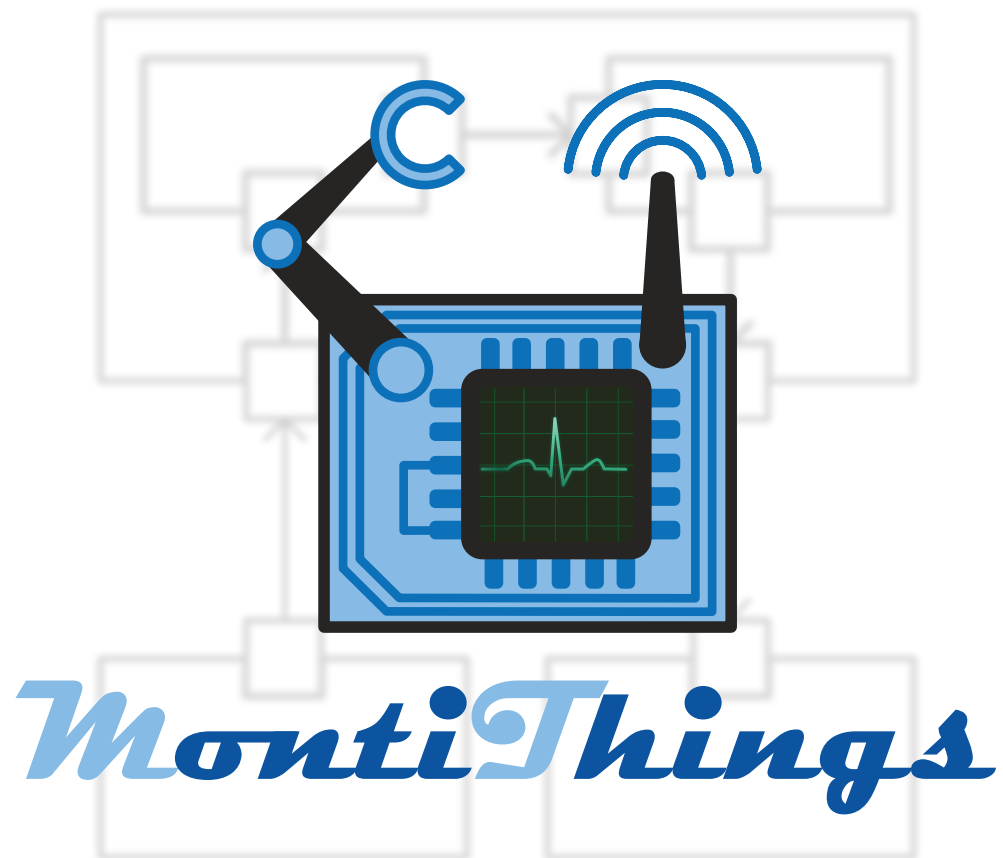


Jörg Christian Kirchhof

Model-Driven Development, Deployment, and Analysis of Internet of Things Applications



Aachener Informatik-Berichte,
Software Engineering

Band 54

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

Model-Driven Development, Deployment, and Analysis of Internet of Things Applications

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

M.Sc. RWTH
Jörg Christian Kirchhof
aus Hilden, Deutschland

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Mag. Dr. Manuel Wimmer

Tag der mündlichen Prüfung: 11. November 2022

D 82 (Diss. RWTH Aachen University, 2022)



[Kir23] J. C. Kirchhof:

Model-Driven Development, Deployment, and Analysis of Internet of Things Applications.

Aachener Informatik-Berichte, Software Engineering, Band 54, ISBN 978-3-8440-8960-8, Shaker Verlag, Februar 2023.

www.se-rwth.de/publications/

Eidesstattliche Erklärung

I, Jörg Christian Kirchhof erklärt hiermit, dass diese Dissertation und die darin dargestellten 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:

[BKK⁺22] Arvid Butting, Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Radoslav Orlov, and Bernhard Rumpe. Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices. In *Proceedings of the 21th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22)*, pages 108–121. ACM, December 2022

[KKR⁺22a] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *ACM Transactions on Internet of Things*, 3(4), 2022

[KKR⁺22b] Jörg Christian Kirchhof, Evgeny Kusmenko, Jonas Ritz, Bernhard Rumpe, Armin Moin, Atta Badii, Stephan Günnemann, and Mohararam Challenger. MDE for Machine Learning-Enabled Software Systems: A Case Study and Comparison of MontiAnna & ML-Quadrat.

In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, page 380–387, New York, NY, USA, October 2022. ACM.

