for the description of road networks and road surfaces. Universal Robot Description Format (URDF)[7] is an XML-based robot description language and part of the ROS ecosystem [QGC+09]. It allows a precise definition of an *isolated* robot's dynamic and kinematic properties. XML Macros (Xacro)[8] is a macro language that enables reusability for URDF models by providing three main concepts: *properties*, *math expressions*, and *macros*.

[5] http://www.opendrive.org/, accessed: July 2020
[6] http://www.opencrg.org/, accessed: July 2020
[7] http://wiki.ros.org/urdf, accessed: May 2019
[8] http://wiki.ros.org/xacro, accessed: July 2020

The CommonRoad project works on the reproducibility of vehicular motion planning experiments. Therefore, it defines an XML-based scenario description format covering a formal representation of the road network, static and dynamic obstacles, and the planning problem of the ego vehicles [AKM17].

GeoScenario is another XML-based scenario description format [QBC19] aiming to tackle experiment reproducibility. It is built on the OpenStreetMap (OSM) format for road network definition and provides a metamodel featuring dynamic agents, metrics, and goals.

Scenario Description Language for Multi-agent Systems (SDLMAS) is a declarative textual simulation language mainly addressing communication aspects of multi-agent simulations (MASs) [CSZ09].

Although EMA can be and has been used in conjunction with a variety of simulators, including SUMO, CARLA, TORCS, and Gazebo, we decided to design a dedicated simulation as a service (SimaaS) framework named MontiSim [GKR+17, FIK+18] to fit the requirements of a simulative testing for cooperating vehicles research platform in the best possible way. While an extensive presentation of MontiSim is out of scope of this thesis, the remainder of this section will give a brief overview of the main concepts.

MontiSim can be classified as a MAS simulator. Each vehicle is an agent consisting of a physical model, an EE-architecture, and the software. Each of these three subsystems is exchangeable and individually configurable using a textual vehicle configuration model. The basic physical model of MontiSim is based on the rigid body approach often used in 3D-simulations and game engines [BET14]. The vehicle is modeled using adjustable parameters like weight, size, and engine power. For more elaborate use cases a Modelica-based implementation of a vehicle dynamics model of the Chalmers university was integrated [GF12].

A vehicle is assembled of different components with different purposes and capabilities such as sensors, actuators, buses, and the hardware executing the driving software. The components are executed in parallel and exchange data between each other over the simulated EE infrastructure in order to realize the driving behavior. To test an autopilot model written in EMAM, the component is referenced in the vehicle configuration file. To render the execution times of the autopilot realistic, a hardware emulator analyzes the assembler code of the compiled model and estimates the execution times for a configurable environment with limited memory and computation resources [KKMR19].

A scenario is defined using a proprietary DSL. It allows the test case designer to create instances of vehicle models and attach goals to them. Goals can include: arriving at one or several given destinations as well as fulfilling constraints, e.g. with respect to velocity, acceleration, etc. Furthermore, each vehicle has a core set of implicit goals including that a vehicle must not collide with other objects and that it needs to comply with the road traffic regulations. From the software engineering point of view, scenarios are the test cases while EMAM models are the code to be tested.

Setting up and using complex simulators can be difficult and might require a lot of

resources. The simulator user should not be responsible for dealing with the technical details, but should rather be able to use a simulator as a black box. To facilitate the integration of MontiSim into an automated test process, we pursue the SimaaS approach. The MontiSim server offers a microservice-based interface, which can be used by a client to request simulations and obtain the simulation results. The MontiSim server dispatches the request to available simulator instances. To enable very large simulations, a simulation scenario can be subdivided spatially. In this case the MontiSim server reserves multiple subsimulators simulating parts of the scenario [FIK+18, KKRZ19]. When a vehicle leaves the area controlled by a subsimulator, it gets handed over to the responsible subsimulator instance. The MontiSim server orchestrates the distributed simulation and aggregates the results while hiding the complexity from the user.

Chapter 3

Dynamics Aspects of EmbeddedMontiArc

3.1 Cooperative Agents

In Chapter 2 our focus was on static architecture modeling of closed, isolated systems such as autonomous vehicles using EMA. The elements of a static architecture are fixed at design time and cannot be altered, removed, or added at runtime. With this approach we can cover the majority of closed systems such as embedded devices and control software. However, in practice many kinds of systems like Internet of Things (IoT) applications or V2X require the ability to restructure or reconfigure parts of their architecture according to changing goals and requirements at runtime. In autonomous system domains many instances of intelligent *agents* share common resources of their environment, e.g. an urban traffic infrastructure. Their efficiency can hence often be improved drastically by cooperation. For this reason, we are going to discuss the architectural properties of cooperative systems and develop an extension for EMA introducing dynamics to architectural elements such as ports, connectors, and components in this chapter based on [KKR19].

To understand the requirements of cooperative systems with regard to architectural modeling, we first need to sharpen the term cooperation. While obeying traffic regulations can already be interpreted as a cooperative behavior, we need to stress here that this is not a sufficient definition of cooperation in the context of this work, but rather a prerequisite thereof. Instead, we say that a system of agents is cooperative if the agents aim to optimize some cost function(s) by explicit information exchange and collaborative planning. In game theory, the notion of the Nash Equilibrium (NE) describes a non-cooperative solution, where each player (or agent) uses a strategy so that switching to another strategy would not lead to a gain [Mye13]. The NE is non-cooperative in the sense that every agent aims to maximize his or her own reward in a competitive setting and cannot rely on the solutions of other agents. Hence, the NE can be achieved without communication and cooperation and is often not an optimal solution as can be easily seen in the prisoner dilemma [RCO65].

The goal of this chapter is to introduce a reconfiguration framework suitable for the design of agents optimizing the expected reward and exceeding the NE of the system *by means of cooperation*. In particular, we want to address the domain of cooperative

driving with a focus on local traffic systems (LTSs) [DDE+17].

LTSs are ad-hoc or static vehicular application-level networks for cooperative behavior and trajectory planning in spatially constrained, i.e. local, scenarios. Vehicles can negotiate and create an ad-hoc LTS if they recognize a relevant situation they want to solve together. Participants can request to join or leave an LTS upon entering or leaving the area of interest. What is more, LTSs can be set up statically, e.g. hosted by a roadside unit (RSU) at an intersection, similar to a traffic light. In the following we give a brief overview of scenarios, which can be handled by cooperation within a spatially constrained area without claim of completeness [ECP+16, LMD+19]:

Intersections. The throughput of intersections can be improved if approaching vehicles can communicate with their peers before arriving. For instance, instead of stopping at the intersection to obey priority rules, vehicles can negotiate individual velocities. This can also eliminate dead locks at intersections with no traffic lights.

Occluded street segments. Traffic participants that cannot see each other, e.g. due to to a building or a hilly landscape, can announce themselves to each other and plan safe trajectories without having to slow down unnecessarily.

Platoons. Vehicles with the same destination can form a platoon to minimize the safety distance between each other as well as to reduce unnecessary acceleration and deceleration by sharing sensor data and their plans.

Overtaking maneuvers. When overtaking, the participants, including oncoming traffic, can negotiate trajectories and velocities to achieve optimal maneuvers.

Priority vehicles. Emergency vehicles and the like can request priority for a street segment. Vehicles can react to this request early and replan their own trajectories.

The LTS concept organizes cooperative planning and decision making in three layers. A cooperative vehicle should implement these layers to be able to participate in LTS-based cooperative traffic systems:

Traffic layer. The traffic layer is a macroscopic layer dealing with the creation and management of cooperating groups and the negotiation of roles inside an LTS cluster. This layer contains functionality to look for potential LTS situations based on a vehicle's situation understanding.

Maneuver layer. The maneuver layer is responsible for cooperative trajectory planning and information exchange. Its output are reference trajectories aiming to achieve the goal of the LTS given by the traffic layer. This layer replaces the trajectory planner of a non-cooperative autonomous vehicle.

Control layer. The control layer is a microscopic layer realizing the trajectory assigned to the vehicle by the maneuver layer. Usually, this layer is not aware of the cooperation and can be seen as ordinary autonomous vehicle control. However, centralized control is possible, as well.

The research question to be answered in this chapter is the following:

Research Question 2. *How can runtime dynamics be modeled in a C&C-based development methodology to support the design of cooperating CPSs at SMArDT levels 2 and 3?*

We are going to derive a set of requirements for a dynamic C&C modeling language applicable to the design of agent-based systems in Section 3.2. Afterwards, we are going to give an overview of state-of-the-art C&C languages and their means for dynamic architectural reconfiguration in Section 3.3. The actual concept for an event-triggered and type-safe runtime reconfiguration dynamics will be discussed in detail in Section 3.4.

3.2 Background & Requirements

Since development processes in the automotive domain such as the V-model or SMArDT heavily rely on component-based and hierarchical decomposition, it is desirable to continue using C&C languages for the cooperative vehicle domain. Different forms of dynamic ADLs are known in the literature tackling different concerns of architectural dynamics [BHK+17]. In particular, the choice of appropriate means of architectural runtime reconfiguration depends on the kind of systems under development and the application domain. The concepts discussed in this chapter are intended for the LTS domain introduced above. Our requirements and design decision will hence be based on the following list of assumptions:

- The agents are instances of compatible types or share a common interface. In the automotive domain, for instance, agents are equal or similar vehicles or RSUs. The agents are independent processes with proprietary goals. They are not part of and do not contribute to the functioning of a bigger system (in contrast to an aircraft architecture designed using a language like Architecture Analysis & Design Language (AADL), where architectural dynamics is used to model functional variations of a single but complex system).
- The agents do not know each other by default and there is no communication between them at the beginning. Furthermore, the total number of agents living in the system is not known to an agent. Each agent's knowledge about its peers is limited to what it perceives through its sensors and communication.

- The number of agents in the system can vary throughout time. Agents can be spawned without existing agents to be notified explicitly. In the cooperative vehicles domain, new vehicle instances can come into existence by being manufactured or by entering the area of interest from outside, e.g. in a sector-based simulation, cf. Section 2.6.3.
- There is a communication channel which can be used by the agents to send and receive messages to and from other agents, respectively. This channel can be used for both directed and broadcast communication. However, since we are dealing with the application layer, we will not care about lower network protocols in this work, assuming an end-to-end channel connecting the logical interfaces, e.g. EMA ports, of two different agents directly.

To be able to model interactions between participants of a dynamically changing traffic system, the C&C language used needs to support changes in the component structure and variations of the dataflows at runtime. Such changes can be induced by specific events, such as the occurrence of a new traffic participant, which the developer should be able to model with the same language, as well. Based on the assumptions introduced above we have elicited a list of requirements for a C&C modeling language supporting dynamic reconfigurations applicable to the domain of cooperating vehicles:

(RD1) Architectural reconfiguration: the architecture modeling language must provide means to describe structural changes of a system at runtime. This requirement is fulfilled automatically if all the subrequirements, cf. (RD1.1)-(RD1.3), are satisfied.

(RD1.1) Dynamic components: the dynamic ADL must provide means for the modeling of the runtime instantiation of new components and the removal of obsolete ones. The concept for runtime component instantiation must cover the creation of new functional components to cope with changing situations. For instance, a vehicle might need to instantiate a component to handle a deadlock at an intersection and to deactivate or dispose this component afterwards. Furthermore, it must be possible to replicate and remove components to deal with changing numbers of communication partners. For instance, an agent might want to model other agents as separate subcomponents. Since the target platform might have limited resources, it must be possible to constrain component replication.

(RD1.2) Dynamic interfaces: the ADL must provide means for the modeling of runtime alterations of component interfaces. In particular, it should be possible to add and remove ports at runtime to serve end points for the communication with dynamically emerging and vanishing agents. Similarly to (RD1.1) it should be possible to create new and replicate existing ports.

Consequently, a mechanism is needed to make the newly created ports usable by external communication partners. On the other hand, it should only be possible to remove ports if they are not needed any more. This requirement does *not* include changes applied to existing port instances such as alterations of a port type. Type safety must be guaranteed at any time.

(RD1.3) Dynamic connectors: as a consequence of (RD1.1) and (RD1.2) the dynamic ADL must provide a way to model runtime creation and disposal of dataflows by interconnecting new components and ports. Furthermore, it must be ensured at compile-time that all possible interconnections are valid, e.g. that a port never becomes the endpoint of two different connectors or that an input port is never floating. A reconfiguration is valid if its result is a valid EMA architecture, which can be modeled without dynamics.

(RD2) Internal event-triggerd reconfiguration: it must be possible to model reconfigurations of a component as reactions to events visible in the component's scope. It should be possible to define events in terms of data observed at the component's ports or the ports of its immediate subcomponents. For instance, if the velocity of a vehicle is written to a corresponding port, it should be possible to change the driving mode by instantiating a highway component if the value at this port exceeds a predefined threshold. Furthermore, it should be possible to define events in terms of architectural changes. A component should be able to react to newly created ports by instantiating more ports and/or subcomponents and by interconnecting them. For instance, when a port is created to communicate with another vehicle, then a subcomponent for mutual trajectory planning might need to be instantiated as a reaction.

(RD3) External service-based reconfiguration: in addition to (RD2), it should be possible for a component (or any other software) to send reconfiguration requests to other components exhibiting a reconfiguration interface. To preserve the black box principle of components, the decision whether to accept or reject the request, however, must remain within the authority of the target component. Such a reconfiguration interface could be used by an agent to request an input port at another agent to establish a new communication channel. The reconfiguration interface should be available not only at model level, but also in the generated code so that it can be used by external software and legacy code, as well. This would allow a caller to initiate reconfigurations if the model is generated and used as a library.

(RD4) Black box reconfiguration: it must be strictly ensured that dynamic components remain accessible for other components only through their interfaces to maintain their self-contained and reusable nature. In particular, a component should not be able to perform or request any architectural changes to the internal

structure of peer or subcomponents or to peer or subcomponents' subcomponent interfaces.

The consequence of excluding invasive component reconfigurations is that (RD3) is limited to interface reconfiguration requests. This ensures that a dynamic component maintains full sovereignty over its internal reconfiguration processes including implementation details and timing aspects. A further benefit is that no details about the internal structure of a component need to be known in order to use a dynamic component and its reconfiguration interface.

(RD5) Reversibility: it must be possible to roll back architectural changes carried out at runtime. In particular, it must always be possible to get back to any past architectural state, e.g. if a situation reoccurs or if components have fulfilled their task and are no longer needed.

3.3 Dynamic ADLs

3.3.1 Brief Overview

ADLs providing runtime dynamics have been studied for decades, resulting in a variety of approaches, a selection of which we are going to introduce in this section. This selection aims to cover different classes of dynamics and to analyze languages related to EMA. We are going to analyze the dynamic reconfiguration features of these languages with respect to the requirements derived in Section 3.2. A tabular overview of this analysis is given in Table 3.1. More exhaustive overviews and classifications covering a wider range of dynamic ADLs are given in [BHK+17, KJKD05].

AADL [FG12]. AADL is an ADL standardized by the Society of Automotive Engineers (SAE) and originally developed for systems engineering in the avionics domains. In an AADL model, an architecture incorporates both software and hardware components. Furthermore, AADL provides analysis and verification tools needed for the quality assurance in safety-critical systems. The ecosystem of AADL is based on a single core language used for both software and hardware components. The language can be extended by adding user-defined properties and by the concept of language annexes. The latter can be subdivided in behavior annexes, error-model annexes, ARINC653 annexes (avionics related patterns) and data-model annexes. Runtime reconfiguration can be modeled using the concept of *modes*. A mode is a self-contained architectural configuration. Transitions between modes are governed by a mode FSM where each mode is represented as a separate state. Different operational states, e.g. of aircrafts or vehicles, can be represented as modes of an architecture. Furthermore, modes can serve as a mechanism for the recovery from component failures. Mode transitions can be propagated from parent components to their respective subcomponents.

	AADL	Autofocus 3	Darwin	EMAD (this chapter)	MontiArc	ROOM	Simulink	Wright
(RD1) - Runtime reconfiguration	p	p	p	√	p	√	p	p
(RD1.1) Dynamic component creation	–	–	p	√	p	√	p	√
(RD1.2) Dynamic interface modification	–	–	p	√	–	√	–	–
(RD1.3) Dynamic connectors	√	√	√	√	√	√	p	√
(RD2) Internal reconfiguration	–	√	–	√	√	–	√	–
(RD3) External reconfiguration	?	√	√	√	√	√	√	√
(RD4) Self-determined reconfiguration	–	√	√	√	√	?	√	√
(RD5) Reversibility	√	√	–	√	√	√	√	√

Table 3.1: Comparison of C&C modeling languages supporting runtime dynamics, √: yes, p: partially, –: no, ?: unknown

AutoFocus [AVT+15]. AutoFocus 3 is a framework and a research platform for the model-driven engineering of embedded and safety-critical systems developed by the fortiss GmbH[1]. It covers important aspects of the targeted development process from the requirement analysis to integration and provides graphical tooling. The FOCUS theory, where specifications of static and dynamic architectures are represented as predicate logic formulas [BS12, Bro14], serves as a theoretical basis for AutoFocus allowing strictly formal analysis and verification of both static and dynamic architectures. Similar to AADL, AutoFocus provides mode-based runtime reconfiguration governed by FSMs enabling modeling of multiple operational regimes and their transitions.

Darwin [MDEK95]. Darwin is another textual ADL enabling modeling of static and dynamic distributed software architectures. A formal foundation of Darwin's operational semantics is given by the agent-based π-calculus [MPW92]. Components, their services, and bindings can be mapped to π-calculus agents. In particular, a binding agent is composed with a request agent to create a binding request. Once composed with the service providing agent, the binding request results in a binding. Dynamic structures can be realized in Darwin based on two different mechanisms: lazy and direct dynamic instantiation. The former requires all the bindings of the architecture to be defined at design

[1] https://www.fortiss.org/veroeffentlichungen/software/autofocus-3, accessed September 29, 2020

time. However, components declared with the keyword `dyn` are only instantiated once their provided services are actually accessed by another component. This mechanism allows for the creation of potentially unbounded component structures such as pipelines when combined with recursion. However, the evolution of a dynamic architecture still underlies a fixed pattern declared at design time.

The direct dynamic instantiation mechanism overcomes this constraint and allows the architecture to evolve in arbitrary ways. This is essentially modeled by binding a required service to a dynamic instantiation service. The instantiated components are anonymous and their services can only be accessed by passing service references in messages. The disadvantage of this approach is that the ADL cannot model bindings with dynamically created instances explicitly. Consequently the model is not able to capture the evolution of such a dynamic architecture adequately. Direct multicast communication between dynamically created groups of components can be realized using abstract services. A peculiarity of Darwin's dynamic architectures is that components, once instantiated, cannot be removed or deactivated any more.

Other ADLs based on the π-calculus are, for instance, LEDA [CPT99] and π-ADL [Oqu04].

MontiArc [HRR12]. MontiArc is a modeling language targeting the design of distributed software architectures in domains like IoT. As already discussed, it shares a large portion of concrete and abstract syntax regarding core language features with EMA. Although MontiArc was originally developed as a modeling language for static architectures, an approach retrofitting the language family to support mode-based dynamics following the example of AADL and AutoFocus 3 was presented in [HKR$^+$16]. Accordingly, mode transitions are controlled by FSMs. Each transition of the mode FSM has a source mode, a target mode, and a guard expression which must be fulfilled for the transition to become activated. Furthermore, the authors distinguish between runtime component *instantiation* modeled using the keyword `component` and *activation* modeled by the keyword `activate`. Components which are *instantiated* at mode entry are disposed when the mode is left and the component has become obsolete. Activated components on the other hand are just deactivated when not needed and retain their inner state until the mode is entered again.

Real-time Object-Oriented Modeling (ROOM) [SGW94]. The aim of ROOM is to cover the complete specification of a system in order to prevent architectural decay. To ensure a reasonable separation of concerns, ROOM is subdivided into two layers: the *schematic* layer is a graphical modeling language for the description of high-level abstract architectural aspects. The *detail* layer on the other hand is concerned with implementation details and can be written in a standard GPL. This separation is similar to EMAM where the structure is described using the EMA ADL while the behavior

description is done in MontiMath.

ROOM targets explicitly the distributed realtime systems domain with event-driven or reactive behavior. Message passing is the main means of communication in ROOM. The language allows the replication of components and ports at runtime for dynamic architectures such as a PBX system featuring a changing number of participants. As such, the set of possible runtime architectures is prescribed at compile-time [RSRS99].

Simulink [Mat16]. Simulink is a graphical modeling tool mainly focusing on static architectures in domains such as control engineering, automotive, CV, and others. The set of components and connectors is fixed at design time and cannot be extended during runtime. However, it is possible to turn components on and off as needed based on external signals. This is modeled using blocks such as *Enabled Subsystems*, *Triggered Subsystems*, and *Function-Call Subsystems*. This kind of blocks comes with a special port controlling the activity of the component. For an Enabled Subsystem, the component is active while the control signal has a positive value. For a Triggered Subsystem the modeler can choose between three variants: rising signal, falling signal or either. A Function-Call Subsystem is activated upon a function call. It is possible to configure the initial and the disabled mode value for the component's output. Furthermore, there are three different ways to handle such a component's state: the state can either be saved or reset while the component is off; alternatively the state behavior can be inherited from the parent component.

While it is not possible to design highly dynamic architectures, the method can be employed to simulate modes. Hence, the designer has to maintain a 150% model containing all the components possibly needed at runtime. Having more than two or three modes however would lead to models, which are difficult to read and understand.

WRIGHT [ADG98]. The aim of WRIGHT is to deliver an ADL which is able to capture both dynamic reconfiguration behavior as well as its non-reconfiguration functionality while maintaining a clear separation of the two concerns. The ADL is based on the communicating sequential processes (CSP) formalism [Hoa78] allowing for an event-based specification of behavior and communication. In WRIGHT dynamic reconfiguration is based on control events which can be used in the configuration program to initiate a reconfiguration. The configuration program itself uses a set of reconfiguration actions to create, delete, connect, and disconnect architectural elements. Being a formal language, WRIGHT provides verification and consistency checks of architectures and reconfigurations, e.g. ensuring that the components which are part of a reconfiguration actually exist or that reconfigurations take place only at permitted points of computation.

3.3.2 Requirements Assessment

Each of the languages presented above carries some concepts of dynamic reconfiguration. Remarkably, these concepts vary a lot from language to language and are obviously developed with a particular kind of applications in mind. In the following we analyze if and how the aforementioned ADLs fulfill our requirements introduced in Section 3.2.

Dynamic components (RD1.1). Component dynamics is supported by all of the studied languages at least to some limited extent. ROOM and WRIGHT are the only ADLs providing full support for component instantiation and removal. Darwin supports runtime component instantiation as well, but lacks the possibility of component removal. Hence, from our perspective it fulfills (RD1.1) only partially. The mode concept employed by MontiArc allows for a runtime creation of components according to the mode description pre-defined at design time. Hence, it can be seen as a restricted variant of component dynamics. Although AADL and AutoFocus 3 support modes as well, these languages only allow for a runtime rewiring of connectors and parameter changes and hence, are not considered to fulfill (RD1.1). Simulink's enabled subcomponents mechanics can be used to enable and disable components based on signal values at specific ports. However, all components need to be modeled at design time and remain in the model all the time. We consider the requirement as partially fulfilled.

Dynamic interfaces (RD1.2). Changing a component's interface at runtime is a severe modification affecting both the overall architecture as well as the component's behavior. For this reason, means for interface modification are provided by a smaller number of languages. ROOM's dynamics model allows the replication of ports to handle many similar signals coming from different instances of the same component type, e.g. to model a changing number of phone connections. In Darwin, services provided by dynamically created component instances can be offered as services of their respective parent components which, in turn, implies an interface modification.

Dynamic connectors (RD1.3). Dynamic connectors is the most common dynamic feature among the analyzed languages. All candidates are able to rewire their components at runtime. AADL, AutoFocus and MontiArc modify their components' interconnections at mode changes. ROOM replicates connections on demand, e.g. to handle an increasing number of phone calls. In Darwin bind requests lead to new connectors. In WRIGHT connectors can be created freely by attaching and detaching ports to and from one another in the configuration program. Simulink provides the most restrictive way to model connector dynamics. Dataflows can be altered using switches and similar components to activate or deactivate connectors based on control signals. This is similar to Simulink's way of dealing with component dynamics.

Internal reconfiguration (RD2). The mode FSM concept of AutoFocus 3 and MontiArc can be regarded as an internal reconfiguration mechanism. A mode transition is activated once a condition related to the ports in a component's scope is met. Similarly, in Simulink, an architecture is reconfigured using enabled, triggered, and function-call subsystems based on control signals. Hence, component activations and deactivations can be derived from port values.

External reconfiguration (RD3). External reconfiguration requests can be imitated by the means of internal reconfiguration by reinterpreting signals arriving from outside, e.g. from other components, as reconfiguration requests. Hence, fulfilling (RD2) implies fulfilling the external reconfiguration requirement (RD3), as well. Darwin and WRIGHT do not support internally triggered reconfiguration, but provide means for external reconfiguration only.

Self-directed black box reconfiguration (RD4). Self-directed black box reconfiguration enables a black box reuse of dynamic components and is available in all the discussed languages except AADL. In AADL it is possible to propagate mode changes from parent components to their children's mode configurations directly. Hence, a component has no control over its mode transitions as these might depend on the mode changes of the enclosing scope.

Reversibility (RD5). In FSM-controlled architecture models, reversibility has to be modeled explicitly. If it is a desired property, it can be proven by ensuring that each FSM state is connected with each other state either directly or by a path including intermediate states. Darwin is the only language prohibiting reversibility by design as it has no means to remove already instantiated components.

3.4 EmbeddedMontiArc Dynamics

The aim of this section is to introduce the main concepts of an EMA language extension for dynamic reconfiguration, which we are going to refer to as EMAD. By developing these concepts our goal was to deviate from the original EMA syntax as little as possible while ensuring that the requirements presented in Section 3.2 are fully met. In particular, we wanted to keep the language extension conservative [HR17], guaranteeing that standard, non-dynamic models can be parsed and generated by EMAD without changes. The MontiCore grammar of core EMAD is given in the appendix in Listing B.10. The syntax for reconfiguration conditions is contained in a separate grammar in Listing B.11.

3.4.1 EMAD Execution Semantics

In Section 2.3.3 we have discussed the synchronous execution semantics of EMA. The system is executed stepwise. In each step all the subcomponents are executed according to an execution order determined at compile-time. To enable reconfiguration and to support dynamically evolving architectures, we adapt the execution semantics of EMA by extending the reconfiguration phase introduced in Section 2.4.6, which was used to adapt component parameters.

There is no effective difference whether the reconfiguration phase takes place before or after the execution phase in an execution cycle, except for the first one. By putting the reconfiguration phase first, we ensure that a reconfiguration can take place before the first computation is performed .This can be helpful, if the architecture needs to be changed before processing data, e.g. if a parameterization requires a specific mode or a component failure requires a failover reconfiguration.

In the reconfiguration phase, reconfiguration triggers are checked and, if present, the corresponding reconfigurations are performed. This possibly activates further reconfiguration triggers which are then handled as well, until the reconfiguration queue is empty.

To live up to the requirements (RD2) and (RD3) of Section 3.2, we introduce two main concepts for runtime reconfiguration in EMAD:

- Data-triggered and
- Service-based reconfiguration.

3.4.2 Data-Triggered Internal Reconfiguration

The simplest way to trigger and model reconfiguration is the data-triggered approach. Thereby, a reconfiguration is initiated when a signal fulfills a given condition, e.g. a port value exceeds a predefined threshold. The reconfiguration is executed and maintained as long as the condition is satisfied. The approach can be easily motivated and illustrated by non-linear components used in electronics. For instance, a diode is conductive only if the applied voltage is higher than the threshold voltage; a multiplexer passes the data signal chosen by a control signal; when a battery electric vehicle (BEV) is connected to a charging station, the connection is signaled to the charging electronics which reacts by enabling the charging process as long as the connection signal is active.

To enable modeling data-triggered reconfiguration, we extend the body of an EMA component definition by a list of reconfiguration blocks. The header of such a reconfiguration block contains a condition formulated as a Boolean expression over port values and architectural properties, which needs to be fulfilled in order to trigger the reconfiguration. The body of the reconfiguration block follows for the most part the same syntax as the body of a standard non-dynamic component and contains a declarative definition of the architectural changes to be performed as a response to the triggering event. These

```
1  component BMux4<T>                                          EMAD
2    ports in T inSig[4],
3          in B ctrSig[2],
4          out T outSig;
5
6    instance BMux2<T> mux2;
7
8    connect ctrSig[1] -> mux2.ctrSig;
9    connect mux2.outSig -> outSig;
10       reconfiguration condition
11   @ ctrSig[2]::value() == true {
12     connect inSig[3]  ->  mux2.inSig[1];   } Value-triggered
13     connect inSig[4]  ->  mux2.inSig[2];   } reconfiguration
14   }
15
16   @ ctrSig[2]::value() == false {
17     connect inSig[1]  ->  mux2.inSig[1];
18     connect inSig[2]  ->  mux2.inSig[2];
19   }
20 }
```

Figure 3.1: A multiplexer component choosing two of its inputs to be passed to the inner multiplexer dependent on a control signal.

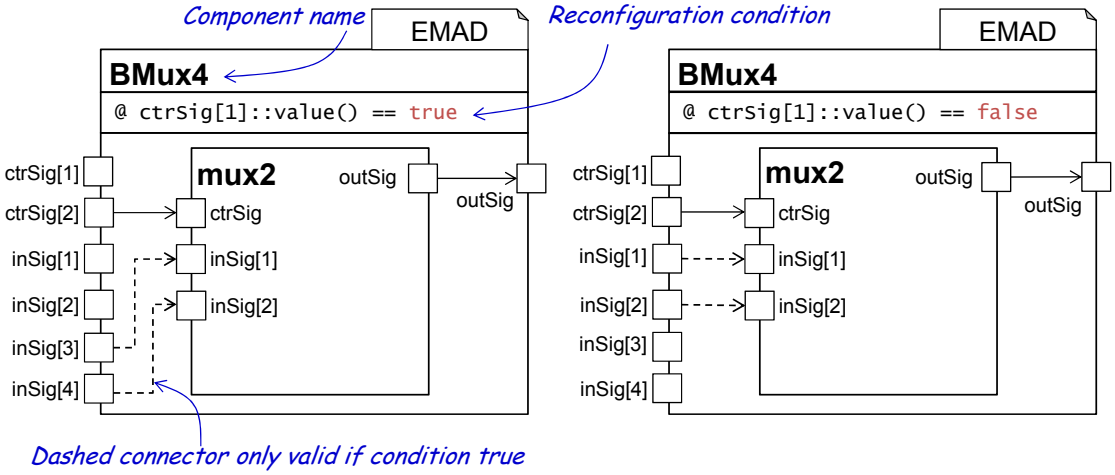


Figure 3.2: The two architectural states of the BMux4 component.

changes are rolled back as soon as the reconfiguration condition in the reconfiguration block header ceases to hold.

To illustrate the syntax and the mechanics behind data-triggered reconfiguration, we introduce a simple multiplexer example in Figure 3.1. The `BMux4` component has four data inputs of a generic type `T` and two Boolean control inputs. The purpose of the component is to choose one of the four input signals of the `inSig` port array based on the values of the control signals (`ctrSig` port array) and to forward it to the output port. The idea is to realize this behavior by altering the connectors corresponding to the control signal. Therefore, we first choose two of the four data signals (the first two *or* the second two ports of the `inSig` array) based on the value of `inSig[1]` and then forward them as well as a further control signal `inSig[2]` to a subcomponent of type `BMux2`, which in turn uses the received control signal `inSig[2]` to choose one of the remaining two data signals. Its choice is then output through the parent component's output port.

The static connectors of the component are defined in L.8-9 to connect the first control signal with the inner multiplexer and its output to the output of the parent `BMux4`. There are two reconfiguration definitions given in L.11-14 and L.16-19. In L.11 and L.16 the @ symbol denotes the beginning of a reconfiguration condition. The actual reconfiguration code is a block enclosed in curly brackets following the condition. As can be seen in L.12-13 and in L.17-18, the configuration code is composed of ordinary connect statements as we know them from the static EMA syntax. The connections defined in these two blocks are established and released in the reconfiguration phase at the beginning of an execution cycle as discussed earlier. In this example, this is used to choose two of the four incoming inputs to be forwarded to the child component `mux2`.

A reconfiguration is executed once the condition becomes true and remains active as long as the condition remains true, i.e. as long as the value at the port `ctrSig[1]` is true in L.11 and as long as it is false for L.16. When the condition of an active reconfiguration goes back to false, the reconfiguration is rolled back, i.e. all the architectural elements defined in the reconfiguration block are removed (irrespective of whether or not another reconfiguration becomes active instead). In our example, the two reconfiguration conditions are mutually exclusive, but their disjunction is always true. Consequently, exactly one of the two reconfigurations is active at any given point in time. In general, arbitrarily many reconfigurations (including zero) can be active in parallel. However, each combination must result in a valid architecture. That is, an input port must not be the target of more than one connector. Furthermore, under no circumstances an input port may be floating. This is verified by context conditions at compile-time, the details are given in Section 3.4.6. Consequently, none of the two reconfigurations can be removed from the component in the multiplexer example: when no dynamic reconfiguration is active, only the static part of the architecture is present. In this case, the `inSig` ports of `mux2` would be floating.

Note that in order to access the value of a port in an EMAD reconfiguration, we use the

port function `value()` accessible for each port of the component using the `::` operator. The syntax highlights that we are not trying to use a model element in a conventional manner (which would require a dot), but want to perform a runtime query related to a model element instead. The function is available in reconfiguration conditions and bodies only. If the port we are referring to belongs to a subcomponent, we can access it by specifying the port's name preceded by the (subcomponents') instance name, e.g. `mux2.outSig::value()`. Note that a component can only query the values visible in its scope, i.e. values of its own or of its immediate subcomponents', but not of its subsubcomponents' or the parent component's ports.

A reconfiguration condition can be an arbitrary Boolean expression. Similarly to other languages the Boolean OR and the Boolean AND operators are denoted by `||` and `&&`, respectively. For equalities and inequalities we use the following operators: ==, <=, >=, <, >.

Reconfiguration conditions can be formulated in terms of an expression sequence in order to identify sequence patterns. A value sequence can be notated similarly to an EMA row vector with the oldest value coming leftmost. To avoid confusions with vector-valued variables, the `tick` keyword is used as a separator instead of a comma. For instance, the condition `ctrSig[1]::value() == [true tick false tick false]` is evaluated to true at execution cycle n if the following sequence of values was observed: `true` at $n-2$, `false` at $n-1$, `false` at n. The type of each expression in the sequence must be compatible with the corresponding port type. The sequence notation implies that past values of the underlying port need to be stored at runtime. In this particular example, in addition to the current value at the `ctrSig[1]` port, the component needs to store two of this port's past values in order to be able to evaluate the reconfiguration condition in each execution step.

To cover more complex patterns, EMAD offers the following language constructs:

Inequality operators. Each entry in the sequence can be preceded by one of the following inequality operators: <, >, <=, >=, !=, meaning that the expression evaluates to true, if the value at the port is less than, greater than, less or equal, greater or equal, or unequal to the expression provided, e.g. `inSig::value()==[>5 tick <=6 tick !=0]` becomes active if the value at $n-2$ was greater than 5, the value at $n-1$ was less or equal to 6, and the current value is not 0. If all values are preceded by the same operator, we can use it instead of the == operator, e.g. `inSig::value()>[5 tick 6 tick 0]`.

Variables. Variables can be introduced as placeholders to compare values from different execution cycles. For instance, `inSig::value()==[a tick 2*a]` has a placeholder `a`, the concrete value of which is irrelevant. The reconfiguration condition is fulfilled as long as the value at the port keeps doubling in each execution cycle. Note that we don't need to specify the type of `a`, since it is resolved as the type of the corresponding port `inSig`. A variable can serve as a placeholder for arbitrary

values, as well. For instance, `b` will match any value in `inSig::value()==[a tick b tick 2*a]`. The condition is true if the current value is twice as large as the value two time steps ago, no matter what the value in between was.

Time Shifts. The `value(Z n)` function can take an integer argument. If the integer argument is greater or equal to 1, it denotes an absolute execution cycle number n. For instance, `inSig::value(10)==5` is true if the tenth value read at port `inSig` was equal to 5. The expression is evaluated to false automatically until the tenth execution cycle is reached. Then it evaluates to the true value of the expression and remains an invariant for all future execution cycles. The component can either store the n-th value or cache the Boolean result of the condition. The latter is more efficient as long as the number of conditions depending on the n-th value of a port is relatively small compared to the size of the port type.

If `value(Z n)` receives a negative argument, i.e. $n < 0$, the function returns the value, which was present at the respective port $|n|$ execution cycles ago. For instance, `inSig::value(-2)==5` is evaluated to true if `inSig` was 5 two execution cycles ago and is false otherwise. If the value to be compared is a series, the negative argument refers to the latest point of the series (this implies that it is not possible to refer to future execution cycles). For instance, `inSig::value(-2)==[5,10]` is evaluated to true at execution step m if `inSig` had a value of 5 at execution step $m+n-1 = m-3$ and a value of 10 at execution step $m+n = m-2$. To be able to evaluate a condition with a negative `n` at runtime, the component needs to store the last $|n| + l - 1$ values of the corresponding port, where l is the length of the sequence to compare against. Regardless of its sign, the parameter `n` must be fixed at component instantiation and cannot be changed later on. Otherwise it would become necessary to save the complete history of the corresponding port.

Until now, we have been using a graphical representation of EMA models to facilitate the understanding of the architecture. Given the fact that there is no single representation of an EMAD model, we need an appropriate extension of the graphical syntax. Diagrams representing the two reconfigurations of the `BMux4` model are depicted in Figure 3.2. Thereby, we introduce two syntactic elements: first, the reconfiguration condition triggering the reconfiguration is specified in a box under the component's name. Second, model elements, which are added in this reconfiguration, are denoted by dashed figures instead of solid ones. In this example, only connectors are created dynamically at runtime. Components and ports can be added in a similar way by the means of dynamic arrays, which will be discussed in Section 3.4.3.

The aim of the example in Figure 3.1 and Figure 3.2 was to introduce the main ideas behind data-triggered reconfiguration. The exactly same behavior can be achieved with a mode model with two states. Using a mode FSM for a system with a small number

of states and state transitions can be favorable as it facilitates a state-centric model analysis. In cases with many, possibly partially overlapping reconfiguration conditions and state transitions between all possible states, however, the data-triggered reconfiguration concept presented in this chapter can lead to much more concise models, since we don't need to define all possible states explicitly and no transitions need to be modeled at all. On the other hand, modes are more powerful since reconfigurations can depend on the current *architectural* state, which is not possible with our concept. We recommend using modes and data-triggered reconfiguration interchangeably depending on the requirements and the nature of the modeled system.

3.4.3 Service-Based External Reconfiguration

To enable the creation of more complex, propagating reconfigurations, we introduce a second way of triggering architectural changes at runtime, the service-based reconfiguration. The idea behind it is to trigger reconfigurations by external architectural change requests and to propagate such requests from component to component.

We are going to present the concepts of service-based reconfiguration by the example of a cooperative collision prediction component given in Figure 3.3. The `CollisionSystem` component receives the planned trajectories from other vehicles of an LTS and checks each of these trajectories for a collision with its own one. Each trajectory is input into the component through a dedicated port. Furthermore, each pairwise collision check is executed by a dedicated subcomponent of type `CollisionCalculator`.

Before we proceed to the discussion of the service-based trigger mechanism, we need to introduce the concept of dynamic component and port arrays. In Section 2.3.2, static component and port arrays were introduced, allowing us to model an arbitrary but fixed number of similar components and ports in a single line of code. In the collision detection example described here we don't know at design time, how many traffic participants will be present in the LTS. Furthermore, the number of peers can change over time. The concept of dynamic arrays enables us to cope with this modeling challenge by allowing us to specify a range instead of a fixed number of elements in the array. At runtime the concrete number of elements in the array can change.

The syntax is based on the range syntax of EMA types: the modeler needs to specify the minimum and the maximum number of elements inside the square brackets of an array declaration separated by a colon instead of a single length value. This is done in L.4 and L.5 of Figure 3.3 to define a dynamic port array and in L.8 to define a dynamic component array. In the case of port arrays it is obligatory to use the `dynamic` keyword. If the component interface contains dynamic port arrays, it is also necessary to mark the component with the `dynamic` keyword in the header, cf. L.1 of Figure 3.3. Technically, the `dynamic` keyword is not necessary. However, it improves readability by making the dynamic interface explicit for component users and developers. Furthermore, it prevents developers from accidentally creating dynamic components. This is in line with abstract

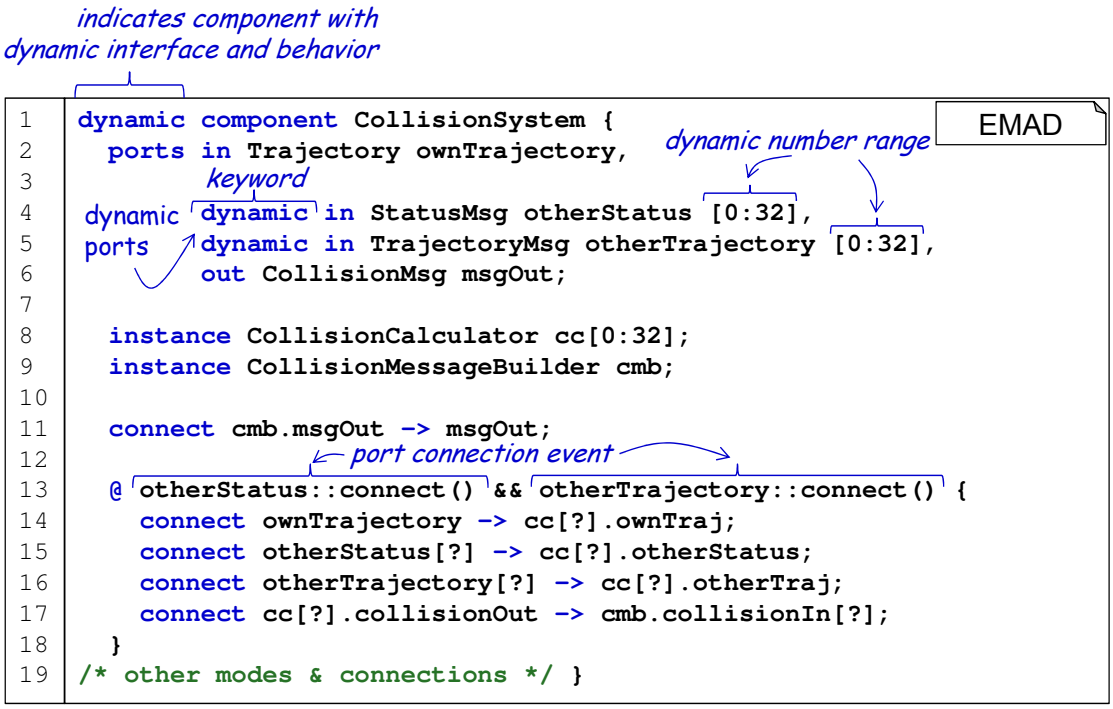


Figure 3.3: Collision system of an autopilot calculating potential collisions with up to 32 other vehicles.

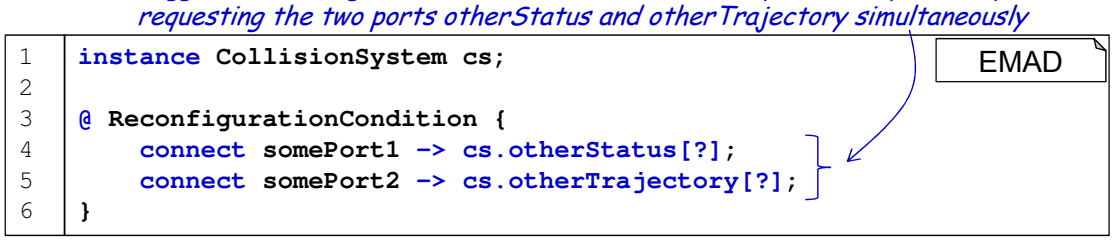


Figure 3.4: The listing shows a valid usage of the reconfiguration service interface of the CollisionSystem component of Figure 3.3 by a parent component.

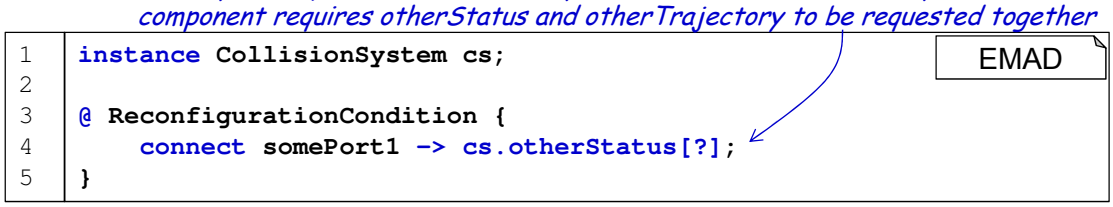


Figure 3.5: The listing leads to a compile-time error since CollisionSystem does not have a reconfiguration triggered by requesting only the otherStatus port.

classes in Java, where abstract methods *and* the class itself have to be defined using the `abstract` keyword explicitly. Technically, the presence of a method without a body is sufficient to infer that the class is abstract.

In case the lower bound of the element count is greater than zero, the minimum number of elements will be created at instantiation of the component. Once the upper bound of the elements in an array has been reached, events leading to an instantiation of further elements are not handled. The availability of free port and/or component slots in an array can hence be regarded as a further implicit condition of a reconfiguration. Upper bounds on elements in an array have been introduced with embedded systems in mind often having very limited resources and strict performance requirements. The upper bound can be set to infinity by putting `oo`, similarly to EMA type bounds. However, since this can have a negative impact on the performance of an overloaded system, this is not an advisable modeling pattern and results in a warning. A system knowing its limits can react to an overly high demand in a controlled manner.

In our collision system example, the port arrays `otherStatus` and `otherTrajectory` are supposed to receive status and trajectory messages from other cooperative vehicles in the LTS. The maximum number of connections is limited to 32. On the other hand, if there are no other vehicles in the network, the port arrays can be empty.

For each connected vehicle, the `CollisionSystem` component provides an individual `CollisionCalculator` component instance. Accordingly, the number of these instances varies between 0 and 32, as well. At system start up, the minimum number of components and ports is instantiated, i.e. zero.

The question arises how the free slots in the component and port arrays can be used and released at runtime. We realize this by introducing a *reconfiguration service interface* as required by (RD3). This interface allows external components or even external software to request reconfigurations. More precisely, it allows external clients to request a port from a dynamic array.

The reconfiguration interface is defined not just by declaring a dynamic port array, but by the reconfiguration conditions using it, cf. L.13 in Figure 3.3. To query reconfiguration requests in a reconfiguration condition, we introduce the new port property `connect`, which is basically a Boolean flag indicating whether a connect request for this port has been issued, bundled with an id to avoid confusions with other requests sent to the same port. Similarly to the value at a port, the `connect` property can be queried using the `::` operator, i.e. as `portName::connect()`. A reconfiguration condition can be composed as a conjunction of arbitrarily many connect atoms, i.e. `portName1::connect() &&,...,&& portNameN::connect()`, where the port names used must be dynamic port arrays declared in the component's interface. Disjunctions and negations of connect atoms are forbidden by a context condition to prevent inconsistencies (in a disjunction we do not know at design-time which port(s) will be actually requested and hence, cannot define meaningful reconfigurations using these ports).

The resulting reconfiguration interface can be used by issuing connect request for all

the ports required by the reconfiguration condition simultaneously. In our example this means that, due to the reconfiguration condition in L.13 of Figure 3.3, connections to the `otherStatus` *and* the `otherTrajectory` port must be requested at once. Such a request is created in an EMAD model in the reconfiguration body of a parent component as connect statements targeting the corresponding dynamic port arrays. This is shown in Figure 3.4, where a component holding an instance of `CollisionSystem` connects to the aforementioned port arrays `otherStatus` and `otherTrajectory` of the latter in L.4-5 of its own reconfiguration body.

Note that the reconfiguration bodies of Figure 3.3 and Figure 3.4 are chained: the reconfiguration of the latter triggers the one of the former. If `ReconfigurationCondition` in L.3 of Figure 3.4 is a data-driven reconfiguration as discussed in Section 3.4.2, the chain starts in Figure 3.4. If `ReconfigurationCondition` defines a reconfiguration interface similar to L.13 in Figure 3.3, it must be triggered from another reconfiguration body itself. Hence, arbitrarily long service-based reconfiguration chains can be initiated by a data-driven reconfiguration.

Note that the reconfiguration request issued by the parent component of the `CollisionSystem` component in L.4-5 of Figure 3.4 matches the reconfiguration interface defined in L.13 of Figure 3.3 exactly. This is verified at compile-time by a context condition. An invalid usage of the reconfiguration interface of the `CollisionService` component is shown in Figure 3.5. Here we are trying to connect to the `otherStatus` port only. However, this is not supported and results in a compile-time error as there is no such reconfiguration condition in the `CollisionSystem` component.

To be able to deal with dynamic port and component arrays in reconfiguration descriptions, we need a syntax allowing us to access the newly created elements. To do so, we introduce the `?`-operator. It is used instead of the element number in square brackets to request and access new elements in a dynamic port or component array, e.g. `myArray[?]`. Usage of the operator is restricted to reconfiguration bodies.

An example is given in L.14-17 of the `CollisionSystem` model in Figure 3.3. In L.14 the `?`-operator is used to connect the `ownTrajectory` port to a new component `cc[?]`. Since this is the first access to `cc[?]` in this reconfiguration body, it implicitly triggers the creation of a new component instance. In contrast, further accesses to `cc[?]` in L.15-17 are pure access operations, no implicit instantiation is involved. If the component type of the component array requires component parameters, the parameter list can be passed in parenthesis right after the array brackets and before the dot operator, e.g. `cc[?](param1, param2,...).ownTraj`.

Since the `cc` array has a maximum capacity which cannot be exceeded, a further implicit reconfiguration condition is that the maximum capacity of this array has not yet been reached. If, however, the array is maxed out, the reconfiguration condition will evaluate to false and the reconfiguration will thus not be activated.

In this example we only need to add one element to the `cc` array. If we needed to create several elements, we would need a syntax to distinguish them. Therefore, the question

```
1  dynamic component DynamicSum {                                  EMAD
2    port dynamic in Q summands[0:32],
3                 out Q sum;
4
5  implementation Math {
6    Q tmp = 0;
7    for i = 1:size(summands)        <- iterates over all ports in the
8        tmp = tmp + summands(i);       input port array summands
9    end
10   sum = tmp;
11 } }
```

Figure 3.6: Adder with 0 to 32 inputs.

mark can be followed by an integer. Then we would use `cc[?1]`, `cc[?2]`,..., `cc[?n]` to create and access `n` different elements. To simplify the syntax, the index can be dropped for the first element (which is sufficient for the majority of cases). As in standard EMA, indices start with 1.

The reconfiguration service interface is available not only at modeling level allowing other components to use it, but also in the generated code. The latter can be used by any client. For instance, the `CollisionSystem` component can be generated and compiled to a library to be deployed as a building block of the vehicle run-time environment (RTE). The RTE receives a stream of vehicle to vehicle (V2V) messages and redirects them to the right ports of the `CollisionSystem` library (each sender is assigned to one port). If a new LTS participant starts sending, the RTE can request a new port from the `CollisionSystem` library by calling a generated request function. The library in turn checks whether the request is satisfiable. If yes, it provides a new port instance the RTE can forward messages of the new vehicle to. Otherwise no reconfiguration is carried out and the library call returns with a `NO_SAT` error. The client can then withdraw the request or wait until the dynamic component satisfies the request in a future reconfiguration cycle.

To facilitate the usage of the generated reconfiguration interface, we generate request methods allowing the client to require all necessary ports to activate a reconfiguration with a single function call, e.g. `requestOtherStatusAndOtherTrajectory(Port<T1> *otherStatus, Port<T2> *otherTrajectory)`, where `Port<T>` is a generic class representing an EMA port of type `T` at C++ level. This way, it is not possible to create invalid request, e.g. requiring only an `otherStatus`, but no `otherTrajectory` port, when using the generated code as a library.

Figure 3.6 shows an example combining a dynamic interface with a MontiMath implementation. The purpose of the component is to compute a sum of all inputs and to output the result. This is a typical data aggregation example working on a varying num-

ber of inputs. The dynamic input port array `summands` can contain 0 to 32 elements, i.e. at instantiation the component has no inputs and outputs zero due to the initial assignment `tmp = 0` in L.6. The loop in L.7-9 iterates over all ports in the `summands` array and adds each port's value to the overall sum, which is accumulated in the `tmp` variable. In this example, we treat the dynamic port array in a stateless anonymous way. We iterate over the port array and are only interested in the value present at each available port without caring about its history. This is the natural way to deal with dynamic port arrays in MontiMath. Tracking states related to dynamic ports using MontiMath is possible but highly discouraged. Instead, to track a concrete dynamic port's history, we need to replicate a dynamic subcomponent for each dynamic port instance, as was done in Figure 3.3. This way, each communication partner requiring a port in a dynamic port array is assigned a dedicated processing subcomponent maintaining the corresponding state. Each of these dedicated processing subcomponents only sees a single input port of the dynamic port array it is assigned to instead of the whole port array. This pattern enforces the separation of concerns and high cohesion principle as the processing related to each communication partner is clearly encapsulated and limited to the actual logic (no explicit iterating over the port array is needed in the behavior implementation).

Based on the reconfiguration mechanism described in this section, we can model whole *reconfiguration chains* to realize deep or flat reconfigurations. A deep reconfiguration means that reconfiguration of a parent component triggers reconfigurations in child components. A `connect` to a subcomponent's port activates this port's connect flag which can in turn be used to trigger a reconfiguration in the subcomponent. In the same way, the subcomponent can trigger reconfigurations in its subcomponents and so on. When a parent component instantiates a static subcomponent in an EMAD model, it can connect its output ports immediately, e.g. as is done in L.11 of Figure 3.3. However, the subcomponent might be dynamic and new output ports might be added throughout the subcomponent's reconfiguration procedures. In this case, the parent component can react to newly created ports of the subcomponent by observing the dynamic ports' `connect` flags in the same way as it would observe connect request to its own input ports. This enables us to create reconfiguration chains propagating downwards into the hierarchy as well as those coming from the bottom and propagating upwards.

A reconfiguration chain is always performed in one single reconfiguration phase as an atomic transaction, i.e. if the chain breaks at some point, the whole reconfiguration is considered infeasible. If a failure occurs after some reconfiguration steps of the chain have already been carried out, these steps will be rolled back.

As in data-triggered reconfiguration, a reconfiguration remains active as long as the respective condition is evaluated to true. As was discussed above, whenever a new port request is issued, the port is created and a connector connected, the `port::connect()` property is activated for this port. This flag and hence, the configuration remain active until the requesting client removes its connector to the dynamic port. If the client created the connector as part of an EMAD reconfiguration, it would remove it, when

the condition of this original reconfiguration ceases to hold. If the client is an external software, it can use the reconfiguration service interface to roll back a reconfiguration available in the generated code. Such a rollback would remove all architectural elements created in the reconfiguration and trigger the rollback of reconfigurations of subcomponents. This way, a reconfiguration chain is rolled back completely. The rollback interface is not usable explicitly in an EMAD model to prevent arbitrary removals of ports leading to inconsistencies in an architecture.

The service-based reconfiguration procedure of EMAD models boils down to the following steps:

1. Request: an external component sends a set of connect requests.

2. Reservation: the receiving component checks if the requested ports are available, i.e. if the corresponding dynamic port arrays do not violate their respective upper limit constraint. If yes, the component returns references for the new ports, i.e. the newly allocated array indices, to the requester so that explicit access is possible in the future. Otherwise, the requester is informed that its request has been rejected.

3. Reconfiguration: in the reconfiguration phase of the component, the reconfiguration bodies of all valid reconfiguration requests, i.e. those complying with a reconfiguration condition, are realized (L.14-17 in the `CollisionSystem` example). Consequently, the component reacts to the external reconfiguration request by internal self-modifications.

4. Follow-up requests: possibly, the reconfiguration instructions of the previous step contain the creation of new ports and/or subcomponents, as well. In this case, the component becomes a requester itself initiating a follow-up reconfiguration in its subcomponents or external components.

In our target domain of interconnected vehicles we mostly need the combination of both data-driven and service-based reconfiguration, which, when used together, can result in a powerful symbiosis. Reconfigurations which emerge as reactions to environmental changes measured by sensors or to incoming messages can be modeled using the following pattern: a data-driven event stands at the beginning of an event chain. The reconfiguration caused by this event requests new components and ports triggering service-based reconfigurations, which in turn trigger further service-based reconfigurations. As soon as the original trigger vanishes, the reconfiguration chain is rolled back completely and the architecture returns to its initial state. A data-driven source event can be based on a sensor measurement (including the vehicle's antenna receiving messages from other cooperating traffic participants). A particular measurement value or the reception of a specific message would trigger a reconfiguration of the controller architecture, the internal reconfigurations of which are mostly service-based.

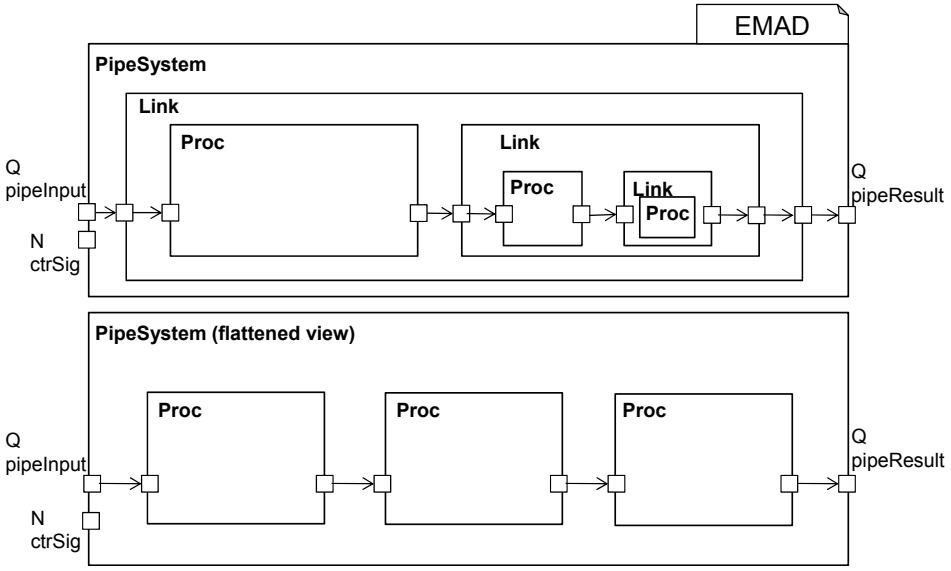


Figure 3.7: The pipe system component on top shows the structure of an arbitrarily long pipeline as defined in Figure 3.8. The last (or deepest) link component contains only a processing but no more linking subcomponents. Since EMA flattens hierarchical components, the effective architecture is a simple pipeline as shown in the bottom model.

An important aspect of EMAD is that there is no explicit way to *remove* architectural elements. Instead, elements are removed implicitly, whenever the triggering reconfiguration condition goes back to false. This guarantees that an architecture can always be put back into its original state as required by (RD5).

A further important property is that all possible reconfigurations are fixed by the design time model. Component and port replication is limited by an upper dynamic array size. Consequently, there is only a finite number of possible architectural states at runtime. This is an important design decision preventing a system to reach unexpected states and behaviors and facilitating verification.

3.4.4 Modeling Component Pipelines

Until now we have been modeling dynamic reconfigurations which can achieve arbitrarily hierarchical depths, but no flat chains such as filter pipelines as mentioned in Section 2.3.2 and which is possible using other ADLs such as Darwin [MK96]. Consider a scenario, where we need to pass a signal through a chain of similar components to process the desired result. For instance, in order to compute the n-th derivative of an input signal (with n being an arbitrary but initially unknown integer), we need to pass it through a

EMAD

```
component PipeSystem(N n) {
  ports in Q pipeInput,
        in N ctrSig,
        out Q pipeResult;

  instance Link topLvlLink[0:1];

  @ ctrSig::value() == 0 {
    connect pipeInput -> pipeResult;
  }

  @ ctrSig::value() > 0 {
    connect myInData -> topLvlLink [?](ctrSig::value()).linkInput;
    //connect myInData -> topLvlLink[?](n).linkInput;
    connect topLvlLink[?].linkResult -> pipeResult;
  }
}
```

Figure 3.8: The pipe system component uses the link pattern to instantiate an arbitrarily long (but finite) chain of processing components.

EMAD

```
component Link(N n) {
  ports in Q linkInput,
        out Q linkResult

  instance Proc proc;
  instance Link sublink[0:1];

  connect linkInput -> proc.dataToBeProcessed;

  @ n == 0 {
    connect proc.processingResult -> linkResult;
  }

  @ n > 0 {
    connect proc.processingResult -> sublink[?](n-1).linkResult;
    connect sublink[?].linkResult -> linkResult;
  }
}
```

Figure 3.9: The link component consists of a processing subcomponent and, optionally, a further link component, which in turn contains a further processing component and, optionally, a link component, allowing us to create arbitrarily long processing chains.

EMAD

```
1  component Proc {
2    ports in Q dataToBeProcessed,
3          out Q processingResult
4
5    implementation Math {
6      processingResult = f(dataToBeProcessed);
7    }
8  }
```

Figure 3.10: An example processing component to be used in the pipe system, where `f` denotes any valid MontiMath program using the data from the input port and writing a result to the output port.

chain of n derivation blocks. An example is depicted at the bottom of Figure 3.7 where a pipe system consists of a serial connection of three processing blocks.

While EMA offers the concept of arrays to cope with parallel structures efficiently, there is no way to define generic serial connections explicitly. However, such architectures can be modeled using the reconfiguration framework of EMAD. In the following, we are going to introduce the link component pattern. A link component contains a processing component and a further, optional link component. The second link component contains another processing and a link component, which allows us to create arbitrarily long chains of processing components.

This is illustrated using the pipe system example at the top of Figure 3.7 and in Figure 3.8. The `PipeSystem` component receives a data stream through its `pipeInput` port and outputs the computation result, which is computed by a chain of subcomponents, through the `pipeResult` port. The port `ctrSig` controls the length of the chain to instantiate. Alternatively we can use a component parameter to control the chain length. Then the length of the chain cannot be changed after instantiation. The reconfiguration in L.8-10 specifies that the data input is forwarded to the output directly if `ctrSig` is set to zero. If, on the other hand, `ctrSig` receives a value greater than zero, the reconfiguration in L.12-16 will be triggered. In this reconfiguration, a subcomponent of type `Link` is created and added to the component array `topLvlLink` defined in L.6, which can hold either one or no instances. Furthermore, the `pipeInput` port is connected to the `Link` input and the output of the `Link` component is forwarded to the parent component's `pipeResult` port. Note that the value at the `ctrSig` port is passed as a parameter to the new `Link` component created in L.13. L.14 depicts a slight variation of L.13, where the component parameter `n` is passed instead of a port value. In this variant, the pipeline is created at instantiation and cannot be altered afterwards.

The `Link` model is given in Figure 3.9. It contains a mandatory, i.e. static, processing component `proc` of type `Proc`, the concrete implementation of which depends on the application. In addition, the `Link` component can contain a further, optional

instance of type `Link`. This is controlled by the reconfigurations in L.10-12 and L.14-17, respectively. Note that the reconfiguration conditions depend only on the value of the component parameter `n` and are mutually exclusive. Since `n` cannot change at runtime, the reconfiguration chosen at instantiation will remain valid for the remaining component lifetime. If `n>0` evaluates to true, an inner `Link` instance is created with the decremented component parameter value `n-1`. This new inner `Link` component creates a further `Link` subcomponent itself if its component parameter is still greater than 0 and so on. The chain terminates when an instance is created with `n=0`. For the sake of completeness an example model for the `Proc` component with a MontiMath implementation is given in Figure 3.10. We abstract away from its concrete functionality by using a placeholder function `f(dataToBeProcessed)` in L.6. Some examples for linkable components from the digital signal processing (DSP) domain are numeric integrators and differentiators, delays, filters, controllers, and the like.

A graphical model of the discussed pipeline system with a chain length of 3 is given at the top of Figure 3.7. As discussed in Section 2.3.3, EMA flattens the component hierarchy internally. Hence, the effectively created model is the one at the bottom of Figure 3.7. Consequently, the overhead of the link pattern as it is visible in the non-flattened model disappears at runtime. To eliminate the need for the link pattern, EMAD syntax can be extended by dedicated syntax elements in the future.

3.4.5 Reconfiguration Views and Graphical Notation

Visualizing C&C architectures can often facilitate the work with complex models. However, grasping all aspects of a dynamic model in a single diagram is difficult. Instead, we propose to apply an adapted variant of the views concept to graphical C&C modeling in order to obtain meaningful EMAD visualizations. The concept of C&C views allows us to create abstractions of concrete C&C models bringing out specific aspects of interest while hiding details which are unnecessary for the understanding of the abstraction. For instance, views can be used to bring out the hierarchical relation of component instances without having to model the complete hierarchy or to specify the concrete types [MMR+17, BMR+17, KKRvW18]. To facilitate the understanding of dynamic reconfigurations, we introduce the notion of (graphical) reconfiguration views. Thereby, each reconfiguration is captured in its own view. A reconfiguration view only contains the actually changing parts and omits showing the static elements of the architecture. The diagram syntax is mostly the same as for normal C&C diagrams with several exceptions. First, the condition leading to the reconfiguration shown by the view is found in a box below the component name. Second, all architectural elements, i.e. ports, components, and connectors, created during the reconfiguration are depicted using dashed lines and boxes. Ports which trigger the reconfiguration are emphasized by an exclamation mark and, optionally, by using another color. Furthermore, if the reconfiguration is data-triggered, the triggering value should be written in the box representing the port

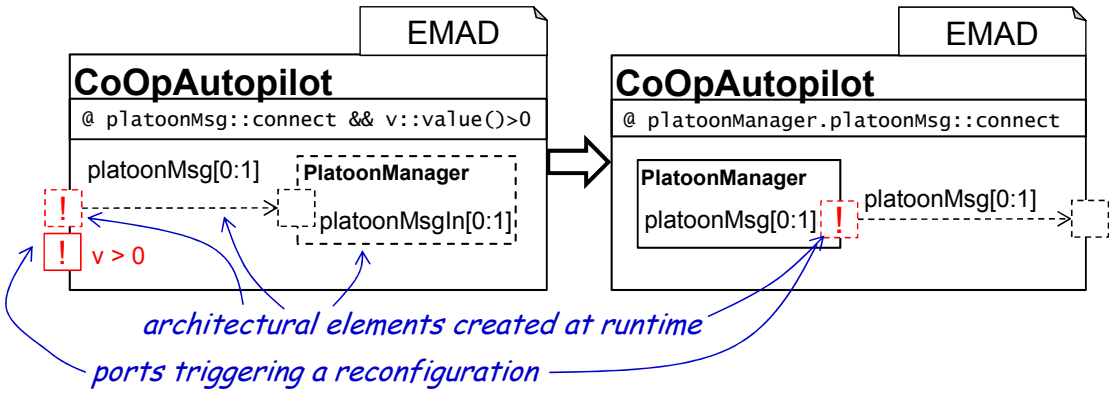


Figure 3.11: A reconfiguration chain involving input and output ports of the `PlatoonManager` component. An arriving platoon message causes the creation of new input ports in the diagram on the left. Follow-up reconfigurations inside the `PlatoonManager` result in a new output port and a new outgoing connector as depicted in the diagram on the right.

or next to it.

Since reconfigurations are mostly executed in chains, it is crucial to group graphical reconfiguration views so that the chain becomes visible. An example reconfiguration chain consisting of two reconfigurations is shown in Figure 3.11. In the first reconfiguration, depicted on the left, the `CoOpAutopilot` component, a controller of a cooperative vehicle, instantiates a platoon manager when a platoon port is requested and the velocity is greater than 0. In a second reconfiguration step, an inner component of the platoon manager requested a new output port and the `CoOpAutopilot` component reacts by creating a new connector. The ports triggering the reconfigurations are emphasized with an exclamation mark. Additionally, the data condition (`v>0`) is set next to the corresponding `v` port. Note that the `PlatoonManager` component is depicted using a dashed line in the left view, while it is solid in the view on the rhs. This is because the component is already there, when the second reconfiguration event is triggered. A big arrow between the two views stresses the order of reconfigurations. Obviously, a reconfiguration must have taken place inside the `PlatoonManager` component to request the creation of its new output port `platoonManager.platoonMsg`. This reconfiguration (chain) is not part of the depicted sequence as it is not in the scope of the `CoOpAutopilot` component and should be visualized in a separate view chain.

3.4.6 Remarks on Architectural Consistency

For static EMA architectures, it is straightforward to verify at compile-time that a model is consistent. For dynamically changing architectures we need to ensure consistency

```
Z3
1  //Declarations
2  (declare-const a Real)
3
4  //Range assertions
5  (assert (<= a 255))
6  (assert (>= a 0))
7
8  //Event condition satisfiability checks
9  (push)
10 (assert (and (> a 0) (< a 127)))
11 (check-sat)
12 (pop)
13 //further checks
```

Figure 3.12: An example of generated Z3 code proving condition satisfiability.

using appropriate context conditions for all possible reconfigurations and reconfiguration sequences. Some of these checks were already mentioned in the discussion of EMAD concepts. A list of checks performed by the EMAD CoCo checker is given in the following.

Static Context Conditions. We apply context conditions from static EMA to reconfiguration block bodies. For instance, each input port of a subcomponent must be fed by a connector. A subcomponent's input port cannot be floating. When a subcomponent is created in a reconfiguration block of an EMAD model, connectors for its input ports must be created, as well. Otherwise, the behavior of the created subcomponent would be undefined.

Valid Reconfiguration Requests. For each reconfiguration block of a component requesting new ports of a subcomponent, it is checked at compile-time whether the reconfiguration interface of the subcomponent is fulfilled. This is the case if the requests of the parent component trigger at least one reconfiguration of the subcomponent. For instance, if a subcomponent has a reconfiguration condition of the form `@ portA::connect() && portB::connect()`, but the parent component only connects to one of these two ports, the subcomponent cannot handle the reconfiguration request adequately and the compiler returns with an error.

To avoid bad usage of the generated code, the generated reconfiguration interface does not provide functions to request separate ports, but to request all the ports required to trigger a reconfiguration. For the example given above this would lead to a single function of the form `requestPortAAndPortB(Port<T1> *A, Port<T2> *B)` requesting port `A` and port `B` in one shot. Thereby, `*A` and `*B` are out-parameters used to return the created ports to the client and `T1` and `T2` are their respective types.

The actual return value provides information on the success of the reconfiguration.

No Removal. There is no way to explicitly remove architectural elements in a reconfiguration block. Reconfigurations are atomic procedures which get rolled back automatically if a part of the reconfiguration chain fails or if the condition causing the reconfiguration is no longer valid. An exception are functions of the form `freeXXX()` offered by the dynamic component interface. They is offered as part of the generated C++ code API and allows an external client to inform the component that it no longer needs some dynamic input ports and will not provide any more input data. As a consequence the informed component will roll back the reconfiguration chain which was triggered when the external client requested the said dynamic ports. For the example given above, the function `freeAAndB()` would dispose the ports `A` and `B` in one shot and unroll the reconfiguration once triggered by their creation.

Satisfiable and Replicated Conditions. A reconfiguration condition is a Boolean expression and hence, might be unsatisfiable. This is checked at compile-time by converting the reconfiguration condition into a Z3 program and letting the Z3 solver check it for satisfiability [DMB08]. Each port is defined as a variable with an equivalent name in Z3. For port array accesses, we append the character `*` to the array name followed by the index, e.g. `a[2]` results in `a*2`. A similar pattern is applied for time series accesses, but instead of an asterisk, an underline is appended. For instance, for `a::value()==[false tick false tick true]` we generate three variables `a_-2`, `a_-1`, and `a`.

The Boolean formula is pretty-printed using the Z3 syntax and then checked for satisfiability. If the port variables have a range, it is encoded in Z3 as an assertion. Consider a port `a` of type `Q(0:255)` and the reconfiguration condition `a::value() > 0 && a::value()< 127`. This condition leads to the Z3 code snippet given in Figure 3.12. In L.2 the variable representing port `a` is declared as a `Real`. More declarations could follow here.

L.5-6 assertions constrain the variable to its allowed range. The actual Boolean expression is printed in L.10. L.11 checks for satisfiability and returns `sat` or `unsat`. If the latter happens, the context condition fails and the model is invalid.

Similarly, we check that there are no reconfigurations triggered by the exactly same condition. Therefore, we create pairwise checks for all pairs of reconfiguration conditions of a component. For each resulting pair of Boolean formulas A and B, we check that $A \neq B$ is satisfiable. If this is not the case, the context condition fails.

Mutual Port Usage. In EMA an input port cannot be accessed by more than one connector. However, it is legal that two different reconfigurations create connectors leading to the same input port as long as their conditions are mutually exclusive. A context condition checks for each pair of reconfigurations of a component whether the

sets of target ports used in the said reconfigurations are disjoint. If this is not the case, a Z3 program is generated checking that $A \wedge B$ is not satisfiable if A and B are the Boolean expressions representing the conditions of the clashing reconfigurations. Recall the setting of the `BMux4` example given in Figure 3.1. Here, we have two reconfigurations, both targeting `mux.inSig[1]` and `mux.inSig[2]`. Therefore, the context condition needs to check, whether the conditions are mutually exclusive. This is generated to Z3 as the assertion `(assert (and (= ctrSig*2 true) (= ctrSig*2 false)))`. Since this is obviously unsatisfiable, the model is valid.

This example shows another interesting context condition: we have discussed that it is forbidden to leave input ports unconnected. In this example `mux.inSig[1]` and `mux.inSig[2]` are not connected in the static part but only in the dynamic reconfiguration blocks. To ensure that the ports are *never* floating, a context condition (having realized that the port is not connected to in the static part) would look up all reconfiguration blocks connecting to these ports. Next, it would create a disjunction of the respective reconfiguration conditions $\bigvee_i A_i$ and check that its negation is unsatisfiable, i.e. that there is always one condition which is satisfied. Furthermore, static input ports connected in the static part cannot be used as targets in reconfiguration blocks.

3.5 Conclusion

In this chapter we discussed EMAD, a conservative extension of the inherently static EMA language enabling the developer to model architectural changes happening at runtime. Due to the conservative extension property, each valid EMA model is also a valid EMAD model [HR17].

EMAD introduces an event-based reconfiguration system which can react to data-driven as well as architectural events. An EMAD component can instantiate ports, subcomponents, and connectors at runtime as a reaction to a triggered event. Thereby, it can trigger further events of its subcomponents, enabling the modeler to define complex reconfiguration chains.

In EMAD, all possible configuration states are implicitly defined at design time, maintaining the possibility to analyze, predict, and verify the behavior of dynamic components at design and compile-time. A set of context conditions ensures that reconfigurations never clash, making the language applicable to safety-critical systems.

In particular, EMAD can be used to model cooperative systems and their dynamically changing communication channels and processing chains, e.g. in the context of local traffic systems. Modeling of flat sequential component chains requires the application of the link pattern. Further syntactical elements can be designed in the future to facilitate the development of such structures.

Chapter 4

Modeling Artificial Neural Networks with MontiAnna

4.1 Deep Learning for Autonomous Systems

In Section 2.4.4, we have discussed how the behavior of EMA components can be described using MontiMath. The main paradigm behind MontiMath is procedural imperative programming offering common control flow elements such as sequences, loops, and conditions. Furthermore, a program can be formulated as an optimization problem. However, many tasks of intelligent autonomous system design including object recognition, planning, and control can be tackled using *machine learning* techniques. In this chapter we will discuss a domain-specific modeling approach for the development and integration of *deep learning* components.

The research question to be answered in this chapter is the following:

Research Question 3. *How can deep neural network models be designed and integrated into CPS architectures at SMArDT level 3?*

4.1.1 Supervised Machine Learning Foundations

Before diving into the design of a modeling approach, we are going to introduce the notation and wrap up foundations of the machine learning domain important for our design decisions. The field of machine learning deals with algorithms learning from data. There are reams of learning models and algorithms, but, in general, the idea behind supervised learning can be condensed as follows: given a *set of training examples* $(x_1, y_1), ..., (x_N, y_N)$ with $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$, we want to *find the best* function $f \in C$, mapping some fixed domain $\mathcal{X}$ to a fixed co-domain $\mathcal{Y}$, in a process called *training*. Here, C is a function space, i.e. a set of functions. What the best function is in a particular context is estimated using a loss function $\mathcal{L}(f(x_i), y_i)$, assessing the output of the function for a given input when the true output is known. The objective of the training problem is the minimization of the estimated loss over all possible functions in

C, i.e.

$$\hat{f} = \arg\min_{f \in C} \mathbb{E}[\mathcal{L}], \tag{4.1}$$

where $\mathbb{E}$ denotes the expectation value of a random variable as introduced in Section 1.7.

Based on these considerations, we can fathom out the main aspects of a machine learning-based system a developer needs to deal with:

1. the function space C,
2. the training procedure,
3. the training data.

In classical programming paradigms, the function space C is the set of all possible programs we can write in the programming language of our choice and f is a concrete program, which we need to write manually. In machine learning on the other hand, we rather let the algorithm try to find an appropriate solution based on data. Many of today's computer languages and development tools have not been designed with data-driven thinking in mind, which leads us to the assumption that a new DSL (or a new DSL family) might be a favorable approach for the design of machine learning-based behavior. The aim of this chapter is to develop and evaluate such a language family focusing on the three aspects listed above. These aspects will be referred to as the *three modeling concerns of machine learning*.

4.1.2 Neural Networks

The choice of the function space C is highly dependent on the application, i.e. the distribution of the data, and therefore requires a lot of expertise, domain understanding, and experimentation. However, a flexible and extensible framework for a modular composition of function spaces is provided by the field of artificial neural networks (ANNs). ANNs belong to the family of parameterized models, i.e. the function space represented by a neural network is spanned by a set of parameters, i.e. $C = \{f_\theta | \theta \in \mathbb{R}^K\}$, where K is the number of parameters. The goal of the training for a parameterized model hence boils down to an optimization over the parameter space, i.e.

$$\theta = \arg\min_{\theta} \mathbb{E}(\mathcal{L}). \tag{4.2}$$

The main building blocks of a neural network are its neurons, which in turn are parameterizable functions. The basic neuron model can be formalized as

$$y = \sigma\left(w^T x\right), \tag{4.3}$$

where w is the vector of neuron parameters and hence, part of the parameter set θ; more precisely, w are the weights which are applied to the neuron inputs represented by x to create a weighted sum of the input using a dot product. For simplicity of notation, we assume that x_0 is a constant 1 and hence, the entry w_0 is an input-independent *bias*. σ is a non-linear, scalar-valued function. By making a neuron non-linear, a composition of a sufficiently large number of neurons can approximate any function, which makes neural networks applicable to a large variety of problems. Without the non-linearity, any composition of neurons boils down to a single linear mapping. A schematic representation of the neuron model described above is given in Figure 4.1. Further neuron models will be discussed in the course of this chapter when needed.

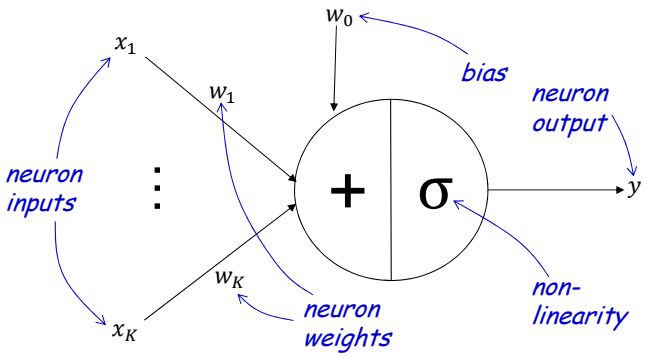


Figure 4.1: The basic neuron model.

A single neuron covers only a relatively simple function space and is of little use for complex tasks. However, arbitrarily complex function spaces can be modeled by interconnecting multiple neurons with each other.

Definition 3 (Artificial Neural Network)**.** *An artificial neural network is a weighted DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$, where each node $v \in \mathcal{V}$ represents a neuron and each connection $e \in \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ denotes a dataflow from the output of the source neuron to a dedicated input port of the target neuron. The data of the connection is weighted with the weight $w(e)$.*

Definition 3 is overly general and therefore not tractable in practice. First, it is cumbersome to describe and understand arbitrarily shaped networks with very large numbers of neurons. Second, it spans an exploding and difficult to cope with parameter space complicating the training procedure. Instead of building a neural network out of single neurons, which can be compared to building an enterprise application using assembler code, it has proven convenient to think in neuron layers.

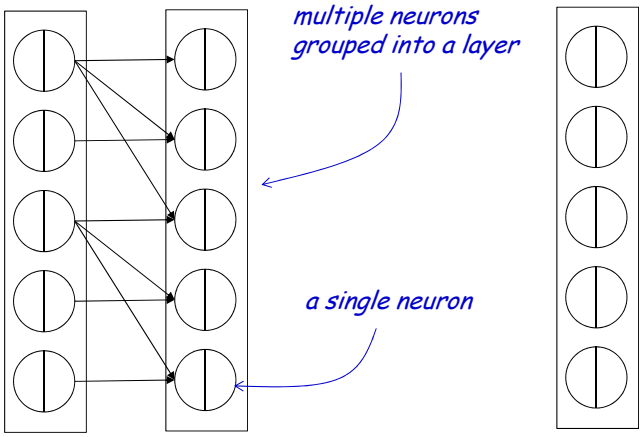


Figure 4.2: The abstraction of layers in artificial neural networks.

Definition 4 (Layered Artificial Neural Network)**.** *A layered artificial neural network is a neural network $\mathcal{G}^{(L)}$ where each neuron is assigned to exactly one of its L layers by a mapping $l : \mathcal{V} \to [1, L]$ and $\forall e = (v_s, v_d) \in \mathcal{E} : l(v_s) < l(v_d)$.*

Since neurons belonging to the same layer cannot be connected, $\mathcal{G}^{(L)}$ is obviously an L-partite graph. Combining nodes of each layer into supernodes, enables us to view a layered neural network as a graph of layers. A layer-oriented view is an important abstraction in deep learning. While it is difficult to understand the role of a single neuron in a deep neural network, a layer can be associated with a specific task and a powerful function space can be assembled using a relatively small number of layers.

Definition 5 (Layer Graph)**.** *A layer graph $\Gamma^{(L)} = (\mathcal{V}_\Gamma, \mathcal{E}_\Gamma)$ is an abstraction graph of a layered neural network $\mathcal{G}^{(L)} = (\mathcal{V}, \mathcal{E})$, where each layer is represented by a single, abstract supernode, i.e. $\mathcal{V}_\Gamma = \{v | v \subseteq \mathcal{V}\}$, $\bigcup_{v_i \in \mathcal{V}_\Gamma} v_i = \mathcal{V}$, and $\forall v_1, v_2 \in \mathcal{V}_\Gamma . v_1 \cap v_2 = \emptyset$. Furthermore, an edge between two supernodes $e = (v_1, v_2) \in \mathcal{E}_\Gamma$ exists iff there are two nodes $v_a, v_b \in \mathcal{V}$ so that $v_a \in v_1$ and $v_b \in v_2$ and $(v_a, v_b) \in \mathcal{E}$.*

A sketch of a layer graph is given in Figure 4.2. We introduce the following notation to facilitate the analysis and implementation of layered networks:

- $w^{[l](n)}$ is the weight vector of the n-th neuron in the l-th layer of the network,
- $W^{[l]}$ is the weight matrix for layer l, with an entry w_{ij} representing the weight of the j-th neuron at its i-th input,
- $a^{[l](n)} = \sigma\left(z^{[l](n)}\right)$ is the output of the n-th neuron in the l-th layer,

- $a^{[l]}$ is the vector of all neuron outputs of the l-th layer with $a^{[0]} = x, a^L = \hat{y}$ being the input and the output prediction of the network, respectively,
- $z^{[l](n)} = w^{[l](n)^T} a^{[l-1]}$ is the weighted sum of the inputs of the n-th neuron in the l-th layer (recall that, for simplicity of notation, we assume for all layers l of a layered network $a^{[l](0)} = 1$ to account for the bias parameter).

Defining the shapes of these parameters falls within the scope of the first modeling concern of machine learning dealing with the function space.

4.1.3 Training of Layered Neural Networks

The aim of the training of a neural network is to optimize the weights of the network with respect to (w.r.t.) an appropriate *loss function*. The appropriateness of a loss function depends on the type of the network output and should reflect the severeness of the deviation of the predicted output from the ground truth. Prominent loss functions include the $\mathcal{L}_2$ loss applicable to metric outputs and defined as

$$\mathcal{L}_2 = \frac{1}{2}(y - \hat{y})^2, \tag{4.4}$$

where y is the ground truth vector and $\hat{y}$ the prediction of the network. Furthermore, the cross-entropy loss is used in classification and is defined as

$$\mathcal{L}_{xe} := -\sum_{i}^{N} y_i \log\left(\hat{y}_i\right), \tag{4.5}$$

where N is the number of classes, y_i is an indicator function being 1 if i is the correct prediction and 0 otherwise, and $\hat{y}_i$ is the probability of class i predicted by the model.

Independent of the concrete loss function, we need to find

$$\underset{W^{[1]},...,W^{[L]}}{\arg\min} \ \mathbb{E}[\mathcal{L}], \tag{4.6}$$

where the expectation is over the training set. In addition, the objective function can be extended by a regularization term $R(W^{[1]}, ..., W^{[L]})$ in order to restrain the weight values (with the purpose to prevent overfitting).

The analytical approach to find the optimum of a function is to look for zeros of its first derivative. Since this is not feasible for the complexity of functions we deal with in machine learning, iterative gradient descend methods are used to minimize a loss function [LNC+11]. Starting from a random (or heuristically determined) set of initial weight parameters, in each step the gradient of the loss is computed w.r.t. the neuron weights and the weights are updated according to the rule

$$w := w - \lambda \nabla_w \mathbb{E}\mathcal{L} \tag{4.7}$$

or a variation thereof. Here $\nabla_w \mathbb{E}\mathcal{L} = \mathbb{E}\left(\frac{\partial}{\partial w_1}, ..., \frac{\partial}{\partial w_N}\right)\mathcal{L}$ denotes the gradient of the expected loss and the expectation is over the underlying data distribution. In practice the latter is approximated by the training set. The learning rate λ is a *hyperparameter*, i.e. a parameter controlling the training process. In contrast to model parameters, e.g. the weights of a neural network, hyperparameters are not optimized during training. The training results depend crucially on the choice of the hyperparameters.

We can obtain the first derivative of the loss function w.r.t. a single weight of a neuron in the last layer by applying the chain rule of calculus as

$$\frac{\partial}{\partial w_i}\mathcal{L}\left(y, \hat{y}\right) = \frac{\partial \mathcal{L}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_i} =: \Delta w_i. \tag{4.8}$$

Repeating this operation for every weight of the neuron yields the gradient vector $\nabla_w \mathcal{L}(y, \hat{y})$. To render this process more tangible, we are going to carry out an exemplary derivation using the $\mathcal{L}_2$ loss function and the well-performing rectifier linear unit (ReLU) function [GBB11] as the neuron's non-linearity, i.e

$$\mathcal{L} = \mathcal{L}_2 = \frac{1}{2}\left(y - \hat{y}\right)^2 \tag{4.9}$$

and

$$\hat{y} = \sigma(z) = ReLU(z) = \max(0, z). \tag{4.10}$$

Then,

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y, \tag{4.11}$$

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \frac{\partial ReLU(z)}{\partial z} = \begin{cases} 0, & \text{if } z \leq 0 \\ 1, & \text{otherwise.} \end{cases} \tag{4.12}$$

and

$$\frac{\partial z}{\partial w_i} = x_i. \tag{4.13}$$

Hence,

$$\frac{\partial}{\partial w_i}\mathcal{L}\left(y, \hat{y}\right) = \begin{cases} 0, & \text{if } z \leq 0 \\ \left(\hat{y} - y\right) x_i, & \text{otherwise.} \end{cases} \tag{4.14}$$

In the same manner, the derivative of the loss w.r.t. any other weight or bias of the network can be computed. Thereby, weight gradients of the last neuron layer are computed

first. The results are reused by the preceding layer, and so on. This process is referred to as *backpropagation*.

To obtain the expectation value $\frac{\partial \mathbb{E}\mathcal{L}}{\partial w_i}$ as demanded by Equation (4.7), we need to average the value of the derivative of the loss function over the whole training set. However, in practice often only a subset of the training set is sampled and used in each training step, cf. stochastic gradient descent (SGD) [Bot10].

The second modeling concern of machine learning deals with the construction of a training algorithm from the ingredients introduced above, including but not limited to the choice of the

- loss function,
- regularization strategy,
- optimization algorithm,
- validation technique,
- as well as the respective hyperparameters, e.g. the learning rate.

Mostly, the training can be put together from available black box algorithms (such as backpropagation for the computation of the derivative and SGD for optimization) and their respective hyperparameters providing a convenient form of reuse and hiding repetitive details from the user. This way, the developer only needs to provide a parameterization.

4.1.4 Deep Network Architectures

We are now going to discuss widespread network architectures, which we will later fall back on for the design of deep learning driven CPS architectures as well as the evaluation of our modeling framework.

Multilayer Perceptron

The most basic neural network architecture beyond a single neuron is the multilayer perceptron (MLP). It is a layered neural network consisting of an input, an output, and at least one hidden layer. The input and output layers represent the input data and the predictions, respectively. Each hidden layer receives data from its preceding and forwards data to the next layer, i.e. the layer graph of an MLP is a linear graph with a path from the input layer node to the output layer node. MLPs can be used to train arbitrary functions, provided there is sufficient training data and an appropriate number of neurons. Although, theoretically applicable to any supervised learning problem, MLPs can be difficult to train due to their generality. Instead, it can be more convenient to work with specialized layers tailored to the application domain. MLPs are particularly

useful in CPSs for decision making based on a (small) set of pre-extracted features, e.g. if we wanted to control the steering wheel, throttle, and brakes based on the actual velocity, the distance to the lane markings, the distance to the front car, and the like.

Convolutional Neural Networks

CNNs constitute a family of networks mostly used in fields related to image processing [LB+95]. The name stems from the introduction of *convolutional* layers, which are an important building block in neural image analysis. A convolutional layer receives an image $I(i, j)$ as its input; for each *channel* it convolves the image input with the channel specific *kernel* $K_c(i, j)$. The result of the convolution operation is the channel output, formally written as

$$C(i,j) = I(i,j) * K(i,j) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} I(k,l)K(i-k, j-l). \tag{4.15}$$

In practice, of course, the convolution is only carried out for a finite area of interest, i.e. within the image and kernel boundaries. The realization of the convolution can be further fine-tuned using the parameters *padding* and *stride*. By padding the image with additional values, image boundaries can be treated in a predefined way. The stride s specifies by how many pixels the filter is moved over the input image. It can be incorporated into Equation (4.15) by replacing $I(k,l)K(i-k, j-l)$ with $I(k,l)K(si-k, sj-l)$.

A kernel can be thought of as a feature detector; each channel of a convolutional layer learns to detect one individual feature. Thereby, each convolutional layer represents a level of abstraction: while the first layers of a CNN aim at detecting low level features such es edges and curves, the subsequent layers detect more elaborate structures composed of the primitive features they get as input.

Convolutional layers are often followed by a pooling operation. The task of a pooling layer is to reduce the amount of data output by the previous layer by applying an aggregation operation, mostly the maximum operator, to neighboring neurons, i.e.

$$P(i,j) = \max_{\substack{k \in [wi, wi+w-1] \\ l \in [wj, wj+w-1]}} I(k,l) \tag{4.16}$$

where w is the width and the height of the pooling window.

The convolutional parts of a CNN are often followed by neuron layers similar to the MLP, also referred to as *fully connected* or *dense* layers (in contrast to convolutional layers where neurons operate on a small receptive field).

In the context of CPSs, CNNs are mostly applied to camera data for object detection, scene understanding, decision making, etc. Thereby, the neural network can deliver different levels of abstraction [CSKX15]:

1. Mediated perception (low level of abstraction): the output of the network describes the detected objects in the original picture domain, e.g. indicating bounding boxes for recognized traffic participants.

2. Direct perception (medium level of abstraction): the output of the network describes features in the problem domain, e.g. the distance to the front vehicle, and can be thought of as sensor imitation [CSKX15].

3. End-to-end networks (high level of abstraction): an end-to-end network realizes a full controller outputting control commands based on the images provided by the cameras and camera-like sensors [BDTD+16].

In the following we introduce a selection of widespread CNN architectures.

AlexNet. AlexNet [KSH12] is a popular CNN architecture consisting of eight layers, the first five of which are convolutional while the remaining three are fully connected. AlexNet is sped up by distributing the training over two graphical processing units (GPUs). Furthermore, it is shown that using the ReLU activation function, the network learns faster than, e.g. with a `tanh` activation.

VGG. The VGG architecture [SZ14] is a very deep CNN architecture with up to 19 weight layers. It is based on the AlexNet and adopts concepts such as its ReLU nonlinearity. However, it uses a very small receptive field of the size of 3×3 pixels and a stride of 1. This reduces the number of weights per layer, thus enabling deeper networks and improving training time.

ResNets. Improving accuracy by adding more and more layers to a CNN leads to the problem of the *vanishing gradient*. As has been discussed in Section 4.1.3, the weight gradient is propagated from the output layer to the earlier layers. By repeated multiplication of small values, the gradient vanishes hindering training. The aim of residual networks or ResNets is to counter this problem by introducing *residual blocks* [HZRS16a]. Assume that a path of layers implements a function $F(x)$. Then, the corresponding residual block is given as $R(x) = F(x) + x$. Forwarding the input x to the output unchanged on a parallel path is referred to as an *identity shortcut connection*. In contrast to the other architectures discussed above, a ResNet cannot be implemented as a linear layer graph. ResNets can incorporate hundreds or even over a thousand layers [HZRS16b]. However, while deeper architectures tend to improve the network accuracy, their training is very slow. This problem is tackled by wide residual networks (WRNs) which offer a trade-off between width and depth [ZK16].

Recurrent Neural Networks

Until now we have been discussing pure feedforward networks. However, feedbacks are indispensable in many applications, particularly if the input is a time series. To tackle this issue, recurrent neural networks (RNNs) have been introduced. While the neuron model defined in Equation (4.3) is a stateless function, the idea behind RNNs is to introduce a feedback taking the neuron's history into account.

Vanilla RNN. This leads us to the vanilla RNN model defined as

$$y[t] = \sigma\left(w^T \begin{pmatrix} x[t] \\ y[t-1] \end{pmatrix}\right). \tag{4.17}$$

Note that we introduced a time dependency to the input and output. The square brackets reflect a *discrete time domain*, as is common in signal processing applications. Now the output of the cell depends not only on its input, but also on its *previous* output. The neuron model is depicted graphically in Figure 4.3.

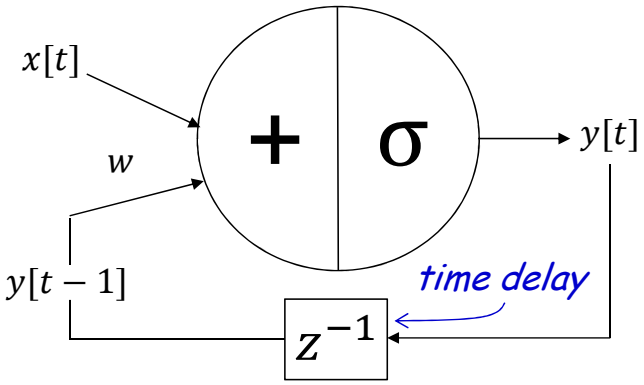


Figure 4.3: The vanilla RNN cell model.

In training, a single example consists of an input and an output time series. To train an RNN, backpropagation through time (BPTT) is used, an extension of the backpropagation algorithm discussed in Section 4.1.3. Thereby, each RNN cell is *unfolded* to "remove" the recurrent connection, c.f. Figure 4.4. The loss for a single example can then be computed as the average loss for all timesteps. The feedback of the neuron acts as a memory.

Long short-term memory. While RNNs are capable of capturing relationships between inputs within a short timespan, they struggle to keep track of long-term dependencies due to a vanishing or exploding gradient in the BPTT process. This issue is tackled by the long short-term memory (LSTM) cell, an extension of the discussed RNN cell

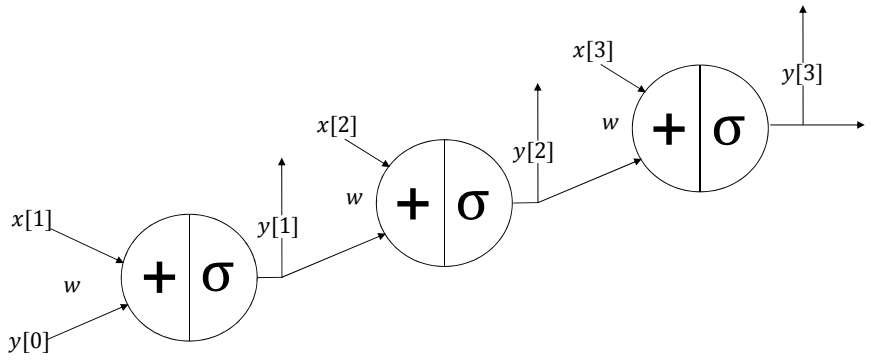


Figure 4.4: An RNN cell unfolded for three timesteps.

[HS97]. Similar to the RNN cell, an LSTM cell depends on its input, the last output but, in addition, also maintains a state $C[t]$. The introduction of *gates* allows the cell to add and remove information to or from its state. A gate is realized as a Hadamard product, i.e. point-wise multiplication, of the vector to control with a gate vector whose entries are in the range between 0=forget and 1=keep. An LSTM cell exhibits three such gates: the input gate $i[t]$, the output gate $o[t]$, and the forget gate $f[t]$, defined as

$$i[t] = \text{sig}\left(W_i^T \begin{pmatrix} x[t] \\ y[t-1] \end{pmatrix} + b_i\right) \tag{4.18}$$

$$o[t] = \text{sig}\left(W_o^T \begin{pmatrix} x[t] \\ y[t-1] \end{pmatrix} + b_o\right) \tag{4.19}$$

$$f[t] = \text{sig}\left(W_f^T \begin{pmatrix} x[t] \\ y[t-1] \end{pmatrix} + b_f\right), \tag{4.20}$$

where W_i, W_o, W_f, b_i, b_o, and b_f are the weight matrices and the bias vectors of the input, output, and forget gates, respectively. Furthermore,

$$\text{sig}(x) = \frac{1}{1+e^{-x}} \tag{4.21}$$

is the *Sigmoid* function applied elementwise to its input vector mapping its elements to $]0,1[$. The cell state $C[t]$ is computed as a superposition of the state candidate $\tilde{C}$ when it passed the input gate and the old cell state when it passed the forget gate. It can be written as

$$C[t] = C[t-1] \circ f[t] + \tilde{C} \circ i[t] \tag{4.22}$$

with

$$\tilde{C}[t] = \tanh\left(W_C^T \begin{pmatrix} x[t] \\ y[t-1] \end{pmatrix} + b_C\right). \tag{4.23}$$

The cell's output is obtained from the state and the input as

$$y[t] = \tanh(C[t]) \circ o[t]. \tag{4.24}$$

There are several variants of the described LSTM cell, e.g. adding peepholes, allowing the gates to look at the cell state [GS00]. A widely used variant is the gate recurrent unit (GRU) [CVMG+14]. It combines the forget and input gate into one update gate and merges the cell state with the cell output.

Attention. A major drawback of recurrent network models such as LSTMs is the linear way a state is propagated through the network. Such networks are bad at capturing long-range dependencies and cannot be parallelized. Usually, the encoder subnetwork first converts a given time-series into a fixed-size vector. A second subnetwork uses this representation to perform a concrete downstream task, e.g. machine translation or sentence classification. However, due to the short-memory nature of RNNs this approach fails for long sequences. This problem has been tackled successfully by the attention mechanism [BCB14]. Thereby, the decoder is not restricted to using a single vector representation, but has full access to all the intermediate states of the encoder. Thereby, it estimates the importance of the different states using an attention network.

An important neural network architecture for the processing of time-series is the transformer [VSP+17]. It is an attention-based encoder-decoder model, which, furthermore, refrains from the usage of recurrent neurons. Instead the network uses the attention mechanism to look at the entire input sequence. In addition to the encoder-decoder attention, the transformer uses self-attention to let the encoder and the decoder attend to their own sequences, respectively. A further benefit of abolishing recurrent neuron models is the possibility to parallelize sequence processing networks. State-of-the-art language models such as BERT [DCLT18] and GPT-3 [RWC+19, BMR+20] are based on the transformer architecture.

4.2 Requirements of a Deep Learning Modeling Framework for Cyber-Physical Systems

We are now ready to develop a modeling methodology for the discussed domain of machine learning with a particular focus on deep neural networks. In particular, we will aim at concepts allowing us to model and train the discussed architectures and neuron types and to cover deep learning applications from the CPS domain. Our target audience are industrial users without much emphasis on machine learning research, who need to design neural networks from known building blocks and integrate them quickly into large systems. This means that we are going to prioritize usability over flexibility.

Based on the analysis of the domain, we have elicited a set of requirements for a model-driven engineering methodology for artificial intelligence (AI)-powered embedded

and cyber-physical systems. These requirements will help us assess the existing GPL- and DSL-based frameworks in Section 4.3 and Section 4.4, respectively. Furthermore, they serve as the basis for our solution, which is the main contribution of this chapter. We distinguish between requirements for the deep learning **l**anguage(s) and the integration **m**ethodology, denoted as **RL** and **RM**, respectively.

(RL1) Appropriate domain representation: we expect from a deep learning language or framework that it has an intuitive representation of the domain concepts and is aware of their semantics. Central concepts of the majority of today's network architectures are *neuron layers* and *layer connections*. Such concepts should be offered by a deep learning language as first-level citizens and/or syntactic elements. The developer should not be obliged to construct them from lower level concepts.

(RL2) Domain specific pragmatics: it should not be necessary to make obvious information explicit in unambiguous contexts. The framework should rather be able to infer as many details as possible. For instance, in many cases the number of neurons in a layer depends on the output of the preceding layer. Although, this information can be inferred automatically, many frameworks require it as explicit parameters. Making the layer size explicit, however, is a problem, as it needs to be updated manually whenever a change occurs in one of the preceding layers.

(RL3) Reusability and modularity: structural patterns can often be found in all kinds of neural networks. For this reason, a neural network modeling language should provide an adequate modularity concept. For instance, a means for *layer composition* should enable the developer to group several layers into a new custom layer type; *network parameterization* should enable the adaptation of single layers but also of whole networks to a specific problem without having to change their internal structure. Furthermore, syntactic sugar can be used to facilitate *layer stacking*, a widely used pattern denoting a successive repetition of the same layer type (with possibly changing parameters).

(RL4) Separation of concerns: deep network engineering consists of several orthogonal concerns, namely the *architecture definition*, where the structure of the network is defined; *network training*, where the neuron weights are optimized based on a given training set and which can include validation and model-selection; the *training data* including its technical realization, preprocessing, and access; the intended *network execution* in the final system. A deep learning framework must separate these concerns clearly, e.g. the structural model must not contain any information on the training. Exchanging one concern should not affect the others.

(RM1) Neural component integrity: to be able to integrate neural networks in complex software, e.g. in the field of robotics, a neural network needs to exhibit a standardized interface, possibly accessible through a middleware such as ROS. The

interface should enable easy integration with other components as well as integrity checks, e.g. making sure that the neural network is compatible with the image format provided by an upstream camera.

(RM2) AI awareness: the engineering methodology must be aware of the neural network components present in the system under development. The compiler should know where to find training data and when to train or retrain individual networks. Unnecessary training needs to be avoided as far as possible. These aspects should not require human supervision, but must be fully automated instead. The developers should focus on design and not on the deep learning lifecycle.

(RM3) Platform independence: a modeling language should describe a neural network architecture and its training independent of the underlying implementation. Models should be reusable across different platforms.

Since the deep learning domain is highly active, new algorithms and features are added to the toolset of an AI engineer on a daily basis. However, often latest developments are only available in a small number of frameworks. A generative approach should hence be able to target different backends and have the ability to exchange the backend without considerable effort.

4.3 Overview of Deep Learning Frameworks

The field of deep learning has been moving fast since its revival due to the steadily increasing computational resources. Due to the gaining popularity of the domain, many frameworks facilitating the development, training, and deployment of neural networks have been proposed. In this section we give a brief introduction of the widely used solutions. The gained insights will contribute to the design of a deep learning language family in the following sections.

A condensed overview of the most important framework features is provided in Table 4.1. Mainly, we differentiate between low level or general purpose frameworks regarding a neural network as a graph of matrix or tensor operations and high level frameworks incorporating domain aspects such as network layers into their domain description.

Keep in mind that our analysis focuses on the frameworks' frontends and usability. Therefore, we omit a detailed discussion of the frameworks' performance and refer the reader to the corresponding benchmarks [SWXC16].

Theano. Theano [ARAA+16] is a low-level general purpose framework operating on symbolic mathematical expressions. It is written in Python and uses Compute Unified Device Architecture (CUDA). The latter ensures a high performance.

In Theano a mathematical expression is represented by a static computation graph. Such a computation graph is a bipartite DAG with two types of nodes: *variable* and

Table 4.1: Comparison of established deep learning frameworks and languages, √: yes, p: partially, −: no, *: depending on chosen backend, **: provided by extensions.

Deep learning framework	C&C integration	Type-safe component interface	Directed acyclic graphs	Low-level operators	Layer composition	Layer stacking	Layer independence	Parameterization	Interfaces	Static/Symbolic computation	Dynamic/Imperative computation	General purpose framework	Mobile deployment
MontiAnna (this thesis)	√	√	√		√	√	√	√	MontiAnna	*	*		*
Theano	−	−	√	√	√	√	√	√	Python	√	−	√	−
Torch	−	−	√	√	√	√	−	√	Lua, C	−	√	√	−
PyTorch	−	−	√	√	√	√	−	√	Python	−	√	√	−
Caffe	p	−	√	−	−	−	√	−	Command Line, C++, MATLAB, Python	√	−	−	−
Caffe2	−	−	√	√	√	√	−	√	Python, C++	√	−	√	√
TensorFlow	−	−	√	√	√	√	√	√	Python, C++	√	−**	√	√
MXNet	p	−	√	√	√	√	√	√	Python, C++, R, Julia, MATLAB, Go, Scala, Perl, JavaScript	√	√	√	√
Keras	−	−	√	−	√	√	√	√	Python	√	*	−	*
MATLAB NNT	√	−	√	−	p	p	√	√	MATLAB	√	−	−	−

apply nodes. The former represent data, while the latter denote mathematical operations. A variable node usually holds a matrix or a higher order tensor with statically typed elements, e.g. as `float32` or `int64`. Variable nodes can represent the provided graph inputs as well as the computed outputs and intermediate results. The syntax for mathematical expressions is similar to the Python library NumPy [WCV11]. The framework computes gradients of such expressions by means of symbolic differentiation.

Being a low-level framework, Theano's focus is on the mathematical aspects of neural networks rather than on a high-level domain-specific modeling. Hence, it violates our language requirements at least partially. Neural networks are assembled from low level operations, which can be a tedious and error-prone task. This makes it more of a framework for an academic audience rather than for industrial applications. Finding errors in a static computational graph is difficult: a problem occurs only when the graph is actually executed and not while it is assembled. Hence, error messages do not provide a reference to the erroneous construction code. Theano does not maintain a C/C++ API which can be a disadvantage in productive systems.

Torch and PyTorch. Torch is a framework for scientific computing and machine learning offering a Lua interface [CKF11, CBM02]. The core functionality is implemented in C and CUDA, ensuring high performance. Torch, in contrast to Theano, represents neural networks as *dynamic* computation graphs facilitating network debugging [LHHN17]. A dynamic computation graph is executed while it is built and can, hence, be debugged line by line. Furthermore, such graphs allow the usage of imperative statements and control structures during the construction and execution of the network. The network architecture can be changed at runtime. Torch offers a dedicated neural network library torch.nn providing the necessary building blocks to build neural networks as acyclic computation graphs. A network has a forward function computing the output for a given input and a backward function calculating the gradient for each parameter using automatic differentiation. A network can be built either in a sequential or a functional way. In a sequential definition a graph is defined as a list of layers, while in a functional definition each layer gets its input as a parameter. PyTorch is a Python implementation of Torch maintained by Facebook.

TensorFlow. Similar to Theano, TensorFlow [ABC+16], as its name suggests, is a low-level computing framework with a focus on tensor processing. It uses static computation graphs, as well, allowing for symbolic differentiation of arbitrary math expressions. Therefore, it suffers from the same disadvantages related to static graphs such as tedious debugging and a complex API. To benefit from the advantages of dynamic computation graphs such as eager execution, a more efficient handling of input data with changing structure and dynamic batching have been added to the framework by libraries such as TensorFlow Fold and Eager. TensorFlow 2 incorporates dynamic computation graphs

and eager execution as a core feature. TensorFlow is often used in conjunction with the higher level library Keras, which is tailored to the deep learning domain and therefore easier to use (see details below).

Caffe & Caffe2. Caffe is a C++ framework developed by the Berkeley Vision and Learning Center (BVLC) [JSD+14] for the design of CNNs with a focus on the reusability and deployment of pretrained models. The possibility to share network architectures and trained state-of-the-art networks contributed to the framework's popularity. Providing interfaces for C++, Python, MATLAB, as well as command line tools for training and prediction, Caffe models are highly portable. In Caffe neural networks are assembled from layers and stored in prototxt files. Four different model types exist: a network description for training, the network architecture for deployment, the hyperparameter configuration, as well as the stored dataset mean. The latter is computed automatically by Caffe for the sake of dataset normalization. Caffe requires two separate descriptions of the same neural network, since the model used in training needs to provide additional information, e.g. the name of the dataset, the initialization parameters of the layers as well as the loss function to be used. This model separation supports the separation of concerns principle as required in (RL4).

Caffe can be used in conjunction with different high performance databases such as LevelDB [GD11], LMDB, or HDF5 [FHK+11]. As a layer-based framework Caffe aims to represent the CNN domain as naturally as possible, thereby fulfilling (RL1). However this comes with the drawback that, unlike Theano and Torch, it does not provide a simple way to create custom layers or loss functions using low-level math operations. Another severe disadvantage of the prototxt format is the lack of reusability concepts and domain-specific pragmatics. In terms of lines of code, Caffe performs by far the worst among the discussed frameworks, as each layer has to be added to the network explicitly. Therefore, Caffe fails to fulfill the requirements (RL2) and (RL3). This makes the construction and maintenance of large networks such as the ResNet [HZRS16a] particularly tedious.

The successor, Caffe2, has abandoned the rather domain-specific, layer-centric prototxt format to replace it by a Python interface introducing the concept of operators. The operators can be employed to define new layers or low-level math operations making Caffe2 more of a general purpose framework, similar to Theano, Torch, and TensorFlow. Caffe2 is now a part of PyTorch.

Apache MXNet / Gluon. Similarly to Theano and TensorFlow, MXNet [CLL+15] models a neural network as a static computation graph. Training is performed by means of automatic differentiation on symbolic functions. Furthermore, the *autograd* package enables automatic differentiation on *NDArray* operation graphs using the eager execution principle. MXNet runs on all major operating systems and offers APIs for multiple languages. MXNet comes with a set of predefined loss functions which are combined

with the prediction output in the output layer. Furthermore, a developer can create custom ones as symbolic expressions.

MXNet serializes the network in two separate files: a JSON description of the actual network architecture and a binary file holding the network's trained weights. MXNet models can be deployed in production software using the C++-based prediction API. The API provides functions to load pretrained models and to execute the neural network for a given input. MXNet can package the complete prediction library as a single file including all the necessary dependencies. This is particularly convenient for neural network deployment in mobile and embedded systems. MXNet provides two interfaces: the original Symbol API provides a rather low-level access to the framework and works with static computation graphs exclusively. The newer Gluon API can be used to define both dynamic and static computation graphs. Thereby, the user can debug the network using a dynamic computation graph and hybridize it later on into a static computation graph to accelerate the execution of the final model.

Keras. Keras[1] is a high-level Python library aiming to provide an easy-to-use deep learning API. It does not provide its own backend. Instead it used to offer a variety of backends including Theano, TensorFlow, CNTK [SA16], and MXNet, thereby implementing the platform independence requirement (RM3). However, Keras 2.3.0 is the last major multi-backend release and is to be superseded by TensoFlow Keras (tf.keras).

Keras offers a layer-oriented API. Similar to Torch, a network can be constructed either as a sequential list of layers or as a functional model, where each layer obtains its input, e.g. the output of a preceding layer, as an argument. The latter variant enables the definition of complex networks. A Keras network architecture can be serialized as JSON or YAML. Trained weights are stored in an HDF5 database.

4.4 Machine Learning Modeling Frameworks

So far we have been discussing widely used frameworks for GPLs. In this section we are going to look at domain-specific modeling alternatives.

MATLAB Neural Network Toolbox. Having been designed as a matrix-processing language, MATLAB [Mat16] seems to provide a strong basis for the deep learning domain which it covers in a dedicated Deep Learning Toolbox (formerly Neural Network Toolbox) [DB09]. A neural network can be defined as a list of layers, similar to the sequential mode of Torch or Keras. Each layer can be assigned a name as an optional parameter if the layer needs to be referred to later on (which is a rather peculiar syntactic way of name declaration). Based on this list, the `layergraph` function then creates a static layer DAG of the neural network. To create a non-sequential architecture, e.g. a ResNet,

[1] https://keras.io/

additional layers can be added to the network using the `addLayers` function. To incorporate the newly created layers into the layer DAG or to add connections between existing ones, the `connectLayers` function can be used, taking a layer graph as well as the names of the source and target layers as its arguments. The `trainNetwork` function trains the network according to the supplied training data and options. Since the Neural Network Toolbox did not support symbolic differentiation, it was necessary to implement both the *forward* and the *backward* functions for custom layers or loss functions. The Deep Learning Toolbox can derive the backward function for a custom layer automatically using automatic differentiation if it consists of `dlarray` objects exclusively. Otherwise a custom backward function needs to be provided.

Due to the possibility of altering the network structure as described above, the toolbox supports a wide variety of linear and non-linear network architectures including recurrent or dynamic neural networks. Furthermore, a library provides pretrained state-of-the-art networks out of the box. If needed, these predefined networks can be adapted to new tasks by individual retraining. Computationally expensive training tasks can be run on one or multiple GPUs using MATLAB's *GPU-coder*.

Since MATLAB code can be used to implement the behavior of Simulink components, neural networks can be encapsulated easily and used as blocks in C&C architectures, as is required by (RM1). However, the compiler is not able to enforce a clear encapsulation of neural networks, due to its ignorance of what a neural network is. Hence, neural network code can be intermingled with unrelated procedural code. In other words, there is a lack of AI awareness as required by (RM2). This hinders an automated lifecycle management, e.g. knowing how and when to train and retrain the neural networks of a large system. Hence, the developer needs to take care of the training phase explicitly.

IBM SPSS Neural Networks. IBM's SPSS Neural Networks[2] is an extension to SPSS [MS17], a statistical analysis software, providing neural network development tools to SPSS users. SPSS supports two types of neural networks: MLPs and radial basis function (RBF) networks. The developer sets up the network by choosing from one of these two types and selecting columns from an SPSS dataset to be used as inputs and outputs, respectively. Further parameters, e.g. the minimum and maximum number of neurons in the hidden layers, can be set using a menu. SPSS then sets up and trains the network automatically, i.e. the user does not have to define the network architecture explicitly. The ease of use comes with the drawback of being restricted to a small class of simple networks with little possibilities for experimentation and research. Image, acoustic signal, and natural language processing are out of scope.