- [KMM⁺22] Jörg Christian Kirchhof, Lukas Malcher, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. Web-Based Tracing for Model-Driven Applications. In *Proceedings of the 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA '22)*. In Press, 2022
- [KKM⁺22] Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. Efficiently Engineering IoT Architecture Languages—An Experience Report (Poster). STAF 2022 Workshop Proceedings: 10th International Workshop on Bidirectional Transformations (BX 2022), 2nd International Workshop on Foundations and Practice of Visual Modeling (FPVM 2022) and 2nd International Workshop on MDE for Smart IoT Systems (MeSS 2022) (co-located with Software Technologies: Applications and Foundations federation of conferences (STAF 2022)), July 2022
- [HKK⁺22] Mattis Hoppe, Jörg Christian Kirchhof, Evgeny Kusmenko, Chan Yong Lee, and Bernhard Rumpe. Agent-Based Autonomous Vehicle Simulation with Hardware Emulation in the Loop. In *2022 IEEE Intelligent Vehicles Symposium (IV)*, pages 16–21, 2022
- [KRSW22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, 183:111087, January 2022
- [KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21)*, pages 197–209. ACM SIGPLAN, October 2021
- [AKKR21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21)*, pages 55–68. ACM SIGPLAN, October 2021

- [KNS⁺21] Jörg Christian Kirchhof, Michael Nieke, Ina Schaefer, David Schmalzing, and Michael Schulze. *Variant and Product Line Co-Evolution*, pages 333–351. Springer, January 2021
- [KMR⁺20b] 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
- [KSGW20] Jörg Christian Kirchhof, Martin Serror, René Glebke, and Klaus Wehrle. Improving MAC Protocols for Wireless Industrial Networks via Packet Prioritization and Cooperation. In *Proceedings of the 21st International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM). Workshop CCNCPS.*, pages 367–372. IEEE, August 2020
- [KMR20a] Jörg Christian Kirchhof, Judith Michael, and Bernhard Rumpe. *Softwarequalität in Energieprojekten*, pages 273–279. Fraunhofer IRB Verlag, Stuttgart, July 2020
- [KRSW20] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Structurally Evolving Component-Port-Connector Architectures of Centrally Controlled Systems. In Maxime Cordy, Mathieu Acher, Danilo Beuche, and Gunter Saake, editors, *International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, February 2020
- [KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019
- [KKMR19] Jörg Christian Kirchhof, Evgeny Kusmenko, Jean Meurice, and Bernhard Rumpe. Simulation of Model Execution for Embedded Systems. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MLE*, pages 331–338. IEEE, September 2019

- [SKS⁺17] Martin Serror, Jörg Christian Kirchhof, Mirko Stoffers, Klaus Wehrle, and James Gross. Code-Transparent Discrete Event Simulation for Time-Accurate Wireless Prototyping. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, pages 161–172, New York, NY, USA, 2017. Association for Computing Machinery

Aachen, 11. Januar 2023

Jörg Christian Kirchhof

Abstract

The Internet of Things (IoT) describes the idea of connecting objects equipped with sensors and actuators to each other and to the Internet. IoT applications are complex to develop for a variety of reasons, including the heterogeneity of the IoT devices, diverse software stacks, the fact that IoT applications are usually distributed applications, and the fragility of the hardware and network connection. Model-driven methods promise to make the complex development of IoT applications manageable by raising the level of abstraction. Related work has proposed a variety of component and connector (C&C) architecture description languages (ADLs) for developing IoT applications. However, these mainly focus on the early development phases and largely neglect reliability aspects.

Accordingly, this work focuses on the model-driven engineering of IoT applications throughout their lifecycle. We present *MontiThings*, an ecosystem for model-driven IoT applications. Based on existing approaches, the MontiThings ecosystem specifies an IoT-focused C&C ADL using the MontiCore language workbench. MontiThings aims at offering an ecosystem that covers the lifecycle of IoT applications starting from the first architecture concepts up to the eventual deployment of the application and its analysis during runtime. At all stages of this process, MontiThings offers reliability mechanisms that can help developers to specify resilient applications.

For design activities, MontiThings provides a C&C language integrated with international system of units (SI) units and the object constraint language (OCL) usable to detect exceptional situations at operating time. Furthermore, MontiThings offers an integration method for hardware drivers that provides a clear separation of concerns and, thus, enables components to be reused and tested independently of their hardware integration. A generator translates the C&C architecture models to C++ code. Based on a tagging language, the IoT components can be integrated with synthesized digital twins. When deploying applications, MontiThings' requirements-based deployment method is able to not only calculate a distribution of components to IoT devices but can also actively propose changes to the user should their requirements be unfulfillable. If devices fail at runtime, MontiThings can automatically adapt the deployment to the changed situation (if possible within the requirements) and restore the previous software state of failed devices. To understand unforeseen situations that may arise at runtime, MontiThings provides developers with model-driven analysis services. Overall, MontiThings demonstrates an end-to-end model-driven approach for designing IoT applications.