[2]https://www.ibm.com/us-en/marketplace/spss-neural-networks, accessed December 10, 2019

RapidMiner. RapidMiner[3] is a graphical C&C-based modeling tool for machine learning and data mining. The developer composes a system from blocks representing complete algorithms and by interconnecting these blocks with each other. RapidMiner offers a large library of learning models including neural networks. Neural network parameters related to both architecture and training can be set using a menu. In particular, the architecture of the neural network can be adapted using the `hidden_layers` list parameter, where each layer can be assigned an individual number of neurons. Hence, the developer has full control over the layers and their sizes, but is restricted to fully connected architectures, i.e. each node of a layer is connected to each node of the subsequent layer. More complex architectures can be realized using the DL4J extension.

Azure Machine Learning Studio. Azure Machine Learning Studio[4] is a cloud-based platform for data analysis and machine learning. The developer can define an experiment by choosing blocks from a built-in library and interconnecting them in a graphical editor. Hence, Azure Machine Learning Studio can be considered a graphical C&C tool. The predefined blocks provide functionality for all stages of an experiment, including data preparation, training, and prediction. Azure Machine Learning Studio provides different machine learning models including multi-class neural networks. Similar to IBM SPSS Neural Networks, the developer does not need to assemble the network architecture manually, but has rather to provide a set of parameters through a menu, e.g. the number of hidden neurons and the learning rate, to set up the neural network. Hence, the graphical platform comes with similar restrictions with regard to modeling of neural network architectures. However, the developer can provide models written in a GPL such as Python or R in order to insert any kind of manually written code. Furthermore, Microsoft offers Net#, a neural network specification language which can be used to customize the architecture of a neural network module.

Net# Neural Network Specification Language. *Net#*[5] is a purely textual neural network specification description language developed by Microsoft with a focus on deep layered neural networks. Azure allows the definition of custom components specified in Net# alongside normal components. A Net# specification model consists of layers and their connections. The developer can define arbitrarily many hidden layers and customize parameters such as the number of nodes and the activation function for each of them. The number of nodes in the input and output layers can be determined automatically from the context using the keyword `auto`, i.e. the shape of the input layer is set to the shape of the input data and the shape of the output layer depends on the problem

[3] https://rapidminer.com/, accessed December 10, 2019
[4] https://studio.azureml.net/, accessed December 10, 2019
[5] https://docs.microsoft.com/en-us/azure/machine-learning/studio/azure-ml-netsharp-reference-guide, accessed December 10, 2019

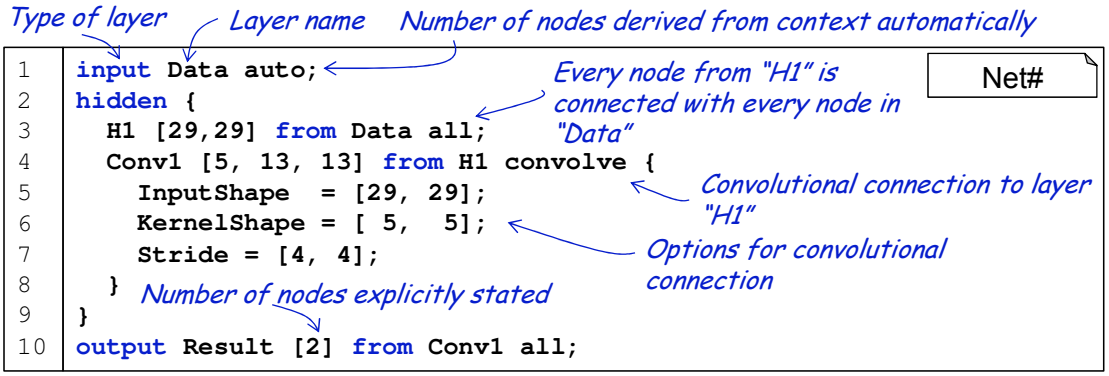


Figure 4.5: Neural network architecture defined in Net# [KPRS19].

type (one output neuron for regression, two output neurons for two-class-classification, etc.). If `auto` is used in the context of a hidden layer, its shape will be initialized from a corresponding constant parameter. To repeat a layer in the architecture, multiple layers have to be defined manually.

The resulting graph must be acyclic. Apart from that there are no restrictions on layer connections. A layer can be connected to multiple other layers enabling DAG architectures such as ResNets.

There are multiple configuration options, how the nodes of two layers can be connected. While the *full* connection mode connects each neuron of the source layer with each neuron of the destination layer, filters can be defined using predicates to describe customized connection patterns, e.g. to connect source neurons to target neurons of the same row of a two-dimensional neuron matrix. The predicate of such a *filtered* connection is a Boolean expression taking the indices of two nodes and connecting them iff it evaluates to true. Furthermore, `convolutional`, `pooling`, and `response normalization` connection bundles enable the definition of CNNs using Net#.

Being a textual DSL, Net# is the closest solution in terms of syntax to the neural network architecture description language to be presented in Section 4.7. Therefore, an example of a simple Net# architecture is given in Figure 4.5. The network has one input layer named `Data`, the two hidden layers `H1` and `Conv1` and one output layer `Result`. The number of nodes in the input layer is derived automatically as specified using the `auto` keyword. Layer `H1` contains $29 \times 29 = 841$ nodes which are arranged as a two-dimensional array. The second hidden layer is a convolutional layer named `Conv1` containing $5 \times 13 \times 13 = 845$ nodes. Further configuration parameters are set in L.5-7. All layers are connected in series using the `from` keyword. Input and output shapes of the convolutional layer need to be specified explicitly, but could be derived automatically instead by looking at the preceding layer, the kernel shape, and the stride.

Net# can only be configured with the predefined parameters, e.g. there are ten dif-

ferent activation functions and it is not possible to add custom ones. This restricts the usability of Net# for elaborate use cases.

DIANNE. Dianne is a graphical modeling tool for the design of artificial neural networks [DCBL+18]. While SPSS and Azure Machine Learning Studio operate on the abstraction level of neural networks, i.e. a neural network is a black box, in DIANNE neural networks can be composed from modules representing neuron layers and related low level operations, e.g. non-linearities. As such, the modeling domain is more similar to Net# and the GPL frameworks presented in Section 4.3. Modules from a library can be added to the neural network by drag-and-drop. Then, they need to be interconnected with each other to define the dataflow. Parameters of single modules (or layers) can be set using context menus. Similarly, the hyperparameters needed to train the network, e.g. the learning rate and the loss function, are set up in a dedicated menu. DIANNE provides learning strategies for different types of neural networks, including generative architectures such as GANs. DIANNE provides a runtime which enables distributed development and deployment of neural networks. The user can control which modules to execute on which device. Providing a high level of abstraction and focusing on usability it has an application-oriented target audience similar to us.

4.5 The MontiAnna Framework

The frameworks and languages presented in Sections 4.3 and 4.4 provide powerful means to develop deep learning applications and cover our requirements introduced in Section 4.2 at least partially. However, multiple software engineering related issues remain unsolved. Most frameworks are used through APIs in GPLs. Consequently, they are bound to the syntax of the host languages which were not designed with deep learning in mind. For instance, *layer stacking* and *composition*, concepts substantial for a clean neural network design, need to be realized using the host language, e.g. Python. Domain-specific deep learning modeling languages exist, as well, e.g. Caffe, Net#, and Dianne. These solutions often tend to suffer from limited expressiveness. For instance, there is no elegant way to instantiate many similar layers in Caffe, which leads to vast amounts of code necessary to define very deep architectures such as the ResNet152 network. What is more, modeling elaborate RNNs and applying advanced learning strategies such as reinforcement learning or generative approaches is out of scope of these solutions.

While syntactic nuances are rather a matter of comfort, a more severe issue, particularly in GPL-based frameworks, is posed by the fact that the compiler or interpreter is unaware of neural networks. This leads to several consequences for the software engineering process. First, avoidable boilerplate code managing the lifecycle of a neural network needs to be written, e.g. code related to loading data, storing the results etc. Second, not being first-level citizens in the host language, neural networks cannot be

clearly separated from other code parts. What is more, different concerns of a neural network itself, e.g. training and architecture, are not well-separated, either.

Software engineers need to handle the integration of neural networks into large systems manually, which includes tasks such as when to train and to retrain them. The automation of the machine learning lifecycle is highly desirable. The discussed model-driven solutions provide a higher degree of automation and require less boilerplate code, but are often limited in terms of expressiveness.

The contribution of this chapter is the introduction of MontiAnna, a deep learning modeling framework for textual neural network specification and design, enabling us to model the architecture of a very deep layered network such as the ResNet152 [HZRS16a] in no more than 31 lines of MontiAnna code, to embed it into a larger EMA architecture with little effort, and to define the training procedure using a concise notation [KNP+19, KPRS19].

The prefix Monti in MontiAnna highlights that the language family is based on the MontiCore language workbench [HR17, MSN17]. The acronym Anna stands for Artificial Neural Network Architectures. Here, the term architectures refers to both the neural architecture of a single neural network as well as to software architectures built from AI components, since we will use MontiAnna as an implementation language for EMA [KRRvW17, KRSvW18a] to embed neural networks into larger architectures.

4.6 An Overview of Modeling Languages

An overview of the MontiAnna framework is given in Figure 4.6. It is organized in six layers which can be understood as parts of the modeling framework, the compiler toolchain, and the generated artifacts. A short overview is given now.

The top layer in Figure 4.6 represents the layer through which a developer communicates with the framework. It enables the developer to model the architecture, the training, and the training dataset of a deep artificial neural network independently. To tackle the three modeling concerns of machine learning, identified in Section 4.1.1 as accurately as possible, we decided to use a dedicated DSL for each of them.

Network architecture description language. The central language of the MontiAnna framework is the neural network architecture description language CNNArc[6]. The purpose of this language is to describe the structure of an ANN in terms of neuron layers and connections or connection patterns between these layers defining the dataflows. What is more, the network architecture language provides means to assign specialized tasks to specific neuron layers, e.g. making them compute a particular activation function or

[6]The name CNNArc stems from the original starting point of designing convolutional neural networks. However, in the meanwhile the language was extended and evaluated on a variety of other network classes and is application-agnostic.

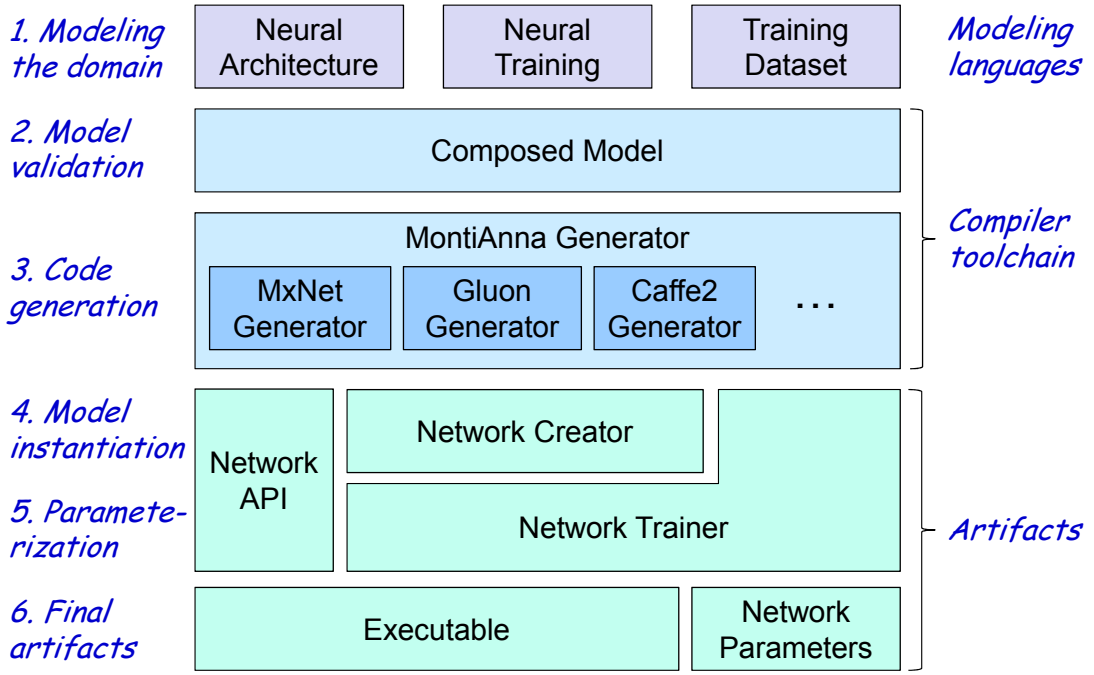


Figure 4.6: Overview of the MontiAnna framework layers [KNP+19].

share weights according to a given pattern. However, it neither specifies how to obtain the neuron weights nor where to get the data from. As such the network architecture modeling language of MontiAnna is a layer graph description language according to Definition 5. We will discuss it in detail in Section 4.7.

Network training. The dedicated training language CNNTrain serves the purpose of declaring the strategy to be used to find appropriate weights for a given network architecture. Such a strategy can be understood as a composition of optimization algorithms and the parameterization thereof as discussed in Section 4.1.3. The training model is defined as a separate artifact with the intent to keep all models concise, loosely coupled, and exchangeable. For instance, a developer can reengineer or replace the training strategy without touching the neural network architecture. We are going to discuss CNNTrain in detail in Section 4.9.

Dataset model. To conduct the training for a neural network architecture, a machine learning framework needs to know where the data is stored and how to retrieve it. For instance, there are multiple widespread high performance database solutions such as

HDF5 [FHK+11], LevelDB [GD11], and LMDB[7], which can be used to hold the training data. When using a GPL, database-specific code needs to be written to interact with the database.

Furthermore, it might be necessary to use multiple datasets, a particular subset of the available dataset only, or to even skip training if the available data has already been learned. Loading a dataset or switching from one database type to another should not require any boilerplate code to be written or rewritten.

In our approach all information regarding the dataset is captured in a declarative dataset model, which, again, is decoupled from the architecture and the training models and, thus, can be exchanged independently. We do not define a dedicated DSL, but use the tagging approach instead [GLRR15]. This is particularly convenient if we need to deal with multiple neural networks in a single software architecture as will be shown in Section 4.10.2. Note that the dataset model is not to be confused with a *data model* capturing types of a system and their relations, e.g. a UML class diagram [Rum17].

4.6.1 The Compiler Toolchain

The compiler toolchain consists of two layers: first, it composes the three independently defined models and performs context condition checks ensuring that the obtained composed model is valid. Then, this composed model is passed to a code generator. To generate the required neural network functionality, MontiAnna makes heavy use of existing frameworks discussed in Section 4.4, hence benefiting from their optimizations, but hiding them from the developer. As can be seen in Figure 4.6, MontiAnna provides generators for several different backends. New generators can be added by extending an abstract generator and providing the corresponding templates. The generated artifacts, again, can be classified according to their tasks.

In addition to context conditions performed on the composed model, the concrete code generator conducts feature checks, ensuring that it can handle all the functionality required by the model. This comprises architectural elements such as layers and graph operations as well as training related aspects such as optimization techniques and the corresponding hyperparameters. Such feature checks are necessary since code generators may be maintained with different effort, by different teams, or for different purposes. Furthermore, a target framework can provide exclusive functionality. Such exclusive functionality can only be used in conjunction with the respective code generator.

Whenever we want to add a new language feature to MontiAnna, we need to make it available in the syntax, i.e. at modeling level. Furthermore, we have to extend the code generators so that they can deal with this feature. Finally, we need to register the feature in the supporting code generators so that they are aware of their capability to generate it. This means in turn that the models defined using the MontiAnna DSLs are completely decoupled from the compiler in terms of features.

[7] http://www.lmdb.tech/doc/, accessed October 12, 2020

4.6.2 The Generated Artifacts

As soon as the generator chosen by the modeler has ensured that it is capable of generating the composed model, artifacts for neural network instantiation and training are generated. Furthermore, the generator provides CMake build scripts to facilitate the dependency management and the compilation of the generated code. Manual changes to the created artifacts or the inclusion of hand-written GPL code are neither needed nor desirable. We are going to discuss the main artifacts in the following paragraphs.

Network Creator. The `NetworkCreator` is an intermediate artifact, i.e. it is not used in the final system. Its purpose is to assemble the neural network defined in the neural network architecture model using the chosen target framework, e.g. MXNet Gluon.

Although our generator toolchain is supposed to deliver C++ code in order to be compatible with the generated code of EMAM2CPP, the EmbeddedMontiArc and MontiMath generator, we have decided to use Python as the target language for our intermediate artifacts. The reason for this decision is that Python APIs are often better documented, have larger communities and in some cases even provide more functionality than their C++ equivalents. Thus, network creation and training are performed in Python while only a small amount of code dealing with loading and execution of the final model is actually generated as C++.

Network Trainer. The `NetworkTrainer` is another intermediate Python module. Its content, primarily determined by the training and the dataset models introduced above, includes the training algorithm, its hyperparameters, as well as code to access the training data. The module performs the training for the neural network assembled by the `NetworkCreator` and outputs a serialized representation thereof (the Network Parameters box in the bottommost layer in Figure 4.6). The latter can then be loaded by another application to be used for its designated purpose, e.g. hand-written digit detection.

Network API. To embed the trained neural network module into a software architecture without having to deal with its internal structure, e.g. to implement the behavior of an EMA component, the MontiAnna framework generates a C++ API providing an execution interface as well as an implementation to load and execute the neural network. This artifact can either be used as source code in other software or it can be compiled to an executable by using the provided CMake build files or a C++ compiler.

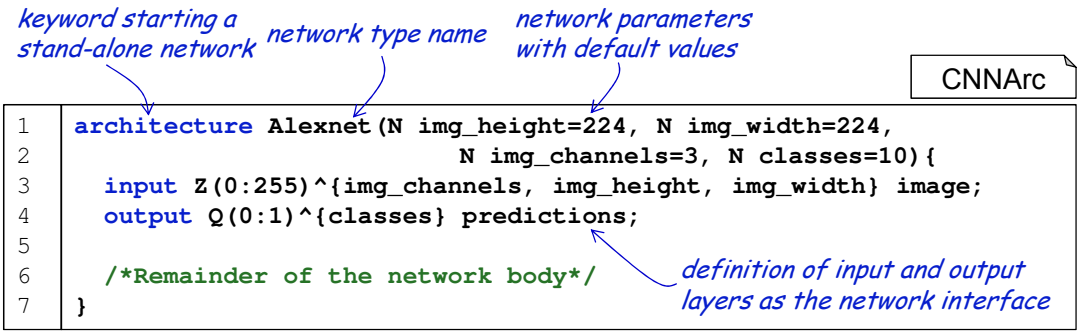


Figure 4.7: Definition of a stand-alone network of type Alexnet (based on the influential Alexnet CNN [KSH12]) and its interface.

4.7 Modeling Feedforward Neural Architectures with CNNArc

4.7.1 Defining a Stand-Alone Network

MontiAnna can be used as a stand-alone framework. In this case the developer models and trains a single isolated neural network without integrating it in a larger software architecture at once. This mode is best used for experimentation or rapid prototyping. In Section 4.10 we will discuss how to obtain a reusable neural network component or to integrate it in a larger EMA software architecture directly.

A stand-alone CNNArc network model is defined in an artifact with a .cnna extension as shown in Figure 4.7. The MontiCore grammar of the CNNArc language is given in the appendix in Listing B.12. The header of the network architecture is initiated with the keyword `architecture` followed by its name and a list of arbitrarily many *architecture model parameters* (not to be confused with hyperparameters which are defined in a corresponding CNNTrain model as discussed in Section 4.9 or network parameters which are the result of the network training step, cf. Section 4.10.3).

The body starts with the declaration of input and output neuron layers. These layers serve as the interfaces of the neural network and are defined using the `input` and `output` keywords followed by the type and a name. Type and shape of the input and output layers must conform to the type and shape of the training examples and labels, respectively.

In the remainder of the body, these special layers can be referenced by name, e.g. to connect them to a neural network. Note that multiple input and output layers may exist. For instance, image style transfer networks receive a content and a style image as inputs. The desired output is the content image enriched with stylistic details of the style image [GEB16].

4.7.2 Modeling Layers and Networks

As discussed in Sections 4.3 and 4.4, neural networks can be analyzed and modeled using different levels of abstractions. The appropriate level of abstraction depends on the complexity of the network but also on the problem to be solved. While very small networks can be composed of single neurons, this approach is not feasible for real world problems. Instead, large neural networks can be thought of as layer graphs or as tensor operations. The latter approach is very powerful and well-suited for experimentation and research, but rather cumbersome to use for typical, well-understood neural network applications and everyday problems. Since our aim is to provide a modeling technique targeting designers of industrial systems, we decided to develop a high-level layer-based language. The language enables the modeler to construct a neural network as a graph of layer instances.

Definition 6 (Layer Instance)**.** *A layer instance is a concrete occurrence of an array of similar neurons with all hyperparameters such as the number of neurons, activation function parameters, etc. set.*

A layer instance can be added as a node to a layer graph, cf. Definition 5. Practical applications reuse typical kinds of neuron layers, such as convolutional, fully connected, pooling layers and the like. To facilitate the creation of layer instances it is therefore helpful to define a library of parameterizable layer instance templates, hereinafter referred to as *layer classes*.

Definition 7 (Layer Class)**.** *A layer class is a blueprint for a layer instance including the behavior, interfaces, and, optionally, a set of instantiation and learnable parameters of the represented neuron model.*

Layer instances are the processing, side-effect free components of a layer graph with their abstract behavior determined by the respective layer classes. The concrete behavior is obtained during training by adapting the learnable parameters.

In the following we present a selection of important layer classes available in MontiAnna out of the box. This overview includes the parameters needed to instantiate a layer, the connection pattern denoting how many neurons of the previous layer are connected to a single neuron of the layer under consideration as well as the function performed by each neuron. For the parameter types we use the EMA type system as introduced in Section 2.2. Parameters can be passed as literals or as expressions of the corresponding type. However, in contrast to component or function parameters, which are evaluated at instantiation and at invocation time, respectively, layer parameters, being a specific kind of hyperparameters, are bound and hence must be known at training time, when the network is constructed for training, and cannot be altered thereafter.

FullyConnected. The fully connected layer class is the basic building block of most ANNs, including MLPs, CNNs, and RNNs. In some layer-based deep learning frameworks and APIs, e.g. in MXNet Gluon and Keras, such a construct is also referred to as a *dense* layer.

- **Parameter:** `N(1,`∞`) units` denotes the number of neurons in the layer instance and hence, its number of outputs.
- **Connection pattern:** all neurons of the preceding layer instance are connected to each neuron of the layer instance, hence the name.
- **Function:** performs the linear part of the basic neuron model given in Equation (4.3), i.e. $z = w^T x$, where $x_0 = 1$ and w_0 is the bias. The weights w need to be learned. The non-linearity can be implemented by connecting a `FullyConnected` layer instance to a non-linearity layer, cf. `Sigmoid` below and further non-linearities in Appendix B.1. Note that this is in contrast to high-level frameworks and APIs such as Keras and MXNet Gluon, where the activation function is passed as a parameter to the dense layer. Our decision to treat the non-linearity as a first-level entity provides more flexibility, reusability, and extensibility. Furthermore, it enforces the layer modeling paradigm of the DSL.

Sigmoid. The `Sigmoid` layer is a non-linearity layer, mostly used after a `FullyConnected` layer.

- **Parameters:** none.
- **Connection pattern:** each neuron of the preceding layer is connected to exactly one neuron of the `Sigmoid` layer, i.e. the number of neurons and hence, both input and output are derived from the preceding layer automatically. This will be referred to as the 1-to-1 pattern in the following. A `FullyConnected` layer followed by a `Sigmoid` layer realizes the basic neuron model as defined in Equation (4.3).
- **Function:** $\sigma(x) = \text{sig}(x) = \frac{1}{1+e^{-t}}$.

Softmax. Softmax is a special non-linearity which is mostly used at the output of a multi-class neural network. Its purpose is to normalize the outputs for each class to a probability distribution, i.e. the sum of all outputs must be one, while maintaining the ratios.

- **Parameters:** none.
- **Connection pattern:** similar to the other non-linearity layers, but the denominator of the Softmax function, i.e. the sum of all neurons' inputs of the layer, needs to be computed once and broadcast to all neurons.

- **Function:** $\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$.

Convolutional. The convolutional layer is the central layer class for CNNs.

- **Connection pattern:** each neuron maps a group of spatially related input neurons to an output. However, each neuron of the input layer can be used as input for multiple neurons.
- **Function:** performs a convolution of the input with one or multiple filters, cf. Equation (4.15). The filter matrices are learned during training. However, the concrete finite realization is controlled by the parameters described in the following.
- **Parameters:**

  `N^2 kernel`: the size of the trainable kernel to convolve the input with.

  `N^2 stride`: the number of neurons by which the filter is moved in each step of the convolution. The bigger the stride, the smaller the size of the resulting convolution output.

  `Padding padding`: In contrast to the theoretical definition of the convolution where the sums are from $-\infty$ to $+\infty$, the practical implementation operates only on the finite input and filter matrices. To keep the information at the matrix borders, new border neurons can be added, which is referred to as padding. Furthermore, padding allows the output to take the same size as the input. The `padding` parameter can be set to one of the three following values. `"valid"` is the default option meaning that no padding will be applied, i.e. only the *valid* area of the input will be used. `"same"` adds a frame of zeros around the original input matrix (zero-padding) so that the shape of the obtained convolution result is equal to the shape of the input matrix divided by the stride. Finally, `"no_loss"` is another zero-padding variant which adds a minimum number of zeros so that no entries of the input matrix are discarded. If the stride parameter is set to one, `"no_loss"` is equivalent to `"valid"`.

  `N channels`: the number of filters to be applied independently. This parameter spans a new dimension of the layer output.

The layer classes listed above represent the basic building blocks for the design of CNNs. Further layer classes will be introduced in the course of this chapter when needed. More layer classes are documented in Appendix B.1. An up-to-date overview of all supported layer classes is given in the documentation of the reference implementation[8]. Layer classes can be instantiated anywhere in the body of a CNNArc model by using the

[8] https://github.com/MontiCore/EmbeddedMontiArc/tree/master/languages/CNNArchLang

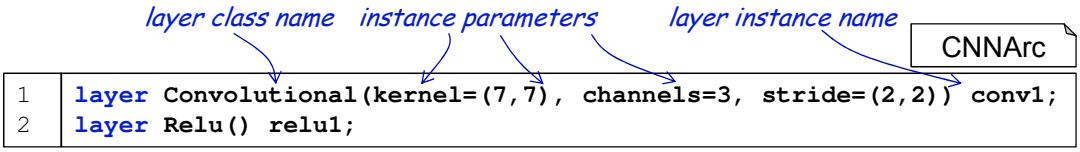


Figure 4.8: Named layer instantiation example.

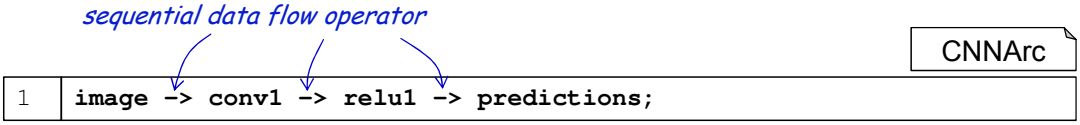


Figure 4.9: Graph expression example with layer instances chained using the sequential dataflow operator.

`layer` keyword followed by the layer class name to instantiate, a list of instance parameters, and a unique name of the layer instance. The instantiation statement is concluded with a semicolon. An example instantiating a `Convolutional` and a `Relu` layer is shown in Figure 4.8. As is common in most computer languages, two layer instances cannot have the same name, which is checked in a corresponding context condition. Furthermore, layer instance names must be declared before they can be referenced. Note that the layer parameter `padding` is not set explicitly for the `Convolutional` layer. In this case, the default (`"valid"`) is used.

As can be seen in the case of the `Relu` layer in Figure 4.8, layers with no parameters must have an empty parameter list, i.e. the brackets are mandatory. The same holds if the modeler intends to use the default parameters of the layer class. This is consistent with constructor calls in object-oriented languages, e.g. Java.

Instantiating layers does not add them to the layer graph representing the neural network. The layer graph is constructed using graph expressions interconnecting layer instances by means of connect operators. The first connect operator we are going to introduce is the sequential dataflow operator denoted as `->`. It creates a unidirectional connection from its left to the right operand, i.e. the output of the layer instance on the left is passed to the layer instance on the right in the forward pass of the neural network. A graph expression can be arbitrarily long. An example graph expression is shown in Figure 4.9.

Note that the sequential dataflow operator has the same syntax as the EMA connector. In fact, if we regard a layer instance as a component instance and its inputs and ouputs as ports, the two operators are very similar. The CNNArc operator, however, has an additional property: the dataflow direction is reverted during training to perform backpropagation. Such a reversion is strictly forbidden in EMA. Hence, the sequential dataflow operator of CNNArc is a variant of an EMA connector with controlled

CNNArc

```
1  image -> conv1 -> relu1;
2  relu1 -> predictions;
```

Figure 4.10: A neural network represented as two graph expressions.

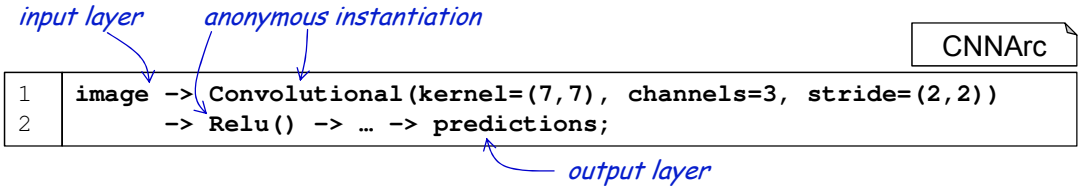


Figure 4.11: Anonymous layer instantiation example.

reversibility.

A neural network does not need to be built as a single graph expression. The CNNArc model body can contain several graph expressions. If multiple graph expressions are present in the model, the compiler will create a single graph internally, e.g. the model given in Figure 4.10 is equivalent to the one in Figure 4.9. The two graph expressions are joined using the `relu1` layer instance as it is present in both expressions.

Combining graph expressions as described above enables us to create arbitrary layer graphs. However, we restrict the allowed set of graphs to the set of all DAGs. Cycles are not allowed. Furthermore, there must be a path from the input to the output layer. Nodes from which the output layer cannot be reached are pruned. Nodes which are not reachable from the input layer are pruned, as well, if they have no data generating predecessor, e.g. a constant layer.

Since in feedforward architectures each layer instance is used exactly once, instantiating and naming layers before building the graph is often unnecessary. Therefore, we introduce the concept of anonymous layer instantiation. Instead of using names of layer instances in a graph expression, we can indicate the layer classes instead as is shown in Figure 4.11. A corresponding layer instance is created and included into the graph at the desired location. A large number of networks are fully or at least partially linear and hence, can be modeled using this syntactic sugar. The resulting CNNArc models are more concise and well readable.

To facilitate the definition of layer graphs with parallel dataflows, we introduce our second connect operator, the parallelization operator `|`. It splits the operands into separate independent processing pipelines taking the same input (layer instances that do not take an input ignore this). A processing pipeline can be left empty denoting a *skip connection*, e.g. in a residual block of a ResNet. The parallelization operator provides a concise syntax to define short parallel dataflows, mostly used in combination

with anonymous layer instantiation. Long parallel pipelines are better modeled using partial graphs with named interface layers.

To model more than two parallel processing pipelines, the operator can be used successively. However, the sequential dataflow operator has a higher priority than the parallelization operator. Therefore, it is necessary to group multiple parallelization operators using parenthesis. For instance, let `m1,...,m4` be layer instances, then the expression `(m1 | m2->m3 | m4)->` creates three parallel processing pipelines, which are combined in a single *parallelization block*. The output of such a parallelization block is a single list of the outputs of each pipeline. This list of data streams can be aggregated into a single stream either by using a merge layer, e.g. `Concatenate` or `Add`, or the selection layer `Get`. On the other hand, we can subdivide a layer's output into multiple parts using the `Split` layer. These layers, needed to create more complex graphs, are introduced in the following.

Concatenate. This layer class is used to concatenate an arbitrary number of input streams into a single stream along a given dimensions. The shapes of the input streams must be equal except the dimension along which to concatenate. In CNNs this layer is mostly used to concatenate multiple channels. Hence, the width and height of all input images must be equal and the number of channels of the output is the sum of the numbers of channels of all input streams. The layer takes an optional parameter `N dim` denoting the dimension along which to concatenate (default is 1).

Add. Aggregates multiple input streams into one output stream by elementwise addition. The shapes of all inputs must be identical. The output shape is equal to the input shapes. Similar layer classes for other arithmetic operations such as `Sub`, `Mult`, and `Div` for subtraction, multiplication, and division, respectively, are conceivable, as well.

Split. This layer class can be considered as the opposite of `Concatenate`. It does not change the input data in any way, but subdivides it into multiple partitions along one dimension and outputs a list of streams. In CNNs subdivision mostly takes place alongside the channel dimension. Hence, in the output height and width remain unchanged, while the number of channels per stream is $\lfloor c/n \rfloor$ in the first $n-1$ partitions and $\lfloor c/n \rfloor + c \mod n$ in the last partition, where c is the number of channels in the input stream and n is the number of partitions to create. `N n` is an optional parameter denoting the number of partitions to create (default is 2). `N dim` is a further optional parameter denoting the dimension along which to split the input.

Get. The `Get` layer class is to be used in conjunction with a `Split` layer and a parallelization operator. Its purpose is to select the partition to be used in the subsequent layer instances. The layer parameter `N n` denotes the partition to select. CNNArc

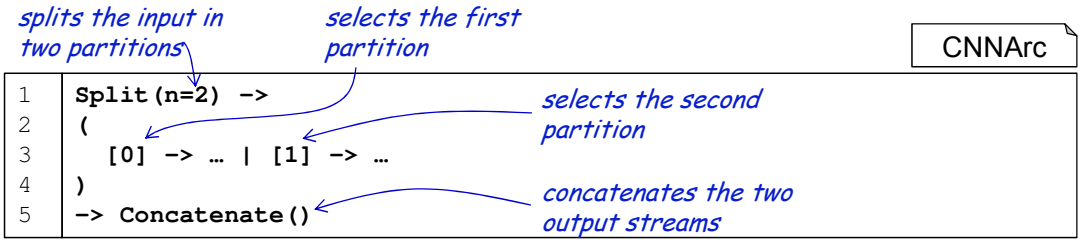


Figure 4.12: A split-parallelization-get pattern example.

offers a dedicated syntax which can be used instead of the usual anonymous instantiation/usage syntax. Thereby, the developer can refer to a particular partition by writing its index into square brackets, i.e. `[index]`. An example of a split-parallelization-get pattern instance is given in Figure 4.12.

The CNNArc layer library includes more layer classes, which can be useful for defining neural networks, e.g. `ImgResize`, `Dot`, `Repeat`, `Squeeze`, `ReduceSum`, `ExpandDims`, `BroadcastMultiply`, `SwapAxes`, `BroadcastAdd`, and `Reshape`. Listing all of them is out of scope. A more complete overview can be found in the code documentation of the reference implementation as mentioned above.

4.7.3 Code Reuse in CNNArc

Deep neural networks often exhibit highly repetitive structures. Therefore, easy-to-use reuse mechanisms are crucial in a neural network DSL. While the concept of layers provides a reuse mechanism for neurons making deep networks controllable in the first place, in this section we are going to introduce layer level reuse mechanisms of CNNArc, further facilitating network design and validation.

Layer Stacking and Structural Parameters. In deep neural networks we often want to create sequences of equal or similar layer instances. To allow concise modeling of such layer stacks, we introduce the concept of structural arguments. Structural arguments can be passed to any layer in addition to normal parameters. MontiAnna supports three different structural arguments: the two data flow operators `->` and `|` as well as the question mark `?`. The former two operators can be assigned an integer expression larger than one (the expression is evaluated at training time and cannot be changed afterwards). Doing so is equivalent to chaining the respective layer that many times. Figure 4.13 depicts two equivalent subnetworks, the first one being defined using a structural operator. The way of chaining layer instances using structural arguments is not only convenient, but also easy to use since the developer can associate the syntax with the dataflow operators.

The third structural argument is used to make layer instances optional. In contrast to the first two structural arguments, it takes a Boolean expression. Whenever used, a

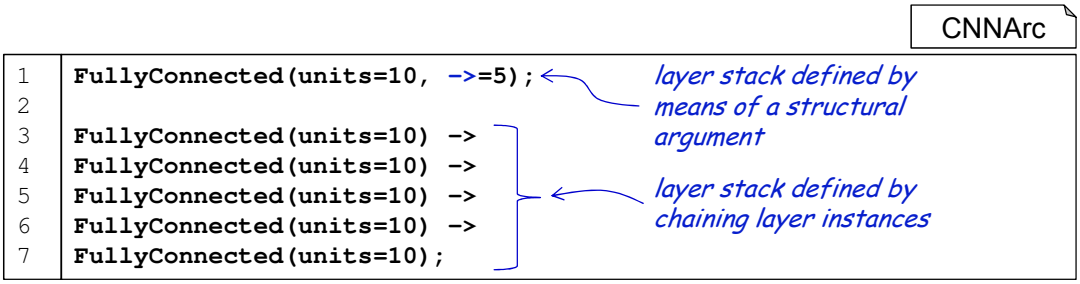


Figure 4.13: Two alternative definitions of the same layer stack consisting of five successive fully connected layers having 10 neurons each. The first definition makes use of a structural parameter, while the second one does not.

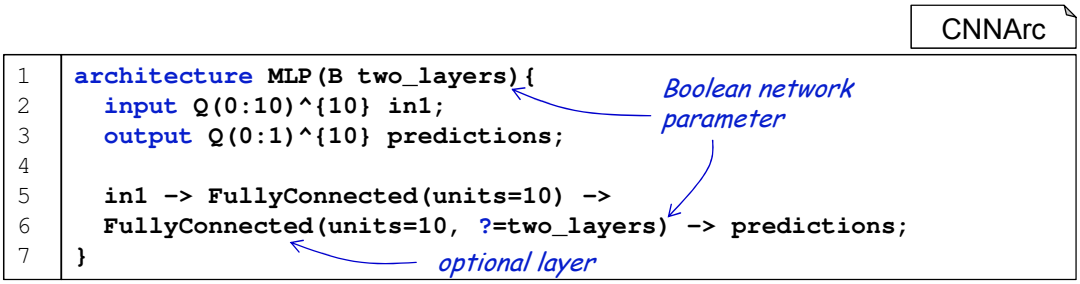


Figure 4.14: The MLP network modeled in this listing can have either one or two hidden layers depending on the value of `two_layers`.

layer instance is only created and added to the layer graph if this structural argument is evaluated to true. This feature is particularly helpful to model variability in neural networks and automate model selection, e.g. by binding the question mark argument to a model parameter as is shown in Figure 4.14.

Argument Sequences. While structural parameters as introduced above are a handy way to instantiate and connect many layers with a small number of lines of code, they often turn out to be inflexible as they require each layer instance to take the exact same set of arguments. We can overcome this problem by introducing argument sequences. Instead of providing a single expression for a layer argument, we can chain the arguments using the usual dataflow operators -> and | to instantiate a layer class multiple times using different sets of arguments. This is illustrated in L.1 of Figure 4.15. The corresponding layer stack modeled without argument sequences is given in L.3-7 below. If the layer class expects multiple arguments, all argument lists must have the same length and the same operator sequence. Alternatively, we can pass single-valued arguments alongside argument lists. In this case the single-valued argument will be replicated for

CNNArc

```
1 FullyConnected(units=[10->20->25->10|5]);
2
3 FullyConnected(units=10) ->
4 FullyConnected(units=20) ->
5 FullyConnected(units=25) ->
6 FullyConnected(units=10) |
7 FullyConnected(units=5);
```

layer stack defined using an argument sequence

layer stack defined by chaining layer instances

Figure 4.15: The definition of five fully connected layers with different numbers of units with and without an argument sequence.

each instance of the layer stack.

Layer Groups. Often combinations of particular layers reoccur in a deep neural network or even across different architectures. Therefore, defining custom reusable layer groups or layer subgraphs from existing layer classes without having to implement their behavior or the backward path is an indispensable feature of our language. Such a reusable layer graph can be defined using the `def` keyword in a CNNArc model followed by a name, a list of parameters, and a body delimited by curly brackets.

The body contains a layer graph definition using the standard syntax as used to define the main layer DAG. The chosen name can then be used as a layer class name to create new layer instances. Since the parameters of the newly defined layer class are passed to existing layers, it is not necessary to specify parameter types. They are inferred automatically from the context. If the type is ambiguous, i.e. a parameter is used in different, incompatible contexts, the compiler will throw a type error. Similarly to predefined layers, default values can be assigned to custom layer parameters.

A layer defined in a `def` block must have exactly one input and one output layer. In contrast to a full architecture, the input and output layers do not have to be declared as such explicitly. Other models can import the architecture containing the layer definition and use it without instantiating the enclosing network.

The combined usage of the three code reuse techniques introduced in this section is illustrated in Figure 4.16 modeling the ResNet152 neural network [HZRS16a]. Having only 31 lines of code, the model is very short compared to ResNet152 definitions in other languages, which may consist of up to hundreds or even thousands lines of code. However, the usage of too much syntactic salt might render the model difficult to read. A slightly longer but probably better readable version omitting the use of argument sequences is therefore given in Figure 4.17. A modeler needs to weigh up readability against compactness. The decision which modeling variant to use might depend on personal preferences. Hence, it is important that the language provides a choice.

CNNArc

```
architecture ResNet152(N1 channels=3, N1 height=224,
                                        N1 width=224, N1 classes=1000){
  def input Z(0:255)^{channels, height, width} image;
  def output Q(0:1)^{classes} predictions;

  image ->
  conv(kernel=7, channels=64, stride=2) ->
  Pooling(pool_type="max", kernel=(3,3), stride=(2,2)) ->
  resLayer(channels=[64->64->128->128->256->256->512->512],
           stride=[1->1->2->1->2->1->2->1],
           skipConv=[true->false->true->false->true->false->true->false],
           ->=[1->2->1->7->1->35->1->2]) ->
  GlobalPooling(pool_type="avg") ->
  FullyConnected(units=classes) ->
  Softmax() ->
  predictions;

  def conv(channels, kernel=1, stride=1, act=true){
    Convolution(kernel=(kernel,kernel),channels=channels,
                stride=(stride,stride)) ->
    BatchNorm() ->
    Relu(?=act);}

  def resLayer(channels, stride=1, skipConv=false){
    conv(kernel=[1->3->1|1], channels=
         [channels->channels->4*channels|4*channels],
          stride=[stride->1->1|stride], act=[true->true->false|false],
          ?=[true->true->true|skipConv]) ->
    Add() ->
    Relu();
}}
```

Figure 4.16: The ResNet152 modeled in CNNArc using structural parameters, argument sequences, and custom layer definitions.

CNNArc

```
1  architecture ResNet152(N1 channels=3, N1 height=224, N1 width=224,
2                                                 N1 classes=1000){
3    def input Z(0:255)^{channels, height, width} image;
4    def output Q(0:1)^{classes} predictions;
5
6    image ->
7    conv(kernel=7, channels=64, stride=2) ->
8    Pooling(pool_type="max", kernel=(3,3), stride=(2,2)) ->
9    resLayer(channels=64, addSkipConv=true) ->
10   resLayer(channels=64, ->=2) ->
11   resLayer(channels=128, stride=2, addSkipConv=true) ->
12   resLayer(channels=128, ->=7) ->
13   resLayer(channels=256, stride=2, addSkipConv=true) ->
14   resLayer(channels=256, ->=35) ->
15   resLayer(channels=512, stride=2, addSkipConv=true) ->
16   resLayer(channels=512, ->=2) ->
17   GlobalPooling(pool_type="avg") ->
18   FullyConnected(units=classes) ->
19   Softmax() ->
20   predictions;
21
22   def conv(channels, kernel=1, stride=1, act=true){
23     Convolution(kernel=(kernel,kernel),channels=channels,
24                                             stride=(stride,stride)) ->
25     BatchNorm() ->
26     Relu(?=act);}
27   def resLayer(channels, stride=1, addSkipConv=false){
28     (
29       conv(kernel=1, channels=channels, stride=stride) ->
30       conv(kernel=3, channels=channels) ->
31       conv(kernel=1, channels=4*channels, act=false)
32       |
33       conv(channels=4*channels, stride=stride, act=false, ? = addSkipConv)
34     ) ->
35     Add() ->
36     Relu();
37 }}
```

Figure 4.17: ResNet152 model equivalent to the one in Figure 4.16 but defined without argument sequences and therefore a better readability.

EMAM

```
1  component FullyConnected<N previousDim, N units>
2                          (Q^{previousDims, units} W, Q^{units} b) {
3    ports in  Q^{previousDim} layerInput,
4          out Q^{units} activations;
5
6    implementation Math {
7      activations = W'*layerInput + b;
8    }
9  }
```

layer implementation as Math code

trainable layer parameters

```
1  component ReLU<N previousDim> {
2    ports in  Q^{previousDim} layerInput,
3          out Q^{previousDim} activations;
4
5    implementation Math {
6      activations = max(0,layerInput);
7    }
8  }
```

elemetwise application of the max function

Figure 4.18: An example showing how MontiMath could be used to implement custom layers for MontiAnna networks.

To define completely new layer classes that cannot be built from existing ones in a `def` block, we need to provide templates for the backends we wish to support and register the layer in the corresponding generators. Often the desired layers already have corresponding counterparts in the target framework. This is the case for a large part of the layer classes presented above. In some cases the existing target framework constructs need to be adapted only slightly to achieve the desired layer functionality, e.g. by modifying the layer parameters or adding a preprocessing step. If, however, a new layer cannot be built from available target framework constructs, it is desirable to model the functionality using an appropriate DSL instead of writing templates for Python code generation. In future work, MontiMath could be evaluated as a layer implementation language to enable layer extensions within the language family. MontiMath seems to be well suited for this purpose due to its matrix-oriented approach, since neuron layers can often be modeled efficiently as matrix operations. Figure 4.18 shows exemplarily how the `FullyConnected` and the `ReLU` layer could possibly be implemented as EMAM components. The generic component parameter `N previousDim` is introduced to hold the number of dimensions of the previous layer's output. In the case of the `FullyConnected` layer, the additional generic parameter `N units` defines the number of neurons in the layer. The component parameters are used to specify the trainable network layer parameters, in this case the weight matrix `Q^{previousDims, units} W` and the bias vector `Q^{units} b`. The actual computations are given as MontiMath

code in the implementation parts of the components. The matrix operations provided by MontiMath enable us to implement both layers in just one line. To make this layer development and integration approach functional, we need to enable automatic differentiation (AD) on MontiMath code or on the generated code so that the backward path can be derived from the forward path implementation automatically. AD is a family of techniques for efficiently and accurately evaluating derivatives of numeric functions expressed as computer programs [BPRS18]. It is widely used in machine learning frameworks to compute the backward path for training [PGC+17]. Alternatively, we can require the layer developer to provide a backward path explicitly, which however is not a convenient solution.

4.8 Modeling Recurrent Neural Networks

4.8.1 Basic Concepts

In many applications, the data a network works on exhibits a sequential nature. In such cases, looking at isolated snapshots of an input signal independently does not lead to the desired results. Interdependencies between the samples of a signal need to be modeled, as well. For instance, the meaning of a single word in a sentence can heavily depend on the context. What is more, input and output sequences can be of arbitrary lengths. Hence, processing each word independently, e.g. to translate a sentence, will likely yield imprecise results.

As discussed in Section 4.1.4, specific neuron models such as LSTM and GRU cells have been developed for neural sequence processing. These neuron models cannot be modeled adequately by the means presented in the previous sections only. Therefore, we introduce the new layer classes representing RNN-based neuron models as well as further layer classes which are essential for RNNs.

RNN. The RNN layer class represents the basic RNN neuron model as introduced in Section 4.1.4. The activation function is set to `tanh` by default. In contrast to the `FullyConnected` layer, the modeler does not have to append a non-linearity explicitly as a stand-alone layer. This goes in line with other deep learning frameworks such as Keras and MXNet Gluon and makes the layer class consistent with the other RNN-based neuron models like LSTMs and GRU, where several non-linearities are applied inside the model and hence, cannot be modeled as a subsequent layer. Furthermore, extracting the non-linearity of the RNN class into an explicit layer would lead to an unnecessarily complex neuron interface.

- **Parameters:**

  `N(1,∞) units` denotes the number of neurons in the layer instance.

`N(1,∞) layers` denotes the number of layers to instantiate. The default is 1.

`B bidirectional` makes the RNN bidirectional if set to true. It is set to false by default.

- **Function:** performs the RNN function given in Equation (4.17) for each layer with the input being the output of the previous layer. The weights w need to be learned.
- **Connection pattern:** consumes and processes an input sequence stepwise.

LSTM. The LSTM layer class represents the LSTM model as introduced in Section 4.1.4. The interface corresponds to the one of the RNN class.

GRU. The GRU layer class represents the GRU model as introduced in Section 4.1.4. The interface corresponds to the one of the RNN class.

To feed words or sentences into neurons, we need to find an adequate numerical representation of words first. The simplest way is to map each word of a vocabulary to a single integer. However, mathematical operations such as a distance cannot be defined on such a representation in a meaningful way. This can be overcome by using a high-dimensional embedding. Layer classes creating a high-dimensional encoding include the `OneHot` and the `Embedding` layers and are discussed in Appendix B.1.

4.8.2 Modeling an Encoder-Decoder Network

We are now going to introduce the syntactic elements required in RNN modeling based on the encoder-decoder architecture. Encoder-decoder architectures are sequence-to-sequence models where an input sequence is mapped to a target sequence. The example we are going to use deals with automated machine translation, where an input sentence is mapped to an output sentence in the target language [CvMBB14]. The source and target sentences can be of arbitrary length and the length of the target sentence may differ from the length of the source sentence. This problem is tackled by mapping the input sentence to some intermediate representation. This mapping is performed by the *encoder* RNN, e.g. consisting of LSTM or GRU cells. The output of the last step of this RNN is input into the *decoder* RNN whose task is to find the best output sequence corresponding to this intermediate, fixed-size representation word by word. This happens in the process of *unrolling*.

A graphical representation of the *unrolled* encoder-decoder network is provided in Figure 4.19. Each replication is depicted as a rectangle with the corresponding discrete timestep index included for the decoder RNN. <SOS> and <EOS> denote the start and end of sequence symbols, respectively, used to model the beginning end the end of

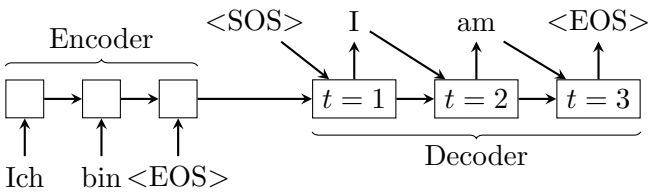


Figure 4.19: RNN encoder-decoder model example [KNP+19].

CNNArc

```
1  architecture RNNencdec<N1 max_length=50, N1 vocab_size=30000,
2                                          N1 hidden_size=1000> {
3      def input N(0:vocab_size-1)^{max_length} source;
4      def output N(0:vocab_size-1)^{max_length} target;
5
6      layer LSTM(units=hidden_size) encoder;
7
8      source ->
9      Embedding(output_dim=hidden_size) ->
10     encoder;
11
12     layer LSTM(units=hidden_size) decoder;
13
14     1 -> target[0];
15     encoder.state -> decoder.state;
16
17     timed<t> GreedySearch(size=max_length) {
18         target[t-1] ->
19         Embedding(output_dim=hidden_size)
20         decoder ->
21         FullyConnected(units=vocab_size) ->
22         Softmax() ->
23         Argmax(dim=0) ->
24         target[t]
25     };
26 }
```

Figure 4.20: MontiAnna model of the RNN encoder-decoder network [KNP+19].

a sentence. In our example the encoder-decoder network translates the German input sentence "Ich bin" to the English equivalent sentence "I am".

The architecture definition of the discussed encoder-decoder machine translation model is given in Figure 4.20. The network input is a vector where each element represents the word at the respective position in the sentence as a natural number. The range of the allowed values is bounded by the vocabulary size, i.e. `(0:vocab_size - 1)`. The vector dimensionality represents the maximum length of a sentence, but the actual sentence can be shorter. The output of the network has a similar structure. Often, the actual length of the output sentence differs from the length of the input sentence. Restrictions on maximum sentence length are imposed by the EMA type system, which prohibits dynamically sized data types in order to guarantee upper bounds on resource usage. This limitation can be overcome easily by extending the EMA type system by dynamic collection types.

The internals of the network do not work on the sentence representation described above. The problem with this representation is that the distance between two different words depends on the chosen concrete encoding, e.g. if the words `house, water, book` are encoded by the numbers `1, 2, 3`, respectively, the similarity of house and water is higher than the similarity of house and book. To eliminate this problem, we use the `Embedding` layer type, cf. Appendix B.1, mapping each word to a fixed-size vector. Hence, the sentence vector is transformed to a sentence matrix, where one dimension represents the word features and the second one the position of the word in the sentence. The `Embedding` layer is used twice in our encoder-decoder model: first it maps the input layer `source` to the input of the encoder in L.9 of Figure 4.20; furthermore, it maps the word at the `target` port of the previous timestep in the same way to be reused by the decoder, cf. L.18-20 of Figure 4.20.

While at first sight the model resembles feedforward architecture models, there are several peculiarities to pay attention to. While in feedforward networks, we relied on anonymous layer instantiation most of the time, this modeling principle is not sufficiently expressive for RNNs; in order to define temporal interdependencies in our network, we need to access a layer instance multiple times. In our encoder-decoder example, the encoder and decoder LSTMs are instantiated as named layers in L.6 and L.12 of Figure 4.20, respectively.

So far we have been working with implicit layer port access. Each layer class, including the basic non-recurrent model, has at least the two ports `input` and `output`. As our aim is to keep the language as simple and pragmatic as possible, by default we refer to a layer's output signal if it is the left operand and to its input if it is the right operand of a connect operator. Recall that LSTM and GRU cells differ from standard feedforward neurons by an inner state or a memory maintained and adapted throughout the course of a recognition task.

When dealing with such cells, we will need to access specific properties of a neuron explicitly. Hence, a more complex neuron model leads to a more complex interface which

cannot be addressed implicitly any more. In the style of object oriented programming, we introduce the access operator "`.`" to retrieve a specific property of a neuron layer and refer to this access mode as *explicit access*. Thereby, `layer1 -> layer2` is semantically equivalent to `layer1.output -> layer2.input`. Explicit access is more important for working with the inner state of a neuron cell such as an LSTM. While, so far, we have introduced the properties `input` and `output` for feedforward neurons and, additionally, a `state` property for RNNs, other neuron models can, of course, provide even more complex interfaces.

The encoder-decoder network consists of multiple subnetwork expressions. The first of these subnetworks is defined in L.8-10 of Figure 4.20 and represents the encoder. The aim of the simple subnetwork in L.15 of Figure 4.20 is to transfer the result of the encoder to the decoder. Note that we use explicit access here to forward the inner state of the encoder to the inner state of the decoder by writing `encoder.state -> decoder.state`.

Furthermore, the decoder is executed in several steps to produce an output sequence until an `<EOS>` symbol marking the end of the sequence is generated or until the maximum output sequence length is reached. In each step the current output needs to be provided as input for the following decoding step. The modeling language hence needs to provide a syntax to access the neuron's input, output, and state from different points in time. In particular, we need to be able to specify *relative* time dependencies, rather than access absolute points in time, e.g. to specify that the input of a neuron is the output of another neuron delayed by a fixed number of timesteps. For this reason we introduce the concept of timed graphs, which is particularly important for time series processing. In a timed graph a layer instance can be replicated for each timestep. In the layer graph, timed replications can be thought of as independent layer nodes. However, if the layer has trainable weights, they are shared across all timed replications. Connections can be created between nodes related to different points in time. To declare a time variable for a subnetwork, we use the `timed` modifier followed by the name of the time variable in angle brackets, e.g. `<t>` in L.17 of Figure 4.20. The subnetwork which is executed stepwise and is ought to use the time variable is enclosed in a block marked by curly brackets following the `timed` modifier. Restricting the time variable to a dedicated scope is a conscious decision: first, we hinder accidental usage of the time variable outside the timed subnetwork; second, the timed subnetwork is highlighted appropriately, making the architecture well readable (graphical representations of encoder-decoder architectures also separate the decoder from the encoder clearly, e.g. in the Transformer architecture paper [VSP+17]).

The time variable `t` does not take an actual value, but is rather used to describe *relative* temporal interdependencies. Inputs, outputs, and other layer properties can now be accessed with a temporal argument in square brackets, cf. L.18 and L.24 of Figure 4.20, which we refer to as timed access. In L.18 of Figure 4.20, we access the network output of the *last* timestep, denoted by `target[t-1]` to be input into the

`Embedding` layer. The result of the subnetwork is fed into the *current* version of the `target` layer, denoted by `target[t]`. Hence, the subnetwork inside the timed block seems to form a cycle where the input and the output are both represented by the `target` layer with a delay of one timestep. To keep the model clean and pragmatic, we interpret the absence of a temporal argument as a reference to the current timestep, i.e. `target` is equivalent to `target[t]`. The temporal argument enables an abstract and concise modeling of temporal interdependencies.

When an RNN is unrolled during prediction, the goal is to find an output sequence with the highest probability. This is similar to non-recurrent classification tasks, where we end up with a `Softmax` layer representing the probabilities of each possible label. Applying an `Argmax` to the result, we finally obtain the best label. In recurrent tasks however, taking the output with the highest probability in each step does not necessarily lead to the sequence with the highest probability at the end. Theoretically, we need to unroll the network for *all* possible output sequences to be able to identify the best one at the end. This, however, is impossible, as the length of an output sequence, e.g. a translated sentence, is unbounded. Limiting the output sequence length to a finite integer n still does not allow for an exhaustive search due to the vast search space. A vocabulary of $|V| = 1000 \approx 2^{10}$ words would require testing 2^{10n} combinations, which, even for short sentences of a maximum length of $n = 10$, would yield $2^{100} \approx 10^{30}$ combinations. For this reason, unrolling RNNs requires search strategies drastically reducing the number of combinations. This is often done using the beam search algorithm in RNNs. In each step, it takes only the w best sequence candidates, where w is a hyperparameter referred to as the beam width. In the subsequent step it generates $w|V|$ new candidates by appending all possible words to each sequence and again chooses the top w ones to be passed to the next step.

The search algorithm is specified after the `timed` modifier, cf. L.17 in Figure 4.20. For instance, the `BeamSearch` strategy takes the parameters `N1 size` bounding the maximum sentence length and `N1 width` being the beam width. The `GreedySearch` is essentially a `BeamSearch` with the `width` parameter set to 1. It is provided as a distinct algorithm for the sake of convenience. The search runs from 1 to `size`.

The question arises, how the network behaves if there is no previous timestep to read from, i.e. at `t=1`. `target[0]` is set in the untimed part of the network, cf. L.14 of Figure 4.20. This is solved using an *invariant network* allowing us to write a constant to a particular property. We initialize `target[0]` with the value of 1, which, in our case, represents the start of sentence symbol `<SOS>`. Hence, the value of `target[0]` is not computed explicitly using the timed subnetwork. Similarly, the initial state of the `decoder` is set from outside the timed block in L.15 of Figure 4.20. Note that although we have introduced a loop in the layer graph by making the `target` layer the input *and* the output layer of the graph expression in L.18-24 of Figure 4.20, the *unrolled* graph still remains a DAG. Hence, we maintain our constraint that the layer graph must be a DAG, cf. Section 4.7.2.

In the last few years, RNN-based models have been steadily replaced by attention-based large memory networks, e.g. the transformer and its descendants such as BERT and GPT [VSP+17, DCLT18, BMR+20]. Modeling such networks is similar to the approach described in this section and requires the same language elements. The required layer classes are provided by MontiAnna and include `DotProductSelfAttention` and large memory layers such as `EpisodicMemory` and `LargeMemory`, which are based on [dMdRKY19, LSR+19].

4.9 Modeling Training

The previous sections have focused on the first modeling concern of machine learning, namely architectural modeling. In this section we shift the focus to the *training* model, which is another essential part of a machine learning system. The aim of the training is to find an appropriate set of parameters or, in the context of neural networks, neuron weights before the module is used. As was described in the beginning of this chapter, the parameter search in deep learning is usually an optimization problem, which can be approximated using numerical iterative algorithms. Our goal is to provide popular training algorithms covering a large part of all deep learning programs as black box libraries. The developer should not need to reimplement these well understood algorithms. The only thing which needs to be done is to specify the training procedure by choosing the algorithms to use and setting their respective hyperparameters, cf. Section 4.1.3 for more details on network training algorithms.

To comply with the separation of concerns requirement (RL4), a dedicated training model is used to set up the training. This model is only expected to choose the algorithms to be used and set their hyperparameters, hence a simple configuration language is sufficient to describe the training. The training description language CNNTrain developed for MontiAnna resembles the JSON format and other similar data description languages. However, there are some syntactic differences and semantic constraints.

As most MontiCore models, a CNNTrain model has a package it belongs to, which is defined using the `package` keyword followed by the fully qualified package name. The header starts with the keyword `configuration` followed by the model name. The body is enclosed by curly brackets. In the body, the training parameters are set in an arbitrary order. The name of a parameter to be set is followed by a colon and the desired value.

Our training language only allows a predefined set of parameters and forbids combinations of mutually exclusive parameters. The correctness of a model is verified against the CNNTrain schema. Since MontiAnna is the only context in which the language is used, no dedicated schema language was developed. Instead, the training schema is implemented by the means of context conditions. However, a schema language should be added in the future to help making the allowed combinations of parameters and their

CNNTrain

```
1  package de.monticore.dl;
2
3  configuration Alexnet {
4                  num_epoch:1,
5                  batch_size:64,
6                  normalize:true,
7                  optimizer:adam{
8                    learning_rate:0.01,
9                    learning_rate_decay:0.8,
10                   weight_decay:0.01 },
11                 seed:12345}
```

Figure 4.21: A simple training model example written in CNNTrain.

types explicit and easier to maintain and to extend.

The training language reuses the EMA type system. Each allowed parameter is linked with a corresponding type which is checked by the schema, as well. The syntax for scalar literals and matrices is the same as in MontiMath. In addition to that, CNNTrain uses the type `String` for string-valued parameters and nested types similar to EMA structs for complex properties. Similar to many other languages a string is enclosed in quotation marks. A major difference to JSON is that a nested parameter is composed of an enum-like value followed by a set of subparameters listed in a block enclosed in curly brackets. For instance, if we want to set the optimization algorithm, we set the respective parameter by choosing one of the available optimizers, e.g. `optimizer: adam`. However, this specific optimizer requires its specific set of hyperparameters. These hyperparameters can be set in the nested block (or are set to their default values). This makes `adam` both a value (of type enum), since it represents a concrete algorithm, and a type, since it defines its own hyperparameters (other optimizers, e.g. `sgd` or `rmsprop`, might require other parameters). This is similar to the notion of clabjects, which are entities combining properties of classes and objects in object-oriented multi-level modeling [HSGP05].

In the following we provide a list of important hyperparameters supported by CNNTrain. Each parameter is preceded by its type and can have a default value assigned to it. We are not going to dive into the details of each parameter as these are common in the machine learning domain and do not contribute to the understanding of the modeling methodology.

N1 num_epoch = 1. The number of epochs to be trained.

N1 batch_size = 1. The number of example used in each training step.

B normalize = false. If this is set to true, the data will be normalized before training.

enum loss. This parameter is used to set the loss function for the training. The available loss function options include: $\mathcal{L}_1$ (`l1`), $\mathcal{L}_2$ (`l2`), cross-entropy (`cross_entropy`), softmax cross-entropy (`softmax_cross_entropy`), endpoint error (EPE) for optical flow [ZLNH17] (`epe`), Sigmoid cross-entropy (`sigmoid_cross_entropy`), and the Huber loss (`huber`).

nested optimizer. Sets the optimizer to be used. Supported optimizers are SGD (`sgd`), Adam [KB14] (`adam`) and its variant decoupling weight decay [LH17] (`adamw`), RmsProp (`rmsprop`), Adagrad [DHS11] (`adagrad`), Nesterov Accelerated Gradient (NAG) [Nes83] (`nag`), and AdaDelta [Zei12] (`adadelta`) . Each optimizer can be parameterized with general loss parameters including, but not limited to: learning rate (`Q learning_rate`), learning rate decay policy (`enum learning_rate_policy` supporting the decay functions `fixed`, `step`, `exp`, `inv`, `poly`, `sigmoid`), `Q learning_rate_decay`, etc. Furthermore, specialized parameters can be available depending on the concrete loss, e.g. `Q momentum` for SGD, `Q beta1` and `Q beta2` for Adam, `Q epsilon` for Adam, AdaGrad, RmsProp, and AdaDelta.

enum context. An enum defining whether the GPU or central processing unit (CPU) is used for training (allowed enum values are `gpu` and `cpu`). GPU training requires a CUDA compatible graphics card and a backend, e.g. MXNet, compiled for CUDA (MontiAnna itself does not have to be compiled for CUDA usage in a particular way).

nested eval_metric. Sets the evaluation metric for the trained network. Allowed values are listed below (cf. Appendix B.2 for more alternatives):

`accuracy`: percentage of the correctly classified examples

`bleu`: bilingual evaluation understudy (BLEU) is a metric for the domain of machine translation of natural languages [PRWZ02]. The subparameter `N1^n exclude` can be used to exclude symbols from the evaluation, e.g. special symbols such as start and end of sentence. The dimensionality `n` does not have to be specified explicitly, but is deduced from the given vector.

Z seed. A seed controlling stochastic algorithms can be used to ensure reproducibility.

A CNNTrain model for the training of an `Alexnet` architecture is given in Figure 4.21. It sets the number of training epochs to 1 and the batch size to 64. Normalization is activated in L.6. The optimizer algorithm is set to Adam [KB14], which in turn requires its own set of hyperparameters: the learning rate, its decay, and the weight decay. Other optimizers might require a completely different or an overlapping set of hyperparameters.

Note that the training model is underspecified and therefore exhibits a non-deterministic semantics. For instance, the training model abstains from defining the concrete sequence

of the training examples to be drawn during a stochastic optimization procedure such as SGD. To ensure reproducibility, a seed controlling random variables can be set, cf. L.11.

The training language as it is presented here is fully declarative and uses aliases to describe the desired functionality, e.g. we use the alias `l1` to reference the $\mathcal{L}_1$ loss function defined as $\mathcal{L}_1 = \sum_i |y_i - \hat{y}_i|$, where the sum is over the given batch of examples, y_i is the true label of example i and $\hat{y}_i$ is the corresponding prediction. When more loss functions beyond the provided set are required, new aliases need to be implemented in the backend. To enhance extensibility and facilitate experimentation with new functions and function variants, future work includes the integration of expressions into the training language and/or the usage of external MontiMath scripts.

For instance, we could model the $\mathcal{L}_1$ loss as `loss = sum(abs(label, prediction))`, where `sum(.)` is over the training batch and hence, adapted to the domain-specific context; `label` is part of the context as well and refers to the true label of a training example and `prediction` references an output layer of the network architecture. If the implementation of the function is encapsulated in a separate EMA component which is referenced from the training model, e.g. `loss = de.monticore.losses.L1`, the component needs to exhibit a to be defined interface to receive the context.

In addition to the loss function, different other aspects such as the optimizer, decay functions, etc. can be modeled in this way.

4.9.1 The Composed Model

The semantics of a neural architecture model is a parameterizable family of functions $f_w(x)$ that can be learned. Composing the architectural model with the training and the dataset model yields a concrete function binding the parameters w.

Although the three modeling subdomains are orthogonal, only together they represent a neural processing system. Therefore, we need to merge them into a single composed model before we can generate the actual application code. The composition is realized on the AST and the symbol table using MontiCore's language aggregation mechanism, i.e. there is no need to create a concrete syntax for this purpose. All models are stored in independent files, but are linked to each other after parsing. The resulting *composed model* serves as a basis for complex inter-model context conditions and code generation. For instance, we need to check whether each neural network component is assigned a valid training configuration.

If not specified otherwise, we assume that a network model uses a training model of the same fully qualified name. This of course requires that a dedicated training model is defined for each network which is contradictory to reuse principles in software engineering. For this reason, the developer can explicitly reference a custom training model for a neural network in a tag model as will be shown in Figure 4.24. The realization details will be discussed in Section 4.10.2.

The composed model undergoes context conditions checking whether the training

model is compatible with the network architecture. For instance, the loss function specified in the training model must be applicable to the network output. A concrete example is the cross-entropy loss. The cross-entropy loss is a measure for the distance of probability distributions. It is often used in classification tasks where the network output is a probability for each possible class. To use the cross-entropy loss, the network hence must output a vector of probabilities, i.e. values between 0 and 1 summing up to 1. This must be ensured by using a `Softmax` as the last layer in the network architecture. Otherwise, a context condition will raise an error.

Another example is the usage of the F1 evaluation metric. Since it is only applicable to binary classification models, we need to check that the output layer of the network is one-dimensional. Hence, inter-model context conditions ensure that the employed training algorithm and the underlying network architecture are compatible on a semantic level. Once the composed model is assembled and validated, it is passed to the code generator.

4.10 EmbeddedMontiArcDL

Our goal is a holistic model-driven engineering (MDE) approach allowing the developer to tackle the heterogeneous problems arising in CPS applications using appropriate paradigms. In particular, we want to be able to design the software architecture including its typed dataflows using the EmbeddedMontiArc ADL. Furthermore, we should be able to model the behavior of an atomic component in an appropriate way: while it makes sense to model signal processing components such as filters and controllers using MontiMath, recognition and detection tasks can often be solved efficiently using a neural network.

Therefore, we are going to extend the EMAM language family to support MontiAnna as an alternative behavior modeling language. In the following, we will refer to the newly created language family as EmbeddedMontiArc Deep Learning (EMADL). EMADL is constructed using the language composition principles *language extension*, *language aggregation*, and *language embedding* of the language workbench MontiCore [HR17, MSN17]. The MontiCore grammar of the EMADL language is given in the appendix in Listing B.13. A graphical overview of the EMADL language family and the respective generators is depicted in Figure 4.22. The end user only works with the composed EMADL language and the corresponding compiler EMADL2CPP. The latter does not generate code itself. Instead, it analyses the EMADL model and delegates parts of it to the responsible subgenerators, i.e. EMA2CPP to generate architectural code and connectors, MontiMath2CPP for MontiMath statements and an implementation of MontiAnna2X for neural network architectures.

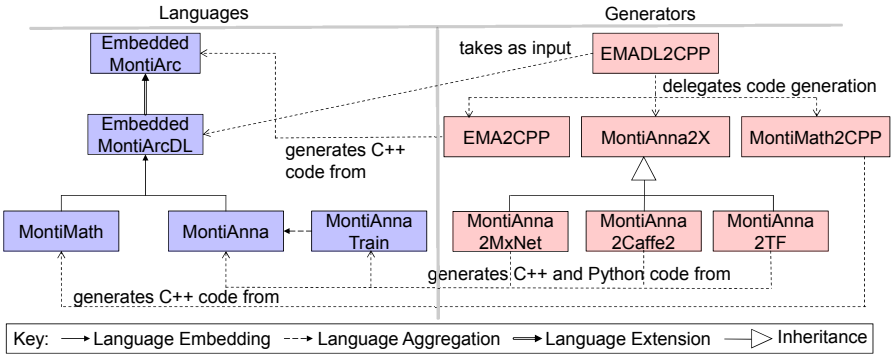


Figure 4.22: An overview of the EmbeddedMontiArcDL modeling language family: languages and code generators are depicted as purple and pink boxes, respectively [KPRS19].

4.10.1 CNNArc as Implementation Language for EmbeddedMontiArc Components

To implement the behavior of an EMADL component as a neural network, the developer has to set the component's implementation language to `CNN` instead of `Math`, cf. Figure 4.23. The corresponding MontiCore grammar is given in Listing B.13. A component's behavior cannot be modeled using both MontiMath and MontiAnna concepts at the same time. This is a conscious design decision aiming to enable neural networks to be trained and reused as black box components in large systems and to leverage the separation of concerns principle.

In contrast to MontiMath, where a single artifact can contain the whole model, a single MontiAnna model consists of multiple artifacts, namely the neural network architecture, the training model, and the dataset model. We only embed the architectural code into the implementation block of a component as we consider it the most representative part of a neural network. The training model is stored in a separate artifact and can be related to the corresponding EMADL component by name, i.e. for a MontiAnna component the compiler will try to resolve a training model of the same fully qualified name. The dataset model is a separate tag model possibly containing information for multiple components, cf. Section 4.10.2.

The definition of a MontiAnna neural network model inside a neural implementation block omits an `architecture` header including the parameter list as well as the definition of input and output layers as given in Figure 4.7. Only the actual layer graph has to be defined in the implementation body. The neural network architecture obtains its parameters from the encapsulating EMADL component. Layer names undefined in the architecture are looked up in the component's port list.

From an abstract mathematical point of view, both EMA and MontiAnna are ma-

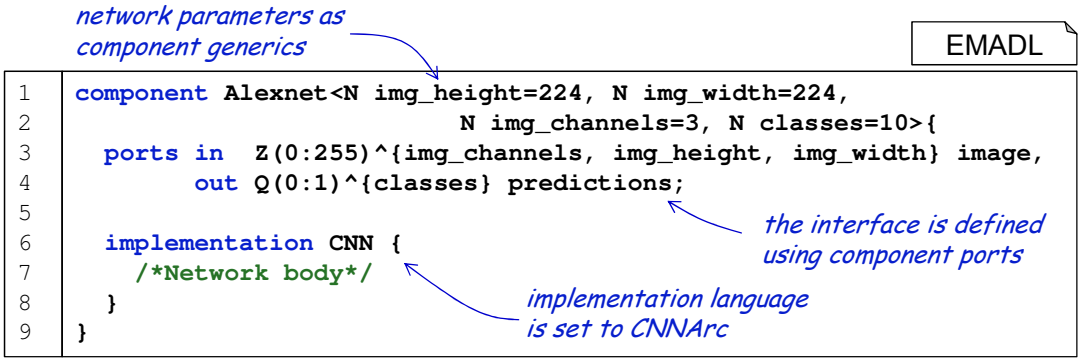


Figure 4.23: Definition of an EMADL component with a MontiAnna neural network implementation.

trix processing languages, which facilitates the integration in the design domain. An EMA matrix is mapped directly to a MontiAnna input layer of the corresponding shape. However, the integration of the two languages has consequences in the generator domain. While EMA uses Armadillo's types `Col<type>`, `Row<type>`, `Mat<type>`, `Cube<type>`, the C++ APIs of common deep learning frameworks expect a simple (multi-dimensional) array as input. Therefore, EMADL2CPP provides a translation library (`CNNTranslator.h`), which is used by the generated code at runtime to adapt the data types when entering and leaving CNNArc components.

4.10.2 Modeling the Dataset

To provide information about the dataset to be used, we employ the tagging mechanism of EMA [vW20] based on [GLRR15, Loo17]. It is designed to enrich symbols of a referenced model with additional information (or tags) conforming to a tag schema. For instance, the tagging approach can be used to add extra-functional properties to C&C models [MRRW16].

In the scope of the EMADL language family, we use tagging for two purposes: first, to capture concrete, implementation specific details mostly used by the generator and, second, to link model artifacts if not possible otherwise. The tag schema which we apply in the context of neural network components is given in Figure 4.24. It consists of a single tag type declared as `Training`. It has two obligatory and one optional entries. `path` is a string indicating the location of the training database to be used. `format` specifies the format of the database and hence, determines how data access is generated by EMADL2CPP for the respective training data. Our implementation currently supports HDF5 and LMDB. Furthermore, by setting the optional variable `training`, the tag model can override the fully qualified name of the training model to be resolved for individual components. This enables the reuse of a single training model

```
1  tagschema TrainingDataToEmadlTagSchema  {                    Tag Schema
2    tagtype Training for EMAComponentSymbol,
3                         EMAComponentInstanceSymbol {
4      path = ${path:String},
5      format = ${format:[HDF5 | LMDB],
6      (training = ${name:String})?}
7    };
8  }
```

Figure 4.24: Tag schema defining the training data and optionally a CNNTrain model reference for a deep learning component.

for multiple neural networks of an EMADL software architecture and across projects.

Note that two kinds of symbols can be enriched with a `Training` tag. If an `EMAComponentSymbol` is tagged, the tag data will apply to all instances of the corresponding component type. Consequently, sharing the same training configuration and training data, all instances of this component type will become identical. Hence, training is performed only once and the trained parameters are shared by all instances of the neural net. Moreover, if the output of a neural network depends only on the current input, only one flyweight instance needs to be created and can be reused for all component instances of the respective type. For now, this is the case for all architectures discussed above, including RNNs. In the scope of the EMA execution model, in each execution cycle an RNN component receives a full sequence as input and unrolls the network to obtain a full output sequence. Having limited the maximum length of the output to the dimensionality of the output port, we constrain not only the required memory consumption but also the network execution time per EMA cycle.

If, on the other hand, the `Training` tag is applied to an `EMAComponentInstanceSymbol`, the configuration is only applied to this particular instance. Other instances of the same component type remain untouched. Tags applied to component instances have a higher priority than tags applied to component types.

4.10.3 The MNISTCalculator Example

To illustrate the EMADL concepts in a system with multiple deep learning and MontiMath components, we are going to introduce the `MNISTCalculator`, a toy application for arithmetic operations on handwritten digits. The top level architecture including an exemplary dataflow is depicted in Figure 4.25, the textual EMADL model is given in Figure 4.26. The input port array `image` expects five 28×28 grayscale images. The first two pairs of digits constitute the two operands while the fifth (in the middle in Figure 4.25) represents the desired operator. The images are fed into individual detector components depicted in purple to highlight that they are implemented in MontiAnna,

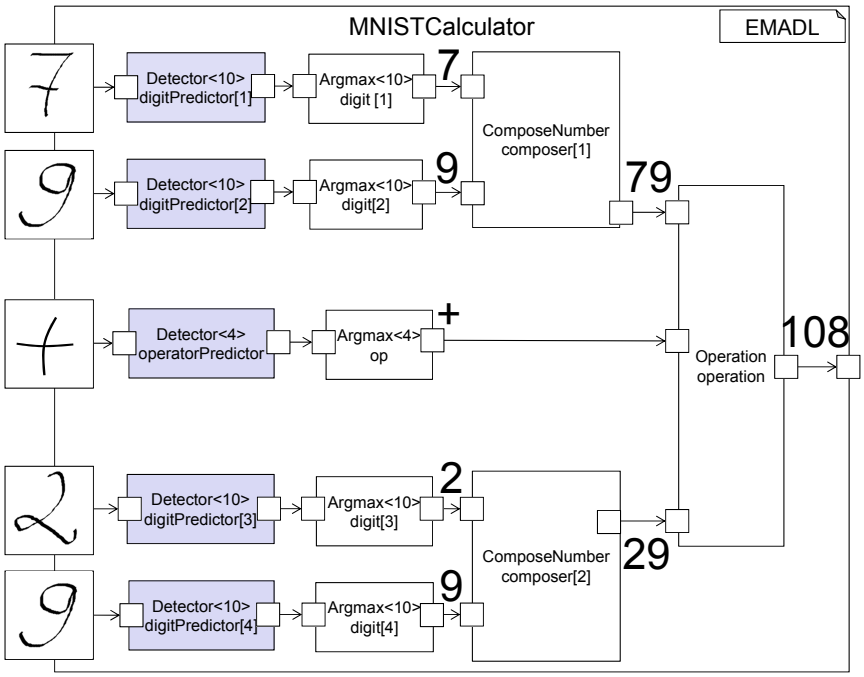


Figure 4.25: Component-and-connector architecture of an extended MNISTCalculator [KNP+19].

MNIST digits are 28x28 grayscale images

EMADL

```
1  component MNISTCalculator {
2    ports in  Z(0:255)^{1,28,28} image[5],
3          out Q(-9801:9801) result;
4
5    instance Detector<10> digitPredictor[4];
6    instance Detector<4> operatorPredictor;
7    instance ComposeNumber composer[2];
8    instance Operation operation;
9    instance Argmax<10> digit[4];
10   instance Argmax<4> op;
11
12   connect image[1:4] -> digitPredictor[:].data;
13   connect image[5] -> operatorPredictor.data;
14   connect digitPredictor[1:4].softmax -> digit[1:4].values;
15   connect operatorPredictor.softmax -> op.values;
16   connect digit[1:2].result -> composer[1].numbers[:];
17   connect digit[3:4].result -> composer[2].numbers[:];
18   connect op.result -> operation.operator;
19   connect composer[1].composed -> operation.operand[1];
20   connect composer[2].composed -> operation.operand[2];}
```

creating 4 CNN components for multiclass classification with 10 possible classes

creating a single CNN component for operator detection (+,-,,/)*

Figure 4.26: Textual EMADL representation of the MNISTCalculator architecture depicted in Figure 4.25.

EMADL

```
component Detector<Z(2:oo) classes = 10>{
  ports in Z(0:255)^{1, 28, 28} data,
  out Q(0:1)^{classes} softmax;

  implementation CNN
  {
    def conv(channels, kernel=1, stride=1){
      Convolution(kernel=(kernel,kernel),channels=channels) ->
      Relu() ->
      Pooling(pool_type="max", kernel=(2,2), stride=(stride,stride))
    }

    data ->
    conv(kernel=5, channels=20, stride=2) ->
    conv(kernel=5, channels=50, stride=2) ->
    FullyConnected(units=500) ->
    Relu() ->
    Dropout() ->
    FullyConnected(units=classes) ->
    Softmax() ->
    softmax;
} }
```

Figure 4.27: Detector component for the MNISTCalculator.

cf. Figure 4.27. The other components are implemented using MontiMath to compose the single digits to operands and to carry out the chosen operation.

Obviously, the kind of data the `Detector` component instances work on is the same for the digit detectors and structurally similar to the data processed by the operator detector. Hence, it makes sense to reuse the same CNN for both tasks, e.g. an AlexNet or a ResNet. Furthermore, we can assume that both tasks can be trained using the same training model. However, while the four digit detector instances should share the same training data, e.g. the MNIST database [LC10] for handwritten digit recognition, the operator detector should be obviously trained on another dataset. This is modeled by the dataset tag model which is given in Figure 4.28. Note that the tag model is compliant with the `TrainingDataToEmadlSchema` schema defined in Figure 4.24.

The first `Training` tag defined in L.4-8 is applied to the component type `Detector`, letting all instances of this type use the dataset information of this tag. The data path is set to `"/home/se/mnistdata"` and the format is HDF5. The fully qualified name of the referenced training model is `de.rwth.se.MyTraining`. The second `Training` tag is applied to the particular component instance `operatorPredictor` of the `MNISTCalculator` model. While the training model and the data format remain the same, the data path is set to `"/home/se/operatordata"`.

```
1  conforms to TrainingDataToEmadlTagSchema                    Tag Model
2
3  tags TrainingTags  {
4    tag Detector with Training = {          <- tag applied to all
5      datapath = "/home/se/mnistdata",        Detector components
6      dataformat = HDF5,
7      training = de.rwth.se.MyTraining;
8    }
9    tag MNISTCalculator.operatorPredictor with Training = {
10     datapath = "/home/se/operatordata",
11     dataformat = HDF5,
12     training = de.rwth.se.MyTraining;   <- tag applied to a specific
13   }                                        component instance only
14 }
```

Figure 4.28: Tag model providing information regarding the data and the training configuration to be used for the training of the detector components.

When a model such as the `MNISTCalculator` is compiled for the first time, the EMADL generator delegates the generation of the different components to the respective subgenerators. For MontiAnna components the training phase is executed after a successful generation. However, network training is not always necessary. First, the EMADL generator does not retrain MontiAnna components of the same type if their training configuration, i.e. the attached tags and the training models, are identical. In the `MNISTCalculator`, we have a `Detector` component array with four identical networks using the same training configuration. The training is executed once and the weights are reused by all four component instances. At runtime only one flyweight instance representing the four digit detectors is created.

Second, after training, the framework stores a metadata file including the creation date of the database used in the repository of the respective model. Whenever generation of the model is requested, the generator checks if a trained network artifact is available. If yes, it checks whether the database used has changed by comparing its actual metadata with the stored metadata file. If this is not the case and the model artifacts have not changed either, the generation of the `NetworkCreator` and `NetworkTrainer` are skipped and the network API reuses the old serialized network. For now, changes to model artifacts are checked based on the modification date and file comparisons. In future work, semantic differencing operators can be designed and implemented for neural network and training configuration models based on [Kau21] to further improve this process.

This is a basis for the automation of the development process for machine learning-based systems. Now, the system designer can handle neural network components in the same way as all other components and does not have to continuously keep track of

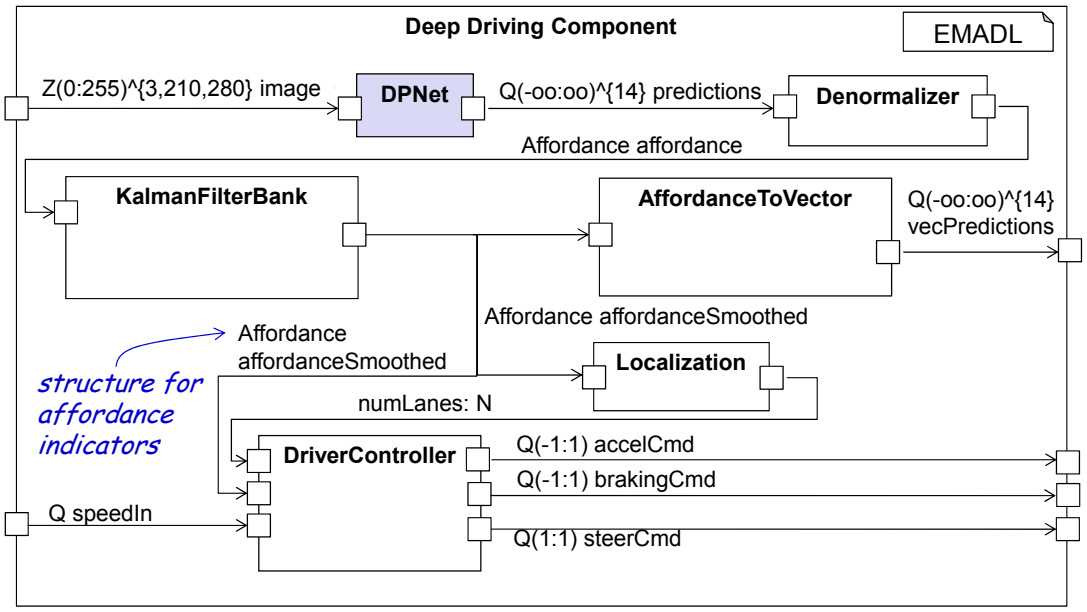


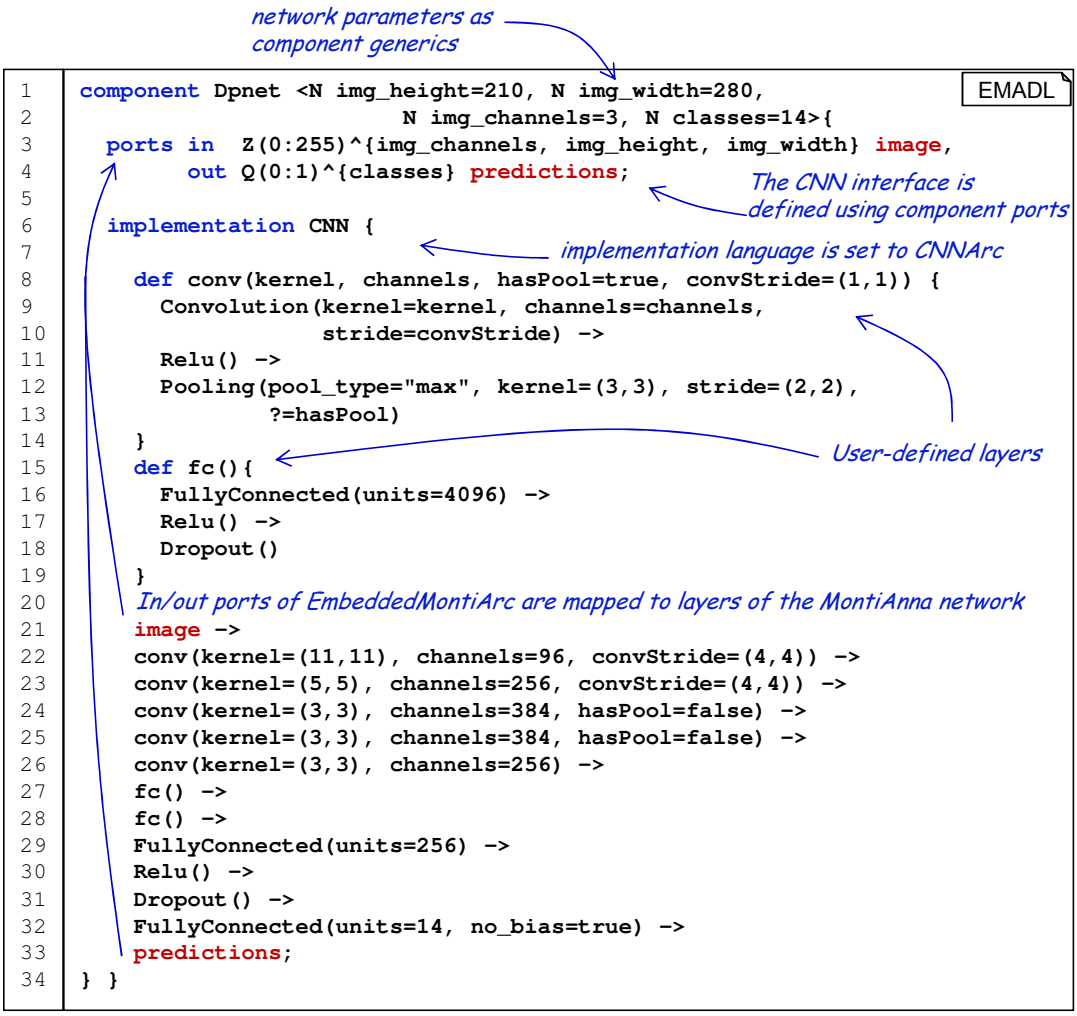
Figure 4.29: The deep direct perception-based architecture of a racing vehicle controller for TORCS [KPRS19].

their states. The lifecycle of such components is fully controlled by EMADL, training is performed automatically and unnecessary retraining is avoided. Furthermore, this approach enables the definition of neural network artifacts including training data and the trained weights as versionable archives, which can be deployed and reused as dependencies in build management systems such as Maven and Gradle. Another approach for handling the machine learning lifecycle is provided by the MLflow development platform [ZCD+18]. It facilitates the tracking of experiments and the underlying configuration and enables reusability by providing formats for packaging code, models, and data.

4.10.4 Modeling a Direct Perception Autonomous Vehicle Controller

A more complex example related to the domain of autonomous vehicles was modeled using EMADL by Svetlana Pavlitskaya in her master thesis and published in [KPRS19]. The goal was to develop an autopilot for The Open Racing Car Simulator (TORCS) following the deep direct perception approach as introduced in [CSKX15].

As described in Section 1.3.3, the goal of the direct perception approach is to use a CNN to extract features out of camera images, which can then be used by a conventional controller to compute the driving commands. A graphical overview of the EMADL architecture is given in Figure 4.29 [KPRS19]. Again, the CNN component of type




```
1  component Dpnet <N img_height=210, N img_width=280,
2                   N img_channels=3, N classes=14>{
3    ports in  Z(0:255)^{img_channels, img_height, img_width} image,
4          out Q(0:1)^{classes} predictions;
5
6    implementation CNN {
7
8      def conv(kernel, channels, hasPool=true, convStride=(1,1)) {
9        Convolution(kernel=kernel, channels=channels,
10                   stride=convStride) ->
11       Relu() ->
12       Pooling(pool_type="max", kernel=(3,3), stride=(2,2),
13               ?=hasPool)
14     }
15     def fc(){
16       FullyConnected(units=4096) ->
17       Relu() ->
18       Dropout()
19     }
20     image ->
21     conv(kernel=(11,11), channels=96, convStride=(4,4)) ->
22     conv(kernel=(5,5), channels=256, convStride=(4,4)) ->
23     conv(kernel=(3,3), channels=384, hasPool=false) ->
24     conv(kernel=(3,3), channels=384, hasPool=false) ->
25     conv(kernel=(3,3), channels=256) ->
26     fc() ->
27     fc() ->
28     FullyConnected(units=256) ->
29     Relu() ->
30     Dropout() ->
31     FullyConnected(units=14, no_bias=true) ->
32     predictions;
33 } }
34
```

Figure 4.30: The direct perception CNN for the extraction of affordance indicators in TORCS [KPRS19].

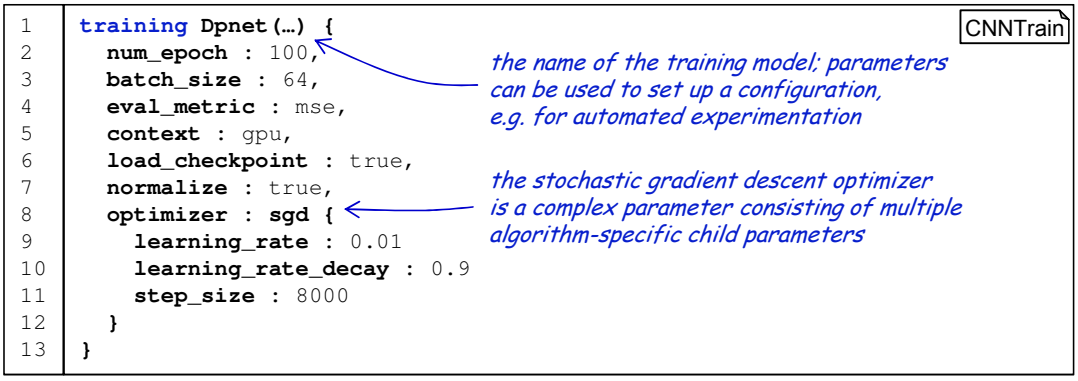


Figure 4.31: The training model of the DPNet component [KPRS19].

`DPNet` is depicted in purple to highlight that it is a neural processing component. All other components are implemented using MontiMath.

Based on [CSKX15], the architecture of the DPNet is a variant of the AlexNet [KSH12] featuring the following modifications: first, there are no local response normalization (`Lrn`) layers; second, two further fully connected layers with 256 and 14 units, respectively, are introduced. Furthermore, a sigmoid layer is added after the last fully connected layer of the network constraining the output to the range [0, 1]. However, due to our observations that the sigmoid layer slowed down convergence significantly, we decided to remove it in our model. The reason for this might be that without squashing the output to a relatively small range, larger deviations from the ground truth lead to larger gradients. Furthermore, when using a sigmoid layer, large errors might actually learn more slowly due to the saturation of the sigmoid function.

The code of the `DPNet` component is given in Figure 4.30. The input port `image` of the wrapping EMADL header is used as the input layer of the network in L.21. Similarly, `predictions`, the last layer of the network, is linked to the corresponding output port of the component.

The network structure is built in L.21-33 and is completely linear, i.e. the sequential operator `->` is sufficient to model the layer DAG. Unsurprisingly, convolutional layers are the main building block of the architecture. Moreover, we employ fully connected, ReLU activation, and pooling layers. A `Dropout` layer is included for regularization purposes in L.18 and L.31. The two custom layer classes `conv` and `fc` are defined in L.8-14 and L.15-19, respectively. Without these two definitions, the model would be much more repetitive. Using default parameters for the `conv` layers instantiated in L.22-26 further enhances model reusability. We therefore restrain from using parameter sequences to squash the `conv` instantiation.

The training model for the DPNet component is given in Figure 4.31. The hyper-parameters are largely based on the original paper [CSKX15]. Since, in contrast to

other applications discussed so far, the DPNet is used to predict features in a continuous space, the mean squared error (MSE) function is used to assess the error. The network is trained for 100 epochs with a batch size of 64. The optimizer is set to SGD. The learning rate is set to 0.01 and is scheduled to decrease by the factor 0.9 every 8000 steps.

The DPNet works with normalized data. Hence, we have to rescale the predictions by inverting the original normalization in the `Denormalizer` component. The obtained denormalized affordance indicators are then written into an `Affordance` struct. Due to the noisy nature of the affordance indicators predicted by the DPNet, a denoising component is necessary in our architecture. We apply the widely used Kalman filtering to smoothen the signals. Since we need to filter multiple streams independently, we encapsulate the required filter components into a `KalmanFilterBank`.

The task of the `Localization` component is to estimate the number of lanes based on the distance between the outer left and right lane markings. This information as well as the affordance indicators themselves and the speed of the vehicle are input into the `DriverController`, which in turn generates an appropriate driving behavior. The `AffordanceToVector` component is a helper component converting the `Affordance` struct to a 14-dimensional vector, which is then connected to one of the main component's output ports for analysis and evaluation reasons.

The `DriverController` applies the corresponding control algorithm of the DeepDriving project [CSKX15] to obtain the actuator commands for accelerating, braking, and steering based on the current speed, distance to the lane markings, and presence of the other cars in the currently occupied and neighboring lanes. The controller tries to keep the vehicle in the middle of the lane using the distance to the lane markings extracted by the DPNet. If another car is detected in front of the ego vehicle and a free lane is available, a lane change is initiated.

The acceleration command is set to a positive (normalized) value in the range $(0, 1]$ if the car drives more slowly than the predefined maximum velocity. Accordingly, the vehicle will brake if its velocity exceeds this limit. Braking and acceleration are also affected by the presence of other vehicles in front. The ego vehicle tries to maintain a minimum distance of 20 m to the front vehicle. Furthermore, the vehicle slows down in curves based on the steering history.

Internally, the `DriverController` uses a first in, first out `Buffer` holding past steering commands, lane change states, as well as lane change timers for several execution cycles. Up to five of the past steering commands are used to compute the deceleration in long turns. To avoid deceleration during lane changes, a corresponding lane change flag is set to inform the controller that the steering is not due to a turn. To be able to specify the kind of lane change being performed, we introduce the `LaneChange` enumeration taking the self-describing values `NO_CHANGE`, `TO_LEFT`, `TO_RIGHT`, `IN_RIGHT`, and `IN_LEFT`. Lane changing timers are used to render smooth lane change maneuvers.

To evaluate the EMADL toolchain, the architecture was generated and trained using the MXNet deep learning backend. To make sure that the logic of the architecture is

correct, we first tested it without the neural network component by providing ground truth affordance indicators to the model directly. Once the vehicle was able to cope with a variety of situations including following a track, overtaking and following other vehicles, we replaced the ground truth input by the neural network extracting the affordance indicators from the camera images. For training of the DPNet, we used the materials provided by the authors of [CSKX15][9]. Thereby, 387.851 samples were used to train and 96.963 to validate the network, each containing a camera image and the 14 corresponding affordance indicators as its label. After 140.000 iterations we achieved an MSE of 0.05 on the validation set. The trained model was used successfully to control a TORCS vehicle. A demonstration is available in the video channel of the Software Engineering department of the RWTH Aachen University[10]. The communication of the presented autopilot with TORCS [WEG+00] was modeled using the middleware tagging approach which will be presented in Chapter 6. Thereby, the whole development process of this AI-driven CPS controller including training, integration, and deployment was conducted in a model-driven manner without the need for hand-written GPL code.

[9] http://deepdriving.cs.princeton.edu/, accessed October 29, 2020
[10] https://youtu.be/hfICK4f-hR4, accessed October 29, 2020

Chapter 5

Modeling Deep Reinforcement Learning Architectures

Until now we have discussed how MontiAnna can be used to model supervised learning systems. While supervised learning has been successfully employed for tasks relevant for this thesis, its major drawback is that it relies heavily on labeled data. Gathering such data is time consuming, expensive, and not always feasible. Reinforcement learning is a subdomain of machine learning which does not require a labeled dataset to be provided for training. The field is of particular interest for behavioral training and hence, for CPSs. For instance, it can be used to teach a system to move in a specific environment using its sensors and actuators. An agent such as a self-driving vehicle controller aims to learn an appropriate policy by maximizing a reward function in a series of experiments.

In this chapter we are going to discuss how MontiAnna can be used for the design, training, and integration of deep reinforcement learning systems based on [GKR19]. In contrast to supervised deep learning, a deep reinforcement learning system requires multiple components, which play different roles throughout the training process: in addition to the actors we actually want to train, we need critic networks, reward functions, and environments. The compiler needs to instantiate and interconnect these components in the different phases of the system's lifecycle, which cannot be achieved by the means of the EMA component model discussed so far.

The research question to be answered in this chapter is the following:

Research Question 4. *How can different training approaches such as reinforcement and adversarial learning be modeled and used at SMArDT level 3?*

5.1 Foundations of Deep Reinforcement Learning

In this section we are going to introduce the terminology and basic concepts of the field needed to understand and design a deep reinforcement learning modeling methodology. The *environment* and the *agent* are two central concepts present in all kinds of reinforcement learning (RL) systems. The agent is the trainable part of the system trying

to learn a behavior policy to achieve some given goal in an environment. Therefore, the agent senses the state s and performs an action a according to its policy π. As a response, the environment and the state of the agent are updated. In the training phase the environment computes a reward measuring the quality of the action based on the new state. The agent's behavior is learned by exploring the action space to find action sequences maximizing the expected reward in a series of experiments. Thereby, the policy can be used in the training phase explicitly, which is referred to as on-policy learning. Algorithms ignoring the policy while learning are classified as off-policy. An important function used in various RL algorithms for learning is the action-value $Q(s, a)$ estimating the overall expected reward if the agent is in state s and performs action a.

In contrast to supervised learning algorithms, in RL the agent does not learn from existing labeled data and there is no correct action or ground truth for a given state. Although in general RL is applicable to continuous time systems [Doy00], we will discuss discrete time systems exclusively, which is in line with EMA semantics and the nature of digital cyber-physical systems. The state and action spaces of a set-up can be either discrete or continuous. Combinations of discrete action with continuous state spaces, and vice versa, are possible, as well.

An important concern of RL is the balance between exploration and exploitation during the learning process. If the agent chooses an action according to the highest estimated reward, it *exploits* its knowledge. Otherwise, it *explores* the action space to gather information and learn to make better choices. *ε-greedy strategy* can be applied to discrete action spaces. In this strategy, the agent acts greedily, but with a probability of ε it chooses a random action. For problems with a continuous action space, we can add exploration noise to the action chosen by the policy instead. Stochastic models such as the Ornstein-Uhlenbeck (OU) process [UO30] or Gaussian noise are suitable for this purpose.

Although RL does not inherently depend on deep learning methods, neural networks have been successfully applied in RL. Alpha Zero is a prominent example of deep RL systems defeating human world champions in complex board games such as chess and go [SHS+17]. Since the focus of MontiAnna is the model-driven development of neural systems, we are going to restrict the further discussion to deep RL algorithms. In particular, MontiAnna supports two standard deep RL algorithms: the deep Q-Network (DQN) approach [MKS+13] and deep deterministic policy gradient (DDPG) [LHP+15]. Furthermore, twin-delayed DDPG (TD3) [FvHM18] is offered as an extension of DDPG. This way, MontiAnna covers learning problems with discrete and continuous state and action spaces. The concept of the modeling framework is suitable for the integration of further state-of-the-art algorithms.

DQN. Q-learning is an off-policy approach applicable to problems with discrete action and state spaces. The idea of Q-learning is to iteratively learn a reward table storing the

rewards for all possible combinations of state-action pairs [Wat89]. However, memorizing such a table is not feasible for large problems. DQN tackles this challenge by using a neural network to learn the Q-function [MKS+13]. The neural network receives the state vector as its input and outputs a Q-value for each possible action. This makes the algorithm suitable for continuous state spaces, as well. The architecture of the network can be chosen arbitrarily to suit the problem.

In the training phase, the network is not trained after each exploration step. Instead, DQN employs a technique called experience replay [Lin93] separating exploration from learning. After each exploration step in the environment, the obtained transition (state, action, reward, next-state) is saved in an intermediate buffer. After a predefined number of updates a minibatch is drawn randomly from this buffer and used to train the Q-network. This allows the algorithm to reuse the same transitions multiple times without having to explore them again. Furthermore, drawing training examples randomly decorrelates the training data preventing the network from overfitting.

In each step the agent inputs the current state into the Q-network and looks for the output neuron with the highest Q-value. Then it executes the action associated with this neuron, i.e. the action is determined as

$$a(s) = \arg\max_{a} Q(s, a). \tag{5.1}$$

Variations of DQN can be applied to problems with continuous action spaces, as well [GLSL16].

DDPG. DDPG is a reinforcement learning algorithm applicable to continuous action spaces exclusively. It combines DQN, deterministic policy gradient (DPG) [SLH+14], and actor-critic methods. Given a continuous action space, the maximization problem in Equation (5.1) cannot be solved by evaluating the Q-function for all possible actions. Solving the problem analytically in each step would be an overly expensive task. DDPG learns both a concrete policy $\pi(s)$ and an action-value function $Q(s, a)$. The policy and the Q-function approximators are referred to as the *actor* and the *critic* networks, respectively. Both approximators are modeled as neural networks. While the critic network is trained similarly to DQN, the actor network is trained using the critic so that the policy maximizes $Q(s, \pi(s))$. Once training is finished only the actor network is required. The critic network is, hence, not part of the logical software architecture.

5.2 Requirements

The general requirements for a deep learning methodology presented in Section 4.2 continue to hold for RL-based systems. However, due to the particular nature of such systems, we have elicited an additional set of RL-specific requirements.

(RRL1) Reinforcement Learning Algorithms: as discussed in Section 5.1 various deep RL algorithms exist. The choice of the right RL algorithm depends on the problem to be solved. A versatile deep RL framework should provide the possibility to specify the learning algorithm to be used explicitly without having to deal with its implementation details. Furthermore, the developer should be able to set the hyperparameters of the chosen approach freely to tailor the solution to the problem adequately.

(RRL2) Network roles: as we have outlined for DQN and DDPG, deep RL employs neural networks as function approximators. Hence, the user should be able to model the architectures of these neural networks individually to fit the RL task to solve. What is more, various RL algorithms such as DDPG make use of multiple neural networks, each having a dedicated purpose. Therefore, the developer should be able to assign roles to the networks, e.g. the actor and critic roles when using DDPG.

(RRL3) Environment integration: RL can only be conducted by letting the agent interact with a preferably simulated target environment. This environment might not be explicitly designed for the interaction with an RL agent. A modeling framework must provide the means to describe the environment interface. Based on this description, the compiler should be able to realize the coupling between the agent and the environment. Thereby, the interface and the behavior of the environment must remain time-invariant from the modeler's view.

5.3 Related Work

Similarly to supervised learning problems, RL-based systems can be realized using an appropriate framework. General-purpose deep learning frameworks such as TensorFlow [ABC+16], Theano [ARAA+16] or MXNet [CLL+15] can be used to model and train the required function approximators. However, the functionality around the neural network, e.g. the exploration strategy, the communication with the environment, and the interaction between the actor and the critic need to be implemented from scratch. A developer needs a profound understanding of RL algorithms in order to be able to use them. For this reason, more specialized frameworks tailored to the RL domain have been developed. In contrast to generic solutions such as TensorFlow, RL frameworks provide RL functionality such as predefined agents out-of-the-box. In the following paragraphs we are going to discuss a selection of popular approaches.

OpenAI Baselines. OpenAI Baselines [DHK+17] is a Python-based RL framework built on top of TensorFlow. It offers open-source implementations of various RL algorithms including DQN and DDPG as well as predefined neural network architectures.

```bash
python -m baselines.run --alg=deepq --env=CartPole-v0\
--num_timesteps=10000 --network=mlp --num_layers=2\
--num_hidden=64 --activation=tf.tanh --save_path=cp_deepq
```

Figure 5.1: Setting up a training in OpenAI Baselines using its CLI.

According to the authors, the intention of OpenAI Baselines is to make RL research reproducible and to provide a basis for further developments. Researchers can use the provided libraries and adapt them to experiment with new ideas. Furthermore, OpenAI Gym [BCP+16] offers a set of playground environments useful for learning, experimentation, and model comparison. The user can control the training by selecting an OpenAI Gym environment and the algorithm to be used via the CLI. Figure 5.1 shows how an MLP agent can be trained by using DQN in a CartPole environment. Being more focused on RL research, the framework targets advanced and experimenting users requiring a lot of flexibility.

DeepMind TensorFlow Reinforcement Learning (TRFL). TRFL [Lim18] is a Python-based library using TensorFlow as its deep learning backend. Its focus is not on out-of-the-box implementations, instead the framework provides components for the realization of RL algorithms. In particular, its main approach is to regard and encapsulate RL update rules as loss functions. The user has to model the neural network architecture in pure TensorFlow, but can apply RL specific loss functions, e.g. `trfl.qlearning(.)` to perform Q-learning. Hence, the framework can be classified as an intermediate solution between general-purpose deep learning and specialized RL frameworks. The user needs an understanding of the applied algorithms. In contrast to OpenAI Baselines, TRFL does not provide predefined environments. The communication between the agent and the environment needs to be developed manually.

Tensorforce. Like TRFL and OpenAI Baselines, Tensorforce [KSF17, SKE+18] relies on TensorFlow as its deep learning backend. However, it is rather a high-level framework with a focus on practical systems. It provides predefined agents and further modules. Furthermore, the framework defines clear interfaces for agents and environments, facilitating reusability and assembly of RL systems. Agents are provided for a variety of well-established RL algorithms such as DDPG or DQN. Moreover, the user is able to set up the RL system by passing a set of parameters to the agent's constructor. These parameters describe aspects concerning the system itself, e.g. the state and the action space, the network architecture, but also the exploration and the training phase as well as the corresponding hyperparameters. The network architecture can be auto-configured based on the information about the state and the action spaces when using the `AutoNetwork` class. On the other hand, the user still has a high degree of configurability.

```python
from tensorforce.agents import DQNAgent
from tensorforce.execution import Runner
from tensorforce.contrib.openai_gym import OpenAIGym
q_network = [{'type': 'dense', 'size': 64, 'activation': 'tanh'},
             {'type': 'dense', 'size': 64, 'activation': 'tanh'},]
agent = DQNAgent(states={'type': 'float', 'shape': (4,)},
          actions={'type': 'int', 'shape': (2,)},
          network=q_network, memory={'type': 'replay', 'capacity':
                                                          10000},
          step_optimizer={'type': 'adam', 'learning_rate': 0.001},
           states_preprocessing={'type': 'divide', 'scale': 10})
environment = OpenAIGym('CartPole-v0')
runner = Runner(agent=agent, environment=environment)
runner.run(episodes=1000)
runner.close()
```

Figure 5.2: Setting up and training a DQN agent using Tensorforce.

For instance, it is possible to set the network depth or to add an LSTM as the last layer. Alternatively, it is possible to assemble custom network architectures from a library of available layer classes. This can be accomplished by creating a Python list of layer specifications if the network has a standard sequential architecture. For more complex networks consisting of multiple sequential layer-stacks, it is possible to create a list of lists of layers, as well. Both the network and the agent can be defined as JSON-files instead of Python.

Tensorforce supports a variety of environments to choose from, including the previously discussed OpenAI Gym but also VizDoom [KWR+16], Deepmind Lab [BLT+16], the Arcade Learning Environment (ALE) [BNVB13], etc. Furthermore, it is possible to create and integrate custom environments by implementing the environment interface including methods such as reset, execution of an action, and retrieval of the specifications of the state and action spaces.

The training is carried out by a runner object which expects an agent and an environment on instantiation. Training can be sped up by using a parallel runner. Tensorforce can accelerate the exploration part of RL by running independent simulations in parallel [RK19]. Being a high-level framework, Tensorforce is supposed to be easier to use than the previously discussed alternatives, since the user does not need to have a deep knowledge regarding the internals of the used RL algorithms.

An example setting up and training a DQN agent in the CartPole environment of OpenAI Gym is given in Figure 5.2. After importing the required modules in L.1-3, the Q-network (`q_network`) is defined as a list of two layers in L.4-5. The agent is instantiated as a `DQNAgent` in L.6-11. Thereby, the state space is set as four-dimensional and `float`-typed, whereas the action space is set to be two-dimensional and `int`-typed. The network parameter is bound to the previously defined `q_network` object. Further parameters regarding the memory, the optimization algorithm, and the preprocessing are

```
gin
1  DQNAgent.gamma = 0.99
2  DQNAgent.update_period = 4
3  tf.train.RMSPropOptimizer.learning_rate = 0.00025
4  tf.train.RMSPropOptimizer.decay = 0.95
5  atari_lib.create_atari_environment.game_name = 'Pong'
6  create_agent.agent_name = 'dqn'
7  Runner.num_iterations = 200
8  Runner.training_steps = 250000
9  WrappedReplayBuffer.replay_capacity = 1000000
10 WrappedReplayBuffer.batch_size = 32
```

Figure 5.3: Excerpt of a Google Dopamine gin configuration to train a DQN agent in the Atari Pong environment.

set, as well. The CartPole environment of OpenAI Gym is instantiated in L.12. Finally, the runner is instantiated with the agent and the environment as its parameters, run, and closed in L.13-15.

Google Dopamine. Like Tensorforce, Google's Dopamine [CMG+18] is a high-level out-of-the-box Python framework providing predefined agents. According to the developers, the focus is on fast prototyping and experimentation. The training of an agent is defined declaratively in a gin configuration file[1], similarly to the Tensorforce way of defining the agent using JSON. In the gin file the user can specify the agent type, the optimization strategy as well as the environment and the corresponding hyperparameters. An example of a gin training configuration is given in Figure 5.3. It sets up a DQN agent and trains it in the Pong environment of ALE. Custom agents can be implemented from scratch or by subclassing one of the available agents. However, there is no common agent superclass or interface. In its first version Dopamine was designed to be used in conjunction with ALE. Dopamine 2 introduced support for general discrete-domain Gym environments.

Mathworks Reinforcement Learning Toolbox. The Reinforcement Learning Toolbox by Mathworks[2] offers RL functionality for MATLAB/Simulink users enabling the block-based development of RL systems. The toolbox provides a series of predefined agents including DQN and DDPG, but custom ones can be created by subclassing the `rl.agent.CustomAgent` class. An agent block exhibits an interface with the input ports *observation*, *reward*, and *isdone* and the output ports *action* and *cumulative reward*. Neural networks representing policy and value functions can be assembled using the graphical Deep Network Designer app, but it is also possible to import Open Neural Network Exchange (ONNX)[3] models created using other frameworks.

[1] https://github.com/google/gin-config
[2] https://www.mathworks.com/products/reinforcement-learning.html, accessed October 14, 2020
[3] https://onnx.ai/, accessed January 20, 2020

Custom environment blocks can be created by implementing the observation, action, and reward functionality required by the environment interface. Predefined MATLAB/Simulink environments are provided, as well. What is more, external environments can be integrated by using an external language interface, a distributed protocol such as ROS, or functional mockup units (FMUs). An agent can be trained using the `train(.)` function which takes the agent, environment, and a configuration object holding hyperparameters as its arguments.

5.4 Modeling Reinforcement Learning

The discussed frameworks employ different approaches including white box and black box solutions to cover the needs of different user groups. All of these frameworks except MATLAB/Simulink are meant to be used with Python as host GPL and TensorFlow as deep learning backend. Integrating agents into larger software architectures is not in the scope of these frameworks and hence, may require additional development effort. In the following we are going to extend the applicability of the supervised learning oriented MontiAnna framework to reinforcement learning tasks. Thereby, we maintain the model-driven and generative paradigm of component-based development for CPS design. Furthermore, in contrast to many of the presented frameworks requiring a profound understanding of the used algorithms, the RL language extension for MontiAnna should enable the designer to use RL algorithms as a black box. Similarly to supervised learning, RL algorithms and hyperparameters should be set up in a declarative way.

To maintain a link to the CPS domain, the methodology will be explained by means of a TORCS [WEG+00] racing controller. In particular, our goal is to replace the MontiMath-based controller component (`DriverController`) presented in Figure 4.29 in Section 4.10.4 with a MontiAnna component trained using RL. The continuous state space of the vehicle is given as $S \subseteq \mathbb{Q}^{29}$ and is a subset of the state space provided by the TORCS competition server interface [LCL13]. It includes information on velocity, track axis angle, as well as data from 20 range finder sensors returning the distance to the edges of the track. The action space $A = [-1, 1]^3$ is continuous, as well. Its three dimensions represent the steering wheel, the throttle, and the braking pedal of the vehicle in this very order. Thereby the steering wheel values of -1 and 1 represent the steering wheel fully turned to the right and left, respectively. For the pedals, -1 and 1 represent a released and a fully pressed pedal, respectively.

5.5 Modeling the Function Approximators

Since the action space is continuous, we are going to use the DDPG algorithm to train the agent. As discussed in Section 5.1, DDPG needs an actor and a critic network for the training procedure. The two network components are given in Figures 5.4 and 5.5. We

EMADL

```
1  package torcs.agent.network;
2  component TorcsActor {
3    ports in  Q^{29} state,
4          out Q(-1:1)^{3} action;
5
6    implementation CNN {
7      state ->
8      FullyConnected(units=300) ->
9      Relu() ->
10     FullyConnected(units=600) ->
11     Relu() ->
12     FullyConnected(units=3) ->
13     Tanh() ->
14     action; } }
```

Figure 5.4: EMADL component of the TORCS actor based on [GKR19]. The implementation is described using the MontiAnna architecture language CNNArc.

EMADL

```
1  package torcs.agent.network;
2  component TorcsCritic {
3    ports in  Q^{29} state,
4          in  Q(-1:1)^{3} action,
5          out Q qvalue;
6
7    implementation CNN {
8      (
9        state ->
10       FullyConnected(units=300) ->
11       Relu() ->
12       FullyConnected(units=600)
13     |
14       action ->
15       FullyConnected(units=600)
16     )->
17     Add() ->
18     FullyConnected(units=600) ->
19     Relu() ->
20     FullyConnected(units=1) ->
21     qvalue;}}
```

Figure 5.5: EMADL component of the TORCS critic based on [GKR19]. The implementation is described using the MontiAnna architecture language CNNArc.

can model these two networks using the standard network description syntax of MontiAnna, since these networks are structurally equivalent to the ones used for supervised learning problems. If DQN or any other RL algorithm is used to train the agent, the function approximator for the actor can be modeled in a similar way as in this example, but no critic might be needed.

The actor model in Figure 5.4 is an MLP with three fully connected layers instantiated in L.8, L.10, and L.12 and consisting of 300, 600, and 3 neurons, respectively. The activation function is set to ReLU for the first two layers in L.9 and L.11 while `Tanh` is used at the output in L.13. The network is supposed to take the state vector sent by the environment, which is TORCS in our case, as its input in L.7 and to produce an action vector to be output in L.14.

The aim of the critic model in Figure 5.5 is to approximate the Q-function. Hence, it has two input ports according to the signature of the Q-function described in Section 5.1, namely `state` and `action`. Note that although most of the neural networks we have dealt with so far exhibited exactly one input and one output, MontiAnna can handle interfaces with more than two ports, as well. While this is a common pattern in RL, some supervised learning architectures make use of multiple input and output ports, as well, cf. the FlowNet [DFI+15] architecture in Figure A.3, a CNN for optical flow estimation.

The two inputs flow through two separate layer pipelines. The path for the state vector is given in L.9-12 while the action vector is processed in L.14-15. The two paths are merged using an `Add` layer in L.17 which sums the two inputs elementwise. The rest of the network is a serial path defined in L.18-21 outputting the Q-value (`qvalue`) at the end. Since the Q-value is a scalar, the last neuron layer, defined in line 20, contains only one neuron and has no non-linearity.

Now having defined the critic and actor components, a new challenge arises, which we have not encountered in C&C modeling so far: we only need one of the components in our final productive system, namely the actor taking the state as input and deciding what to do based thereon. The final runtime system architecture is given in the upper part of Figure 5.6. For the sake of simplicity we omit the direct perception part. Instead we assume that our system receives sensor data as inputs instead of a camera image. Furthermore, we abstain from filtering the noisy sensor inputs.

The critic, on the other hand, is only required at training time and is not part of the runtime architecture. To cope with this issue, we are going to introduce the notion of roles for components. While role modeling has been mostly studied for object-oriented structures [Ste00, KLG+14] which are more dynamic than C&C models by nature, we can reuse at least some of the concepts for our purpose. In particular, a component role consists of a role interface and the definition of its relationships to other roles in specific contexts. In contrast to object-oriented modeling, we abstain from dynamic properties of roles such as the ability of acquiring and abandoning roles dynamically. General role modeling for components could be realized with the help of C&C views [BMR+17] by

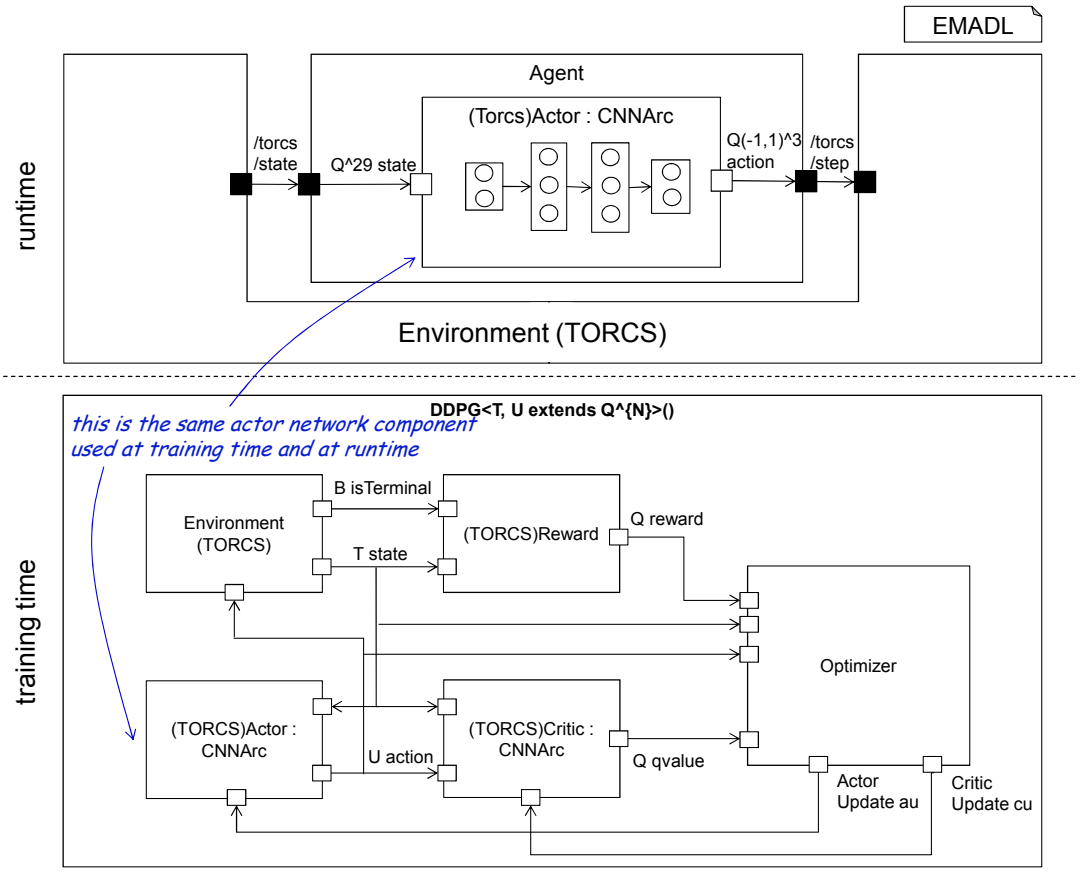


Figure 5.6: The upper part of the figure depicts the top level runtime C&C model of the DDPG-based TORCS controller. The filled ports highlight the reinforcement learning interface. The connections between them are generated based on the middleware tagging approach, which will be presented in Chapter 6. The training reference architecture of the DDPG algorithm is depicted in the bottom part and contains a critic, a reward, an environment, and a trainer component, which are thrown away at runtime.

providing component interfaces and interconnections for different contexts, e.g. compile-time and runtime, and by mapping component instances to roles. This, however, is not in the scope of this thesis. Instead we are going to use predefined roles for the usage in actor-critic methods, e.g. the actor, the critic, and the reward role. Furthermore, we are going to use the predefined contexts *training time* and *runtime*. The architecture for the runtime context is defined as usual in EMADL, cf. upper part of Figure 5.6. In the training context, additionally critic, environment, and reward components are used to train the actor. The training time architecture of an RL system trained with the DDPG algorithm is depicted in the lower part of Figure 5.6. The environment is exchangeable between the training and the runtime phases. Critic and reward components, on the other hand, are thrown away after training time. The training time model in Figure 5.6 can be thought of as a reference architecture. The reference training architecture is fixed for a given training algorithm and defines the roles required for the algorithm to function as subcomponents.

The port types in the role definitions are kept generic and are adapted to the application. In our TORCS example, the state type `T` is `Q^{29}` and the action type `U` is `Q(-1:1)^{3}`. The generic type parameters of the roles are bound by providing concrete component types for the corresponding roles as is done for the actor and critic roles in Figures 5.4 and 5.5. Assigning roles to components is in the scope of the following section. Note that the training reference architecture is a conceptual model. It can serve to generate stubs and interfaces as well as to check for consistency, e.g. whether the state type of the actor is the same as the one of the critic. However, in contrast to (runtime-oriented) EMADL models, we do not generate fully functional code out of the reference training architecture. The generator developer needs to provide templates for training code generation complying with the interfaces of the reference architecture.

5.6 Modeling the Training

As discussed in Section 4.9, training is modeled using the declarative language CNNTrain and we continue to employ this concept for RL, as well. However, we obviously need to extend the language by a set of new parameters. Furthermore, we will use CNNTrain to assign roles to EMADL components.

5.6.1 General Reinforcement Learning Parameters

In the following, we give a brief overview of general RL parameters applicable to all RL algorithms.

enum learning_method=supervised. In Chapter 4 we were dealing with supervised learning exclusively. Hence, this learning method was assumed implicitly. However,

since we need to discriminate between different learning methods now, we introduce an explicit learning method parameter. The allowed values are `supervised` for supervised learning, `reinforcement` for RL, and `gan` for generative adversarial networks, cf. Section 5.9. To keep old models valid, we use `supervised` as default. The parameters introduced below are only allowed if the learning method is set to `reinforcement`.

enum rl_algorithm. This parameter needs to be set in order to choose a concrete RL algorithm for training. The allowed values are `dqn-algorithm` for DQN, `ddpg-algorithm` for DDPG, and `td3-algorithm` for TD3. If the parameter is not set explicitly, `dqn-algorithm` is used as default. Each algorithm comes with its own individual parameters which are introduced in Appendices B.3.2 and B.3.3.

N1 num_episodes=50. An episode is a full simulation from the beginning to the final state. An RL agent is usually trained in multiple episodes. The `num_episodes` parameter is typed as an integer and specifies the number of episodes to be trained. The default value is 50.

N1 num_max_steps=99999. This integer-valued parameter sets the number of steps within an episodes before the environment is forced to reset the state, e.g. to avoid the agent being stuck forever. The default value of this parameter is 99.999.

Q(0,1) discount_factor=0.99. The discount factor γ is a hyperparameter weighting future rewards for an update step. It is a rational between 0 and 1. A reward r which occurs n steps from the current step is considered as $r\gamma^n$. The default value is $\gamma = 0.99$

Q target_score. The target score is an optional `Q`-typed parameter. If set, training will be stopped when the average score of the last 100 episodes reaches the parameter value.

EMADLComponent reward_function. The reward function parameter expects the fully qualified name of the EMADL component used to calculate the reward. The referenced reward component must exhibit an interface consisting of two input ports and one output port: an arbitrarily typed `state` input port depending on the application, a Boolean input port `isTerminal` which expects to receive a true signal if the current state is a terminal one, as well as a `Q`-typed output port `reward`, through which the reward for the current state is output. The chosen component type is used by the generated training system, even if it has not been instantiated in the main EMADL model being generated. An isolated instance is created and connected to the environment automatically to play the reward role. Alternatively, an external ROS-based reward service can be used by setting the `reward_topic` of the `ros_interface` (see below).

nested environment. The environment parameter is essential to connect the agent to the environment. The parameter is required and can be set to `ros_interface` or to `gym`. In either case more information needs to be provided through further nested configuration parameters. The details are given below.

gym. This option allows us to train the agent with one of the existing OpenAI Gym environments. It has the following configuration parameter:

String name. The `gym.name` parameter is typed as string and denotes the name of the OpenAI Gym environment to be used. The code generator will produce code to instantiate the chosen environment and to connect the agent to it.

ros_interface. Setting the environment to `ros_interface` lets the agent and the environment communicate via ROS. This enables us to use any external program as the training environment as long as it supports our ROS-based training interface. The user can map this interface to arbitrary ROS topics with the help of the following configuration parameters:

String state_topic="/state". The parameter is used to define the name of the ROS topic to which the current state of the environment is published. The default value is set to "`/state`".

String action_topic="/action". The parameter is used to define the name of the ROS topic to which the agent publishes the chosen actions. The default value is set to "`/action`".

String reset_topic="/reset". The parameter is used to define the name of the ROS topic which can be used by the training algorithm to reset the environment. The default value is set to "`/reset`".

String terminal_state_topic="/terminal". The parameter is used to define the name of the ROS topic to which the environment writes that the last state was a terminal state. The default value is set to "`/terminal`".

String reward_topic. The parameter is used to define the name of the ROS topic to which the environment publishes the reward. This topic is required if no EMADL component for the reward, cf. `reward_function` parameter above, is provided by the user. If neither is set, an error will be issued.

The following additional parameters are only available for DDPG and TD3.

EMADLComponent critic. This parameter is required when DDPG or TD3 is used. It expects the fully qualified name of the EMADL component encapsulating the critic network. Similarly to the reward component, the chosen component type for the critic is used by the generated training system, even if it has not been

instantiated in the main EMADL model being generated. An isolated instance is created automatically to play the critic role.

Q soft_target_update_rate=0.001. Represents the interpolation factor τ of the critic and actor target network. This hyperparameter controls the update of the target network weights according to $W_{target} := (1-\tau)W_{target} + \tau W$. Based on [LHP+15], the default value is set to 0.001.

nested actor_optimizer. Sets the optimizer for the actor network training. More information and the available optimizer configuration parameters are given in Section 4.9.

nested critic_optimizer. Sets the optimizer for the critic network training. More information and the available optimizer configuration parameters are given in Section 4.9.

A complete overview of CNNTrain parameters for reinforcement learning applications is out of scope of this section. Information on parameters concerning the replay memory, exploration strategy, etc. can be found in Appendix B.3.1.

As was demonstrated in this section, linking the different roles for the training phase is done via the CNNTrain model. The concrete linkage implementation is not visible to the modeler and is restricted to supported training algorithms. The approach is easy to use and exploits the component-based idea of EMA. Introducing a metalayer to model the learning architecture including the roles and their relationships using EMA and/or process-oriented modeling techniques has the potential to improve the extensibility of MontiAnna, to facilitate experimentation with new learning algorithms, and to automate the creation of context conditions in future work.

5.6.2 The TORCS Training Model

Figure 5.7 shows a CNNTrain model for the TORCS actor of Figure 5.4. In L.4 we specify that the component is trained using reinforcement learning by setting the `learning_method` parameter to `reinforcement` making the new parameters discussed earlier in this chapter available. As already mentioned, due to the continuous action space, we are going to apply the DDPG algorithm for this use case, which we specify in L.5. Due to this choice, the component the training model is attached to, i.e. `torcs.agent.network.TorcsActor` in our example, becomes the actor automatically.

To assign the critic role to our critic component `torcs.agent.network.TorcsCritic` modeled in Figure 5.5 and hence, let the trainer use it during training according to the reference training architecture of Figure 5.6, we use its fully classified name for the `critic` parameter in L.6. This parameter is required due to the critics role defined in the reference training architecture of the DDPG algorithm. Note that

CNNTrain

```
package torcs.agent.network;
configuration TorcsActor {
  context : gpu
  learning_method : reinforcement
  rl_algorithm: ddpg-algorithm
  critic: torcs.agent.network.TorcsCritic
  environment : ros_interface {
    state_topic : "/torcs/state"
    terminal_state_topic : "/torcs/terminal"
    action_topic : "/torcs/step"
    reset_topic : "/torcs/reset"
  }
  reward_function: torcs.agent.network.Reward
  num_episodes : 3000
  discount_factor : 0.99
  num_max_steps : 900000
  training_interval : 1
  start_training_at: 0
  evaluation_samples: 50
  soft_target_update_rate: 0.001
  snapshot_interval : 150
  replay_memory : buffer{
    memory_size : 120000
    sample_size : 32
  }
  strategy : ornstein_uhlenbeck{
    epsilon : 1.0
    epsilon_decay_method: linear
    epsilon_decay_start: 10
    epsilon_decay : 0.0001
    min_epsilon : 0.0001
    theta: (0.6, 1.0, 1.0)
    mu: (0.0, 0.0, -1.2)
    sigma: (0.3, 0.2, 0.05)
  }
  actor_optimizer: adam { learning_rate : 0.0001 }
  critic_optimizer: adam { learning_rate : 0.001 }
  target_score : 100000}
```

Figure 5.7: Configuration of the TORCS actor training modeled using the training language of MontiAnna based on [GKR19].

the critic component is instantiated implicitly and must not be part of the runtime architecture depicted in the upper part of Figure 5.6. The same holds for the reward component.

Furthermore, we configure all necessary hyperparameters, e.g. the number of episodes to train in L.14, the discount factor in L.15, as well as DDPG related parameters such as the target update rate in L.20. The optional parameter `target_score` is set in L.38 and serves as a stopping criterion for the training. Once the average reward of the last 100 episodes is equal to or greater than the target score, the training is finished. The parameter `snapshot_interval` used in L.21 lets the trainer save the weights of the agent once per 150 episodes. By default, if `snapshot_interval` is not set explicitly, the weights are only stored when the training is finished.

For the replay memory, we use the operation mode `buffer` as introduced in [MKS+13, Lin93], which stores state-action-reward pairs in a buffer. The memory size and the sample size are set using the respective nested parameters in L.23 and L.24. Alternatively, we could have used the `online` mode which does not use a replay memory at all or the `combined` mode. The latter is an implementation of a replay memory variant introduced in [ZS17] where only the last state-action-reward pair is added to the minibatch.

The operation mode of the exploration strategy is set in L.26-35 of the training model. The framework supports the ε-greedy strategy for discrete action spaces and the OU process as well as Gaussian noise for continuous problems. Independent of the operation mode used, the modeler is able to set the randomness or noise level for the action selection through the parameter epsilon, cf. L.27. Since the action choice should become more and more robust throughout training, we set up a linear decay of epsilon after the tenth period by 0.0001 per episode in L.28-30. This linear reduction is stopped, when the minimal allowed value of 0.0001 as defined in L.31 is reached. Each strategy requires a set of specific parameters to set up the noise. For instance, we initialize the parameters `theta`, `mu`, and `sigma` controlling the OU process in L.32-34. Note that since in our example the action space is three-dimensional, the generated noise and hence, the noise parameters are three-dimensional, as well. This is why each of the parameters set in L.32-34 is a three-dimensional array modeling individual values for steering, braking, and acceleration.

5.6.3 Reward Function

A central concept of RL is the reward function assessing the performance of the agent throughout learning. It can be provided by the environment directly, which is often the case for playground environments such as OpenAI Gym designed with RL in mind. If this is the case, the reward can be received through the environment interface after each action. However, if the simulator used is not designed explicitly for RL and, thus, does not offer a reward function out of the box or if a customized reward function is required, an external reward function needs to be made available to the training algorithm.

EMADL

```
1  package torcs.agent.network;
2  component Reward {
3    ports in  Q^{29} state,
4          in  B isTerminal,
5          out Q reward;
6
7    implementation Math {
8      Q speedX = state(22);
9      Q angle = state(1);
10     Q trackPos = state(21);
11     reward = speedX * cos(angle);
12     if abs(trackPos) > 1.0
13       reward = -20;
14     end } }
```

Figure 5.8: EMADL component with a MontiMath implementation encapsulating an exemplary reward function for the TORCS agent training based on [GKR19].

Sticking to the component-centric approach of EMA and MontiAnna, we have decided to let designers encapsulate reward functions as EMA components. Since the reward function is usually a mathematical operation on the state space, MontiMath turns out to be an appropriate language for the concrete implementation. The reward component obviously plays a special role in an RL model. Similar to critic components it is only needed at training time. To make it usable by the trainer, it needs to comply with a predefined reward interface. In MontiAnna, the reward interface requires two inputs. The first input named `state` receives the current state of the environment. The type of this port is not prescribed by the framework, but depends on the application or the simulator used. The second input is a Boolean flag named `isTerminal` indicating whether the last received state is a terminal state. This port is optional and the compiler will emit a warning if it is declared but not used in the implementation. The reward component must exhibit a single output port named `reward`, which is interpreted as the reward for the last action. Its type must be compatible with `Q`, i.e. it can also be a subset such as `Z(-1:1)`.

Figure 5.8 shows the EMADL component encapsulating the reward algorithm used to train our TORCS agent. Conforming with the state space of TORCS, it has a state input typed as `Q^{29}`. The reward type is not constrained, i.e. typed as `Q`. The MontiMath implementation of the reward function is given in L.7-14. For the sake of convenience, the variables of interest for the computation of the reward function are extracted from the `state` vector and assigned to local variables with meaningful names in L.8-10. In this concrete example we require the speed of the vehicle, its angle to the track axis, as well as the normalized distance between the car and the track axis w.r.t. the track width. Since we want our vehicle to drive as fast as possible and to exhibit the least

possible deviation from the track, we reward high velocities and a low angle to the track by multiplying the speed with the cosine of the vehicle's angle in L.11. This means, on the other hand, that we penalize the agent for increasing angles and low speeds. What is more, leaving the track leads to a hard penalty of -20 in L.13. This is the case if the absolute value of the normalized distance to the track axis is greater than 1, which is checked in L.12.

As was mentioned above, the overall RL model and training infrastructure needs to be aware of the reward component, although it is not part of the runtime agent C&C model. We accomplish this similarly to the integration of the critic component by referencing the fully qualified component name in the training model. For the reward this is done in L.13 of Figure 5.7 setting the parameter `reward_function` to `torcs.agent.network.Reward`.

At training time, the state vector is passed to the specified reward component after each step in order to retrieve a numeric reward. If needed, the reward component can be instantiated and connected inside the agent's runtime architecture, as well. The modeler does not have to model a reward component if the reward can be provided by an external source, e.g. the environment. In this case, instead of setting the `reward_function` parameter to an EMADL component, the developer can assign a ROS topic to the training parameter `environment.ros_interface.reward_topic` in the CNNTrain model.

5.7 Environment

The environment is a central building block of any RL application. In the training phase the agent is trained by submitting its actions to the environment, assessing the resulting state changes using a reward function and by updating the agent according to the training algorithm. The final, trained model can then be used in the same or a similar environment for its actual purpose. The environment can be realized as an EMADL component implementing the required interface. In this case the developer needs to set the `environment` parameter to the fully qualified name of the component in the CNNTrain model. Through its interface the environment component has to provide the following data:

T state. The current state of the environment. The type is generic and mostly a rational feature vector or matrix depending on the application.

Boolean isTerminal. A Boolean flag indicating whether a terminal state has been reached.

Q reward. An optional port representing the reward for the last action. It is only used if no explicit reward component has been provided by the modeler as described in Section 5.6.3. If this port is not present, the designer must model an explicit reward function and provide it as an EMADL component.

Furthermore, the environment component must be able to consume the following data through its input ports:

U action. Represents the action information sent by the agent in each step. In discrete action spaces, `U` is an enum or integer-based type, while it can be a rational vector for continuous problems. After sending an action message, the actor expects a response from the environment, serving the next state, the terminal information and a reward.

Boolean reset. A simulated environment should offer a reset port. If it receives a Boolean `true`, the environment undertakes a clean restart. As a response, the environment should emit its initial state, a terminal flag set to `false` and, optionally, a reward.

In the training phase, our toolchain will use the environment component in a synchronous, blocking manner, i.e. the trainer will wait for the actor to produce an action, forward this action to the environment, execute one step and forward the state back to the actor to produce the next action. This is different from asynchronous environments, e.g. the real world, where environment and training advance independently.

Mostly however, instead of defining an own environment in an EMADL component, we rather want to integrate an existing external software to train an agent, e.g. TORCS. Furthermore, an agent is often trained in a simulated environment to speed up training, to reduce costs, and for safety reasons, but is intended for a final deployment in a real CPS. For this reason, it is crucial to maintain a loose coupling to the environment and to be able to exchange it whenever needed. We realize this requirement by integrating environments through the publish/subscribe pattern. In particular, we employ the ROS middleware in our implementation [QGC+09]. Any application can be employed as a training or execution environment for MontiAnna if it provides a ROS interface using ROS topics, which can be mapped to the component interface described above. In this case, the environment needs to be set to `ros_interface` in the CNNTrain model.

The concrete ROS topic names used by the external simulator can differ from the port names listed above. A mapping is defined in the CNNTrain file by setting the following nested topic parameters for the `ros_interface`: `state_topic`, `terminal_state_topic`, `action_topic`, `reset_topic`, and, optionally, `reward_topic`. If a simulator does not support ROS out of the box, it can be made compatible with the MontiAnna framework by providing a ROS adapter.

The usage of a ROS interface is modeled in L.7-12 of Figure 5.7 in the context of our TORCS example. After setting the environment to have a ROS interface in L.7, the names of the state, terminal, action, and reset topics are set in L.8-11. This information is used at training time to connect the trainer with the environment.

The final, trained system can then be connected with any other environment using the middleware tagging approach, which will be discussed in Chapter 6. The concrete

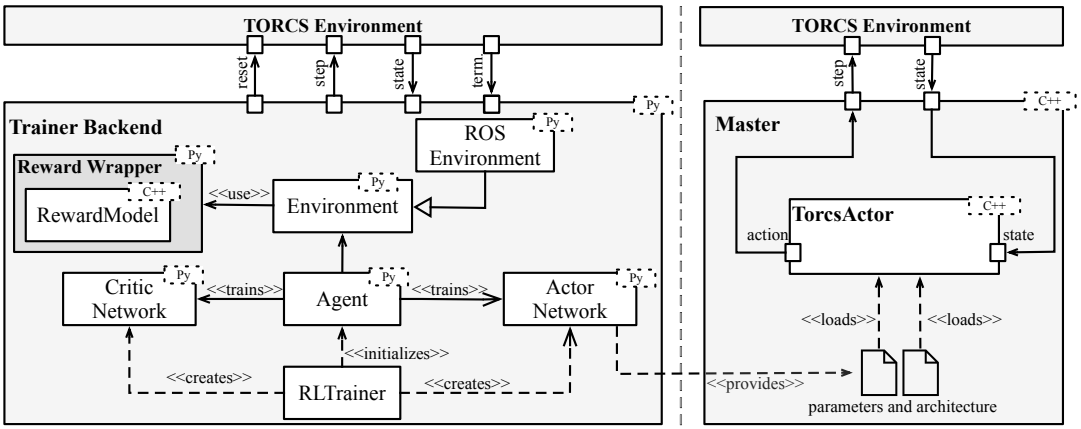


Figure 5.9: Generated artifacts: the left side shows the artifact architecture during the training. The right side is the artifact architecture at the execution time of the model [GKR19].

middleware is not fixed by design. While ROS was chosen for the reference implementation due to its wide usage in the domain, any other publish/subscribe middleware could be employed, as well. Note that in contrast to the synchronous environment execution described for EMADL environments, external simulators might run asynchronously, i.e. without waiting for new action messages to arrive. This means that they might provide more state and reward messages than the trainer can handle. This can degrade the training quality drastically if not taken into account.

As a third way of environment integration, we provide built-in environments of OpenAI Gym, which can be used by setting the `environment` parameter to `gym` with a nested string-typed parameter `name` identifying the concrete environment. Some examples are: `"BipedalWalker-v2"`, `"CartPole-v0"`, and `"RoboschoolHalfCheetah-v1"`. Further built-in environments can be added easily to the toolchain as long as they are compatible to the Python interface of OpenAI Gym environments.

5.8 Code Generation

The generation of RL models is inherently different from supervised learning models. Therefore, the EMADL code generator uses a dedicated RL generation pipeline to generate and train a fully functional RL system from an EMADL model. This generation mode is activated by setting the parameter `learning_method` to `reinforcement` in the training model as is done in L.4 of Figure 5.7.

First of all, the reinforcement learning pipeline comes along with a set of RL specific context conditions which need to be checked in addition to the standard MontiAnna

context conditions. Of particular interest are context conditions on the training model as well as the composed architecture ensuring a meaningful algorithm set-up and the conformity of the role component interfaces. A selection of context conditions is listed in the following:

Reward integration. As discussed, the reward for an action can be computed by an explicitly provided EMADL component or obtained from a remote application through a middleware. This context condition ensures that a CNNTrain model defines exactly one reward source.

Actor/critic conformance. For actor/critic algorithms such as DDPG, the context condition ensures that a critic component is provided in the training model and that the referenced component exists. Furthermore, the context condition ensures that the types and dimensions of the state and action ports of the critic component correspond to those of the actor.

Problem domain typing. This context condition ensures that the types of the action and the state spaces are compatible with the chosen RL algorithm. DQN can only be used for discrete action spaces; DDPG can only be used for continuous action spaces.

Exploration strategy. The exploration strategy must be suited for the chosen action space. If the action space has a continuous type, e.g. in the case of DDPG and TD3, the exploration strategy must be either OU or Gaussian noise. Otherwise, i.e. if the type of the action space is discrete, the exploration strategy has to be ϵ-greedy.

Fixed target network. If DQN is used and `use_fixed_target_network` is set to `true`, this context condition ensures that the corresponding target weights update interval, i.e. the property `target_network_update_interval`, is set as well.

If all context conditions are successful, a fully functional solution including the training scripts and the final deliverable code is created. The target software architecture including the generated artifacts is depicted in Figure 5.9. The figure separates training related artifacts on the lhs from the runtime artifacts on the rhs. As in the supervised learning pipeline, the training program is generated as Python code using the chosen deep learning backend. The evaluation of this pipeline was mostly conducted using MXNet Gluon.

The `RLTrainer` module is the entry point for the training phase; it creates the neural networks modeled in MontiAnna and maintains a parameter store realized as a Python dictionary holding the parameters and values of the training model.

The created neural networks as well as the training parameters are passed to the `Agent` module encapsulating the actual, executable RL training algorithm to be applied.

This agent module interacts with the environment in the course of the training phase in order to exchange states and actions. If an explicit reward component is referenced in the training model, EMADL2CPP generates C++ code for this model using its EMAM2CPP subgenerator. This code is then compiled to an executable. To integrate it into the Python-based trainer, it is wrapped and instantiated by a Python-based reward wrapper module providing access to the reward interface. This enables the trainer to pass states to the reward component and to receive its reward values.

Furthermore, the training system keeps track of training related information such as the actor loss and the average reward. Once training is finished, the learned weight parameters of the neural network are serialized to a file which can be loaded by the predictor of the final executable model, i.e. the `TorcsActor` on the rhs of Figure 5.9. At this point the target architecture, here represented by the `Master` component, is ready to be deployed and executed. Further statistical information about the training process is saved for analysis and tuning in a separate report.

5.9 Modeling Generative Adversarial Networks

In this section we are going to introduce another generation and training pipeline provided by MontiAnna to design and process GANs. Although GANs are not directly related to RL, the two areas share the commonality that at training time a helper network is used. Therefore, we are going to reuse the component role concept introduced in this chapter to integrate critic networks and rewards into an RL training procedure for GANs. The basic idea behind the GAN concept is to train a *generator* network to produce data according to some training distribution. To do so, the generator is trained in an interaction with an adversary network, the *discriminator*. While the goal of the discriminator is to be able to decide correctly whether a given example is real or fake, the aim of the generator is to be able to create examples which cannot be distinguished from real data by the discriminator [GPAM+14]. GANs can be employed to enrich sparse datasets by additional artificial but genuinely looking data. In software engineering this technique can potentially be used for test data generation and augmentation. The GAN concept has already been applied to automotive problems such as the recovery of blurred road markings [LLH+19], the generation of cyclist images [ZHL+19], as well as for the post-processing of aerial photos to obtain map images [IZZE17]. Hence, we consider GANs as an important tool for modern software engineering processes, particularly in the area of testing and simulation. The GAN pipeline of MontiAnna has been evaluated primarily on the family of deep convolutional generative adversarial networks (DCGANs) producing image data and where both the generator and discriminator are CNNs [RMC15].

The GAN training pipeline can be selected in the training model by setting the learning method accordingly as `learning_method=gan`. Similarly to the reinforcement

learning pipeline the GAN pipeline supports the standard supervised learning parameters in the training model, but adds additional GAN-specific parameters. In particular, this includes the configuration of the discriminator network, which, similar to a critic network in the RL pipeline, is only necessary during training:

EMADLComponent discriminator_name. The fully qualified name of the EMADL component encapsulating the discriminator network. This component is only used during the training phase.

nested discriminator_optimizer. This parameter is derived from the supervised learning parameter `optimizer` and has the same type, but is related to the optimization algorithm of the discriminator network.

enum discriminator_loss. This parameter is derived from the supervised learning parameter `loss` and has the same type, but specifies the loss function of the discriminator network.

The discriminator component has to provide a valid discriminator interface exposing two input ports: the port `T data` takes the output of the generator as its input, where `T` is the corresponding, application-specific data type; another input port `U target_label` takes the label which was used to create the data by the generator network. It can have an arbitrary numeric type `U`, e.g. a ten-dimensional vector in the case of handwritten-digit generation or a whole image for image-to-image mappings. Furthermore, a discriminator network is expected to have an output telling whether the given image is genuine or not. 0 represents a false and 1 a genuine example. As is common in classification tasks, the discriminator outputs a value between 0 and 1, which can be interpreted as the probability of the given example being genuine.

The bottom line is that the role-based approach can be used to implement different training pipelines. However, the more pipeline variability is needed, the more complexity is introduced in the generator, the training configuration, and the context conditions and the more desirable a pipeline modeling approach gets. A schema language could, furthermore, facilitate the configuration parameter management of the CNNTrain language, e.g. by providing a modeling means to specify which parameters can be combined and which are mutually exclusive.

5.10 Evaluation

5.10.1 TORCS and Open AI Gym

The evaluation of the RL modeling extension and the corresponding generation and training pipeline were mainly evaluated using the running example of this chapter, where an agent was modeled and trained to control a TORCS vehicle using the DDPG algorithm.

EMADL

```
1  package torcs.agent.network;
2  component TorcsCritic {
3    ports in  Q^{29} state,
4          in  Q(-1:1)^{3} action,
5          out Q(-oo:oo)^{1} qvalue;
6
7    implementation CNN {
8      (state | action)->
9       Concatenate() ->
10      FullyConnected(units=300) ->
11      Relu() ->
12      FullyConnected(units=600) ->
13      Relu() ->
14      FullyConnected(units=1) ->
15      qvalue; } }
```

Figure 5.10: EMADL component of the TD3 critic for the TORCS reinforcement learning model.

While the result is similar to the one of Section 4.10.4, the RL approach did not require any labeled data. The agent was trained solely by exploring the action space in the simulator. For training we used Gym-TORCS[4], a python wrapper for TORCS exhibiting an OpenAI-Gym-like interface. To make Gym-TORCS compatible with the ROS-based training interface of MontiAnna, we developed a ROS adapter for a ROS-based data exchange. This way the code generated from our machine learning models using the RL pipeline can be used for training and execution without any adaptations or inclusion of hand-written GPL code. The models of the actor and the critic network as well as the training configuration and the reward were given as examples in the course of this chapter, cf. Figures 5.4, 5.5, 5.7 and 5.8. In addition to DDPG, which turned out to perform poor for this example, an agent was trained using TD3. For TD3 training, the critic model differs slightly and is given in Figure 5.10. What is more, the value of the parameter `rl_algorithm` needs to be changed to `td3-algorithm` in L.5 of the CNNTrain model in Figure 5.7.

We used the CG track 2 with a length of 3185.83 m for training and the CG Speedway number 1 with a length of 2057.56 m for evaluation, cf. Figure 5.11. The DDPG variant had to be trained for 3500 episodes to achieve rewards of up to 100.000-150.000. What sounds like a large number, was not enough to control the vehicle correctly for more than a few seconds. The better performing TD3-based agent achieved rewards of over 500.000 after less than 2000 episodes while able to drive collision-free for multiple laps on both the training and the evaluation maps.

[4]https://github.com/ugo-nama-kun/gym_torcs, accessed March 01, 2021

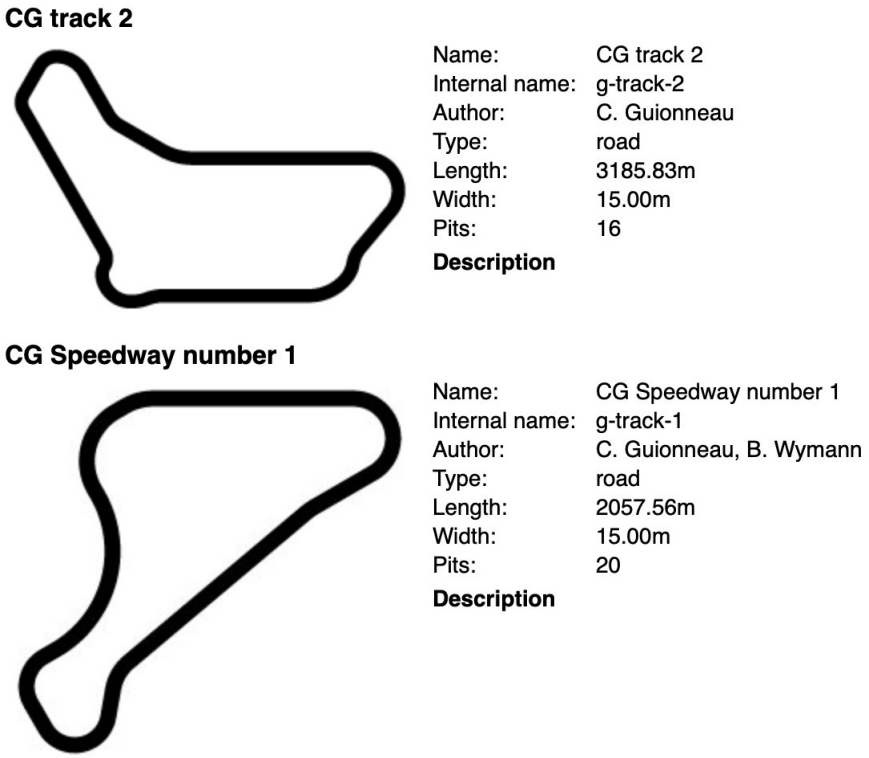


Figure 5.11: The racing tracks used for training (top) and evaluation (bottom) of the TORCS agent. Copied from the TORCS racing board website `http://www.berniw.org/trb/tracks/tracklist.php`, accessed April 26, 2021.

In addition to the TORCS experiment, agents for the OpenAI Gym environments[5], namely, CartPole, Lander (discrete and continuous), Pendulum, HalfCheetah, Bipedal Walker, and Atari Pong were modeled and trained successfully by Nicola Gatto in his master thesis using the presented methodology. Due to the similarity of the used neural networks we omit the presentation of the individual models. The training results (average rewards) are wrapped up in Figure B.1 in the appendix and are comparable with the ones provided by OpenAI. These experiments show that the presented methodology is applicable to reinforcement learning and can operate with a wide range of training environments.

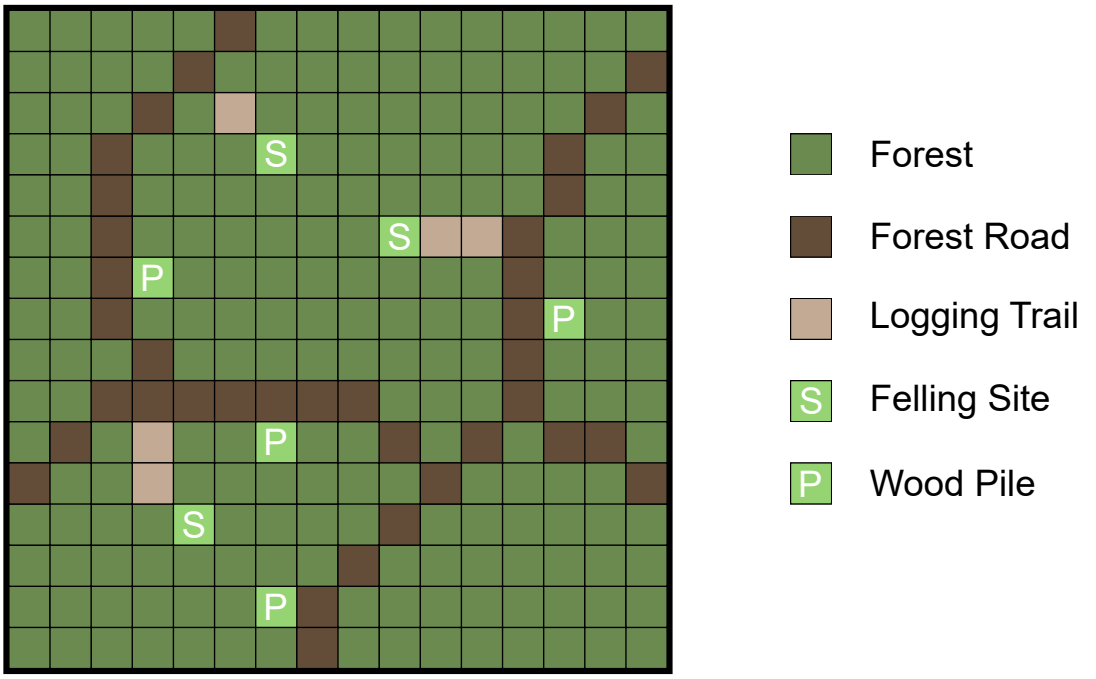


Figure 5.12: An example map of the CTL forestry simulator by WZL.

5.10.2 Decision Making in Forestry 5.0

A further evaluation of MontiAnna for reinforcement learning was conducted in the industry 5.0 or, more specifically, the forestry 5.0 domain in collaboration with Stephan Wein from the Werkzeugmaschinenlabor (WZL) of the RWTH Aachen University. While the applications described above use RL to train low-level controllers, in this example our goal was to obtain a more abstract decision making network dealing with job planning tasks for cut-to-length logging (CTL), thereby optimizing economical and ecological performance indicators. CTL is the predominant harvesting system in Europe, where a *harvester* fells and cuts trees into logs that are stored on the ground until a *forwarder* picks them up and carries them to the landing sites [Rin11]. Causes of logging trail damage in forestry were discussed in [Hil14]. Soil compaction can be caused by heavy machinery moving over the logging trails. The severity of compaction depends on the machinery weight, but also on changing soil properties such as moisture. This needs to be taken into account in the planning process to avoid or at least reduce possible damages.

WZL provided a simulator for this work enabling us to simulate the CTL process as well as the caused soil compaction. In the simulation, a harvester first fells the trees

[5]https://gym.openai.com/envs/, accessed March 1, 2021

and cuts them into logs of a certain length. Afterwards, a forwarder transports these logs to a wood pile associated with the corresponding job. The agent trained in this evaluation was meant to control the forwarder, not the harvester, i.e. it was dealing with the transportation part exclusively. Over the course of an episode the simulator increases the supply of the felling sites in random intervals to imitate a continuous felling of new trees. The simulated environment consists of a virtual representation of a forest and a forwarder that can be controlled by the agent. The forest is modeled as a 32×32 grid map, where each cell represents one of the following partition types: forest, road, logging trail, felling site, or wood pile. An example map is given in Figure 5.12. The map is fixed at the beginning of an episode. Each episode consists of two phases. First, the agent has to choose a finite number of jobs from a list provided by the simulator and to decide, where to put the wood piles for these jobs. In the second phase, the agent carries out the selected jobs by delivering logs to their associated wood piles. To finish a job, the required number of logs of a given wood type and length need to be delivered. At each felling site exactly one type of wood and a fixed log length are available. Environmental damages can be estimated by the agent based on the logging trail properties, which are also provided by the simulator. As an additional constraint, each job has a deadline. Exceeding a deadline leads to additional fines. Once all jobs are completed, the episode is over and the next episode is started.

The simulator accepts six different types of actions, which we are going to discuss in the following:

Job selection (phase 1). The simulator presents the job offers as an $8 \times N$ matrix, where N is the total number of available jobs. Each job offer contains 8 properties including the required log length and type, the pile coordinates (x and y), the total demand, offered reward, the deadline, as well as the penalty for missing the latter. To select jobs the agent needs to send an indicator array, where a 0 represents a declined and a 1 an accepted job.

Pile placement (phase 1). Once the agent has chosen the jobs, it will assign pile positions to them. Wood piles may only be placed in unused parts of the forest, e.g. the piles cannot be placed on a road. What is more, the piles need to be accessible by a road so that the forwarder is able to reach them.

Forwarder movement (phase 2). The remaining actions are available in the second phase of an episode. To move a forwarder the agent needs to specify a destination. The simulator then checks if it is actually reachable, computes a path using the A* algorithm and updates the forwarder position. Thereby, it records the time taken as well as the fines for caused logging trail damage.

Loading wood (phase 2). The forwarder can specify the number of logs it wants to load. The simulator then checks, whether the forwarder is actually at a felling

EMADL

```
package forestrl.agent.networks;

component ForestActor {
    ports
        in Q^{488} state,
        out Q(-1:1)^{63} action;

    implementation CNN {
        state ->
        FullyConnected(units=400) ->
        Relu() ->
        FullyConnected(units=300) ->
        Relu() ->
        FullyConnected(units=63) ->
        Tanh() ->
        action;
    }
}
```

EMADL

```
package forestrl.agent.networks;

component ForestCritic {
    ports
        in Q^{488} state,
        in Q(-1,1)^{63} action,
        out Q qvalue;

    implementation CNN {
        (state | action)->
        Concatenate() ->
        FullyConnected(units=400) ->
        Relu() ->
        FullyConnected(units=300) ->
        Relu() ->
        FullyConnected(units=1) ->
        qvalue;
    }
}
```

Figure 5.13: Actor and critic networks of the forestry 5.0 agent.

```
configuration ForestActor {
  context: cpu
  learning_method: reinforcement
  rl_algorithm: td3-algorithm
  critic: forestrl.singlestep.agent.networks.forestCritic
  environment: ros_interface {
    state_topic: "/preprocessor/state"
    terminal_state_topic: "/sim/terminal"
    reward_topic: "/sim/reward"
    action_topic: "/postprocessor/action"
    reset_topic: "/sim/reset" }
  discount_factor: 0.7
  policy_noise: 0.2
  noise_clip: 0.5
  policy_delay: 2
  num_episodes: 1
  start_training_at: 1
  num_max_steps: 10000
  training_interval: 1
  snapshot_interval: 50
  evaluation_samples: 5
  soft_target_update_rate: 0.005
  replay_memory: buffer{ memory_size : 100000
    sample_size : 100 }
  strategy : gaussian { epsilon : 1.0
    min_epsilon : 0.05
    epsilon_decay_method: linear
    epsilon_decay_start: 500
    epsilon_decay : 0.005
    epsilon_decay_per_step: false
    noise_variance : 0.1 }
  actor_optimizer : sgd { learning_rate : 0.002 }
  critic_optimizer : sgd { learning_rate : 0.0005 }
}
```


Figure 5.14: Training configuration of the forestry 5.0 agent.

site, whether the felling site has enough supply, and whether the forwarder's bunk capacity is sufficient for the requested load. If at least one of these checks fails, the action is ignored. Loading consumes time. The time needed to finish the action depends on the number of logs to be loaded as well as the crane capacity of the forwarder specifying how many logs the crane can load onto the bunk of the forwarder at once.

Unloading wood (phase 2). Similarly to loading wood, the forwarder can unload wood by specifying the amount to unload. The simulator checks whether the forwarder is at a wood pile, whether the loaded wood type and log length match those of the pile, and whether the number of logs to unload does not exceed the number of loaded logs. If the action is valid, the simulator will update the number of logs on the bunk of the forwarder and advance the time in the same way as for loading.

Waiting (phase 2). The forwarder can remain idle for a specified amount of time.

Once an action has been submitted and executed, the simulator checks for missed deadlines, updates the trail and supply information and, finally, returns an updated state, a reward, as well as a Boolean flag indicating whether the new state is a terminal state.

The actor and critic networks are given in Figure 5.13. For the actor, we use three fully connected layers followed by ReLU and tanh activation functions, respectively. The structure of the critic is similar, but it omits the final tanh layer and outputs only one scalar Q-value. The agent is trained using the TD3 algorithm. For both networks we use the SGD optimizer. The connection with the simulator is realized using the ROS interface. The corresponding training configuration including all hyperparameters is given in Figure 5.14. To asses the quality of the agent, the total monetary reward consisting of rewards for finishing jobs and penalties for missed deadlines and the soil damage caused to the logging trails is identified as the main performance indicator. We implement the reward function for an action as the total monetary reward added after a simulation step. Although time is not explicitly present in this definition, it is still represented through the missed deadline fines. The training results are depicted in Figure B.2. The agent was evaluated against a randomly acting and a simple rule-based agent. While the trained agent is on par with the rule-based agent when we compare the total monetary reward, it outperforms the rule-based agent with regard to money-to-time ratio, which is 6.23 for the RL agent and 5.53 for the rule-based agent in our experiment. Needless to say, both agents outperform the randomly acting agent clearly with regard to both measures (money and money-to-time ratio). Further tuning of hyperparameters and more complex networks with more layers and/or neurons per layer can probably further improve the quality of the agent.

5.11 Conclusion and Future work

In Chapters 4 and 5 we have introduced the MontiAnna language family and discussed how it is integrated with the EMA language family. It has been shown that the MontiAnna framework can be used to model neural processing components for the usage in CPSs including supervised learning, reinforcement learning, and generative adversarial networks. To cover these three training types, MontiAnna provides three independent generation and training pipelines. However, there is no difference at syntax level, the modeler uses the same modeling language family for all types of networks, which means that models can be reused easily. The differences are hidden in the actual, generated training process and the context conditions ensuring that the training set-up is semantically correct.

The differences in structure of the generated training processes for the three supported training pipelines are hidden in the code templates of the generator. Adapting a training pipeline therefore requires adapting the respective templates. This is easy to manage for a small number of training pipelines. However, if the demand for training pipeline variability grows, models capturing the steps and dependencies of the training process explicitly will become inevitable and hence are subject of future work. C&C modeling could then be used to define formal reference architectures explicitly capturing the dataflows between the training components and to generate context conditions automatically, e.g. checking whether the actor and the critic networks are compatible. More elaborate pipelines should be captured by training workflow models, e.g. if multiple nets have to be trained in a specific order or if they provide data for each other.

The MontiAnna framework has been evaluated on a variety of different ANN architectures and architectural styles ranging from simple MLP architectures and widely used CNNs such as AlexNet and the VGG16 to recurrent networks including encoder-decoder models with and without attention as well as mixes of convolutional and recurrent architectures. The three training pipelines have been applied successfully to tackle objection detection, control, natural language translation, segmentation, and test data generation tasks using supervised, reinforcement, and adversarial learning.

Embedding MontiAnna into the component-based language family EMA makes artificial neural networks first level citizens in a software architecture enabling us to automate the training process and data management. Avoiding unnecessary retraining and being able to package and publish weights enhances the reusability of compiler results.

For now, the MontiAnna framework is mainly focused on deep ANNs. Other learning techniques such as support vector machines (SVMs), decision trees, Bayesian methods, and clustering are not covered by MontiAnna. However, deep learning is not appropriate for many learning tasks. For instance, learning of error patterns from diagnosis trouble codes (DTCs) in the automotive domain can be handled using unsupervised methods such as spectral clustering [KKRS19]. An extension of MontiAnna by further supervised and unsupervised learning techniques is therefore highly desirable and subject of further

research. A combination of deep learning with such techniques is desirable as well, e.g. using SVMs in the last layer of a neural network [Tan13].

Chapter 6

Modeling Distributed Architectures

6.1 The Need for Distributed Systems

So far we have been dealing with holistic EMA architectures, which are meant to be executed as a single program on a target device. Communication between subcomponents can therefore be realized in the target language as standard function calls, which is computationally very cheap and efficient. In practice however, CPS architectures are often highly distributed. Different parts of the system are developed by dedicated expert teams and are deployed on multiple computation nodes connected by a network. A smooth communication is then best realized using a *middleware*. A middleware not only abstracts away from the technical nature of the underlying network protocols, thereby facilitating the implementation of the distributed communication, but also ensures a low coupling between components.

In this chapter, we are going to discuss how middleware aspects can be modeled in the scope of the EMA language family, partially based on [HKKR19]. In particular, we pursue the following goals:

- Integrating EMA models with external third party components such as sensors, actuators, simulators, and the like. In this case only ports at the system border need middleware capabilities. Ports of subcomponents can still be realized using standard function calls.
- Using EMA to model distributed systems whose subcomponents need to run in separate processes, e.g. on different ECUs of a vehicle, communicating via a middleware.
- Automating deployment scheme searches optimizing the partitioning of large models with respect to some cost function and a set of given constraints. For instance, if an EMA model has to be deployed on multiple ECUs, the subcomponents need to be assigned to these ECUs so that the amount of necessary communication is minimized.

The research questions to be answered in this chapter are the following:

Research Question 5. *How can middleware communication be modeled in a C&C-based development methodology at SMArDT level 3?*

Research Question 6. *How can the distribution of components be automated at SMArDT levels 3 and 4?*

6.2 Existing Approaches for Middleware Integration

Given the tremendous necessity of middleware, most methodologies for cyber-physical system design support some kind of middleware modeling. In the following, we are going to discuss several relevant approaches:

Simulink. In Simulink [Mat16] all kinds of network communication, including middleware, are modeled using dedicated blocks providing the functionality. For instance, ROS communication is provided by the Robotics System Toolbox. The developer needs to use the *Blank Message* block to define empty messages. These messages can then be modified using bus assignments. Publishing and subscribing to ROS topics is realized via the *Publish* and *Subscribe* blocks, respectively. This approach has a major drawback: developers can mix up middleware aspects with business logic, thereby creating dirty models with side effects. Such models violate the separation of concerns principle and are difficult to maintain and to test. In the simplest case, where only the boundary ports need to exchange data via a middleware, this can be avoided by using the wrapper pattern, i.e. the logical component is wrapped by an infrastructure component connecting the ports of the former to *Publish/Subscribe* blocks. However, the situation becomes more intricate if middleware communication is deeply intertwined with logical components inside the model. This is referred to as code *scattering* and *tangling* and can make it difficult to exchange a middleware configuration. Furthermore, middleware contamination can hinder black box testing of logical components.

This can get particularly painful in layered development processes with multiple abstraction levels such as SMArDT: each development stage might require a specific quality assurance strategy for the abstraction level under consideration, e.g. model in the loop (MiL), software in the loop (SiL), and hardware in the loop (HiL) testing. Furthermore, different aspects of a model might need to be tested in different simulators. Model integration might therefore vary between development stages. The same may hold for different product variants, e.g. if variants of a CPS use different operating systems or hardware. Hence, if middleware aspects are highly intertwined with the logical model, the maintenance of a series of integration scheme variants will require unbearable efforts.

Platform-based Design. Another component-based modeling approach from the domain of embedded systems focusing on the orthogonalization of concerns is platform-based design (PBD) [KNRSV00]. A platform is a layer of abstraction hiding its implementation details from layers using it; a design is organized as a stack of layers and their mappings. Each platform is composed of behavior and communication model components. The latter can be used to model middleware aspects. Hence, similar to Simulink, communication components can lead to a middleware contamination of behavioral models.

In PBD multiple components of a platform can solve the same problem and can therefore be interchangeable. The selection of the best alternative is performed automatically using search space exploration according to specified constraints. Such a selection is called a *platform instance* and is used to generate code for the system. If a standard platform with different middleware communication components is created, a user can exchange the middleware by changing a single constraint, leading to the desired platform instance without touching behavior-related parts of the model. Metro II [DDM+07, BDD+09] is a framework designed for PBD. It contains a code generator targeting C++, an integrated simulator, and tools for formal verification.

MontiArcAutomaton. MontiArcAutomaton [Wor16] is a language family for C&C modeling. Due to its relation to MontiArc it has a syntax very similar to EMA. However, it exhibits differences in its semantics and targets the Java ecosystem, including the type system and generated code. The behavior of a MontiArcAutomaton component can be modeled as an automaton or be implemented directly as hand-written code in Java [AHRW17]. MontiArcAutomaton focuses on logical aspects of a system, as well. The models are completely free of middleware aspects. To employ MontiArcAutomaton in the robotics domain, the authors propose a generator concept, which flattens the component hierarchy and realizes the remaining components as ROS nodes and the connectors as ROS topics in the target language. This leads to fully distributed architectures, where each component can run in an independent process. The approach is very easy to use as the modeler does not have to model the middleware at all. On the other hand, hiding the complexity comes with the drawback that the developer is not able to define custom middleware schemes. It is not possible to select a subset of ports to be generated as ROS nodes or to map different topic names to each other. Furthermore, the integration of multiple middleware solutions in one model is not covered by the proposed concept.

RobotML. RobotML is an EMF-based [SBMP08] graphical modeling framework for the robotics domain providing code generation for a variety of robotics platforms, e.g. OROCOS-RTT and RTMaps [DKS+12]. The designer starts with a platform-independent model (PIM) describing the functionality of the system and omitting technical details.

```
1  component IntersectionController(N n) {
2    ports in Time timeCutoff,
3          in Trajectory trajectory[n],
4          in Boolean isActive,
5          out Boolean stop[n];
6
7    instance SingleSetCompare compare;
8    instance TrajectoryCollision trajCollision[n*(n-1)/2];
9    instance CollisionToStop collisionToStop;
10
11   connect trajectory[:] -> compare.trajectoryIn[:];
12   connect timeCutoff -> trajCollision[:].timeCutoff;
13   connect collisionToStop.stop[:] -> stop[:];
14   /* further connections */   }
```

EMA

ports at system border to be adapted to a middleware

components to be distributed across multiple ECUs

Figure 6.1: EMA definition of an intersection controller component receiving trajectories from nearby vehicles, checking for potential collisions and sending stop signals to endangered vehicles [HKKR19].

Afterwards, a deployment-platform model (DPM) needs to be created. It contains advices for the code generation toolchain concerning the middleware and simulators to use. The code generation toolchain consists of several subgenerators supporting a variety of robotic middleware solutions and simulators. Hence, to exchange the middleware, the developer does not have to touch the PIM.

6.3 Running Example and Use Cases.

Consider the EMA model in Figure 6.1 and the corresponding graphical diagram in Figure 6.2 (ignore the different port representations of the latter for now). The model captures a simple collision warning system which might be deployed at an RSU or in the manager of an LTS to support automated cooperative driving [DDE+17]. Its purpose is to warn vehicles with a possible collision ahead. Each traffic participant provides its planned trajectory as input to the model and expects a feedback either confirming the trajectory or instructing it to adapt the trajectory. For each vehicle in the network, the model provides a `trajectory` input port to receive the respective trajectory and a `stop` output port to send the feedback.

In this model we work with a fixed number of traffic participants `n` for the sake of simplicity. A scenario for `n=2` vehicles approaching an intersection is sketched in Figure 6.3. A dynamic version, where the number of traffic participants can change over time, can be realized using EMAD constructs.

Internally, the `SingleSetCompare` component creates all possible $x = n(n-1)/2$ pairs of trajectories and forwards each pair to a dedicated instance of a `Trajecto-`

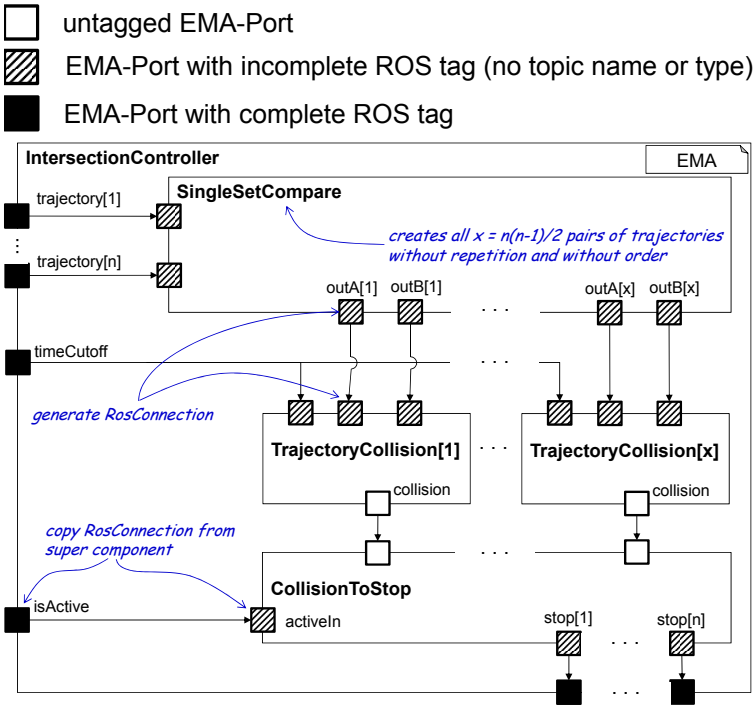


Figure 6.2: Intersection controller model with middleware tags attached to its ports [HKKR19].

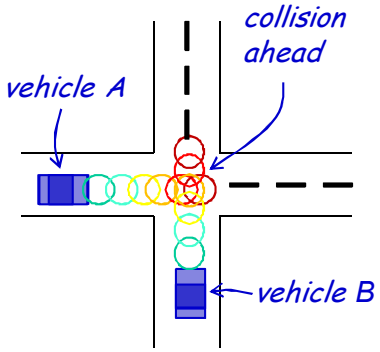


Figure 6.3: Two vehicles approaching an intersection. The planned trajectories are depicted as circles.

`ryCollision` component. The latter checks whether the two input trajectories collide and outputs the result. The `CollisionToStop` component takes the results of the x `TrajectoryCollision` components and maps them to n `stop` ports.

Obviously, traffic participants are distributed agents and have to provide their trajectory data over some network protocol or middleware before it can be fed into the outer ports of the intersection controller. Such an integration of a black box EMA component with remote (third party) software is the first use case of our tagging approach. It leaves the internals of an EMA component untouched. For the realization, we only need to provide some kind of wrappers or adapters to exchange data between the filled black ports of the model in Figure 6.2 and a network.

As a second use case, we might need to deploy the modeled components as a distributed system, e.g. if the application we are modeling consists of multiple loosely connected parts deployable on a distributed hardware platform such as an automotive or a robotics system featuring an ECU network. Of course, this can be achieved by modeling independent EMA systems or agents for each target ECU and interconnecting them using the first use case approach described above. However, by doing so, we would loose the advantages of an explicit architecture definition such as integrity and consistency checking on the distributed system level. Instead, it is much more desirable and practical to use EMA not only to model the agents, but also to capture the distributed top tier architecture of the overall system including the interaction between the agents. In this case the generator needs to produce several independently deployable artifacts running in separate processes and communicating via middleware instead of one black box system. The connectors should then be interpreted as middleware connections instead of simple function calls.

The question arises whether the synchronous execution order semantics of standard EMA as discussed in Section 2.3.3 is still a good fit for this use case. It might slow down the execution of a distributed system significantly due to delays in the communication network. Therefore, asynchronous parallelizing execution modes should be considered for this use case.

6.4 Requirements

To extend the EMA framework with middleware modeling functionality covering the two use cases discussed in the running example section, we derive a set of requirements for the modeling language as well as the underlying code generation toolchain:

(RC1) Middleware-agnostic modeling: logical EMA models must remain free of technical aspects including middleware. Middleware models must be defined as separate artifacts ensuring a clear separation of concerns and avoiding model scattering and tangling. The aim of this requirement is to ensure model reusability in different environments as well as independence of the technical realization.

(RC2) Middleware-agnostic core code generation: the EMA architecture and behavior code generators must not produce code related to technical aspects such as middleware communication to ensure a clear separation of middleware and core EMA generation concerns.

(RC3) Minimization: as middleware communication is expensive, it must not be used unless required explicitly or implicitly by the model. Dataflows not eligible for middleware communication must use the standard tightly coupled and synchronous communication pattern in the generated code, e.g. based on function calls as offered by the EMA code generator.

(RC4) Semantics invariance: EMA has an efficient compile-time scheduling mechanism and a synchronous weakly causal execution semantics which must remain preserved for non-distributed parts of the system in order to keep obtaining highly-optimized generated code.

(RC5) Middleware coupling: the integration of multiple middleware solutions in a single model must be possible, i.e. each input port should be capable of receiving data from an individually configured middleware source. Furthermore, each output port must be able to send data to arbitrarily many different middleware networks.

(RC6) Modeling distributed systems: it should be possible to use EMA to design a distributed system, i.e. given a single EMA model it should be possible to declare a distribution and communication scheme for its subcomponents.

(RC7) Build infrastructure: generating code using the EMA core code generator coupled with arbitrarily many middleware generators will probably result in a complex project and dependency structure. To ensure out-of-the-box usage, in addition to the actual code artifacts, the generator system must be able to generate a build infrastructure taking care of all project dependencies.

6.5 Tagging-Based Middleware Modeling

We have several possibilities to design a middleware modeling extension for EMA. First, by providing middleware component libraries similar to the Simulink Robotics Toolbox, middleware connections could be modeled as normal EMA components. Such a solution would lead to *scattering* and *tangling*, i.e. a single aspect is distributed across the whole model and each component polluted with middleware becomes unnecessarily complex. Modularity, reuse and testability are highly constrained.

Second, EMA could be extended by new language constructs to cover middleware connectivity. For instance, a port declaration could be extended with a modifier, optionally parameterized with the middleware to use. This solution has the potential to enhance

readability compared to using middleware components, since middleware information present in a model could be seen at one glance in the interface definition. Scattering and tangling are barely reduced in such an approach and the solution still suffers from bad modularity and variability support while making the language more complex.

A modular solution can only be achieved by capturing technical communication and middleware aspects in a dedicated model type which is clearly separated from the logical model. Furthermore, it is desirable to be able to compose the logical model with the middleware model, e.g. to enable navigation from a port symbol directly to its optional middleware information to facilitate model analysis and code generation. Instead of developing a dedicated DSL, we reuse the *tagging* mechanism of EMA. The main idea is to enrich ports with middleware related tags, which can be used as advices by the code generator to produce the corresponding communication code. These tags may include the type of middleware or protocol to use as well as further technical, possibly platform-dependent details[1].

By applying the tagging concept to our problem, we benefit from the following useful properties:

- All middleware information related to an EMA model is gathered in one textual model; a developer can quickly get an overview of all ports communicating via a middleware.
- The tagging mechanism ensures that a tagged element exists and is of correct type.
- All middleware information is attached directly to the symbol table of the tagged EMA model and is, hence, easy to find.

Since each middleware follows its own specific communication pattern in combination with proprietary parameters, we need to define a tag type for each supported middleware in a tag schema. In this work we focus on publisher/subscriber middleware employed in various application domains from robotics and IoT to automotive. The idea behind this communication pattern is that senders, instead of sending messages to the designated recipients directly, publish messages to *topics*. Receivers can subscribe to topics they are interested in to receive the messages. The communication is usually realized over a message broker known to all participants. The broker manages the participants of the network and forwards messages to their subscribers. Hence, communication participants do not need to know each other, which ensures a low coupling and a high degree of extensibility. Prominent examples of publisher/subscriber middleware are ROS in the robotics domain [QGC+09], Message Queuing Telemetry Transport (MQTT) for IoT applications [BG14, Nai17], MQTT-S for wireless sensor networks (WSNs) [HTSC08]

[1] Recall that we also used the tagging mechanism in Section 4.10.2 to model the technical aspects of the training dataset for an EMADL component, which are mainly relevant for the code generation, as well.

Tag Schema

```
1  tagschema MiddlewareToEmamTagSchema  {
2    tagtype RosConnection {
3      (topicName = (${topicName:String},
4        topicType =  ${topicType:String})
5      (, msgField = ${msgField:String})?)?
6    } for PortInst, ComponentInst ;
7
8    tagtype MqttConnection {
9      (topicName = ${topicName:String})?
10   } for PortInst, ComponentInst ;
11 }
```

Figure 6.4: Tag schema defining the middleware tag structure for EMA component instances and ports.

and Scalable Service-Oriented Middleware for IP (SOME/IP) [SSG+15, HPHC16] as well as OpenDaVinci [Ber10, Ber16] in the automotive domain. The middleware modeling and generation concept discussed in this chapter can be applied to any such frameworks.

We define the tag schema for the modeling of middleware-based communication in Figure 6.4. The schema contains two separate tags for ROS and MQTT. Consider the definition of the first tag type named `RosConnection` in L.2-6. The syntax of the tag is defined in L.3-5. Special characters are interpreted as MontiCore syntax. A `$` initiates a placeholder for the actual data to be input by the user and the accepted data type for this entry. In a `RosConnection` tag, the whole body is optional due to the outer parentheses group followed by a `?`. Furthermore, the entry `msgField` is optional if `topicName` and `topicType` are given. The `for` keyword in L.6 specifies which symbol kinds this tag type can be applied to. In this case it is applicable to port and to component instances. By tagging a port instance with such a tag, we indicate that the communication of this port is realized through ROS. Alternatively, we can apply the tag to a component instance. In this case all ports of the tagged component will communicate through the middleware. In both cases, the middleware communication is parameterized using the three parameters mentioned above. The `topicName` parameter sets the topic the tagged port will listen to (in the case of an input port) or to which it will publish its data (in the case of an output port). The topic type can be set to one of the predefined ROS message types or a composed type (similar to struct types in C/C++). Since ROS and EMA have different type systems, a conversion concept is necessary. However, we can reuse the conversion table used by the EMAM2CPP code generator, cf. Section 2.5, since ROS types have a straightforward mapping to C++ types. Furthermore, a message field (`msgField`) can be specified. If the topic type is a struct-like composed type, but the receiver is only interested in a specific field, it can access it by setting the `msgField` parameter. Graphically, we depict tags with topic

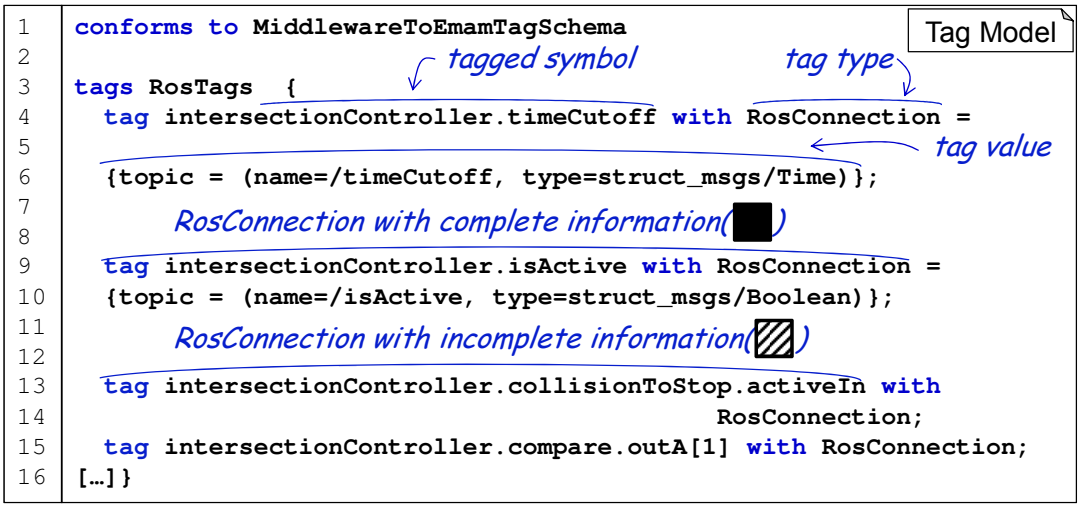


Figure 6.5: Tag model for the running example [HKKR19].

and type information set explicitly by filling the ports these tags are attached to black in the EMA diagram, cf. Figure 6.2.

Note that all middleware parameters are optional. In the case of an empty tag, the parameters are derived from the tagged EMA ports. If the tagged port is a port of the main component, the topic name can be derived from its name directly, no matter whether it is an input or an output port. If we have an output port connected to one or multiple input ports *inside* a model tagged with empty middleware tags, a unique topic name is derived from the output port's name. All receivers then listen to this topic. The type is derived from the port's type if the middleware supports typing, e.g. ROS. Tags without middleware information are depicted as striped ports, cf. Figure 6.2. As mentioned above, instead of tagging each port individually, it is also possible to tag component instances. This is interpreted as tagging *all* (outer) ports of the tagged instance with empty tags. Obviously, tagging a component with topic information makes no sense and is therefore forbidden by a context condition. The ROS tag model for the intersection system of Figure 6.2 is given in Figure 6.5.

As long as we only tag the outer ports of a model, it can be generated as a black box according to our first use case. Assume that we need to model a distributed system as demanded by the second use case. To do so, we can tag subcomponent ports with middleware tags, as well. Consider the striped ports in Figure 6.2, i.e. these ports have middleware tags attached, but no topic names and types were specified. These ports belong to subcomponents of the model, but due to the fact that the connectors need to be realized using a middleware, it becomes possible to generate them as independent applications. Components which are only connected through middleware tagged ports

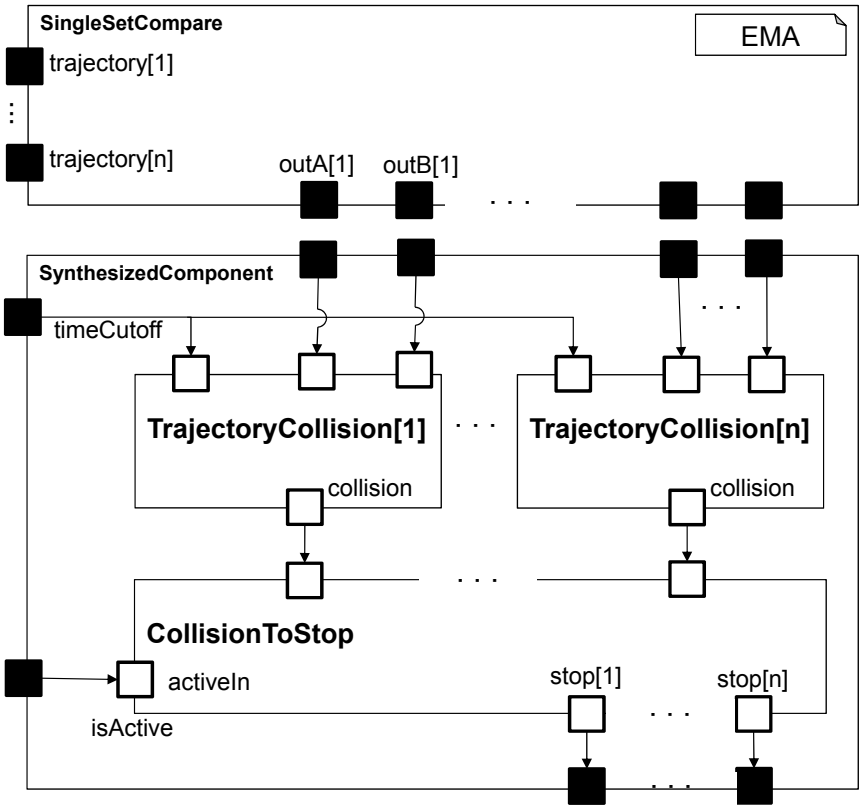


Figure 6.6: True deployment scheme of the intersection controller with two separately deployable components.

can reside in two different target applications.

Consequently, the `SingleSetCompare` component should run in its own process as it only communicates via middleware. The other subcomponents of the intersection controller on the other hand can be combined into a single independently deployable executable since they do not communicate via middleware. The two desired separately deployable models are shown in Figure 6.6. The activity needed to obtain these separated deployment models is an automatic preprocessing step of the overall generation process, which we refer to as *slicing*. For each deployment model or slice, an independent code generation process is used. This allows us to model a distributed system in one EMA model as required by (RC6).

As can be seen in the tag schema in Figure 6.4, the tag type `MqttConnection` can be used to model MQTT connections. In contrast to the ROS tag type, the MQTT tag type allows only one parameter, the topic name. The data is sent as a char stream and there is no further typing.

Note that an output port can be tagged with multiple middleware tags of different types. This means that the port will have a dedicated adapter to the infrastructure of each tagged middleware type, thereby allowing (and requiring) it to publish its data into the respective middleware networks simultaneously at runtime, e.g. ROS and MQTT. The choice of the middleware solutions to be used is fixed at compile-time by the tag model and cannot be changed at runtime.

Given that the C&C model and the middleware model, i.e. the tag model, reside in two separate files, cf. Figure 6.1 and Figure 6.5, the tagging approach fully supports the middleware-agnostic modeling requirement (RC1). The logical models remain untouched by the middleware information. Nevertheless, the tagging mechanism ensures that the tagged element names are actually present in the target model.

6.6 Code Generator Composition

Obviously, the EMADL code generation toolchain needs to be extended to support middleware tagging as presented above. To maintain a clean separation of concerns and extensibility, a modular and extensible generator composition approach is required, allowing us to keep the core EMADL generator middleware-agnostic. Middleware aspects need to be generated by dedicated middleware code generators. Furthermore, it should be easy to add new middleware generators to the toolchain.

Luckily, middleware code generators can work independently and do not have to know about each other. The only interface which is needed to glue the generated artifacts together is the way how ports are represented in target code by the core EMA generator so that the middleware generators can interact with them.

We tackle the composition problem with a variant of the commonly used *bridge pattern* [GHJV95], which is referred to as the star-bridge (or *-bridge) pattern. In the bridge pattern, several concrete implementations might exist, but only one is active at runtime. The star-bridge on the other hand has a list of implementations and all of them are used whenever the implementation is called.

A class diagram representing the star-bridge-based generator composition mechanism is shown in Figure 6.7. In our use case, the abstraction of the bridge is the `MainGenerator`. This class provides methods to manage the implementations as well as a `generate(.)` method to call them in a loop. The concrete class `CoordinatingGenerator` is the actual implementation of the bridge abstraction containing the skeleton algorithm for the generation of the EMA project together with the middleware. Additionally, this class generates code for the orchestration of all artifacts at runtime. For each EMA component, it calls its own `generate(ECIS comp)` method which in turn delegates the generation to the implementation objects of the bridge. Each such implementation object needs to implement the `GeneratorImpl` interface containing a `generate(ECIS comp)` method. Note that the core C++-generator implementing

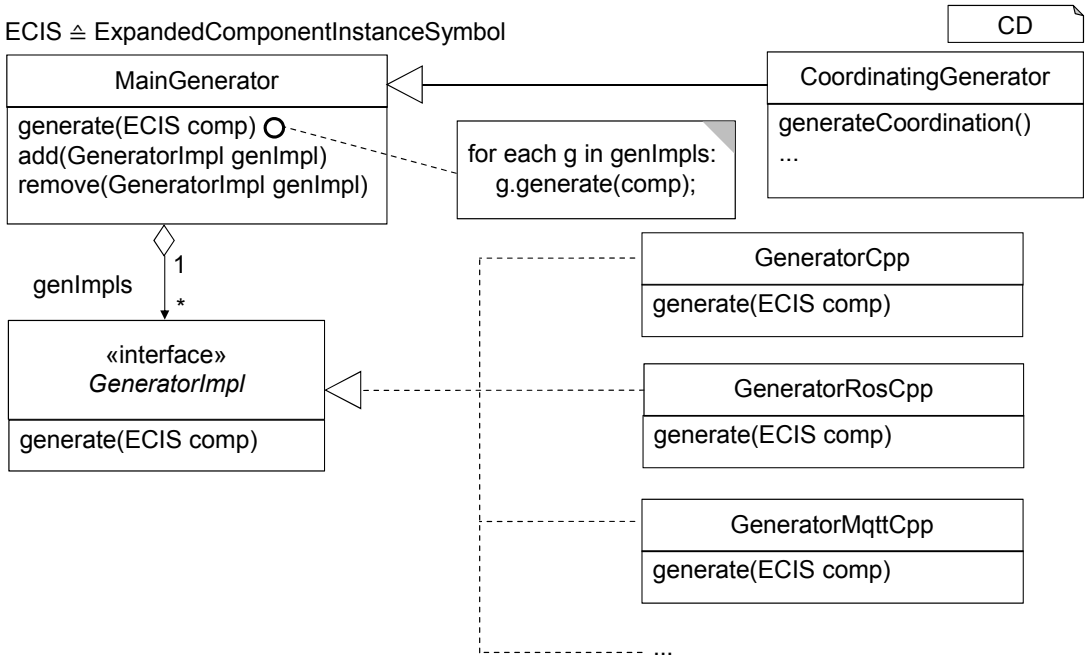


Figure 6.7: Overview of the generator coupling architecture [HKKR19].

`GeneratorImpl` is treated in the same way as the middleware generators.

While the core code generator produces the actual component code, the middleware generators generate middleware adapters for these components according to the information in the middleware tags. Given that the C&C and the middleware generators are completely independent of each other, the coupling approach obviously satisfies the middleware-agnostic generation requirement (RC2).

The class diagram of the generated code is depicted in Figure 6.8. Each middleware generator produces an adapter which is responsible for the middleware communication of the ports tagged in Figure 6.5. Such an adapter implements the `IAdapter` interface requiring the methods `init(.)` and `publish()`, which can be called by the coordinator controlling the model execution and communication.

On instantiation, each middleware adapter is initialized with a reference to the adaptee, i.e. the object representing the component instance to be adapted. The middleware adapter joins the middleware network, e.g. by subscribing to all relevant ROS topics for input ports. A middleware-specific callback function is triggered by the middleware as soon as new data becomes available. After reception this callback function forwards the data to the targeted input ports of the EMA component instance. At the end of each execution cycle the coordinator triggers the adapter's `publish()` method which then collects data from the output ports of the component instance it represents and sends it

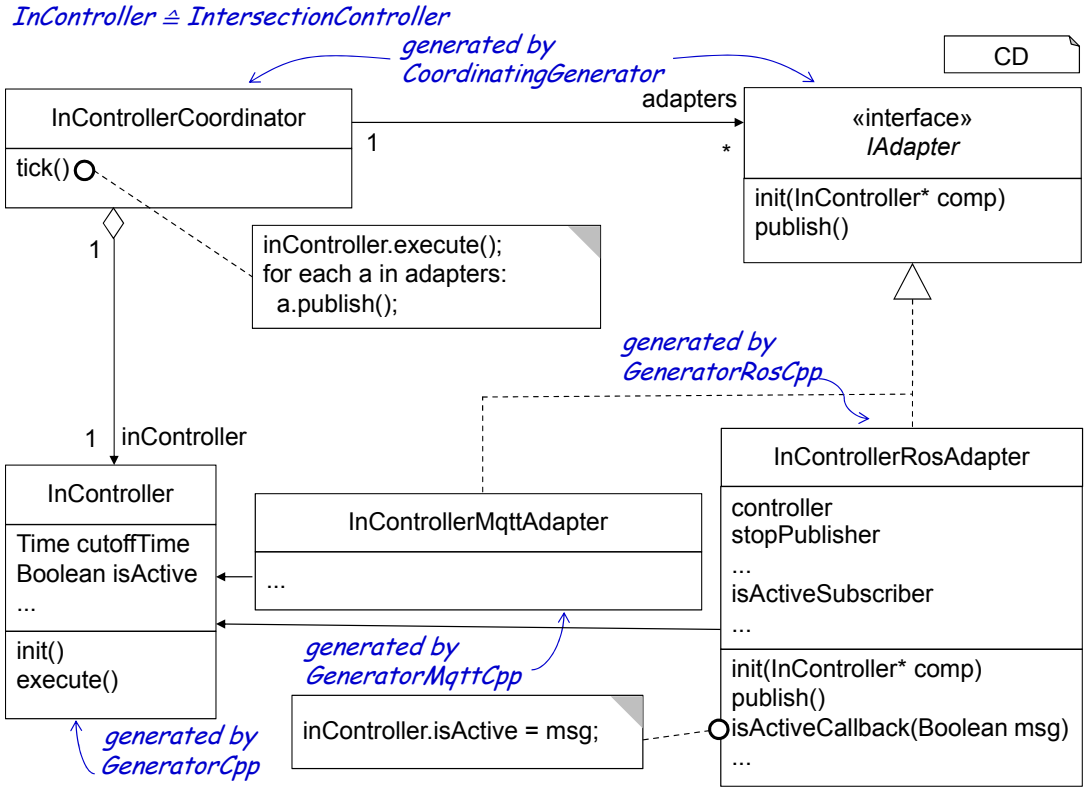


Figure 6.8: Overview of the generated C++ code [HKKR19].

to the corresponding middleware topics.

Each output port can be tagged with multiple middleware tags. In this case the coordinator instantiates multiple adapters enabling a component instance to send its data through different middleware networks. This fulfills the middleware coupling requirement (RC5). Note that for an input port, a tag counts as an incoming connector. According to the rule that an EMA input port can only have one incoming connector, an input port can only have one middleware tag attached.

Recall that the generation process described in this section is applied to each slice of the model independently and might even use tailored variants of the generator depending on the middleware types used by the respective slices. Inside each slice, the communication is generated as standard C++ function calls according to the EMA semantics as described in Section 2.3.3 and Section 2.5 sticking to the synchronous and weakly causal execution model, which supports (RC3) and (RC4). Applying the same execution model to the distributed communication, however, is infeasible. Bottlenecks in the communication network would slow down the execution as each slice would need to

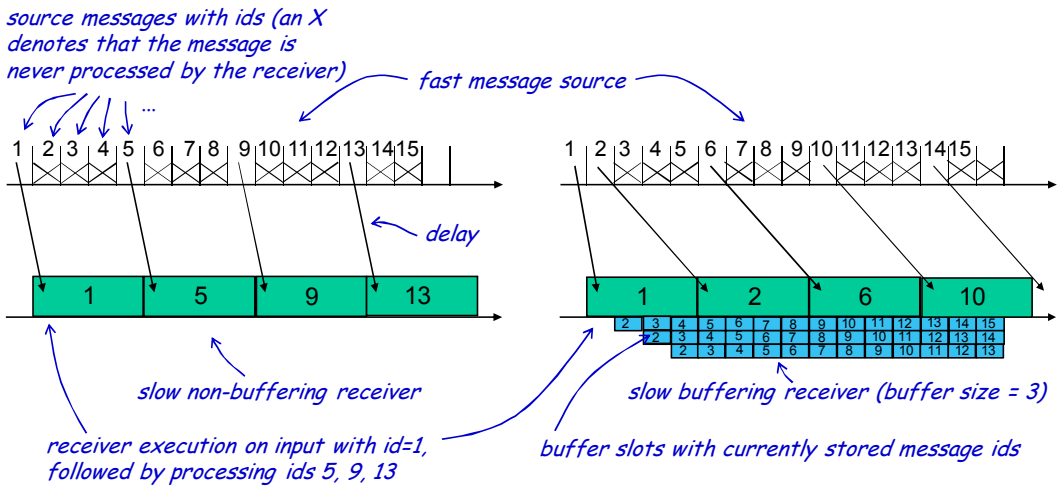


Figure 6.9: Setting in which a fast sender serves a slow receiver with and without buffering.

wait for the slowest component. What is more, sensors provide measurement data with different frequencies; control tasks need to run with a higher frequency than planning tasks. A synchronous execution model would hinder quick reactions in safety-critical situations, e.g. if a sensor detects an obstacle in front of a self-driving vehicle, but the controller cannot process this input because another component is still computing the gear shift.

This leads to the decision to refrain from a synchronization between the distributed model slices and switch to an asynchronous execution mode, where each model slice is controlled by its own coordinator and can run at its own pace without waiting for external inputs from the middleware (recall that inside a model slice the synchronous model continues to hold). The EMA language as introduced in Chapter 2 assumes perfect communication with no latencies and losses. Hence, there is no need to wait for messages and there is exactly one message at each input port when a component is executed. Message queues are not possible by design. This is different in an asynchronous execution model with distributed components and we need to extend our understanding of an EMA port. Therefore, we introduce the following assumptions: a port always holds exactly one message; there is no buffer and hence, no message queues are possible. What is more, an incoming message overrides the message currently present at the port even if the latter has not been processed yet. On the other hand, a message is cached and remains accessible until it is overridden by a successor message. This has two implications. First, a component always operates on the latest message.

Second, if the pace of incoming messages is not the same as the execution frequency of

the processing component, messages might get skipped or processed multiple times. The rationale behind this seemingly unintuitive design decision is described in the following. As discussed in Chapter 1, the fundamental structure of a cyber-physical system consists of sensors capturing the environment, the control software operating on the sensor signals, and the actuators. If the control software is slower than the sensors, it is still crucial that it operates on the most up-to-date information. Older sensor inputs, even if they have not been processed yet, do not represent the current state of the environment and are, hence, obsolete. Processing buffered inputs first would introduce an unnecessary delay letting the controller operate in the past. This is illustrated in Figure 6.9.

At the top of the figure, two identical timelines divided into execution steps of a message source, which is sending messages with a frequency f_S, are shown. Below these two timelines, the processing timelines of a non-buffering receiver (left) and a buffering receiver with a buffer size of three messages (right) are depicted. The slow receiver has an execution frequency $f_R = \frac{1}{4} f_S$, but this ratio can vary arbitrarily. Execution cycles of the receiver are depicted as rectangles with the message ids being processed in the respective execution steps. There is no synchronization between the execution cycles of the sender and the receiver. Source messages which are never processed by the receiver are labeled with an X in the sender's timeline. While the non-buffering receiver always uses the latest available message (e.g. the latest sensor sample), the buffering receiver introduces a processing delay which is equal to the buffer size. Since the buffer size is finite, message loss cannot be avoided completely, the lost messages are just distributed differently. An infinite (or very large) buffer size would lead to a linearly increasing delay between the production of a message by the source and its processing by the receiver.

On the other hand, if the processing component is faster than some of the sensors, it will take the last available sensor inputs as still valid. Consider a situation, where an optical sensor has detected an obstacle in front of the vehicle, but the velocity sensor has not provided an update yet. The last sample provided by the velocity sensor was a measurement of 50 km/h. It makes sense to reuse this value instead of waiting for the next update while approaching the obstacle – in the distributed setting there is no guarantee when the next sensor measurement will arrive.

The propagation of a signal throughout a distributed EMA network is illustrated in Figure 6.10. The top plot shows a true physical signal, e.g. a vehicle's velocity. In the second plot, a digital sensor samples this signal and forwards the measurements to another component, e.g. a controller. For the sake of simplicity, the measurements are shown as ideal, i.e. without a measurement error or delay. The third plot shows the value present at the target port of the receiver component. This is a piecewise constant function, which is sampled by the receiver whenever it is executed. The samples seen by the receiver are depicted in the bottommost plot: while the value 47 is sampled twice by the receiver (at 0.3 and 0.5), the rise of the signal to 52 is ignored as the component is not executed in the respective timespan. As long as the execution frequency of the receiver is high enough for a flawless functioning of the overall system, losing signal samples is not a

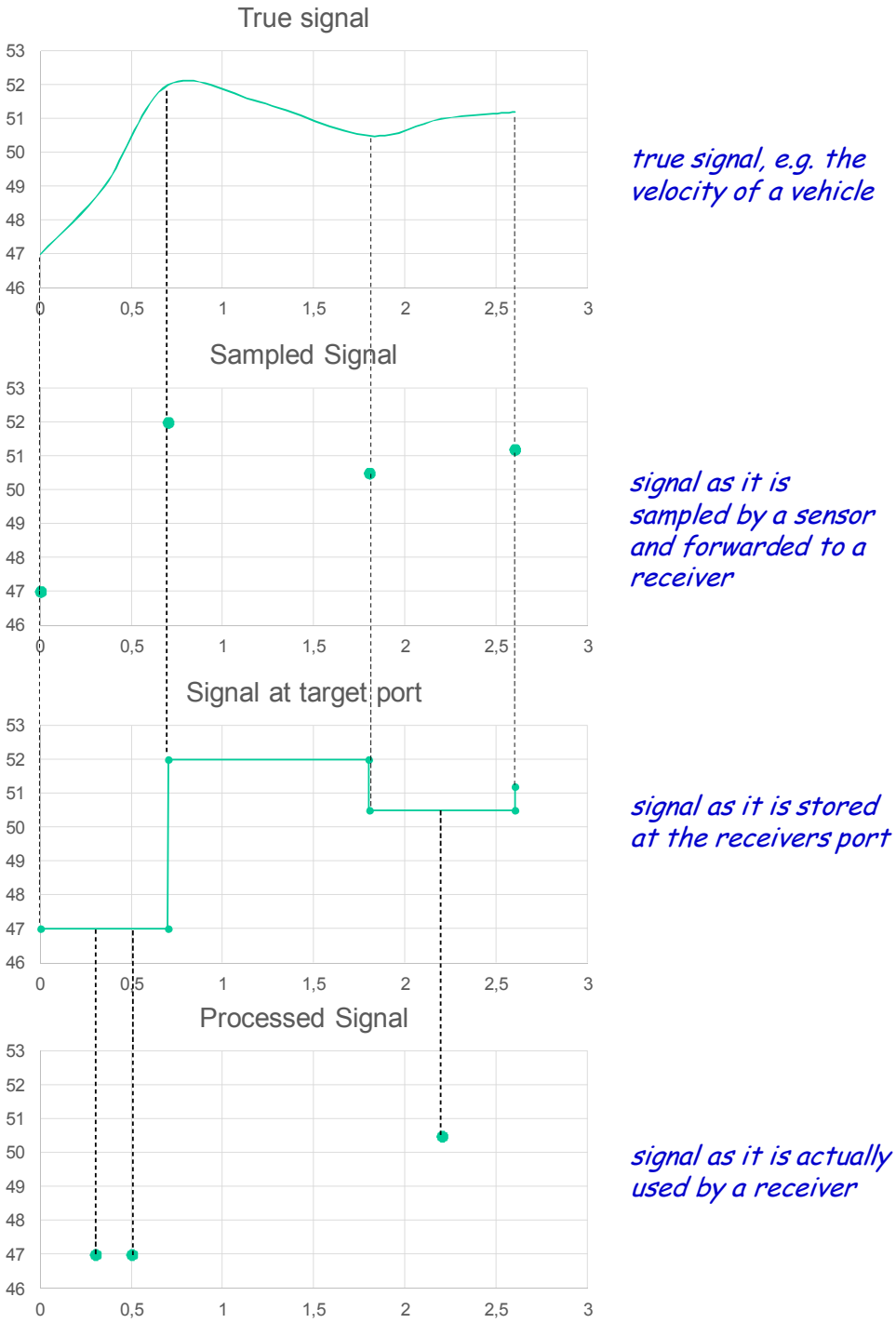


Figure 6.10: The four plots show an exemplary propagation of a signal through a middleware-based EMA component network.

problem. The original signal cannot be sampled with arbitrarily high precision anyway. A minimum execution frequency for each model slice needs to be found by system tests and physical modeling.

DSP systems often use zero-stuffing instead of processing the same input multiple times when upsampling a signal. This can be achieved with the proposed scheme by letting a component reset its input ports to zero when it has finished an execution cycle. Consequently, the component will read zeros at these ports until a new update is received from the source.

If processing a message twice is not appropriate in the modeled system, it can be prevented by extending each message by a unique id and letting the receiver check this id before starting the actual computation. If message loss is not tolerable, synchronization can be mimicked by implementing a corresponding protocol on EMA level, i.e. a sender would wait for a response by the receiver before sending the next message. Again this can be accomplished by using unique message ids. A cleaner solution is to extend the synchronous EMA semantics as introduced in Section 2.3.3 for distributed systems and to provide it as an alternative operation mode. Such an approach is not in the scope of this thesis, but might be useful to extend the applicability of EMA to further domains, e.g. web and business applications.

A further possible operation mode is the event-triggered component execution. A component's behavior is executed as a reaction to an incoming message. As long as no new messages are present, a component remains idle. Since messages rarely arrive synchronously, the component is able but does not have to use the cached old values of the other ports.

A generated ROS adapter example implementing the asynchronous semantics as described above is given in Figure 6.11 for the `IntersectionController` component. In L.1 the header of the adapted component is included. The `init(.)` method is called once at startup by the coordinator and receives a pointer to the adaptee object, which is shared between all of its adapters and the coordinator. Furthermore, for each input port a ROS subscriber and for each output port a ROS publisher is initialized. L.14-19 show the publish method of the adapter. It is called by the coordinator in each execution cycle after the component logic has been executed; it reads the value of each output port and publishes it to the respective ROS topic using a dedicated publisher object.

While there is only a single publish method for all output ports, a dedicated callback is available for each input port. For instance, a callback for the `isActive` port is given in L.20-22. The respective subscriber has a reference to this callback function and calls it whenever new data arrives. The received message is unpacked and assigned to the target port.

Figure 6.12 shows an overview of the generated project structure, the contained artifacts, as well as the responsible generators. Here, `SynthesizedComponent` is the name of the model slice based on Figure 6.6. The `SingleSetCompare` slice would lead to a similar generator output. Note that only those middleware adapters are gen-

C++

```
1  #include "IntersectionController.h"
2  [...] /* other includes*/
3  class IntersectionControllerRosAdapter : public IAdapter{
4    IntersectionController* controller;
5    ros::Subscriber isActiveSubscriber;
6    ros::Publisher stopPublisher;
7    [...]
8    public:
9    void init(IntersectionController* comp){
10     controller = comp;
11     /*init publishers, subscribers and start ROS thread*/
12     [...]
13   }
14   void publish(){
15     struct_msgs::Boolean tmpMsg =
16         msgFromStructBoolean(controller->stop[1]);
17     stop1Publisher.publish(tmpMsg);
18     [...]
19   }
20   void isActiveCallback(struct_msgs::Boolean& msg){
21     controller->isActive = structFromMsgBoolean(msg);
22 } [...] };
```

instance shared between the coordinator and other adapters

called once by IntersectionControllerCoordinator

called in a defined interval by IntersectionControllerCoordinator

called by ROS every time a message is published on the topic /isActive

Figure 6.11: Generated ROS adapter for the IntersectionController component.

Generator	Generated files
CoordinatingGenerator	├── CMakeLists.txt ├── coordinator │ ├── CMakeLists.txt │ ├── SynthesizedComponentCoordinator.cpp │ └── IAdapter.h
GeneratorCpp	├── cpp │ ├── CMakeLists.txt │ └── SynthesizedComponent.h
GeneratorMqtt	├── mqtt │ ├── CMakeLists.txt │ └── SynthesizedComponentMqttAdapter.h
GeneratorRosCpp	└── roscpp │ ├── CMakeLists.txt │ └── SynthesizedComponentRosAdapter.h
GeneratorRos2Cpp	└── ros2cpp ├── CMakeLists.txt └── SynthesizedComponentRos2Adapter.h

Figure 6.12: Overview of the generated project structure and artifacts.

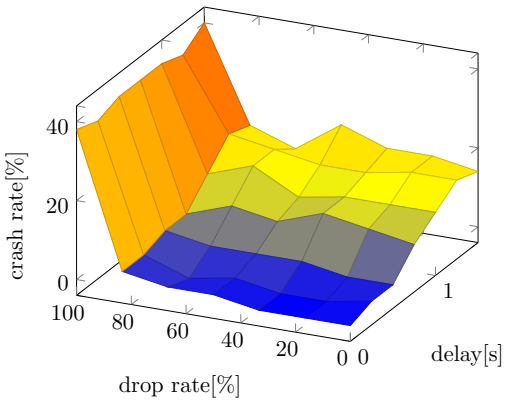


Figure 6.13: Crash rate over network delay and drop rate [HKKR19].

erated, which are actually required by the tag model. MQTT and ROS2 adapters are not needed according to the tag model in Figure 6.5, but are mentioned here to stress the multi-platform capability of the approach.

As can be seen in Figure 6.12, each generator delivers its own `CMakeLists.txt` file. Furthermore, to glue the project together, the coordinating generator produces a `CMakeLists.txt` at top level. This build file can be used by the developer to finally build the whole project in one go. This approach covers the build infrastructure requirement (RC7).

6.7 Evaluation

To demonstrate the presented toolchain, we have integrated our running example model with the ROS-based CoInCar simulator of the DFG SPP1835 program on cooperatively interacting vehicles [NPLS18]. Our goal was to ensure that the middleware coupling approach is feasible by verifying that the intersection controller and tagging model compile to a ROS-based system, which is in fact capable of preventing collisions at an intersection in a series of simulations under non-ideal communication conditions.

In our test scenario, two vehicles approach an intersection as depicted in Figure 6.3. Fifty configurations of this scenario with the starting positions varying by up to $\pm 10\,\mathrm{m}$ in the direction of the respective lane and starting velocities equally distributed between 18 and 33 km/h were defined. A total of 49 pairs of network delay between 0 and 1.5 s and message drop chance of 0 to 100% were applied to each of these starting configurations. The vehicles have no intelligence except starting to decelerate upon a signal from the intersection controller. The collision rate is recorded to evaluate the system performance. The experiment was set up and conducted by Alexander Hellwig in his bachelor's thesis. The results are depicted in Figure 6.13. The sampled collision probability is plotted

over the network delay and drop rate. As can be seen, for delays of up to 250 ms and package drop rates of up to 30% the controller is able to prevent collisions completely in our experiment. An increase of the drop rate up to 80% while the delay is kept below 250 ms still results in a low collision rate of below 4%. Once the drop rate surpasses the threshold of approximately 85%, the crash rate increases significantly – when no or almost no messages come through, collisions cannot be avoided. As long as the drop rate remains below this threshold, the collision rate is almost a linear function of the delay.

We can summarize that the toolchain supports the developer by providing all necessary code and build files out of the box. Thanks to the separation of concerns principle, the logical model can be reused in another simulator by exchanging or adapting the tag model. The presented middleware modeling approach was used in a variety of EMA projects in conjunction with different simulators including Gazebo [KH04], TORCS [WEG+00, LCL13], Carla [DRC+17], and others.

6.8 Automating Model Slicing for Distributed Deployment

6.8.1 Motivation

In the course of this chapter, we have introduced the notion of model slicing, which can be used to subdivide an EMA model into several stand-alone parts. This is accomplished by manual port tagging, i.e. the modeler must provide the information on how the logical model is transformed into a set of deployment models. Manually analyzing a model, deciding how to deploy it, and tag its subcomponents accordingly might seem to be a straightforward task for small models featuring only a few components.

However, real world architectures consist of hundreds or thousands of components, cf. the Daimler advanced driver assistance system (ADAS) models used in [BMR+17]. What is more, a deployment scheme needs to take into account various constraints such as binding a specific software component to a specific ECU. In addition, the resulting configuration should be efficient with regard to certain criteria. For instance, it is desirable to keep the required amount of communication and the latencies between the distributed nodes as low as possible.

A manual analysis of how to split up a logical model and the maintenance of the corresponding middleware tagging model is, hence, a tedious and error-prone task leading to suboptimal results concerning the optimization goals with high probability. It is therefore our goal to replace this manual procedure by an automated toolchain building on top of the EMA compiler and the middleware tagging toolchain.

The question we aim to answer in this section is: how can the derivation of final, deployable artifacts from the functional architecture be completely automated by an appropriate toolchain? The input of the toolchain should consist of a functional architecture (defined as an EMA model in our case), a description of the target environment, e.g. on how many computation nodes the architecture should be deployed, as well as

information about the target architecture for the compiler. No user feedback should be required for the generation and compilation of the final artifacts.

6.8.2 Component Clustering

A natural way to subdivide an architecture into several parts is to analyze its top level structure. Recall the intersection controller model in Figure 6.2 and let $n = 2$. Then the architecture consists of three subcomponents in total: `SingleSetCompare`, `CollisionToStop`, and one `TrajectoryCollision` instance. If our aim were to deploy it on three distributed computing nodes, the naive approach would be to deploy each of the three components on its own node. However, this strategy might turn out to be suboptimal, e.g. if the connectors between these components carried much more data than connectors *inside* the respective components or if the computational burden were distributed highly unequally between the three components.

To address this problem, model flattening as described in Section 2.3.3 is carried out before identifying a suitable slicing scheme. Having only one granularity level consisting of small basic blocks enables us to find the most flexible boundaries.

We are going to tackle the partitioning problem by employing unsupervised learning techniques, also referred to as clustering. In unsupervised learning there are different ways to handle the data. We can map the domain elements to be clustered into some metric space with a distance function and perform the clustering in that space. For instance, we could map each component of our model to an n-dimensional vector space and use the Euclidean distance as a basis for clustering. Algorithms like k-means/k-means++ [AV07] or Mean Shift [Che95] require such an explicit representation as they need to create new points in the data space, e.g. updated cluster centers.

However, finding an appropriate representation with a meaningful distance function for software components seems fallacious. Fortunately, various clustering algorithms exist which are meant to operate on an adjacency matrix, where each entry represents the similarity (or alternatively the distance) of two instances of the dataset. In such a case the distances or adjacencies can be looked up and don't have to be computed during the course of the clustering procedure. This saves us from the need to find an explicit high-dimensional embedding for component models in a vector space as well as a meaningful similarity function. All we need to do is to find a mapping from the set of (flattened) C&C models to the set of adjacency matrices (or, equivalently, the set of undirected weighted graphs).

Given a flattened C&C model, we use the following injective transformation to construct a corresponding graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$ with $\mathcal{V}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ being the nodes and the edges of the graph, respectively, and $w : \mathcal{E} \to \mathbb{R}_0^+$ being a function assigning a non-negative weight to each edge of the graph:

1. We set $\mathcal{V} = \mathcal{C}$, i.e. each component instance of the input model is represented by a node in $\mathcal{G}$.

2. We insert an undirected edge $e \in \mathcal{V} \times \mathcal{V}$ between two nodes of the graph iff there is at least one connector between the two corresponding components in the C&C model.

3. For each edge $e \in \mathcal{E}$ we look up all the connectors in the original C&C model and sum up their costs, e.g. by estimating the total data rate transmitted over these connectors based on the ports' data types.

4. Finally, we set the edge weights $w(e)$ to be this total cost. Alternative definitions of the edge weights are possible, e.g. based on latencies, and highly depend on the use case.

5. In some cases, structural deployment constraints need to be incorporated. This will be discussed in Section 6.8.4

Having mapped our C&C model to a graph, we have a huge arsenal of graph processing and analysis techniques at our disposal enabling an easy integration into MDSE and visualization tools. Of particular interest for this work are graph-based clustering or unsupervised learning techniques. These techniques enable us to find disjoint graph partitions. A partition, also referred to as a cluster, can be interpreted as a set of graph nodes corresponding to components to be deployed on the *same* execution node, e.g. an ECU.

Hence, components belonging to the same cluster can be run in a single process and share the same memory. Intuitively, we want to find a deployment scheme, so that most of the communication takes place *inside* a cluster, since it can be realized by simple function calls, while inter-cluster communication, i.e. expensive middleware communication, should be kept at minimum. In graph terminology this means that we would like to minimize the weights of edges connecting nodes of different clusters while trying to distribute the components in a fair way to achieve a balanced utilization of computing resources. Hence, we need to avoid naive solutions deploying all components on a single computing node and thereby having no middleware connections at all.

The desired partitioning can theoretically be found by minimizing a graph cut function such as the *RatioCut*:

$$RatioCut(\mathcal{A}_1, ..., \mathcal{A}_k) := \frac{1}{2} \sum_{i=1}^{k} \frac{W\left(\mathcal{A}_i, \overline{\mathcal{A}}_i\right)}{|\mathcal{A}_i|}, \tag{6.1}$$

where $\mathcal{A}_i$ are disjoint node sets with $\cup_{i=1}^{k} \mathcal{A}_i = \mathcal{V}$; $W\left(\mathcal{A}, \mathcal{B}\right)$ denotes the sum of edge weights connecting nodes from the two node sets $\mathcal{A}$ and $\mathcal{B}$. Here, $\overline{\mathcal{A}} := \mathcal{V} \setminus \mathcal{A}$ denotes the set containing all nodes of $\mathcal{V}$ not contained in $\mathcal{A}$. k is the number of clusters we want to obtain.

In most clustering applications the number of clusters is unknown and needs to be estimated using computationally intense algorithms, e.g. the Silhouette method [Rou87].

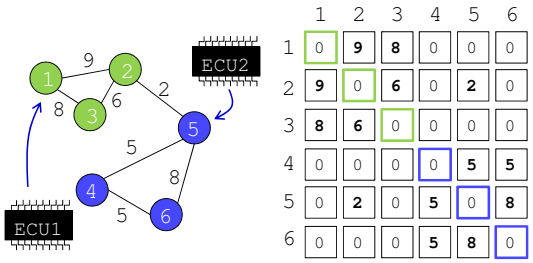


Figure 6.14: Clustering software components for deployment.

In our context however, the number of desired clusters k is inherent as it is given by the number of heterogeneous ECUs we want to deploy our CPS software on (in a hardware/software codesign scenario where the software architecture is known, but a hardware architecture still needs to be found, estimating the number of clusters using such methods might still be appropriate).

Intuitively, minimizing the RatioCut corresponds to finding a clustering which avoids inter-cluster connections while trying to maintain a reasonably large number of nodes in each cluster. Minimizing the RatioCut function however is an NP-hard problem and hence infeasible for a large number of components. Fortunately, it has been shown that unnormalized spectral clustering yields a fair approximation of the RatioCut minimization problem and has a complexity of $\mathcal{O}\left(|\mathcal{V}|^3\right)$ [NJW02, VL07]. The algorithm is described in the following.

Based on the graph weights, a symmetric graph similarity matrix W is constructed, with the entry $W_{ij} = W_{ji}$ representing the weight of the edge connecting the nodes i and j. This is depicted in Figure 6.14 for a graph representing a simple software architecture featuring six components. For the actual clustering we need to obtain a matrix known as the graph Laplacian, which is defined as

$$L := D - W, \tag{6.2}$$

where

$$D := \operatorname{diag}(W\mathbb{1}) \tag{6.3}$$

is a diagonal degree matrix. Note that the values on the diagonal of W do not affect L. Therefore, we can set them to zero without loss of generality, i.e. we can omit loop edges. The first k eigenvectors of L form a low-dimensional representation of the data which we cluster using the k-means algorithm, optionally using a k-means++ initialization to avoid degenerate solutions [AV07]. In the example given in Figure 6.14 we set $k = 2$ to deploy our six components on two different ECUs.

The result is a cluster indicator vector assigning each component to an ECU. Nodes assigned to the same cluster share the same color in the graph and the corresponding

similarity matrix in Figure 6.14. Each cluster forms a stand-alone model slice. The EMA compiler is run separately on each slice to obtain an independent executable for each ECU as discussed in Section 6.5.

For ports receiving or sending data from or to another cluster, the toolchain automatically generates middleware tags for the desired middleware (evaluated for ROS in the reference implementation). Consequently, all the necessary middleware adapters are generated and don't need to be specified by the developer explicitly. All type and middleware information is inferred from the EMA model as mentioned in Section 6.5.

6.8.3 Weights

For the proof-of-concept we use the data rate, i.e. the number of bytes transmitted in each cycle, to define the graph weights. If all components are executed and transmit data with the same frequency, the data load transmitted between two components per cycle can be estimated as the sum of the connector type sizes. We assume a static size typing with `B`, `Z`, `Q`, and `C` being generated as `boolean` (1 Byte), `int` (4 Bytes), `double` (8 Bytes), and two `doubles` (16 Bytes), respectively. This scheme however highly depends on the code generator used since EMA types are abstract. A `Z` variable with a range from 0 to 1000 can be represented as an `int` in target code, while very large numbers might need to be generated using several longs. For this reason, the generator should offer an interface, through which it can indicate how many bytes a given abstract type will require at runtime. A more general weight construction process should hence consult the code generator to estimate the expected data rates.

6.8.4 Encorporating Structural Constraints

Until now, we have discussed how to get a clustering, which optimizes communication costs while trying to achieve a fair workload distribution. Another important aspect in embedded systems design is that data is produced by a multitude of different, specialized sensors. To reduce latencies or to fulfill timing constraints, much of the data needs to be processed locally, near the sensor. This means that we need to be able to enforce our toolchain to deploy particular computations using a fixed ECU assignment. To achieve this, we have *constrained spectral clustering* approaches at our disposal enabling us to formalize such requirements [WD10, WQD14]. In constrained spectral clustering, the optimization problem of spectral clustering is extended by a symmetric constraint matrix Q encoding colocation constraints as must-link (ML) and cannot-link (CL) constraints. The choice $Q_{ij} = Q_{ji} = 1$ means that nodes i and j must be assigned to the same cluster while $Q_{ij} = Q_{ji} = -1$ means that an assignment of i and j to the same cluster is forbidden. If no supervision is available for i and j, the entry is zero.

To define a constraint matrix, the user needs to be able to mark components with the id of the target ECU. This can be accomplished using tagging, as well. Thereby, instead

of tagging ports for middleware code generation, we now tag *components* with ECU ids. If a component can be deployed on one of a series of ECUs, it can be tagged with a list of ECU ids instead of a single id.

In the flattening process, ECU tags of composed components are propagated automatically to their subcomponents. Hence, it is sufficient to tag the image detection component to be deployed at the image processing ECU – all of its subcomponents will inherit this parameterization.

Constraint tagging requires minor changes in our graph creation procedure. For each ECU id explicitly mentioned in the deployment tag model, we create an additional node in the graph $\mathcal{G}$. All such nodes corresponding to an ECU instead of a software component are pair-wise marked with a cannot-link (CL) constraint in the constraint matrix Q by setting the respective entries to -1. Next, all nodes representing a software component and tagged with exactly one ECU id obtain a must-link (ML) constraint with respect to the node representing the tagged ECU id. If a software component is tagged with a list of possible ECUs, this is realized in the constraint matrix as cannot-link constraints to all other available ECUs, i.e. those which are not present in the tagged list. The clustering result fulfills the deployment constraints while still aiming to minimize communication overhead.

6.8.5 Related Work on Automated Deployment

In AIRES [WMS04] the software architecture is modeled as a weighted directed graph which is referred to as the structural model. It consists of nodes representing components with known resource consumption and edges representing connectors connecting output ports to input ports. The resource consumption is subdivided into the computation, communication, and memory consumption. Computation and memory consumption are assigned to nodes, while communication costs are the edge weights.

The target platform is modeled as a weighted graph, as well. The nodes of this graph represent computation and memory resources. The graph is undirected, since the communication infrastructure is modeled as a shared communication link for all devices. Resource availability functions assign computation, memory, and communication resources to the computation nodes and the communication link, respectively. An informed branch-and-bound algorithm with a competence function and a forward checking mechanism is then used to partition the structural model so that each partition can be allocated to the computational nodes of the platform model with sufficient resources. The algorithm aims to optimize the competence function, which is defined as a weighted combination of the normalized computation, memory, and communication consumption. The latter is used in our work, as well.

In our approach, different constraints can be combined by redefining the edge weights of the initial graph as long as these weights can be interpreted as component similarities. This restriction implies that our clustering-based approach does not allow constraints

on the resources of single components such as memory or computational resource consumption. The concrete implementation of informed branch-and-bound allows for colocation constraints (must-link and cannot-link) to be ensured, which can be solved by the means of constrained spectral clustering in our approach. A difference between the two approaches is that our solution does not require an explicit model of the target platform (except the number of target ECUs). Therefore, it can be used in an earlier stage of development as well as in hardware/software codesign.

A model-driven allocation engineering approach is proposed by Pohlmann and Hüwe [PH19]. The system under development consists of a component model and a platform model. Again, the goal is to map the component model to the ECUs of the hardware model under certain constraints. The two models are defined using MechatronicUML [BDG+14]. Constraints are specified using a DSL incorporating the Object Constraint Language (OCL) and the Viatra Query Language (VQL) [BURV11] for graph querying. The tuples returned by a query can be used for colocation of components, to bind components to a specific ECU, to constrain the memory usage, etc. The query results are transformed into an integer linear program (ILP) from which a valid allocation scheme can be calculated using existing ILP solvers.

In contrast to our approach, resource limitations of ECUs, as well as the resource consumption of components can be modeled. The resulting ILP guarantees that a generated allocation scheme violates no constraints, but, in contrast to our clustering-based approach, it will not optimize for the resulting costs (e.g. inter-ECU communication). Since a model of the target hardware is needed to create the ILP, this algorithm can only be used for fixed hardware architectures. The approach presented in this chapter can be used earlier in the development process and inform the choice of hardware, e.g. with respect to memory and communication bandwidth.

Moser and Mostaghim propose an approach based on a bi-objective optimization problem formulation to find component deployment schemes for fixed vehicular ECU-platforms [MM10]. The hardware architecture is modeled using a set of parameters including ECU capacities, their processing speed, and failure rate. Furthermore, the communication infrastructure is characterized by the data rates, delays, and the reliability of the preferred buses.

The two optimization goals are reliability of data communications and communication overhead. The search space is reduced drastically by three types of constraints. The memory constraint ensures that the capacity of an ECU is not exceeded. The location constraint is employed to restrict the set of possible target ECUs for a given component. The colocation constraint can be used to forbid two components to be deployed on the same ECU. The constrained optimization problem is solved using NSGA-II [DAPM00], an improved version of the non-dominated sorting genetic algorithm (NSGA). A repair algorithm eliminates infeasible solutions.

6.9 Conclusion and Future Work

In this chapter we presented a tag-based approach for the distribution of EMA models and their integration with third-party software such as simulators and testing frameworks. The approach enforces the separation of concerns principle with regard to logics and communication. To keep the code generation process adaptable and maintainable, a modular generator composition approach is used. New middleware generators can be added by implementing the required interfaces. Future work includes the support for further execution modes, e.g. the fully synchronous regime, as well as the integration of more middleware solutions and paradigms not based on the publish/subscribe pattern.

Furthermore, we presented an automated approach deciding how to distribute the components of an EMA model so that the distribution costs, e.g. the required communication overhead, are minimized. Without human supervision the toolchain is able to create a distribution model, subdivide the system accordingly into multiple independent parts, add appropriate middleware adapters and compile the result for the chosen target platform. Supervision can be added to define deployment constraints, e.g. to bind software components to particular ECUs. Future work comprises the integration of further constraint types, e.g. dealing with limited ECU capacities, but also the development of mechanisms for redundant deployment, robustness, and failover.

Chapter 7

Conclusion

The development of CPSs is a complex process requiring experts from different fields. It must be supported by a cross-paradigm model-driven engineering methodology to cope with interdisciplinary challenges and ensure consistency.

An overview of how the methodology developed in this thesis can support the automotive SMArDT process is given in Figure 7.1. It has been shown in this thesis that the nature of CPSs can be represented very well using dataflow-centric modeling techniques. For this reason, the C&C modeling paradigm is the core of the developed methodology. However, components and connectors alone are not sufficient for an adequate architectural modeling. A type system tailored to the domain and featuring abstract mathematical types, physical units, and matrices has proven to be indispensable.

In early phases of the process, the top level architecture is designed using the EMA ADL, cf. level 2 of SMArDT in Figure 7.1. To enable dynamic architectural changes at runtime, EMAD, an event-based reconfiguration language for EMA, was designed. Using EMAD, components, ports, and connectors can be created on demand enabling the developer to model cooperative and self-adaptable CPSs.

The logical architecture serves as a basis for the development of the technical concept at SMArDT level 3. High-level components can be hierarchically refined or – if no further refinement is desirable – their behavior can be modeled using existing library blocks or one of the two available behavior modeling languages of the EMADL language family. Many of the required computations throughout all relevant parts of a CPS depend on matrix processing. The matrix-oriented scripting language MontiMath is therefore a central element for behavior specification. Furthermore, it can be used to model optimization problems often occurring in control tasks of CPSs.

In cases, where a manual behavior specification is tedious, components can sometimes be learned based on data. Despite machine learning being a fast-paced field changing from day to day, in many practical scenarios solutions can be composed of existing learning models. An efficient application of machine learning to industrial problems requires domain-oriented modeling techniques focusing on simplicity and reuse rather than research-oriented algorithmic freedom. To tackle this challenge, a second behavior modeling language, the practice-oriented deep learning framework MontiAnna was designed. Making MontiAnna a stand-alone modeling language instead of extending MontiMath

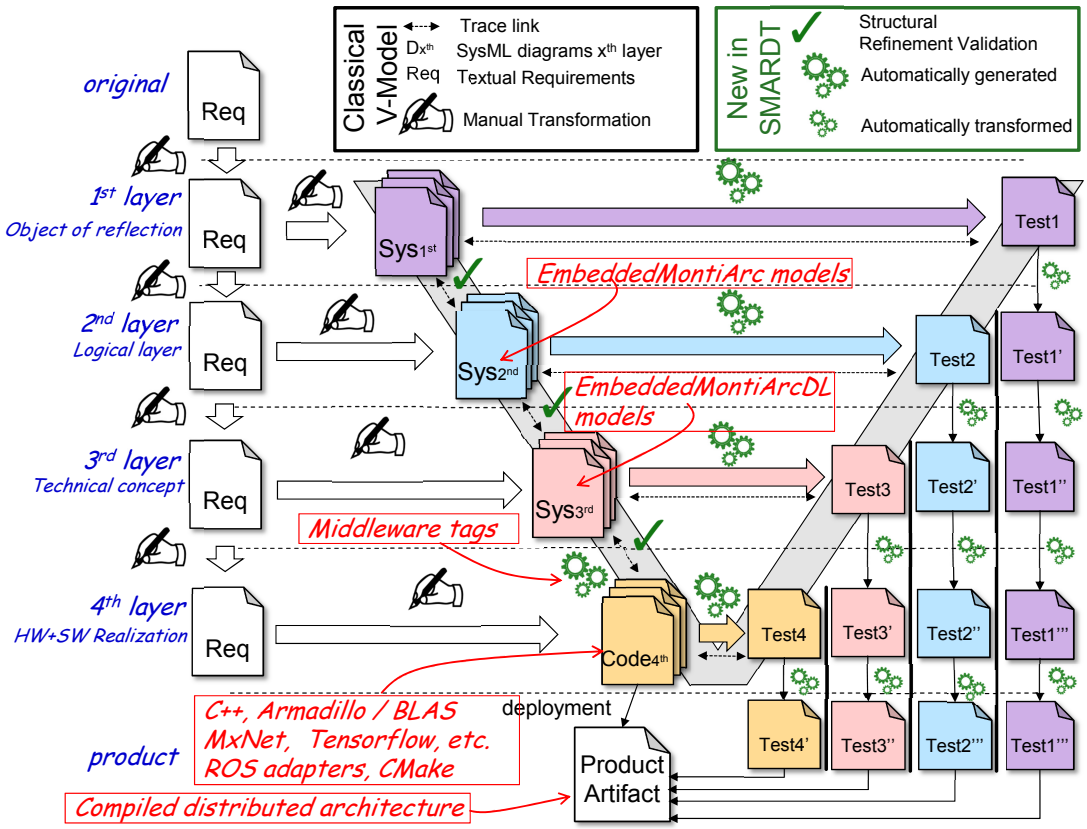


Figure 7.1: EMA in the SMArDT process.

with a deep learning library is an important design decision. It enables the code generator to distinguish between standard and trainable code, thereby offering opportunities for the automation of deep software engineering.

With its focus on existing technology, MontiAnna does not target low level deep learning research with experimentation on completely new neuron models and training methods. Instead, it has proven to be very efficient for the composition of deep artificial neural networks as layer graphs from common building blocks due to its DAG-oriented domain-specific syntax and a simple training configuration language. MontiAnna has been evaluated successfully on a series of state-of-the-art neural networks from fields including image processing, object detection, decision making, data generation, and machine translation. Such networks can be used in sensor signal processing, planning, and decision making of CPSs.

Quality assurance is a central part of development processes such as SMArDT. Each development phase might require a specific Q/A strategy. In our methodology, high-level

architectural models cannot be tested conventionally, since there is no functionality at this point, but structural properties can be verified against a structural specification, e.g. EmbeddedMontiView models. Technical concept models, where components are enriched with MontiMath or MontiAnna specifications, can be tested in a unit test manner using the EMA stream testing language to assert that expected output streams are generated by a model under test for given inputs. Furthermore, a simulator can be used as a validation framework to evaluate the desired model behavior in complex situations with a virtual environment in the loop.

The ability to deploy models in a distributed network of computation nodes is a crucial feature of a CPS development methodology. It is particularly important to separate the technical communication details from the pure logical models. In this thesis a tagging-based approach was used to attach communication-related parameters to EMA models. This solution facilitates the reuse of logical models in different hardware and network settings and when using different testing frameworks. To further support the process, an optional unsupervised learning step can be used to split and tag large models automatically. The tag models enable a transition from a level 3 technical model to the level 4 hardware/software realization of SMArDT. Again, simulations can be used for the quality assurance of the final product.

In this way, the presented methodology supports a CPS development process such as SMArDT from an early conceptual phase to the final deployable product.

Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016. 4.3, 5.3

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *International Conference on Fundamental Approaches to Software Engineering*, pages 21–37. Springer, 1998. 3.3.1

[AHRW17] 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. 6.2

[AKM17] Matthias Althoff, Markus Koschi, and Stefanie Manzinger. CommonRoad: Composable benchmarks for motion planning on roads. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 719–726. IEEE, 2017. 2.6.3

[ARAA+16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, 2016. 4.3, 5.3

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007. 6.8.2, 6.8.2

[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. *Joint proceedings of ACES-MB 2015–Model-based Architecting of Cyber-physical and Embedded Systems*, pages 19–26, 2015. 1.7.6, 3.3.1

[BBB+08] Christian Basarke, Christian Berger, Kai Berger, Karsten Cornelsen, Michael Doering, Jan Effertz, Thomas Form, Tim Gülke, Fabian Graefe, Peter Hecker, Kai Homeier, Felix Klose, Christian Lipski, Marcus Magnor, Johannes Morgenroth, Tobias Nothdurft, Sebastian Ohl, Fred W. Rauskolb, Bernhard Rumpe, Walter Schumacher, Jörn-Marten Wille, and Lars Wolf. Team CarOLO – Technical Paper. Informatik-Bericht 2008-07, Technische Universität Braunschweig, October 2008. 1.3.1

[BBF+08] Andrew Bacha, Cheryl Bauman, Ruel Faruque, Michael Fleming, Chris Terwelp, Charles Reinholtz, Dennis Hong, Al Wicks, Thomas Alberi, David Anderson, et al. Odin: Team Victortango's entry in the DARPA urban challenge. *Journal of field Robotics*, 25(8):467–492, 2008. 1.3.1

[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. 1.7.2

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. 4.1.4

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016. 5.3

[BD95] Adolf-Peter Bröhl and Wolfgang Dröschel. *Das V-Modell*. Oldenburg-Verlag, München, Wien, 1995. 1.7.2

[BDD+09] Felice Balarin, Abhijit Davare, Massimiliano D'Angelo, Douglas Densmore, Trevor Meyerowitz, Roberto Passerone, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Alena Simalatsar, Yoshinori Watanabe, et al. Platform-based design and frameworks: METROPOLIS and METRO II. *Model-Based Design for Embedded Systems*, page 259, 2009. 6.2

[BDG+14] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Wilhelm Schäfer, Matthias Meyer, and Uwe Pohlmann. The MechatronicUML method: model-driven software engineering of self-adaptive mechatronic systems. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 614–615, 2014. 6.8.5

[BDTD+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Mon-

fort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016. 1.3.3, 3

[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. 6.5

[Ber16] Christian Berger. An open continuous deployment infrastructure for a self-driving vehicle ecosystem. In *IFIP International Conference on Open Source Systems*, pages 177–183. Springer, 2016. 6.5

[BET14] Jan Bender, Kenny Erleben, and Jeff Trinkle. Interactive simulation of rigid body dynamics in computer graphics. In *Computer Graphics Forum*, volume 33, pages 246–270. Wiley Online Library, 2014. 2.6.3

[BG14] Andrew Banks and Rahul Gupta. MQTT version 3.1. 1. *OASIS standard*, 29:89, 2014. 6.5

[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. 3.2, 3.3.1

[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. 7, 1.6

[BLT+16] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016. 5.3

[BMR+17] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Component and Connector Views in Practice: An Experience Report. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 167–177. IEEE, September 2017. 2.3.1, 3.4.5, 5.5, 6.8.1

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 4.1.4, 4.8.2

[BNVB13] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. 5.3

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010. 4.1.3

[BPRS18] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018. 4.7.3

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. 2.5

[Bro14] Manfred Broy. A model of dynamic systems. In *From programs to systems. The systems perspective in computing*, pages 39–53. Springer, 2014. 3.3.1

[BS12] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement.* Springer Science & Business Media, 2012. 1.7.6, 1.7.6, 2.3.3, 2.3.3, 3.3.1

[BURV11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In *International Conference on Theory and Practice of Model Transformations*, pages 167–182. Springer, 2011. 6.8.5

[CA13] Eduardo F. Camacho and Carlos Bordons Alba. *Model predictive control.* Springer Science & Business Media, 2013. 1.3.2

[CBM02] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002. 4.3

[Che95] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence*, 17(8):790–799, 1995. 6.8.2

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and control*, 2(2):137–167, 1959. 1.7.2

[CKF11] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011. 4.3

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MxNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 4.3, 5.3

[CMG+18] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018. 5.3

[CPT99] Calos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *Working Conference on Software Architecture*, pages 107–125. Springer, 1999. 3.3.1

[CSKX15] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. DeepDriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015. 1.1, 1.3.3, 4.1.4, 2, 4.10.4

[CSZ09] Igor Cavrak, Armin Stranjak, and Mario Zagar. Sdlmas: A scenario modeling framework for multi-agent systems. *J. UCS*, 15(4):898–925, 2009. 2.6.3

[CvMBB14] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. *Syntax, Semantics and Structure in Statistical Translation*, page 103, 2014. 4.8.2

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. 4.1.4

[DAPM00] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000. 6.8.5

[DB09] Howard Demuth and Mark Beale. Matlab neural network toolbox user's guide version 6. *The MathWorks Inc.*, 2009. 4.4

[DCBL+18] Elias De Coninck, Steven Bohez, Sam Leroux, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. DIANNE: a modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure. *Journal of Systems and Software*, 141:52–65, 2018. 4.4

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 4.1.4, 4.8.2

[DDE+17] Jens Dankert, Christian Dernehl, Lutz Eckstein, Stefan Kowalewski, Evgeny Kusmenko, and Bernhard Rumpe. RapidCoop - Robuste Architektur durch geeignete Paradigmen für Kooperativ Interagierende Automobile. In *Automatisiertes und Vernetztes Fahren (AAET'17)*, February 2017. 7, 1.6, 3.1, 6.3

[DDM+07] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A next-generation design framework for platform-based design. In *Conference on using hardware design and verification languages (DVCon)*, volume 152, 2007. 6.2

[DFI+15] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015. 5.5, A.3, B.4

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019. 7, 1.6, 1.7.5

[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. 1.7.2

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai

Wu, and Peter Zhokhov. OpenAI Baselines. `https://github.com/openai/baselines`, 2017. 5.3

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011. 4.9

[DJM+19] Manuela Dalibor, Nico Jansen, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. Towards Sustainable Systems Engineering-Integrating Tools via Component and Connector Architectures. In Georg Jacobs and Jonas Marheineke, editors, *Antriebstechnisches Kolloquium 2019: Tagungsband zur Konferenz*, pages 121–133. Books on Demand, February 2019. 1.1

[DKS+12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. RobotML, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012. 6.2

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 3.4.6

[dMdRKY19] Cyprien de Masson d'Autume, Sebastian Ruder, Lingpeng Kong, and Dani Yogatama. Episodic memory in lifelong language learning. In *Advances in Neural Information Processing Systems*, pages 13143–13152, 2019. 4.8.2

[Doy00] Kenji Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1):219–245, 2000. 5.1

[dPeM19] Bureau International des Poids et Mesures. *SI Brochure: The International System of Units (SI)*. The address, 9 edition, 2019. bipm.org. 1.7.4, A.2, A.3, A.4, B.4

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017. 2.6.3, 6.7

[DS13] Alexander Munro Davie and Andrew James Stothers. Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh Section A: Mathematics*, 143(2):351–369, 2013. 2.4.3

[Eat93] John W. Eaton. Octave – a high-level interactive language for numerical computations. 1993. 2.5

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309, 2010. 1.7.3

[ECP+16] Cristofer Englund, Lei Chen, Jeroen Ploeg, Elham Semsar-Kazerooni, Alexey Voronov, Hoai Hoang Bengtsson, and Jonas Didoff. The grand cooperative driving challenge 2016: boosting the introduction of cooperative automated vehicles. *IEEE Wireless Communications*, 23(4):146–152, 2016. 3.1

[FG12] Peter H. Feiler and David P. Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language.* Addison-Wesley, 2012. 3.3.1

[FHK+11] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011. 4.3, 4.6

[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. 7, 1.6, 2.6.3

[FLV06] Peter H. Feiler, Bruce A. Lewis, and Steve Vestal. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211. IEEE, 2006. 1.1

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language.* Morgan Kaufmann, 2014. 1.7.5

[Fow10] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010. 1.7.3

[FV10] Martin Fellendorf and Peter Vortisch. Microscopic traffic flow simulator VISSIM. In *Fundamentals of traffic simulation*, pages 63–93. Springer, 2010. 2.6.3

[FvHM18] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. 5.1

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011. 4.1.3

[GD11] Sanjay Ghemawat and Jeff Dean. LevelDB. *URL: https://github.com/google/leveldb, http://leveldb. org*, 2011. 4.3, 4.6

[GEB16] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016. 4.7.1

[GF12] Jorge Gómez Fernández. A vehicle dynamics model for driving simulators (master thesis). 2012. 2.6.3

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley, 1995. 1.7.3, 2.5, 6.6

[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. 1.7.2

[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. 7, 1.6, 2.6.3

[GKR19] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In *Proceedings of MODELS 2019. Workshop MDE Intelligence*, pages 196–202, September 2019. 7, 1.6, 5, 5.4, 5.5, 5.7, 5.8, 5.9, B.4

[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. 4.6, 4.10.2

[GLSL16] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep Q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016. 5.1

[GLSU13] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The KITTI dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013. 1.1, 1.3.3

[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. 1.7.2

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. 5.9

[Gro17] Object Management Group. OMG systems modeling language™: Version 1.5. 2017. 1.7.5

[GS00] Felix A. Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000. 4.1.4

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016. 1.7.3, 1.7.6

[Hil14] Ernst Hildebrand. Forstliche bodenbewirtschaftung. *Handbuch der Bodenkunde*, pages 1–22, 2014. 5.10.2

[HJ90] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1990. 2.2.3, 2.1, 2.4.3, B.4

[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. 7, 1.6, 1.1, 1.7.5, 2.4.6, B.4

[HKKR19] Alexander David Hellwig, Stefan Kriebel, Evgeny Kusmenko, and Bernhard Rumpe. Component-based integration of interconnected vehicle architectures. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 153–158. IEEE, 2019. 7, 1.6, 6.1, 6.1, 6.2, 6.5, 6.7, 6.8, 6.13, B.4

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, volume 9839 of *LNCS*, pages 175–182. Springer, December 2016. 1.7.6, 3.3.1

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 1.7.2

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 3.3.1

[HPHC16] Nadine Herold, Stephan-A Posselt, Oliver Hanka, and Georg Carle. Anomaly detection for SOME/IP using complex event processing. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pages 1221–1226. IEEE, 2016. 6.5

[HR00] David Harel and Bernhard Rumpe. Modeling languages: Syntax, Semantics and All That Stuff (Part I: The Basic Stuff). Technical Report. *MCS00-16, Mathematics & Computer Sience, Weizmann Institute Of Science*, 2000. 1.7.2

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004. 1.7.2

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017. 1.7.3, 3.4, 3.5, 4.5, 4.10

[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. 1.7.6, 2.3.1, 3.3.1

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 4.1.4

[HSGP05] Brian Henderson-Sellers and Cesar Gonzalez-Perez. The rationale of powertype-based metamodelling to underpin software development methodologies. In *Conferences in Research and Practice in Information Technology Series*, 2005. 4.9

[HSK+12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012. B.1

[HTSC08] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. MQTT-S - a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE, 2008. 6.5

[HW08] Mordechai Haklay and Patrick Weber. OpenStreetMap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008. 2.6.3

[HZRS16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 4.1.4, 4.3, 4.5, 4.7.3

[HZRS16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016. 4.1.4

[IEE08] IEEE. IEEE-754, Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–58, 01 2008. 2.2.1

[Ins98] National Instruments. BridgeView and LabView: G Programming Reference Manual. Technical Report 321296B-01, National Instruments, 1998. 1.7.5

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. B.1

[IZZE17] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017. 5.9

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014. 4.3

[KAE+12] Roozbeh Kianfar, Bruno Augusto, Alireza Ebadighajari, Usman Hakeem, Josef Nilsson, Ali Raza, Reza S. Tabar, Naga VishnuKanth Irukulapati, Cristofer Englund, Paolo Falcone, et al. Design and experimental validation of a cooperative driving system in the grand cooperative driving challenge. *IEEE transactions on intelligent transportation systems*, 13(3):994–1007, 2012. 1.3.1

[Kau21] Oliver Kautz. *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Aachener Informatik-Berichte, Software Engineering, Band 46. Shaker Verlag, April/May 2021. 4.10.3

[KB98] Wojtek Kozaczynski and Grady Booch. Component-based software engineering. *IEEE software*, 15(5):34, 1998. 1.7.6

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 4.9

[KH04] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004. 2.6.3, 6.7

[KJKD05] Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. In *ICEIS (3)*, pages 189–195, 2005. 3.3.1

[KK03] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP: A Practitioner's Guide to the RUP*. Addison-Wesley Professional, 2003. 1.7.2

[KKM+19] Maximilian Kloock, Ludwig Kragl, Janis Maczijewski, Bassam Alrifaee, and Stefan Kowalewski. Distributed model predictive pose control of multiple nonholonomic vehicles. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1620–1625. IEEE, 2019. 1.3.2

[KKMR19] Jörg Christian Kirchhof, Evgeny Kusmenko, Jean Meurice, and Bernhard Rumpe. Simulation of Model Execution for Embedded Systems. In *Proceedings of MODELS 2019. Workshop MLE*, pages 331–338. IEEE, September 2019. 7, 1.6, 2.6.3

[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. 1.7.2

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications. 7, 1.6, 3.1

[KKRS19] Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Igor Shumeiko. Learning error patterns from diagnosis trouble codes. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 179–184. IEEE, 2019. 7, 1.6, 5.11

[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. 7, 1.6, 1.7.5, 3.4.5

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019. 7, 1.6, 2.6.3

[KLG+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A metamodel family for role-based modeling and programming languages. In *International Conference on Software Language Engineering*, pages 141–160. Springer, 2014. 5.5

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In *Conference on Model Driven Engineering*

Languages and Systems (MODELS'19), pages 283–293. IEEE, September 2019. 7, 1.6, 4.5, 4.6, 4.19, 4.20, 4.25, B.4

[KNRSV00] Kurt Keutzer, A. Richard Newton, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE transactions on computer-aided design of integrated circuits and systems*, 19(12):1523–1543, 2000. 6.2

[Koz16] Štefan Kozák. From PID to MPC: Control engineering methods development and applications. In *2016 Cybernetics & Informatics (K&I)*, pages 1–7. IEEE, 2016. 1.3.2

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Driven Systems. In *ASE'19. Software Engineering Intelligence Workshop (SEI'19)*, pages 126–133. IEEE, November 2019. 7, 1.6, 4.5, 4.5, 4.22, 4.29, 4.10.4, 4.30, 4.31, B.4

[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. 7, 1.6, 1.7.6, 2, 4.5

[KRRvW18] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc: Textual Modeling Alternative to Simulink. In *Proceedings of MODELS 2018. Workshop EXE*, October 2018. 7, 1.6

[KRSvW18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018. 7, 1.6, 2, 2.5, 4.5

[KRSvW18b] Evgeny Kusmenko, Bernhard Rumpe, Ievgen Strepkov, and Michael von Wenckstern. Teaching Playground for C&C Language EmbeddedMontiArc. In *Proceedings of MODELS 2018. Workshop ModComp*, October 2018. 7, 1.6, 2.5

[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. 1.7.3

[KSB+19] Maximilian Kloock, Patrick Scheffe, Lukas Botz, Janis Maczijewski, Bassam Alrifaee, and Stefan Kowalewski. Networked model predictive vehicle race control. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 1552–1557. IEEE, 2019. 1.3.2

[KSF17] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a TensorFlow library for applied reinforcement learning. https://github.com/tensorforce/tensorforce, 2017. 5.3

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 4.1.4, 4.7, 4.10.4, B.1, B.4

[KSM+19] Maximilian Kloock, Patrick Scheffe, Sascha Marquardt, Janis Maczijewski, Bassam Alrifaee, and Stefan Kowalewski. Distributed model predictive intersection control of multiple vehicles. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 1735–1740. IEEE, 2019. 1.3.2

[KSRvW18] Evgeny Kusmenko, Igor Shumeiko, Bernhard Rumpe, and Michael von Wenckstern. Fast Simulation Preorder Algorithm. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 256 – 267. SciTePress, January 2018. 7, 1.6

[KWR+16] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. VizDoom: A doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016. 5.3

[LAB+11] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168, 2011. 1.1

[LB+95] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995. 4.1.4

[LBBW+18] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes

Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using SUMO. In *The 21st IEEE International Conference on Intelligent Transportation Systems*, pages 2575–2582. IEEE, November 2018. 2.6.3

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. 4.10.3

[LCL13] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *CoRR*, abs/1304.1672, 2013. 2.6.3, 5.4, 6.7

[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. 1.1

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303, 2014. 2.4.3

[LH17] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 4.9

[LHHN17] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. *arXiv preprint arXiv:1702.02181*, 2017. 4.3

[LHP+15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 5.1, 5.6.1, B.3.1

[Lim18] DeepMind Technologies Limited. Tensorflow reinforcement learning: TRFL. https://github.com/deepmind/trfl, 2018. 5.3

[Lin93] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993. 5.1, 5.6.2, B.3.1

[LLH+19] Younkwan Lee, Juhyun Lee, Yoojin Hong, YeongMin Ko, and Moongu Jeon. Unconstrained road marking recognition with generative adversarial networks. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1414–1419. IEEE, 2019. 5.9

[LMD+19] Ignacio Llatser, Thomas Michalke, Maxim Dolgov, Florian Wildschütte, and Hendrik Fuchs. Cooperative Automated Driving Use Cases for 5G V2X Communication. In *2019 IEEE 2nd 5G World Forum (5GWF)*, pages 120–125. IEEE, 2019. 3.1

[LNC+11] Quoc V. Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In *ICML*, 2011. 4.1.3

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017. 4.10.2

[LSR+19] Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. In *Advances in Neural Information Processing Systems*, pages 8548–8559, 2019. 4.8.2

[Mat16] Mathworks Inc. Simulink User's Guide. Technical Report R2016b, MATLAB & SIMULINK, 2016. 1.7.5, 1.7.6, 2.3.3, 3.3.1, 4.4, 6.2

[MB12] Jacob Mattingley and Stephen Boyd. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012. 2.4.5

[MBB+08] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, 25(9):569–597, 2008. 1.3.1

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *European Software Engineering Conference*, pages 137–153. Springer, 1995. 3.3.1

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14, 1996. 2.3.2, 3.4.4

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 5.1, 5.1, 5.6.2, B.3.1

[MM10] Irene Moser and Sanaz Mostaghim. The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimisation. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010. 6.8.5

[MMR+17] Shahar Maoz, Ferdinand Mehlan, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. OCL Framework to Verify Extra-Functional Properties in Component and Connector Models. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017. 1.7.3, 3.4.5

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and computation*, 100(1):41–77, 1992. 3.3.1

[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)*, volume 1723 of *CEUR Workshop Proceedings*, pages 19–24, October 2016. 4.10.2

[MS17] Keith McCormick and Jesus Salcedo. Model complex interactions with IBM SPSS neural networks. In *SPSS® Statistics for Data Analysis and Visualization*, pages 325–353. April 2017. 4.4

[MSK+12] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 400–411. Springer, 2012. 2.6.3

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017. 1.7.3, 4.5, 4.10

[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. 1.7.6

[Mye13] Roger B. Myerson. *Game theory*. Harvard university press, 2013. 3.1

[Nai17] Nitin Naik. Choice of effective messaging protocols for iot systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE international systems engineering symposium (ISSE)*, pages 1–7. IEEE, 2017. 6.5

[Nes83] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence O(1/kˆ 2). In *Doklady ANSSSR*, volume 269, pages 543–547, 1983. 4.9

[NHW14] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A survey on domain-specific languages in robotics. In *International conference on simulation, modeling, and programming for autonomous robots*, pages 195–206. Springer, 2014. 1.7.2

[Nin97] Jim Q. Ning. Component-based software engineering (CBSE). In *Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies*, pages 34–43. IEEE, 1997. 1.7.6

[NJW02] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856, 2002. 6.8.2

[NPLS18] Maximilian Naumann, Fabian Poggenhans, Martin Lauer, and Christoph Stiller. CoInCar-Sim: An Open-Source Simulation Framework for Cooperatively Interacting Automobiles. In *IEEE Intl. Conf. Intelligent Vehicles*, 2018. 6.7

[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. 3.3.1

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. 1.7.2, 1.7.3

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 4.7.3

[PH19] Uwe Pohlmann and Marcus Hüwe. Model-driven allocation engineering: specifying and solving constraints based on the example of automotive systems. *Automated Software Engineering*, 26(2):315–378, Jun 2019. 6.8.5

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. 1981. 1.7.2

[Pom89] Dean A. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989. 1.3.3

[PRL+08] M. Piórkowski, M. Raya, A. Lezama Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. TraNS: Realistic Joint Traffic and Network Simulator for VANETs. *SIGMOBILE Mobile Computing and Communications Review*, 12(1):31–33, January 2008. 2.6.3

[PRWZ02] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002. 4.9

[PWH+18] Christoph Pallasch, Stephan Wein, Nicolai Hoffmann, Markus Obdenbusch, Tilman Buchner, Josef Waltl, and Christian Brecher. Edge powered industrial control: Concept for combining cloud and automation technologies. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 130–134. IEEE, 2018. 1.1

[QBC19] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. GeoScenario: An open DSL for autonomous driving scenario representation. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 287–294. IEEE, 2019. 2.6.3

[QGC+09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, May 2009. 2.6.3, 5.7, 6.5

[RBL+08] Fred W. Rauskolb, Kai Berger, Christian Lipski, Marcus Magnor, Karsten Cornelsen, Jan Effertz, Thomas Form, Fabian Graefe, Sebastian Ohl, Walter Schumacher, et al. Caroline: An autonomously driving vehicle for urban environments. *Journal of Field Robotics*, 25(9):674–724, 2008. 1.3.1

[RCO65] Anatol Rapoport, Albert M. Chammah, and Carol J. Orwant. *Prisoner's dilemma: A study in conflict and cooperation*, volume 165. University of Michigan press, 1965. 3.1

[Rin11] Ola Ringdahl. *Automation in forestry: development of unmanned forwarders*. PhD thesis, Institutionen för datavetenskap, Umeå Universitet, 2011. 5.10.2

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014. 1.7.6

[RK19] Jean Rabault and Alexander Kuhnle. Accelerating deep reinforcement learning strategies of flow control through a multi-environment approach. *Physics of Fluids*, 31(9):094105, 2019. 5.3

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015. 5.9

[RMK+13] Michele Rondinone, Julen Maneros, Daniel Krajzewicz, Ramon Bauza, Pasquale Cataldi, Fatma Hrizi, Javier Gozalvez, Vineet Kumar, Matthias Röckl, Lan Lin, et al. iTETRIS: a modular simulation platform for the large scale evaluation of cooperative ITS applications. *Simulation Modelling Practice and Theory*, 34:99–125, 2013. 2.6.3

[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. 1.7.2

[Rou87] Peter J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987. 6.8.2

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015. 2.5

[RSRS99] Bernhard Rumpe, Maurice Schoenmakers, Ansgar Radermacher, and A Schuerr. UML + ROOM as a standard ADL? In *Proceedings Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)(Cat. No. PR00434)*, pages 43–53. IEEE, 1999. 3.3.1

[Rum11a] Bernhard Rumpe. *Modellierung mit UML*. Springer Berlin Heidelberg, 2011. 1.7.3

[Rum11b] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011. 1.1, 1.7.2

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017. 4.6

[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. 1.1

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019. 4.1.4

[SA16] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016. 4.3

[San16] Andrew Sanders. *An introduction to Unreal engine 4*. CRC Press, 2016. 2.6.3

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008. 1.7.2, 1.7.3, 6.2

[SC16] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1(2):26, 2016. 2.5

[SGD11] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, January 2011. 2.6.3

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling*, volume 2. Elsevier, 1994. 3.3.1

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014. B.1

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017. 5.1

[SKE+18] Michael Schaarschmidt, Alexander Kuhnle, Ben Ellis, Kai Fricke, Felix Gessert, and Eiko Yoneki. LIFT: Reinforcement learning in computer

systems by learning from demonstrations. *CoRR*, abs/1808.07903, 2018. 5.3

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org, 2014. 5.1

[SSG+15] Jan R. Seyler, Thilo Streichert, Michael Glaß, Nicolas Navet, and Jürgen Teich. Formal analysis of the startup delay of SOME/IP service discovery. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 49–54. IEEE, 2015. 6.5

[Ste00] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000. 5.5

[SWXC16] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pages 99–104. IEEE, 2016. 4.3

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 4.1.4

[Tan13] Yichuan Tang. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*, 2013. 5.11

[UBB+07] Chris Urmson, J. Andrew Bagnell, Christopher Baker, Martial Hebert, Alonzo Kelly, Raj Rajkumar, Paul E. Rybski, Sebastian Scherer, Reid Simmons, Sanjiv Singh, et al. Tartan Racing: A multi-modal approach to the DARPA Urban Challenge. 2007. 1.3.1

[UO30] George E. Uhlenbeck and Leonard S. Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930. 5.1, B.3.1

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016. B.3.2

[VL07] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007. 6.8.2

[VR77] Herbert B. Voelcker and Aristides A.G. Requicha. Geometric modeling of mechanical parts and processes. *Computer*, 10(12):48–57, 1977. 1.1

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 4.1.4, 4.8.2

[vW20] Michael von Wenckstern. *Verification of Structural and Extra-Functional Properties in Component and Connector Models for Embedded and Cyber-Physical Systems*. PhD thesis, RWTH Aachen University, Germany, 2020. 1.7.6, 2, 2.3.1, 4.10.2, 15, 28, 100, 199, 20, 12

[Wat89] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989. 5.1

[WB06] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006. 2.5

[WCV11] Stéfan van der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. 4.3

[WD10] Xiang Wang and Ian Davidson. Flexible constrained spectral clustering. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 563–572. ACM, 2010. 6.8.4

[WEG+00] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. TORCS, the open racing car simulator. *Software available at http://torcs.sourceforge.net*, 4(6), 2000. 2.6.3, 4.10.4, 5.4, 6.7

[Wil11] Virginia Vassilevska Williams. Breaking the Coppersmith-Winograd barrier. *E-mail address: jml@ math. tamu. edu*, 2011. 2.4.3

[WM95] Robert W. Weeks and John J. Moskwa. Automotive engine modeling for real-time control using Matlab/Simulink. *SAE transactions*, pages 295–309, 1995. 1.1

[WMS04] S. Wang, J. R. Merrick, and K. G. Shin. Component allocation with multiple resource constraints for large embedded real-time software design. In *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, pages 219–226, May 2004. 6.8.5

[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. 1.7.6, 6.2

[WPR+08] Axel Wegener, Michał Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. TraCI: An Interface for Coupling Road Traffic and Network Simulators. In *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS '08, pages 155–163, New York, NY, USA, 2008. ACM. 2.6.3

[WQD14] Xiang Wang, Buyue Qian, and Ian Davidson. On constrained spectral clustering and its applications. *Data Mining and Knowledge Discovery*, 28(1):1–30, 2014. 6.8.4

[XBK+15] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015. A.4, B.4

[YCC+17] Guang-Zhong Yang, James Cambias, Kevin Cleary, Eric Daimler, James Drake, Pierre E. Dupont, Nobuhiko Hata, Peter Kazanzides, Sylvain Martel, Rajni V. Patel, et al. Medical robotics—regulatory, ethical, and legal considerations for increasing levels of autonomy. *Science Robotics*, 2(4):8638, 2017. 1.1

[ZCD+18] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018. 4.10.3

[Zei12] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012. 4.9

[ZHL+19] Miankuan Zhu, Lei Han, Fujian Liang, Chaoxing Xi, Lei Wu, and Zutao Zhang. A novel vehicle open door safety system based on cyclist detection using fisheye camera and improved deep convolutional generative adversarial nets. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2195–2201. IEEE, 2019. 5.9

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 4.1.4

[ZLNH17] Yi Zhu, Zhenzhong Lan, Shawn Newsam, and Alexander G. Hauptmann. Guided optical flow learning. *arXiv preprint arXiv:1702.02295*, 2017. 4.9

[ZS17] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017. 5.6.2

Appendix A

Diagrams and Listings

Tag	Description
bash	bash script
C++	C++ Code
CD	Class Diagram
CNNArc	MontiAnna Architecture (CNNArc)
CNNTrain	MontiAnna Training Configuration (CNNTrain)
EMA	EmbeddedMontiArc
EMAD	EmbeddedMontiArc Dynamics
EMAM	EmbeddedMontiArc with MontiMath
EMADL	EmbeddedMontiArc with MontiMath and Deep Learning
Enum	EmbeddedMontiArc enumeration
gin	Google Dopamine gin configuration
Maven	Maven Build Script
Net#	Net# neural network definition
MontiMath	MontiMath
Prolog	Prolog
Python	Python
Stream	EmbeddedMontiArc stream test
Struct	EmbeddedMontiArc structure

Tag Model	EmbeddedMontiArc tag model
Tag Schema	EmbeddedMontiArc tag schema
Z3	Z3 program

Table A.1: Explanation of the used flags in listings and figures.

Unit	Syntax	Quantity
minute	min	time
hour	h	time
day	d	time
hectare	ha	area
ton	t	mass
litre	l, L	volume
astronomical unit	au	length
neper	Np	logarithmic ratio quantity
bel	B	logarithmic ratio quantity
decibel	dB	logarithmic ratio quantity
electronvolt	eV	energy
dalton	Da	mass

Table A.2: Accepted non-SI units [dPeM19].

Unit	Syntax	Quantity
meter	m	length
gram	g	mass
second	s	time
ampere	A	electric current
kelvin	K	temperature
mole	mol	amount of substance
candela	cd	luminous intensity
hertz	Hz	frequency
newton	N	force
pascal	Pa	pressure
joule	J	energy
watt	W	power
coulomb	C	electric charge
volt	V	voltage
farad	F	capacitance
ohm	Ω, Ohm	resistance
siemens	S	electrical conductance
weber	Wb	magnetic flux
tesla	T	magnetic flux density
henry	H	inductance
degree Celsius	°C	temperature
lumen	lm	luminous flux
lux	lx	illuminance
becquerel	Bq	radioactivity
gray	Gy	absorbed dose
sievert	Sv	equivalent dose
katal	kat	catalytic activity
radian	rad	plane angle
steradian	sr	solid angle

Table A.3: Base and derived units of the SI system [dPeM19].

Prefix	Syntax	Factor
yotta	Y	10^{24}
zetta	Z	10^{21}
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	k	10^{3}
hecto	h	10^{2}
deca	da	10^{1}
deci	d	10^{-1}
centi	c	10^{-2}
milli	m	10^{-3}
micro	μ, u	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}
zepto	z	10^{-21}
yocto	y	10^{-24}

Table A.4: SI unit prefixes [dPeM19].

Prolog

```
%psd(m1,m2,op).
psd(X,Y,' + '):- psd(X), psd(Y).
psd(X,Y,' - '):- psd(X), nsd(Y).
psd(X,Y,' * '):- psd(X), scal+(Y).
psd(X,Y,' * '):- psd(X), int+(Y).
psd(X,Y,' * '):- psd(Y), scal+(X).
psd(X,Y,' * '):- psd(Y), int+(X).
psd(X,Y,' + '):- diag(X), pos(X), diag(Y), pos(Y).
psd(X,Y,' - '):- diag(X), pos(X), diag(Y), neg(Y).
psd(X,Y,' * '):- diag(X), pos(X), diag(Y), pos(Y).
psd(X,Y,' * '):- diag(X), neg(X), diag(Y), neg(Y).

%pd(m1,m2,op).
pd(X,Y,' + '):- pd(X), pd(Y).
pd(X,Y,' - '):- pd(X), nsd(Y).
pd(X,Y,' * '):- pd(X), scal+(Y).
pd(X,Y,' * '):- pd(X), int+(Y).
pd(X,Y,' * '):- pd(Y), scal+(X).
pd(X,Y,' * '):- pd(Y), int+(X).

%nsd(m1,m2,op).
nsd(X,Y,' + '):- nsd(X), nsd(Y).
nsd(X,Y,' - '):- nsd(X), psd(Y).
nsd(X,Y,' * '):- nsd(X), scal+(Y).
nsd(X,Y,' * '):- nsd(X), int+(Y).
nsd(X,Y,' * '):- nsd(Y), scal+(X).
nsd(X,Y,' * '):- nsd(Y), int+(X).

%inv(m1,m2,op).
inv(X):- pd(X).
inv(X):- nd(X).).
inv(X,'inv'):- inv(X)
```

Figure A.1: Prolog rules for the derivation of matrix properties (PSD, PD, NSD, invertible).

```prolog
%nd(m1,m2,op).
nd(X,Y,' + '):- nd(X), nd(Y).
nd(X,Y,' - '):- nd(X), psd(Y).
nd(X,Y,' * '):- nd(X), scal+(Y).
nd(X,Y,' * '):- nd(X), int+(Y).
nd(X,Y,' * '):- nd(Y), scal+(X).
nd(X,Y,' * '):- nd(Y), int+(X).

%norm(m1,m2, * ).
norm(X,Y,' * '):- norm(X), scal(Y).
norm(X,Y,' * '):- norm(X), int(Y).
norm(X,Y,' * '):- norm(Y), scal(X).
norm(X,Y,' * '):- norm(Y), int(X).
norm(X,Y,' ^ '):- norm(X), int(Y).
norm(X,Y,' + '):- herm(X), herm(Y).
norm(X,Y,' - '):- herm(X), herm(Y).
norm(X,'inv'):- herm(X), inv(X).
norm(X,'trans'):- herm(X).

%skewHerm(m1,m2,op).
skewHerm(X,Y,' + '):- skewHerm(X), skewHerm(Y).
skewHerm(X,Y,' - '):- skewHerm(X), skewHerm(Y).
skewHerm(X,Y,' * '):- skewHerm(X), scal(Y).
skewHerm(X,Y,' * '):- skewHerm(X), int(Y).
skewHerm(X,Y,' * '):- skewHerm(Y), scal(X).
skewHerm(X,Y,' * '):- skewHerm(Y), int(X).
skewHerm(X,'inv'):- skewHerm(X), inv(X).
skewHerm(X,'trans'):- skewHerm(X).
```

Figure A.2: Prolog rules for the derivation of matrix properties (ND, normal, skew Hermitian).

EMADL

```
component FlowNet{
ports in Q(0:255)^{3, 384, 512} data_0,
      in Q(0:255)^{3, 384, 512} data_1,
      out Q(-oo:oo)^{2, 96, 128} target_0,
      out Q(-oo:oo)^{2, 48, 64} target_1,
      out Q(-oo:oo)^{2, 24, 32} target_2,
      out Q(-oo:oo)^{2, 12, 16} target_3,
      out Q(-oo:oo)^{2, 6, 8} target_4;
implementation CNN {
def conv(channels, kernel=3, stride=2, pad="same"){
             Convolution(kernel=(kernel,kernel), stride=(stride,stride),
             channels=channels, padding=pad) -> Relu()}

(data_0 | data_1) -> Concatenate() ->
conv(channels=64, kernel=7, stride=2, pad="same") ->
conv(channels=128, kernel=5, stride=2, pad="same") ->
( conv(channels=256, kernel=5, stride=2, pad="same") -> //3
conv(channels=256, kernel=3, stride=1, pad="same") -> //3_
(conv(channels=512, kernel=3, stride=2, pad="same") -> //4
conv(channels=512, kernel=3, stride=1, pad="same") -> //4_1
( conv(channels=512, kernel=3, stride=2, pad="same") -> //5
 conv(channels=512, kernel=3, stride=1, pad="same") -> //5_1
( conv(channels=1024, kernel=3, stride=2, pad="same") -> //6
conv(channels=1024, kernel=3, stride=1, pad="same") -> //6_1
( UpConvolution(channels=512, kernel=(4,4), stride=(2,2), padding="same") ->
Relu() | Convolution(kernel=(3,3), stride=(1,1), channels=2, padding="same") -> (
target_4 |
UpConvolution(kernel=(4,4), stride=(2,2), channels=2, padding="same")
) -> [1]) -> Concatenate()|[0] ) -> Concatenate() ->
(UpConvolution(channels=256, kernel=(4,4), stride=(2,2), padding="same") ->
Relu() |Convolution(kernel=(3,3), stride=(1,1), channels=2, padding="same") ->
( target_3 |
UpConvolution(kernel=(4,4), stride=(2,2), channels=2, padding="same")
) -> [1]) -> Concatenate() | [0]) -> Concatenate() ->
(UpConvolution(channels=128, kernel=(4,4), stride=(2,2), padding="same") ->
Relu()|Convolution(kernel=(3,3), stride=(1,1), channels=2, padding="same") ->
( target_2 |
UpConvolution(kernel=(4,4), stride=(2,2), channels=2, padding="same")
) ->[1]) -> Concatenate()  | [0]) -> Concatenate() ->
(UpConvolution(channels=64, kernel=(4,4), stride=(2,2), padding="same") ->
Relu() |
Convolution(kernel=(3,3), stride=(1,1), channels=2, padding="same") ->
(target_1 | UpConvolution(kernel=(4,4), stride=(2,2), channels=2, padding="same")
) -> [1]) -> Concatenate() | [0]) -> Concatenate() ->
Convolution(kernel=(3,3), stride=(1,1), channels=2, padding="same") ->
target_0;}}
```

Figure A.3: FlowNet [DFI+15] modeled in EMADL by Julian Steinsberger-Dührßen during his lab work.

CNNArc

```
package showAttendTell;
component ShowAttendTell {
  ports in Z(-oo:oo)^{64,2048} data,
        in Z(0:255)^{3,224,224} images,
        out Z(0:37758)^{1} target[25];
  implementation CNN{
    layer LSTM(units=512) decoder;
    layer FullyConnected(units = 256) features;
    layer FullyConnected(units = 1, flatten=false) attention;
    0 -> target[0];
    images ->
    Convolution(kernel=(7,7), channels=128, stride=(7,7)) ->
    Convolution(kernel=(4,4), channels=128, stride=(4,4)) ->
    Reshape(shape=(64, 128)) -> features;
    timed <t> GreedySearch(max_length=25){
      (((features.output -> FullyConnected(units=512, flatten=false)
      |
      decoder.state[0] -> FullyConnected(units=512, flatten=false))->
      BroadcastAdd() -> Tanh() -> FullyConnected(units=1, flatten=false)->
      Softmax(axis=0) -> attention | features.output) ->
      BroadcastMultiply() -> ReduceSum(axis=0) -> ExpandDims(axis=0)
      |
      target[t-1] -> Embedding(output_dim=256)) ->
      Concatenate(axis=1) -> decoder -> FullyConnected(units=37758) ->
      Tanh() -> Dropout(p=0.25) -> Softmax() -> ArgMax() ->
      target[t]};
  }
}
```

Figure A.4: The Show, Attend and Tell network architecture [XBK+15] modeled in CNNArc by Christian Fuß in his master thesis. The aim of the network is to generate textual descriptions for given images. To do so the architecture combines convolutional layers for image processing, attention, and RNNs.

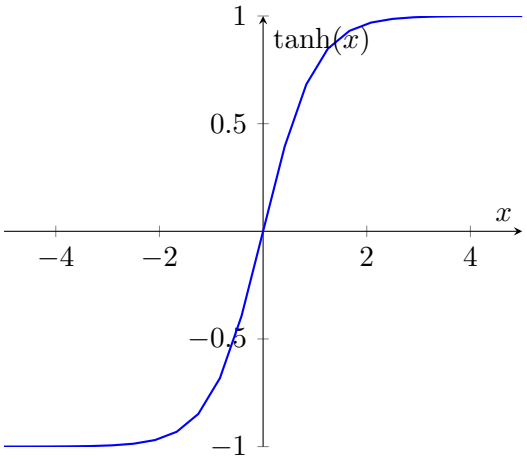
Appendix B

Further Documentation

B.1 CNNArc Layer Classes

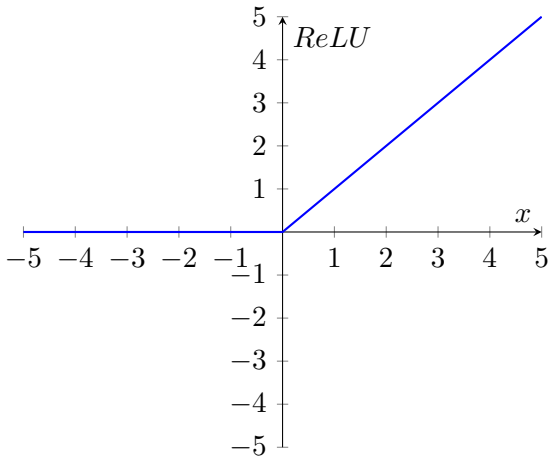
Tanh. The Tanh is a non-linearity often used for binary classification as it provides a sign preserving mapping.

- **Parameters:** none.
- **Connection pattern:** 1-to-1.
- **Function:** $\sigma(x) = \tanh(x) = 1 - \frac{2}{e^{2x}+1}$.



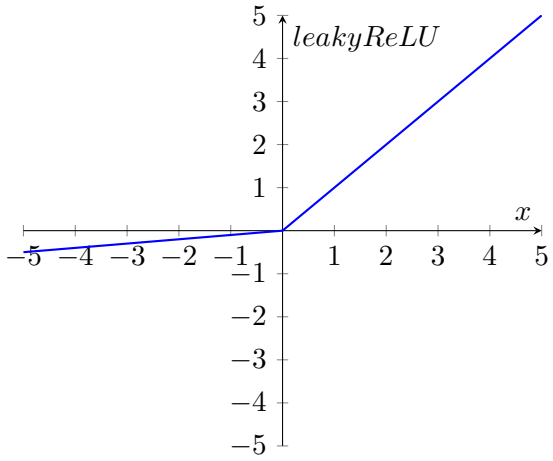
Relu. The ReLU is a widely used activation function, which maps the input to itself for positive values and to zero otherwise.

- **Parameters:** none.
- **Connection pattern:** 1-to-1.
- **Function:** $\sigma(x) = \max(0, x)$.



LeakyRelu. A variant of the ReLU layer, where inputs less than zero are assigned a small slope greater than zero. Consequently, inactive neurons can have a non-zero gradient.

- **Parameter:** `Q epsilon` is a coefficient controlling the slope of the leaky part of the ReLU.
- **Connection pattern:** 1-to-1.
- **Function:** $\sigma(x) = \begin{cases} x & \text{if } x \geq 0 \\ \epsilon x & \text{else.} \end{cases}$



Pooling. Pooling is another important operation in the context of CNNs. It is a data reduction technique mostly applied after a Convolutional layer.

- **Parameters:** the Pooling layer takes the same sliding window parameters as the Convolutional layer, i.e. `N^2 kernel`, `N stride`, and `Padding padding`. Furthermore, the layer supports the `Pool_type pool_type` parameter, cf. function.
- **Connection pattern:** Same as for the convolutional layer.
- **Function:** Similarly to a Convolutional layer the Pooling function can be thought of as a sliding window. However, the Pooling layer has no weights to be learned. Instead, it outputs a single scalar per sliding step by applying a fixed aggregation function to the data inside the window. There are two supported operation modes controlled by the parameter `Pool_type pool_type`: `"max"` outputs the biggest value present inside the window. `"average"` computes and outputs the arithmetic mean.

GlobalPooling. Global pooling is similar to standard pooling, but it has no sliding window. Instead it applies the aggregation operation to the whole input matrix at once.

Lrn. The Lrn layer class implements local response normalization (LRN), a lateral inhibition technique [KSH12]. Its purpose is to emphasize the most excited neurons and dampen their relatively weaker neighborhood activations. The output of the normalization operation has the same shape as the input. The layer has no weights to be learned.

- **Parameters:** the layer function is controlled by the hyperparameters presented in the original paper, namely `Q alpha`, `Q beta`, `Q k`, and `Q n`. They can be tuned with the help of a validation set.
- **Connection pattern:** 1-to-1.

BatchNorm. Batch normalization is another normalization technique [IS15] and is available in MontiAnna though the BatchNorm layer class. It subtracts the minibatch mean of the layer output it is applied to and divides it by the minibatch standard deviation. Batch normalization helps stabilizing and accelerating the training and has a regularization effect.

Dropout. Dropout is a regularization technique temporarily removing a random selection of neurons from a neural network in each learning step [HSK+12, SHK+14]. Dropout regularization can be applied to a layer instance by appending a Dropout layer.

This layer introduces randomness into the model and might hence lead to irreproducible results. To ensure reproducibility a seed can be used, cf. Section 4.9.

- **Parameters:** `Q(0:1) rate` represents the dropout rate controlling the ratio of neurons to be dropped out at each step.
- **Connection pattern:** 1-to-1.

OneHot. One-hot encoding represents categorical data in a vector space. Each category is mapped to a unique vector where exactly one entry is 1, whereas all the remaining entries are zero. The distance between two arbitrary, non-equal one-hot vectors is a constant. A further advantage is that the encoding does not need to be learned. However, spanning a dedicated dimension for each category, the one-hot representation is highly inefficient, particularly for large vocabularies. Furthermore, it cannot represent semantic similarities between categories. This is a drawback for language processing as words can have different degrees of similarity, ranging from being synonyms, describing realizations of the same concept, e.g. colors, or being completely unrelated. The OneHot layer class can be used to map a scalar input to a one-hot vector.

- **Parameter:** `N(1:∞) size` denotes the dimensionality of the one-hot vector. This parameter can be omitted. Then the compiler would try to deduce the expected output dimensionality from the subsequent layers.
- **Function:** creates an `N^size` vector with a 1 at the index equal to the scalar input of the layer and zeros otherwise.

Embedding. The layer class Embedding provides a more sophisticated high-dimensional encoding of a vocabulary tackling the drawbacks of one-hot encoding. It aims to model semantic similarity by mapping words, which tend to occur together, to similar vectors, whereas vectors representing unrelated words are farther apart.

- **Parameters:**

  `N(1:∞) input_dim` is the size of the vocabulary to be mapped.

  `N(1:∞) output_dim` is the dimensionality of the feature space. Note that the feature space can have a lower dimensionality than one-hot encoding.

- **Function:** looks up a feature vector as a row in an embedding matrix given a scalar index as input. The embedding matrix is a learnable parameter.

Argmax. While the loss function is usually computed using the output of the Softmax layer of a neural network, the end user is mostly interested in the concrete result rather than the probability distribution. Similarly, in RNNs the feedback used by an RNN cell needs to be the actual result of the last step. This can be accomplished by placing an Argmax layer instance at the output of the network. This layer can only be used as the last layer in a network and is ignored during the computation of the loss function in the training phase.

- **Function:** $\arg\max_i(x_i)$, where x is the input vector and x_i is the i-th element.
- **Parameters:** none.

B.2 CNNTrain Evaluation Metrics

f1. the F1 score is defined as

$$F1 := 2\frac{precision \cdot recall}{precision + recall}, \tag{B.1}$$

where

$$precision := \frac{TP}{TP + FP} \tag{B.2}$$

and

$$recall := \frac{TP}{TP + FN} \tag{B.3}$$

with TP, FP, and FN being the true positives, false positives, and false negatives, respectively. The score is only applicable to a binary classification function. An inter-model context condition ensures that the neural network has a one-dimensional output.

mae. represents the mean absolute error (MAE) defined as

$$MAE := \frac{\sum_{i=1}^{N} \|y_i - \hat{y}_i\|}{N}, \tag{B.4}$$

where N is the number of samples. y_i and $\hat{y}_i$ are the ground truth and the predicted labels of i-th sample, respectively.

mse. represents the MSE defined as

$$MSE := \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{N}, \tag{B.5}$$

where N is the number of samples. y_i and $\hat{y}_i$ are the ground truth and the predicted labels of i-th sample, respectively.

perplexity. an information theoretic evaluation measure indicating how well a probability distribution predicts a sample. It is defined as

$$perplexity := \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log q\left(x_i\right)\right), \tag{B.6}$$

where N is the number of samples and $q(x_i)$ is the probability for the ground truth label of the sample x_i predicted by the model under evaluation.

rmse. represents the root mean squared error (RMSE) defined as

$$RMSE := \sqrt{\frac{\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2}{N}}, \tag{B.7}$$

where N is the number of samples. y_i and $\hat{y}_i$ are the ground truth and the predicted labels of i-th sample, respectively.

top_k_accuracy. similar to accuracy, but prediction is considered as correct as long as the true label is in the top k predictions, with k set using the subparameter `N1 top_k`. If `top_k` = 1, `top_k_accuracy` is equal to `accuracy`.

B.3 CNNTrain for Reinforcement Learning

B.3.1 General Reinforcement Learning Parameters

N1 training_interval=1. The parameter is a positive integer specifying the number of steps between two trainings. The default value is 1 meaning that the agent is trained after each single step.

N start_training_at=0. This integer-valued parameter determines in which episode the training of the network starts.

N1 evaluation_samples. This integer-valued parameter determines the number of sample games for evaluating the network.

nested replay_memory. The parameter determines the operation mode of the replay memory. We can choose between the variants listed in the following.

nested buffer. This is the default replay memory mode introduced in [MKS+13, Lin93]. A circular buffer is used to store the state-action-reward-next-state transitions. For training, a batch is sampled from the buffer. This mode has the following additional configuration parameters:

N1 memory_size. The integer-valued parameter determines the size of the replay buffer. When the number of elements inside the buffer reaches the size limit, the buffer starts to overwrite the first elements as is typical for circular buffers.

N1 sample_size. Size of the batch sampled for each training step. The parameter is typed as a positive integer.

online. With this option no buffer is used. The update step of the neural network is carried out based on the last transition only.

combined. This option is a combination of the two variants `buffer` and `online`. A sample from the buffer as well as the last transition are used together for each training step. The configuration parameters for `combined` are the same as for `buffer`.

nested strategy. This parameter determines the exploration and exploitation strategy of the agent during training. The developer can choose from the following options:

nested epsgreedy. This option represents the ϵ-greedy strategy. Thereby, the highest Q-value is chosen with probability $1-\epsilon$. Otherwise, i.e. with probability ϵ, a discrete uniform distribution is used to sample a random action from the action space. This strategy is only applicable to discrete action spaces. The strategy needs to be configured using the following parameters:

Q(0:1) epsilon. The probability of choosing a random action.

enum epsilon_decay_method. This parameter defines the strategy to decrease epsilon throughout exploration. It can be set to `linear` for a linear decrease or `no` for no decrease.

N1 epsilon_decay_start. This integer-valued parameter determines the episode after which the decrease of epsilon sets in according to `epsilon_decay_method`. Until that episode no decrease takes place.

Q(0:1) epsilon_decay. The epsilon decay parameter is a rational value between 0 and 1 used to control the linear decay function. If the decay method is set to linear, the epsilon value will be reduced by this parameter's value after each episode, but only starting with the episode number `epsilon_decay_start`.

Q(0:1) min_epsilon. The value of epsilon is not reduced below the value of this parameter.

Boolean epsilon_decay_per_step=false. If this parameter is set to true, the decay will be performed after each step. Otherwise, the decay will be executed after each episode only.

nested ornstein_uhlenbeck. With this strategy correlated noise is drawn from an OU process [UO30] and added to the action selected by the current policy network [LHP+15]. This strategy is only applicable to continuous action spaces. The OU process can be controlled by the following parameters:

Q^{n} mu. This parameter represents the mean reversion level μ of the OU process generating the noise to be added to the action output. It has to have the same dimensionality as the action vector output of the actor network.

Q theta. This parameter represents the mean reversion speed of the OU process generating the noise to be added to the action output. It controls how strong the noise is pulled towards the parameter μ.

Q sigma. This parameter determines the weight of randomness in the noise generation. The higher the value of this parameter is, the stronger the random component of the process.

gaussian. This strategy is an alternative to the OU noise for continuous problems. It produces uncorrelated Gaussian noise and adds it to the current policy action. It is refined using the following configuration parameters:

Q^{n} mu=0. This parameter represents the expectation value of the Gaussian random variable added to the action output. It has to have the same dimensionality as the action vector output of the actor network. It is set to a zero vector by default.

Q^{n,n} cov=eye(n). This parameter represents the covariance matrix of the Gaussian random variable added to the action output. It is set to an identity matrix by default.

B.3.2 DQN Exclusive Parameters

In this section we introduce the training parameters available in CNNTrain for DQN, i.e. if `rl_algorithm=dqn-algorithm`.

enum loss. This parameter is used to set the loss function for the training of the Q-network. For more information and the available configuration values, cf. Section 4.9.

Boolean use_fixed_target_network=false. This Boolean parameter determines, whether an additional target network with fixed parameters should be used to estimate the targets for the update step of the Q-function. false is the default value, i.e. no such network is used.

N1 target_network_update_interval. This integer-valued parameter is required if `use_fixed_target_network` is set to true. Every fixed number of steps the weights

of the Q-network are copied to the target weights. The value of `target_network_update_interval` determines the number of steps between two updates of the target network's weights.

Boolean use_double_dqn. If this Boolean parameter is set to true, the Double DQN algorithm [VHGS16] is used to train the Q-function.

nested optimizer. This parameter is used to select the update function for the training of the Q-network weights. The available alternatives include SGD, the Adam optimizer, RMSprop[1], AdaGrad, NAG[2], and AdaDelta represented by the values `sgd`, `adam`, `rmsprop`, `adagrad`, `nag`, and `adadelta`, respectively, cf. Section 4.9 for more information.

B.3.3 TD3 Exclusive Parameters

In the following we introduce the training parameters available in CNNTrain for TD3, i.e. if `rl_algorithm=td3-algorithm`.

Q^{n} policy_noise. This parameter is a rational vector determining the standard deviation of the random variable added to the actions predicted by the target actor network as noise when calculating the targets.

Q^{n} noise_clip. This parameter is a rational vector determining the maximum absolute values for the elements of the policy noise.

N1 policy_delay. This integer-valued parameter determines after how many steps the actor network is updated.

[1] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, accessed February 19, 2020
[2] https://jlmelville.github.io/mize/nesterov.html, accessed on February 19, 2020

B.3.4 Training Results

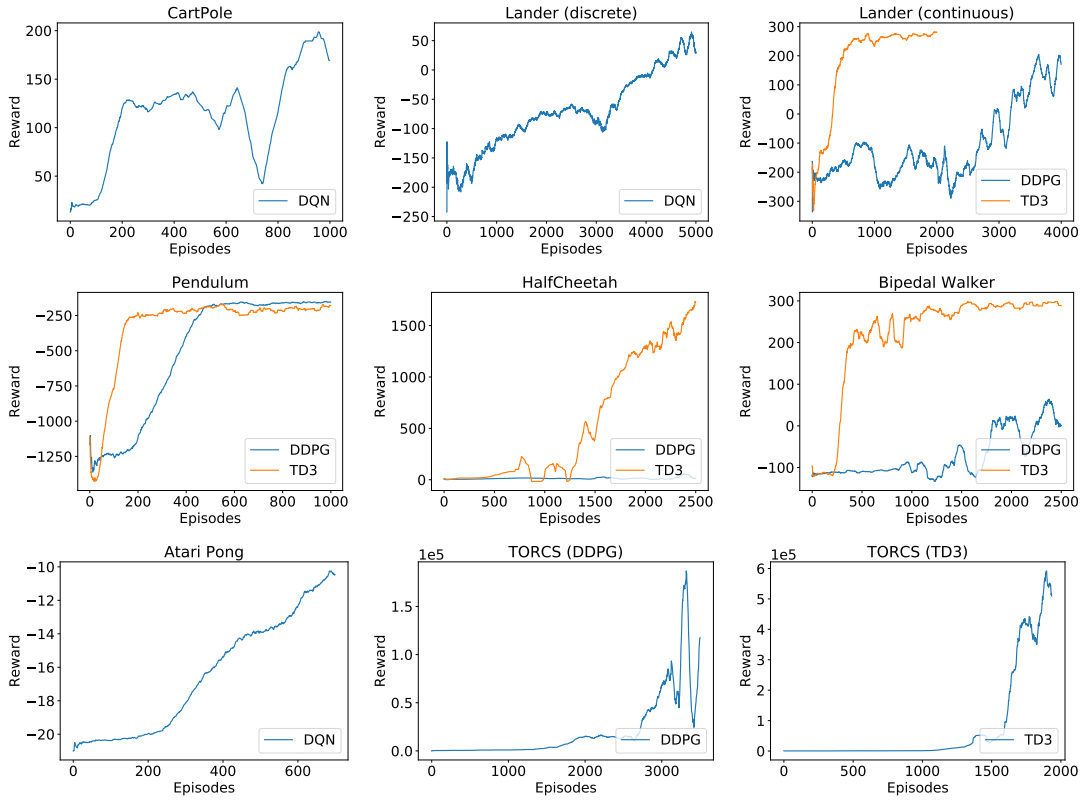


Figure B.1: The average reward over the last 100 episodes during training for OpenAI Gym and TORCS experiments using MontiAnna models conducted by Nicola Gatto in his master's thesis.

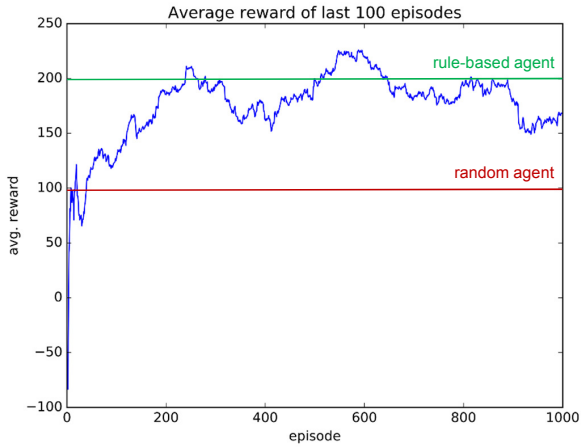


Figure B.2: Training results of the MontiAnna-based forestry 5.0 agent compared to a randomly acting and a rule-based agent. The experiments were conducted by Sascha Dewes in his bachelor's thesis.

B.4 MontiCore 5 Grammars

The following grammars are taken from the internal EmbeddedMontiArc Gitlab repository of the Software Engineering department of the RWTH Aachen University. Some of the following grammars have been co-developed with Michael von Wenckstern. Furthermore, grammars may contain contributions by others, in particular bachelor and master students supervised by the author of this dissertation.

```
1 /* (c) https://github.com/MontiCore/monticore */
2
3 package de.monticore.lang;
4
5 component grammar Matrix extends de.monticore.ExpressionsBasis  {
6    MathMatrixValueExplicitExpression implements Expression<220> =
7       "[" (MathMatrixAccessExpression || ";")* "]";
8     MathMatrixAccessExpression =
9       MathMatrixAccess (","? MathMatrixAccess)*;
10    MathMatrixAccess =
11      doubleDot:":" | Expression;
12    MathVectorExpression  implements Expression<210>   =
13      start:Expression ":" ( steps:Expression ":")?
14    end:Expression ;}
```

Listing B.1: MontiCore 5 grammar for matrix literals (co-developed with Michael von Wenckstern [vW20]).

```
/* (c) https://github.com/MontiCore/monticore */

package de.monticore.lang;

component grammar MatrixExpressions extends de.monticore.
    ExpressionsBasis {
    //Arithmetic expressions for matrices
    MathArithmeticMatrixLeftDivideExpression implements
      Expression<160> = Expression "\\" Expression;

    MathArithmeticMatrixTransposeExpression implements
      Expression<270> = Expression ".\'";

    MathArithmeticMatrixComplexTransposeExpression implements
      Expression<270> = Expression "\'";

    MathArithmeticMatrixEEPowExpression implements
      Expression<290> = Expression ".^" Expression;

    MathArithmeticMatrixEEMultExpression implements
      Expression<280> = Expression "." "*" Expression;

    MathArithmeticMatrixEERightDivideExpression implements
      Expression<250> = Expression "./" Expression;

    MathArithmeticMatrixEELeftDivideExpression implements
      Expression<250> = Expression ".\\" Expression;
}
```

Listing B.2: MontiCore 5 grammar for matrix expressions (co-developed with Michael von Wenckstern [vW20]).

```
/* (c) https://github.com/MontiCore/monticore */


grammar Math extends de.monticore.NumberUnit,
              de.monticore.AssignmentExpressions,
              de.monticore.CommonExpressions,
              de.monticore.lang.Matrix,
              de.monticore.lang.MatrixExpressions,
              de.monticore.lang.monticar.Types2 {

    MathCompilationUnit =
        ("package" package:QualifiedName ";")?
```

```
13          (ImportStatement)*
14          MathScript;
15
16      interface Statement;
17
18      symbol scope MathScript =
19          "script" Name statements:Statement* "end";
20
21      MathDottedNameExpression implements Expression<300> =
22          Name "." Name;
23
24      MathMatrixNameExpression implements Expression<300> =
25          Name "(" MathMatrixAccessExpression ")";
26
27      //Boolean expressions
28      MathTrueExpression implements Expression<240> =
29          "true";
30      MathFalseExpression implements Expression<240> =
31          "false";
32
33      //for loop
34      MathForLoopExpression implements Statement =
35          "for" head:MathForLoopHead body:Statement* "end";
36
37      MathForLoopHead =
38          Name "=" (NameExpression | Expression);
39
40      //if and else conditions
41      MathIfStatement implements Statement =
42          MathIfExpression MathElseIfExpression*
43          MathElseExpression? "end";
44
45      MathIfStatementShort implements Statement =
46          Expression "?" trueCase:Statement* ":"
47                         falseCase:Statement*;
48
49      MathIfExpression =
50          "if" condition:Expression body:Statement* ;
51
52      MathElseIfExpression =
53          "elseif" condition:Expression body:Statement* ;
54
55      MathElseExpression =
56          "else" body:Statement* ;
57
58      MathDeclarationStatement implements Statement =
```

```
59          type:AssignmentType Name ";";
60
61      MathAssignmentDeclarationStatement implements Statement =
62          type:AssignmentType
63          Name MathAssignmentOperator
64          (Expression) ";";
65
66      MathAssignmentStatement implements Statement =
67          (Name | MathMatrixNameExpression |
68          MathDottedNameExpression) MathAssignmentOperator
69          (Expression) ";";
70
71      MathAssignmentOperator =
72          operator:"=" | operator:"+=" | operator:"-=" |
73          operator:"*=" | operator:"/=";
74
75      //Assignments
76      AssignmentType =
77          matrixProperty:Name* ElementType dimension:Dimension?;
78
79      //Expression for all Numbers with units
80      NumberExpression implements Expression<291> =
81          NumberWithUnit;
82
83      NameExpression implements Expression<295> =
84          Name;
85
86      MathArithmeticPowerOfExpression implements Expression <190> =
87          leftExpression:Expression operator:"^"
88          rightExpression:Expression;
89
90      IncSuffixExpression implements Expression <220>, Statement =
91          Expression "++" ";" ;
92
93      DecSuffixExpression implements Expression <220>, Statement =
94          Expression "--" ";" ;
95
96      // remove incompatible expressions
97      BinaryXorExpression implements Expression <110> =
98    "----- will not be used ---- this removes BinaryXorExpression";
99  }
```

Listing B.3: MontiCore 5 grammar of the MontiMath language (co-developed with Michael von Wenckstern [vW20]).

```
/* (c) https://github.com/MontiCore/monticore */

package de.monticore.lang;


grammar MathOpt extends de.monticore.lang.Math
{   MathOptCompilationUnit = MathCompilationUnit;

    enum OptimizationType =
        minimization: "minimize" | maximization: "maximize";

    OptimizationCompareOperator =
        operator:"==" | operator:"<=" | operator:">=";
    OptimizationVariableDeclaration =
        type:AssignmentType?  Name;
    OptimizationSimpleCondition =
        left:Expression operator:OptimizationCompareOperator
        right:Expression ";";
    OptimizationBoundsCondition =
        lower:Expression "<=" expr:Expression "<="
        upper:Expression ";";
    OptimizationCondition =
        (simpleCondition:OptimizationSimpleCondition |
         boundedCondition:OptimizationBoundsCondition |
         forLoopCondition:OptimizationForLoop);
    OptimizationForLoop = "for" head:MathForLoopHead
              body:OptimizationCondition+ "end";

    OptimizationObjectiveValue = type:ElementType Name "=";

    OptimizationStatement implements Statement =
        optimizationType:OptimizationType
        ("<" stepSize:Expression ">")?
            optimizationVariable:OptimizationVariableDeclaration
            ("," optimizationVariable:
             OptimizationVariableDeclaration)* ";"
        "in" objectiveValue:OptimizationObjectiveValue?
             objectiveFunction:Expression ";"
        "subject to" independentDeclaration:
            MathAssignmentDeclarationStatement*
            constraint:OptimizationCondition*
        "end"; }
```

Listing B.4: MontiCore 5 grammar of the MathOpt optimization extention for MontiMath.

```
/* (c) https://github.com/MontiCore/monticore */
package de.monticore.lang.monticar;

/**
 * Grammar for Struct.
 */
grammar Struct extends de.monticore.lang.monticar.Types2 {
  StructCompilationUnit =
    ("package" package:(Name& || ".")+ ";")?
    (ImportStatement)*
    Struct;

  symbol scope Struct implements Type =
    "struct" Name "{" StructFieldDefinition+ "}";

  symbol StructFieldDefinition =
    Type Name ";";
}
```

Listing B.5: MontiCore 5 grammar of the Struct language for EMA.

```
/* (c) https://github.com/MontiCore/monticore */
package de.monticore.lang.monticar;

/**
 * Grammar for Enum.
 */
grammar Enum extends de.monticore.lang.monticar.Types2 {
  EnumLangCompilationUnit =
    ("package" package:(Name& || ".")+ ";")?
    EnumDeclaration;

  symbol scope EnumDeclaration implements Type =
    "enum" Name "{"
      (EnumConstantDeclaration || "|")+ ";"
    "}";

  symbol EnumConstantDeclaration =
    Name;
}
```

Listing B.6: MontiCore 5 grammar of the Enum language for EMA.

```
1 /* (c) https://github.com/MontiCore/monticore */
2
3 package de.monticore.lang.embeddedmontiarc;
4
5 /**
6  * Grammar for EmbeddedMontiArc.
7  *
8  */
9 grammar EmbeddedMontiArc extends de.monticore.lang.monticar.
    Common2 {
10
11  /** ASTEMACompilationUnit represents the complete component
12   * @attribute package          The package declaration of this
       component
13   * @attribute importStatements List of imported elements
14   * @attribute Component the root component of the component
15   */
16  EMACompilationUnit =
17    ("package" package:(Name& || ".")+ ";")?
18    (ImportStatement )*
19    Component;
20
21
22  /* ========================================================*/
23  /* ===== Modified but based on old ARCD Grammar =========*/
24  /* ========================================================*/
25
26  /**
27  * A component may contain arbitrarily many Elements.
28  * This interface may be used as an extension point to
29  * enrich components with further elements.
30  */
31  interface Element;
32
33
34  /**
35  * A component is a unit of computation or a data store.
36  * The size of a component may scale from a single
37  * procedure to a whole application. A component may be
38  * either decomposed to subcomponents or is atomic.
39  *
40  * @attribute name type name of this component
41  * @attribute head is used to set generic types, a
42  *   configuration and a parent component
```

```
43   * @attribute instanceName if this optional name is given,
44   *   a subcomponent is automatically created that
45   *   instantiates this inner component type. This is only
46   *   allowed for inner component definitions.
47   * @attribute body contains the architectural elements
48   *   inherited by this component
49   * A components head is used to define generic type
50   * parameters that may be used as port types in the
51   * component, to define configuration parameters that may
52   * be used to configure the component, and to set the
53   * parent component of this component.
54   *
55   * @attribute genericTypeParameters a list of type
56   *   parameters that may be used as port types in the
57   *   component
58   * @attribute parameters a list of Parameters that
59   *   define a configurable component. If a configurable
60   *   component is referenced, these parameters have to be
61   *   set.
62   * @attribute superComponent the type of the super
63   *   component
64   */
65
66   symbol scope Component implements Element =
67     ComponentModifier* "component" (["interface"])? Name
68   genericTypeParameters:TypeParameters2?
69     (  "("  (Parameter || ",")+  ")" )?
70     ("implements" superComponent:ReferenceType)?
71     body:ComponentBody;
72
73   interface ComponentModifier;
74
75   VirtModifier implements ComponentModifier =
76     VIRTUAL;
77   enum VIRTUAL =
78     "virtual" | VIRTUAL:"virt" | "nonvirtual" |
79     NONVIRTUAL:"non-virt";
80
81   DFModifier implements ComponentModifier =
82     DIRECTFEEDTHROUGH;
83   enum DIRECTFEEDTHROUGH =
84     DF:"direct feedthrough" | "df" |
85     NONDF:"nondirect feedthrough" | NONDF:"non-df";
86
87
88   /**
```

```
89    * The body contains architectural elements of
90    * this component.
91    *
92    * @attribute elements list of architectural elements
93    */
94    ComponentBody =
95      ("{"
96        Element*
97      "}" );
98
99    /**
100   * An Interface defines an interface of a component
101   * containing in- and outgoing ports.
102   * @attribute ports a list of ports that are contained in
103   *   this interface
104   */
105   Interface implements Element =
106      ("port"|"ports")
107     ports:(Port || ",")+ ";" ;
108
109  /**a
110   * An incoming port is used to receive messages, an
111   * outgoing port is used to send messages of a specific
112   * type. Ports can now also be specified as an array.
113   *
114   * @attribute incoming true, if this is an incoming port
115   * @attribute outgoing true, if this is an outgoing port
116   * @attribute type the message type of this port
117   * @attribute name an optional name of this port
118   */
119   /**might support auto type adding based on last previous type
120    * declaration later on
121    */
122    Port =
123           AdaptableKeyword? (incoming:["in"] | outgoing:["out"])
124           Type (Name? | Name ( "[" UnitNumberResolution "]" )?) ;
125
126
127   /**
128     * A subcomponent is used to create one or more instances
129     *  of another component. This way the hierarchical
130     * structure of a component is defined.
131     * @attribute type the type of the instantiated component
132     * @attribute arguments list of configuration parameters
133     *   that are to be set, if the instantiated component is
134     *   configurable.
```

```
135     * @attribute instances list of instances that should be
136     *   created
137     */
138     SubComponent implements Element =
139       "instance"
140       type:ReferenceType
141       ("(" arguments:(Expression || ",")+ ")" )?
142       ("{" (PortInitial || ",")+ "}")?
143       instances:(SubComponentInstance || ",")+ ";" ;
144
145
146   /**
147   * A subcomponent instance binds the name of an instance
148   * with an optional list of simple connectors used to
149   * connect this instance with other subcomponents/ports.
150   * It does also support component arrays.
151   * Simple connectors directly connect outgoing ports of the
152   * corresponding subcomponent declaration with one or
153   * more target ports.
154   *
155   * @attribute name the name of this instance
156   * @attribute connectors list of simple connectors
157   */
158
159   SubComponentInstance =
160     Name
161     ("[" UnitNumberResolution "]")?;
162
163   PortInitial =
164     Name ( "[" UnitNumberResolution "]" )?
165     "(" "t=0" ")"
166     (guess:["~"])? "="
167     Expression;
168
169   /**
170     * port1
171   * port1[2]
172   * port1[:]
173   * sub1.port1
174   * sub1[2].port1[3]
175   * sub1.*
176   * sub1[2].*
177   */
178   QualifiedNameWithArrayAndStar = QualifiedNameWithArray DotStar?;
179   DotStar = "." {noSpace()}? "*";
180   /**
```

```
181   * A connector connects one source port with one or many
182   * target ports.
183   *
184   * @attribute source source port or component instance
185   *   name
186   * @attribute targets a list of target ports or component
187   *   instance names
188   */
189   Connector implements Element=
190     "connect" (source:QualifiedNameWithArrayAndStar |
191                 boolLiteral:BooleanLiteral |
192                 stringLiteral:StringLiteral|
193                 UnitNumberResolution) "->"
194     targets:ConnectorTargets ";" ;
195
196     ConnectorTargets = ["#"] | // route symbol terminates inputs
197                        (QualifiedNameWithArrayAndStar || ","  )+;
198 }
```

Listing B.7: MontiCore 5 grammar of the EmbeddedMontiArc ADL (co-developed with Michael von Wenckstern [vW20]).

```
1 /* (c) https://github.com/MontiCore/monticore */
2
3 package de.monticore.lang.embeddedmontiarc;
4
5
6 component grammar EmbeddedMontiArcBehavior extends
7     de.monticore.lang.embeddedmontiarc.EmbeddedMontiArc{
8
9   /**
10    * External to embed languages that allow implementing
11    * component behavior.
12    */
13   external BehaviorEmbedding;
14   external BehaviorName;
15
16   BehaviorImplementation implements Element =
17     "implementation" BehaviorName
18     "{" behavior:BehaviorEmbedding "}";
19 }
```

Listing B.8: MontiCore 5 grammar of EmbeddedMontiArcBehavior enabling the integration of implementation languages into EMA components (co-developed with Michael von Wenckstern [vW20]).

```
/* (c) https://github.com/MontiCore/monticore */

package de.monticore.lang.embeddedmontiarc;

grammar EmbeddedMontiArcMath extends
  de.monticore.lang.embeddedmontiarcdynamic.
      EmbeddedMontiArcDynamic,
  de.monticore.lang.embeddedmontiarc.EmbeddedMontiArcBehavior,
  de.monticore.lang.MathOpt {
    start EMACompilationUnit;
    BehaviorEmbedding =  Statement+;
    BehaviorName = name:"Math"; }
```

Listing B.9: MontiCore 5 grammar of EmbeddedMontiArcMath (co-developed with Michael von Wenckstern [vW20]).

```
/* (c) https://github.com/MontiCore/monticore */

package de.monticore.lang.embeddedmontiarcdynamic;

grammar EmbeddedMontiArcDynamic extends
  de.monticore.lang.embeddedmontiarcdynamic.Event,
  de.monticore.lang.embeddedmontiarc.EmbeddedMontiArc,
  de.monticore.lang.monticar.Common2{

    start EMACompilationUnit;

    DynamicModifier implements ComponentModifier = "dynamic";


    // Handler for events
    scope EventHandler implements Element =
    (EventReferenceExpression | "@" Expression) "{"
        body:Element*
     "}" ;

    // Dynamic Port(s) [isDynamic is new]
    //
    Port =
        (dynamic:["dynamic"])?
        AdaptableKeyword? (incoming:["in"] | outgoing:["out"])
        Type (Name? | Name ( "[" UnitNumberResolution
```

```
27          (":" dynamicNumberOfPorts:UnitNumberResolution)? "]" )?);
28
29
30      // Dynamic SubComponent(s)
31
32      // SubComponentInstance extension
33      // Dynamic number
34      SubComponentInstance =
35          Name
36          ("[" UnitNumberResolution (":"
37          dynamicNumberOfInstances:UnitNumberResolution)? "]")?;
38
39      ArrayAccess = ("[" ( intLiteral:UnitNumberResolution |
40          [":"] |
41          lowerbound:UnitNumberResolution ":"
42          upperbound:UnitNumberResolution |
43          dynamicNewPort:["?"]
44          dynamicElementIndex:NumericLiteral? ) "]" );
45  }
```

Listing B.10: MontiCore 5 grammar of EmbeddedMontiArc Dynamics.

```
1  /* (c) https://github.com/MontiCore/monticore */
2
3  package de.monticore.lang.embeddedmontiarcdynamic;
4
5  grammar Event extends de.monticore.CommonExpressions,
6                        de.monticore.lang.monticar.Types2,
7                        de.monticore.lang.monticar.Common2,
8                        de.monticore.NumberUnit {
9
10     EventCompilationUnit =
11         ("package" package:(Name& || ".")+ ";")?
12         ComponentEvent;
13
14     symbol scope ComponentEvent =
15         "event" Name
16         genericTypeParameters:TypeParameters2?
17         (  "("  (Parameter || ",")+  ")" )?
18         ("for" forComponent:ReferenceType)?
19         "{"
20             condition:Expression
21         "}";
22
23     PortResolutionDeclaration implements
```

```
24        ResolutionDeclaration = "Port" "<" type:Type ">" Name;
25
26      EventReferenceExpression implements Expression <200> =
27          "@" type:ReferenceType?
28              "(" arguments:(PortValue || ",")*  ")";
29
30      //Boolean expressions
31      TrueExpression =
32          "true";
33      FalseExpression =
34          "false";
35
36      // Port Expressions
37      PortExpression implements Expression <200>  =
38              portName:QualifiedNameWithArray
39              "::" (PortExpressionContent);
40
41      interface PortExpressionContent;
42      interface Operator;
43
44      LTOp implements Operator = "<";
45      GTOp implements Operator = ">";
46      LTEOp implements Operator = "<=";
47      GTEOp implements Operator = ">=";
48      EOp implements Operator = "==";
49      UEOp implements Operator = "!=";
50
51      PortExpressionValue implements PortExpressionContent =
52        "value" "(" Expression? ")" operator:Operator
53                    (PortValue | PortValueList);
54      PortExpressionConnect implements PortExpressionContent =
55          "connect";
56      PortExpressionFree implements PortExpressionContent = "free";
57
58      // Definitions for Port Expressions
59      PortValue = PortSingleValue | PortArrayValue;
60      PortValueList = "[" ( PortValue || "tick")+ "]";
61
62      PortSingleValue = NumberWithPrecision | NumberRange | CTV;
63      PortArrayValue = "[" (PortArrayValueContent |
64                            PortArrayValueMatrixContent) "]" ;
65
66      PortArrayValueContent = ( PortSingleValue || ",")+ ;
67      PortArrayValueMatrixContent = PortArrayValueContent
68          ( ";" | PortArrayValueContent )+ ;
69
```

```
70      ValueInput = NumberWithUnit | NameWithArray |
71                   TrueExpression | FalseExpression;
72
73      NumberWithPrecision = ValueInput
74                            ("+" "/" "-" precision:ValueInput)? ;
75      NumberRange = "(" lowerBound:ValueInput ":"
76                        upperBound:ValueInput ")";
77
78      ast CompareToValue =
79          method public ASTValueInput getCompare(){}
80          method public String getOperator(){};
81
82      interface CompareToValue;
83
84      CompareToValueGreater implements CompareToValue =
85          operator:">" compare:ValueInput ;
86      CompareToValueGreaterEquals implements CompareToValue =
87          operator:">=" compare:ValueInput ;
88      CompareToValueLower implements CompareToValue =
89          operator:"<" compare:ValueInput ;
90      CompareToValueLowerEquals implements CompareToValue =
91          operator:"<=" compare:ValueInput ;
92      CompareToValueNotEquals implements CompareToValue =
93          operator:"!=" compare:ValueInput ;
94
95      CTV = (CompareToValueGreater | CompareToValueGreaterEquals |
96                  CompareToValueLower | CompareToValueLowerEquals |
97                  CompareToValueNotEquals);
98  }
```

Listing B.11: MontiCore 5 grammar of the event definition language (reconfiguration conditions) for EmbeddedMontiArc Dynamics.

```
1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore.lang.monticar;
3
4  grammar CNNArch extends de.monticore.CommonExpressions,
5                          de.monticore.lang.Math,
6                          de.monticore.lang.monticar.Common2 {
7
8
9      /* ================================*/
10     /* ========== PRODUCTIONS ==========*/
11     /* ================================*/
12
```

```
13      /* ========== Declarations =========*/
14
15      /**
16          The complete file.
17          Use nonterminal Architecture for embedding in another
18          language (e.g. EmbeddedMontiArc)
19      */
20      symbol scope CNNArchCompilationUnit =
21        ("package" package:(Name& || ".")+ ";")?
22          "architecture"
23          name:Name
24          ( "(" (ArchitectureParameter || ",")* ")" )? "{"
25          ioDeclarations:IODeclaration*
26          Architecture
27          "}";
28
29      LayerDeclaration =  "def"
30                          Name "("
31                          parameters:(LayerParameter || ",")* ")" "{"
32                          body:Stream  "}";
33
34      IODeclaration =  "def"
35                      (in:"input" | out:"output")
36                      type:ArchType
37                      Name
38                      (ArrayDeclaration)?;
39
40
41      /* ============== Type =============*/
42
43      /**
44           Similar to EmbeddedMontiArc port types.
45           ArchType and Shape are not used if the Architecture is
46           integrated into EmbeddedMontiArc
47      */
48      ArchType = ElementType "^" Shape;
49
50      Shape = "{" dimensions:(ArchSimpleExpression || ",")* "}";
51
52
53      /* ========= Architecture  =========*/
54
55      /**
56          Defines the architecture of the neural network.
57          This NT is used for integration in EmbeddedMontiArc.
58          @attribute methodDeclaration*
```

```
59          A list of new layers which can be used
60          in the architecture.
61          @attribute body
62          The architecture of the neural network.
63      */
64      Architecture = methodDeclaration:LayerDeclaration*
65                     instructions:(Instruction || ";")+ ";";
66
67      Instruction = (LayerVariableDeclaration | NetworkInstruction);
68
69      LayerVariableDeclaration = "layer" Layer Name;
70
71      interface NetworkInstruction;
72
73      StreamInstruction implements NetworkInstruction = body:Stream;
74
75      UnrollInstruction implements NetworkInstruction =
76        "timed" "<" timeParameter:TimeParameter ">"
77          Name "(" arguments:(ArchArgument || ",")* ")"
78          "{" body:Stream "}";
79
80      Stream = elements:(ArchitectureElement || "->")+;
81
82      interface ArchitectureElement;
83
84      Variable implements ArchitectureElement =
85          Name ("." (member:"output" | member:Name))?
86               ("[" index:ArchSimpleExpression "]")?;
87
88      Constant implements ArchitectureElement =
89                               ArchSimpleExpression;
90
91      Layer implements ArchitectureElement =
92                  Name "(" arguments:(ArchArgument || ",")* ")";
93
94
95      ParallelBlock implements ArchitectureElement =
96        "("
97          groups:Stream  "|"
98          groups:(Stream || "|")+ ")";
99
100     ArrayAccessLayer implements ArchitectureElement =
101                              "[" index:ArchSimpleExpression "]";
102
103     /* ====== Variables/Arguments ======*/
104
```

```
105     interface ArchParameter;
106
107     ArchitectureParameter implements ArchParameter =
108                     Name ("=" default:ArchSimpleExpression)? ;
109
110     LayerParameter implements ArchParameter =
111                       Name ("=" default:ArchSimpleExpression)? ;
112
113     TimeParameter implements ArchParameter =
114         Name ("=" default:ArchSimpleExpression)? ;
115
116     interface ArchArgument;
117
118     ArchParameterArgument implements ArchArgument =
119                               Name "=" rhs:ArchExpression ;
120
121     ArchSpecialArgument implements ArchArgument =
122       (serial:"->" | parallel:"|" | conditional:"?") "="
123         rhs:ArchExpression ;
124
125
126     /* ======= Value Expressions =======*/
127
128     /**
129         Expression used for method arguments.
130     */
131     ArchExpression = (expression:ArchSimpleExpression |
132                       sequence:ArchValueSequence);
133
134     interface ArchValueSequence;
135
136     ArchParallelSequence implements ArchValueSequence =
137         "[" parallelValues:(ArchSerialSequence || "|")+ "]";
138
139     ArchSerialSequence = serialValues:(ArchSimpleExpression ||
140       "->")*;
141
142     ArchValueRange implements ArchValueSequence =
143         "[" start:ArchSimpleExpression
144         (serial:"->" | parallel:"|")
145         ".."
146         (serial2:"->" | parallel2:"|")
147         end:ArchSimpleExpression "]";
148     /**
149         Expressions for parameter and variable values.
150     */
```

```
151     ArchSimpleExpression =
152         (arithmeticExpression:ArchArithmeticExpression
153         | booleanExpression:ArchBooleanExpression
154         | tupleExpression:TupleExpression
155         | string:StringLiteral);
156
157     interface ArchMathExpression extends Expression;
158     interface ArchArithmeticExpression extends ArchMathExpression;
159     interface ArchBooleanExpression extends ArchMathExpression;
160
161     ArchSimpleArithmeticExpression implements
162         ArchArithmeticExpression =
163             (NumberExpression
164             | NameExpression
165             | MathDottedNameExpression
166             | MathAssignmentDeclarationStatement
167             | MathAssignmentStatement);
168
169     ArchComplexArithmeticExpression implements
170         ArchArithmeticExpression =
171             leftExpression:ArchMathExpression
172             (   operator:"*"
173             |   operator:"/"
174             |   operator:"%"
175             |   operator:"^"
176             |   operator:"+"
177             |   operator:"-"
178             )
179             rightExpression:ArchMathExpression;
180
181     TupleExpression = "(" expressions:ArchArithmeticExpression ","
182         expressions:(ArchArithmeticExpression || ",")* ")";
183
184     ArchSimpleBooleanExpression implements
185         ArchBooleanExpression = (BooleanExpression
186                         | BooleanNotExpression
187                         | LogicalNotExpression);
188
189     ArchComplexBooleanExpression implements
190         ArchBooleanExpression = leftExpression:ArchMathExpression
191         (   operator:"=="
192             operator:"!="
193             operator:"&&"
194         |   operator:"||"
195         )
196         rightExpression:ArchMathExpression;
```

```
197
198     ArchBracketExpression implements ArchMathExpression,
199         ArchBooleanExpression,
200         ArchArithmeticExpression = "(" ArchMathExpression ")";
201
202     ArchPreMinusExpression implements ArchMathExpression,
203         ArchBooleanExpression,
204         ArchArithmeticExpression = "-" ArchMathExpression ;
205
206
207     /* ================================*/
208     /* ============ ASTRULES ===========*/
209     /* ================================*/
210
211     ast ArchParameter = method String getName(){};
212
213     ast ArchSpecialArgument =
214         method public String getName(){return "";};
215     //Override is necessary
216
217     ast ArchArgument = method String getName(){}
218                        method ASTArchExpression getRhs(){};
219
220 }
```

Listing B.12: MontiCore 5 grammar of the CNNArc language.

```
1 /* (c) https://github.com/MontiCore/monticore */
2
3 package de.monticore.lang.monticar;
4
5 grammar EMADL extends
6     de.monticore.lang.embeddedmontiarc.EmbeddedMontiArcMath,
7     de.monticore.lang.monticar.CNNArch {
8
9     start EMACompilationUnit;
10
11    BehaviorEmbedding = Architecture | Statement+;
12
13    BehaviorName = name:"CNN" | name:"Math";
14 }
```

Listing B.13: MontiCore 5 grammar of EmbeddedMontiArcDL.

List of Figures

1.1 An overview of the SMArDT process [HKK+18]. 18

2.1 A simplified taxonomy of matrices containing the matrix properties relevant for the EMA type system based on [HJ90]. 27
2.2 A struct example encapsulating the primitive variables longitude and latitude into a type named Location. 29
2.3 An enum example representing different weather conditions. 30
2.4 A basic example of an EMA architecture. The component `Main` contains two subcomponents `Add adder` and `Mult multiplier`. 31
2.5 An EMA architecture example featuring port and component arrays. The component `Main` contains `n Add2` components, each operating on one of `n` operand pairs coming from the port arrays `A` and `B`. The `Mult2n` component computes the product of `2n` operands passed through the port arrays `A` and `B` of the `Main` component to the port array `factors` of `Mult2n`. 33
2.6 Graphical views of the component defined in Figure 2.5. On the lhs, the elements of two port arrays are connected to target ports of a component array. On the rhs, a port array is connected to another port array. 34
2.7 This example shows two C&C architectures `Main1` and `Main2`, which are semantically equivalent in EMA due to its synchronized and weakly causal execution model, but which might have different interpretations in a language with strongly causal semantics. 37
2.8 This example shows an architecture featuring an algebraic loop (`Main1`) and its loop-free equivalent synthesized automatically at compile-time (`Main2`). 37
2.9 The component on the left contains a loop consisting of two non-flattened components. The model in the middle displays the contents of the `Processing` subcomponent making obvious that there are no circular dependencies on port level. The figure on the right shows how the component is seen by the compiler after the flattening step. 39
2.10 This listing shows a simple MontiMath example exhibiting the main language constructs including variable declarations, matrix literal definitions, loops and if statements. 40
2.11 An excerpt of the Prolog rule set for the derivation of matrix properties. . 43

2.12 An example of matrix property derivation for operations. 44
2.13 An EMAM model embeds a MontiMath script into an EMA component, thereby leveraging the latter from SMArDT level 2 to SMArDT level 3. . 47
2.14 A MontiMath optimization statement representing a quadratic problem. . 48
2.15 A statically parameterized and an extended, runtime-adaptive PID component interface. 49
2.16 The PID component with parameters adaptable through dedicated ports. 50
2.17 The PID component interface with adaptable parameters defined in EmbeddedMontiArc. 51
2.18 The components LegacyController and TunableController depict the static and the adaptable usage of the same PID component. 52
2.19 The main controller has an adaptable parameter param1. This parameter is used to initialize the (also adaptable) parameter of the PID subcomponent. Whenever param1 of the parent component gets updated at runtime, the update is propagated immediately to the PID subcomponent. 53
2.20 The Abs component under test. 56
2.21 A stream model testing the Abs component. 57
2.22 Usage of the streamtest plugin in a Maven-based EmbeddedMontiArc project. 58

3.1 A multiplexer component choosing two of its inputs to be passed to the inner multiplexer dependent on a control signal. 75
3.2 The two architectural states of the BMux4 component. 75
3.3 Collision system of an autopilot calculating potential collisions with up to 32 other vehicles. 80
3.4 The listing shows a valid usage of the reconfiguration service interface of the CollisionSystem component of Figure 3.3 by a parent component. . . 80
3.5 The listing leads to a compile-time error since CollisionSystem does not have a reconfiguration triggered by requesting only the otherStatus port. . 80
3.6 Adder with 0 to 32 inputs. 83
3.7 The pipe system component on top shows the structure of an arbitrarily long pipeline as defined in Figure 3.8. The last (or deepest) link component contains only a processing but no more linking subcomponents. Since EMA flattens hierarchical components, the effective architecture is a simple pipeline as shown in the bottom model. 86
3.8 The pipe system component uses the link pattern to instantiate an arbitrarily long (but finite) chain of processing components. 87
3.9 The link component consists of a processing subcomponent and, optionally, a further link component, which in turn contains a further processing component and, optionally, a link component, allowing us to create arbitrarily long processing chains. 87

3.10 An example processing component to be used in the pipe system, where `f` denotes any valid MontiMath program using the data from the input port and writing a result to the output port. 88
3.11 A reconfiguration chain involving input and output ports of the `PlatoonManager` component. An arriving platoon message causes the creation of new input ports in the diagram on the left. Follow-up reconfigurations inside the `PlatoonManager` result in a new output port and a new outgoing connector as depicted in the diagram on the right. 90
3.12 An example of generated Z3 code proving condition satisfiability. 91

4.1 The basic neuron model. 97
4.2 The abstraction of layers in artificial neural networks. 98
4.3 The vanilla RNN cell model. 104
4.4 An RNN cell unfolded for three timesteps. 105
4.5 Neural network architecture defined in Net# [KPRS19]. 115
4.6 Overview of the MontiAnna framework layers [KNP+19]. 118
4.7 Definition of a stand-alone network of type Alexnet (based on the influential Alexnet CNN [KSH12]) and its interface. 121
4.8 Named layer instantiation example. 125
4.9 Graph expression example with layer instances chained using the sequential dataflow operator. 125
4.10 A neural network represented as two graph expressions. 126
4.11 Anonymous layer instantiation example. 126
4.12 A split-parallelization-get pattern example. 128
4.13 Two alternative definitions of the same layer stack consisting of five successive fully connected layers having 10 neurons each. The first definition makes use of a structural parameter, while the second one does not. . . . 129
4.14 The MLP network modeled in this listing can have either one or two hidden layers depending on the value of `two_layers`. 129
4.15 The definition of five fully connected layers with different numbers of units with and without an argument sequence. 130
4.16 The ResNet152 modeled in CNNArc using structural parameters, argument sequences, and custom layer definitions. 131
4.17 ResNet152 model equivalent to the one in Figure 4.16 but defined without argument sequences and therefore a better readability. 132
4.18 An example showing how MontiMath could be used to implement custom layers for MontiAnna networks. 133
4.19 RNN encoder-decoder model example [KNP+19]. 136
4.20 MontiAnna model of the RNN encoder-decoder network [KNP+19]. 136
4.21 A simple training model example written in CNNTrain. 141

4.22 An overview of the EmbeddedMontiArcDL modeling language family: languages and code generators are depicted as purple and pink boxes, respectively [KPRS19]. . . . 145
4.23 Definition of an EMADL component with a MontiAnna neural network implementation. . . . 146
4.24 Tag schema defining the training data and optionally a CNNTrain model reference for a deep learning component. . . . 147
4.25 Component-and-connector architecture of an extended MNISTCalculator [KNP+19]. . . . 148
4.26 Textual EMADL representation of the MNISTCalculator architecture depicted in Figure 4.25. . . . 148
4.27 Detector component for the MNISTCalculator. . . . 149
4.28 Tag model providing information regarding the data and the training configuration to be used for the training of the detector components. . . . 150
4.29 The deep direct perception-based architecture of a racing vehicle controller for TORCS [KPRS19]. . . . 151
4.30 The direct perception CNN for the extraction of affordance indicators in TORCS [KPRS19]. . . . 152
4.31 The training model of the DPNet component [KPRS19]. . . . 153

5.1 Setting up a training in OpenAI Baselines using its CLI. . . . 161
5.2 Setting up and training a DQN agent using Tensorforce. . . . 162
5.3 Excerpt of a Google Dopamine gin configuration to train a DQN agent in the Atari Pong environment. . . . 163
5.4 EMADL component of the TORCS actor based on [GKR19]. The implementation is described using the MontiAnna architecture language CNNArc. 165
5.5 EMADL component of the TORCS critic based on [GKR19]. The implementation is described using the MontiAnna architecture language CNNArc. 165
5.6 The upper part of the figure depicts the top level runtime C&C model of the DDPG-based TORCS controller. The filled ports highlight the reinforcement learning interface. The connections between them are generated based on the middleware tagging approach, which will be presented in Chapter 6. The training reference architecture of the DDPG algorithm is depicted in the bottom part and contains a critic, a reward, an environment, and a trainer component, which are thrown away at runtime. . . 167
5.7 Configuration of the TORCS actor training modeled using the training language of MontiAnna based on [GKR19]. . . . 172
5.8 EMADL component with a MontiMath implementation encapsulating an exemplary reward function for the TORCS agent training based on [GKR19]. 174

5.9 Generated artifacts: the left side shows the artifact architecture during the training. The right side is the artifact architecture at the execution time of the model [GKR19]. . . . 177
5.10 EMADL component of the TD3 critic for the TORCS reinforcement learning model. . . . 181
5.11 The racing tracks used for training (top) and evaluation (bottom) of the TORCS agent. Copied from the TORCS racing board website `http://www.berniw.org/trb/tracks/tracklist.php`, accessed April 26, 2021. . . . 182
5.12 An example map of the CTL forestry simulator by WZL. . . . 183
5.13 Actor and critic networks of the forestry 5.0 agent. . . . 185
5.14 Training configuration of the forestry 5.0 agent. . . . 186

6.1 EMA definition of an intersection controller component receiving trajectories from nearby vehicles, checking for potential collisions and sending stop signals to endangered vehicles [HKKR19]. . . . 194
6.2 Intersection controller model with middleware tags attached to its ports [HKKR19]. . . . 195
6.3 Two vehicles approaching an intersection. The planned trajectories are depicted as circles. . . . 195
6.4 Tag schema defining the middleware tag structure for EMA component instances and ports. . . . 199
6.5 Tag model for the running example [HKKR19]. . . . 200
6.6 True deployment scheme of the intersection controller with two separately deployable components. . . . 201
6.7 Overview of the generator coupling architecture [HKKR19]. . . . 203
6.8 Overview of the generated C++ code [HKKR19]. . . . 204
6.9 Setting in which a fast sender serves a slow receiver with and without buffering. . . . 205
6.10 The four plots show an exemplary propagation of a signal through a middleware-based EMA component network. . . . 207
6.11 Generated ROS adapter for the IntersectionController component. . . . 209
6.12 Overview of the generated project structure and artifacts. . . . 209
6.13 Crash rate over network delay and drop rate [HKKR19]. . . . 210
6.14 Clustering software components for deployment. . . . 214

7.1 EMA in the SMArDT process. . . . 220

A.1 Prolog rules for the derivation of matrix properties (PSD, PD, NSD, invertible). . . . 256

A.2 Prolog rules for the derivation of matrix properties (ND, normal, skew Hermitian). 257
A.3 FlowNet [DFI+15] modeled in EMADL by Julian Steinsberger-Dührßen during his lab work. 258
A.4 The Show, Attend and Tell network architecture [XBK+15] modeled in CNNArc by Christian Fuß in his master thesis. The aim of the network is to generate textual descriptions for given images. To do so the architecture combines convolutional layers for image processing, attention, and RNNs. 259

B.1 The average reward over the last 100 episodes during training for OpenAI Gym and TORCS experiments using MontiAnna models conducted by Nicola Gatto in his master's thesis. 270
B.2 Training results of the MontiAnna-based forestry 5.0 agent compared to a randomly acting and a rule-based agent. The experiments were conducted by Sascha Dewes in his bachelor's thesis. 271

List of Tables

3.1 Comparison of C&C modeling languages supporting runtime dynamics, √: yes, p: partially, −: no, ?: unknown 69

4.1 Comparison of established deep learning frameworks and languages, √: yes, p: partially, −: no, *: depending on chosen backend, **: provided by extensions. 109

A.1 Explanation of the used flags in listings and figures. 252
A.2 Accepted non-SI units [dPeM19]. 253
A.3 Base and derived units of the SI system [dPeM19]. 254
A.4 SI unit prefixes [dPeM19]. 255

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview on related work done at the SE Group, RWTH Aachen. More details can be found on the website `https://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 [Rum04]: "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 how digital and organizational techniques help to cope with 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 [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08, HR17] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, size, and number of the 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] was introduced. [BGRW18] explains its applicability in systems engineering based on MDSE projects.

An artifact model basically 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 [KRRvW17] 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 [AKKR21]. 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. Smart semantic differencing operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], 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 [KR18].

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. And we have applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, HRR12] and based it on the core ideas of Broy's Focus theory [RR11, BR07]. 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 and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11a, KLPR12] and city quarters [GLPR15] to optimize the operation efficiency and prevent unneeded CO2 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 based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HR17]. In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKRS06, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12], we demonstrate how to systematically derive a transformation language in concrete syntax. [HMSNRW16] presents how to generate extensible and statically typesafe visitors. In [MSNRR16], 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 [HLMSN+15b, HLMSN+15a]. To understand the implications of executability for UML, we discuss needs and advantages of

executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] respectively [Rum12, Rum13] 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" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98], and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HR17] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HR17] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11, HLMSN+15b, HLMSN+15a, HRW18, BEK+18a, BEK+18b, BEK+19] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages in [KRV14]. [Wei12] presents a tool that allows to create 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]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to

engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15] and the concern-oriented language development approach [CKM+18]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HRW18, BEK+19]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08, HLMSN+15b, HLMSN+15a, HMSNRW16, HR17, BEK+18a, BEK+18b, BEK+19] and the backend [RRRW15, MSNRR16, GMR+16, HR17, BEK+18b]. In [GHK+15b, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV14] describes the roles in software development using domain specific languages. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages [HRW18]. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets. The derivation of internal DSLs from grammars is discussd 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 and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. In [RRW13a], we introduce a code generation framework for MontiArc. 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 [NPR13] and with the robotics domain [AHRW17a, AHRW17b]. [GHK+07] and [GHK+08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14b] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. Co-evolution of architecture is discussed in [MMR10] and modeling techniques to describe dynamic architectures are shown in [HRR98, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HR17] that can even be used to develop modeling tools in a compositional form [HR17, HLMSN+15b, HLMSN+15a, HMSNRW16, MSNRR16, HRW18, BEK+18a, BEK+18b, BEK+19]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respec-

tively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Metamodeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [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 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 for the development of CPS.

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. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability 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+08a] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability modeling languages in order 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 compositional reuse [BEK+18b], and applied it as a semantic language refinement on Statecharts in [GR11].

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied for 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], autonomous driving [BR12a, 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 for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. In [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition on contributing to systems engineering in automotive [GHK+08b], 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 modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for machining based on these concepts [BKL+18]. 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, BCR07b, BCGR09b, BCGR09a], (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 [RRW13a, RRW14, RRW13a, RW18] as well as 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 behaviour (2) based on information from previously stored and real-time monitored structural context and behaviour 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] or UML/P based languages [MNRV19]. [MM15] describes a process how languages for assistive systems can be created.

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 [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view on 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.

Modelling 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 composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15, HR17] that perfectly fit robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we

define a modular architecture modeling method for translating architecture models into modules compatible to different robotics middleware platforms.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08a]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we introduce a framework for modeling the dynamic reconfiguration of component and connector architectures and apply it to the domain of cooperating vehicles. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big

Data, App, and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Model-Driven Engineering of Information Systems

Information Systems provide information to different user groups as main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17], we developed several generators for such data-centric information systems. *MontiGem* [AMN+20] 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], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017. B.4, B.4

[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. B.4, B.4

[AKKR21] Abdallah Atouani, Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. *20th International Conference on Generative Programming: Concepts & Experiences (GPCE'21)*, 2021. B.4

[AMN+20] 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. B.4

[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. B.4, B.4

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009. B.4, B.4, B.4

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009. B.4, B.4, B.4

[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. B.4, B.4

[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. B.4, B.4, B.4

[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. B.4, B.4, B.4

[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. B.4, B.4

[BEK+18a] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018. B.4, B.4, B.4

[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. B.4, B.4, B.4, B.4

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019. B.4, B.4, B.4

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich. B.4

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998. B.4, B.4

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017. B.4

[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. B.4

[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. B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4

[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. B.4

[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*, 149:437–461, March 2019. B.4

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007. B.4, B.4, B.4, B.4, B.4

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012. B.4, B.4

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012. B.4

[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. B.4, B.4

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015. B.4, B.4

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014. B.4

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008. B.4, B.4

[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. B.4, B.4

[CKM+18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018. B.4

[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. B.4

[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. B.4, B.4

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019. B.4, B.4

[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. B.4

[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. B.4

[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. B.4

[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. B.4, B.4

[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. B.4

[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. B.4

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998. B.4

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008. B.4, B.4, B.4

[FLP+11a] 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. B.4

[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. B.4, B.4

[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. B.4, B.4, B.4

[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. B.4, B.4, B.4

[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. B.4, B.4, B.4

[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. B.4

[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 Computer and Information Science 580, pages 112–132. Springer, 2015. B.4, B.4, B.4

[GHK+15b] 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. B.4

[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. B.4

[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. B.4

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996. B.4, B.4

[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. B.4, B.4, B.4

[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. B.4

[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. B.4, B.4, B.4

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67-81, 2006. B.4, B.4

[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 Heidelberg, April 2015. B.4

[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. B.4, B.4, B.4

[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. B.4

[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. B.4, B.4, B.4, B.4

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995. B.4

[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. B.4, B.4, B.4, B.4

[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. B.4, B.4, B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015. B.4

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015. B.4

[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. B.4, B.4

[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. B.4

[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. B.4, B.4

[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. B.4

[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. B.4, B.4

[HLMSN+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. B.4, B.4, B.4, B.4

[HLMSN+15b] 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. B.4, B.4, B.4, B.4

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019. B.4, B.4

[HMSNRW16] 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. B.4, B.4, B.4

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004. B.4

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017. B.4, B.4, B.4, B.4, B.4, B.4, B.4

[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. B.4

[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. B.4, B.4

[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. B.4, B.4, B.4

[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. B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4, B.4

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018. B.4, B.4, B.4

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018. B.4

[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. B.4

[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. B.4, B.4

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications. B.4, B.4, B.4

[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. B.4, B.4, B.4

[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. B.4

[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. B.4, B.4

[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. B.4

[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. B.4, B.4

[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. B.4, B.4, B.4

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE'19. Software Engineering Intelligence Workshop (SEI'19)*, pages 126–133. IEEE, November 2019. B.4

[KR18] 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. B.4

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010. B.4, B.4

[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. B.4

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing.* Springer, Schweiz, December 2014. B.4

[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. B.4

[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. B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4, B.4, B.4

[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. B.4, B.4, B.4, B.4, B.4

[KRV14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modeling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*. CoRR arXiv, 2014. B.4, B.4

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020. B.4

[LMK+11] Philipp Leusmann, Christian MÃ¶llering, 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. B.4

[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. B.4, B.4, B.4, B.4

[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. B.4

[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. B.4

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013. B.4

[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. B.4

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010. B.4, B.4, B.4

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017. B.4

[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. B.4

[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. B.4

[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. B.4, B.4, B.4

[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. B.4, B.4

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011. B.4, B.4

[MRR11d] 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. B.4

[MRR11e] 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. B.4

[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. B.4, B.4

[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. B.4

[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. B.4

[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. B.4, B.4

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014. B.4

[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. B.4

[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. B.4

[MSNRR16] 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. B.4, B.4, B.4

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013. B.4, B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4, B.4

[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. B.4

[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. B.4, B.4

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014. B.4

[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. B.4

[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. B.4

[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. B.4

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015. B.4, B.4, B.4

[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. B.4

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013. B.4, B.4, B.4, B.4

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013. B.4, B.4, B.4

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton.* Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014. B.4, B.4, B.4

[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. B.4

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996. B.4

[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. B.4, B.4, B.4

[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. B.4, B.4

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004. B.4, B.4, B.4, B.4

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage.* Springer Berlin, September 2011. B.4

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage.* Springer Berlin, Juni 2012. B.4, B.4, B.4, B.4, B.4

[Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013. B.4

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods.* Springer International, July 2016. B.4, B.4, B.4

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring.* Springer International, May 2017. B.4

[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. B.4, B.4, B.4, B.4, B.4, B.4

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. B.4, B.4, B.4

[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. B.4

[SM18] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0 (Extended Abstract): Integration of non-smart resources into cognitive assistance systems. *EMISA Forum*, 38(1):35–36, Nov 2018. B.4

[SM20] Claudia Steinberger and Judith Michael. *Using Semantic Markup to Boost Context Awareness for Assistive Systems*, pages 227–246. Computer Communications and Networks. Springer International Publishing, 2020. B.4

[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. B.4, B.4, B.4

[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. B.4

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011. B.4, B.4, B.4

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012. B.4, B.4, B.4, B.4

[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. B.4, B.4