Kurzfassung

Das Internet der Dinge (IoT) beschreibt die Idee, mit Sensoren und Aktuatoren ausgestattete Gegenstände untereinander und mit dem Internet zu verbinden. Die Entwicklung von IoT-Anwendungen ist aus verschiedenen Gründen komplex. Dazu gehören die Heterogenität der IoT-Geräte, die Tatsache, dass es sich bei IoT-Anwendungen normalerweise um verteilte Anwendungen handelt, und die Fehleranfälligkeit der Hardware und der Netzwerkverbindung. Modellgetriebene Methoden versprechen, die komplexe Entwicklung von IoT-Anwendungen durch die Anhebung des Abstraktionsniveaus handhabbar zu machen. In verwandten Arbeiten wurde eine Vielzahl von Komponenten- und Konnektor (C&C) Architekturbeschreibungssprachen (ADLs) zur Entwicklung von IoT-Anwendungen vorgestellt. Diese konzentrieren sich jedoch hauptsächlich auf die frühen Entwicklungsphasen und vernachlässigen weitgehend Zuverlässigkeitsaspekte.

Dementsprechend konzentriert sich diese Arbeit auf die modellgetriebene Entwicklung von IoT-Anwendungen über ihren gesamten Lebenszyklus hinweg. Wir stellen *MontiThings* vor, ein Ökosystem zur modellgetriebenen Entwicklung von IoT-Anwendungen. Basierend auf bestehenden Ansätzen spezifiziert das MontiThings-Ökosystem eine IoT-fokussierte C&C ADL unter Verwendung der MontiCore Language Workbench. MontiThings zielt darauf ab, ein Ökosystem anzubieten, das den Lebenszyklus von IoT-Anwendungen abdeckt, angefangen bei den ersten Architekturkonzepten bis hin zur Bereitstellung der Anwendung und der Analyse der Anwendung während der Laufzeit. In allen Phasen dieses Prozesses bietet MontiThings dabei Zuverlässigkeitsmechanismen, die den Entwicklern helfen können, robuste Anwendungen zu spezifizieren.

Für Designaktivitäten bietet MontiThings eine C&C-Sprache, die das internationale Einheitensystem (SI) und die Object Constraint Language (OCL) integriert, um Ausnahmesituationen zur Laufzeit zu erkennen. Außerdem bietet MontiThings eine Integrationsmethode für Hardwaretreiber, die eine klare Trennung von Zuständigkeiten und somit die Wiederverwendung und das Testen von Komponenten unabhängig von ihrer Hardwareintegration ermöglicht. Ein Generator übersetzt die C&C-Architekturmodelle in C++-Code. Basierend auf einer Tagging-Sprache können die IoT-Komponenten mit synthetisierten digitalen Zwillingen integriert werden. Beim Deployment von Anwendungen ist die anforderungsbasierte Deployment-Methode von MontiThings in der Lage, nicht nur eine Verteilung der Komponenten auf die IoT-Geräte zu berechnen, sondern dem Nutzer auch aktiv Änderungen vorzuschlagen, sollten seine Anforderungen nicht erfüllbar sein. Fallen Geräte zur Laufzeit aus, kann MontiThings das Deployment automatisch an die geänderte Situation anpassen (sofern es im Rahmen der Anforderungen möglich ist) und den vorherigen Softwarestand der ausgefallenen Geräte wiederherstellen. Zum Verständnis unvorhergesehener Situationen zur Laufzeit stellt MontiThings Entwicklern modellgestriebene Analysedienste zur Verfügung. Insgesamt demonstriert MontiThings eine durchgängig modellgetriebene Methode zur Entwicklung von IoT-Anwendungen.

Danksagung

Viele Menschen haben mich auf dem Weg zu meiner Promotion begleitet bei denen ich mich an dieser Stelle bedanken möchte.

An erster Stelle möchte ich mich gerne bei meinem Doktorvater Prof. Dr. Bernhard Rumpe bedanken. Zum einen bedanke ich mich für die konstruktiven wissenschaftlichen Diskussionen, durch die ich stets nochmal einen anderen Blickwinkel auf die Dinge erhalten habe, und zum anderen auch für die Möglichkeit mich in den Projekten mit Ford weiterzuentwickeln. Insbesondere bin ich dankbar für die Freiheit, immer an den Themen arbeiten zu dürfen, die mich am meisten interessieren.

Ich danke Prof. Mag. Dr. Manuel Wimmer für die Zweitbegutachtung dieser Arbeit. Darüber hinaus möchte ich mich bei Prof. Dr. Erika Ábrahám für die Bereitschaft bedanken, meine theoretische mündliche Prüfung abzunehmen, sowie bei Prof. Dr.-Ing. Stefan Kowalewski für die Übernahme des Vorsitzes meiner Prüfungskommission.

Insbesondere möchte ich mich auch bei den zahlreichen Kolleginnen und Kollegen am Lehrstuhl für Software Engineering für erfolgreiche und angenehme Zusammenarbeit bedanken. Durch euch wurde die Zeit meiner Promotion zu einer schönen Zeit, die mir auf ewig in Erinnerung bleiben wird. Ich danke Jun.-Prof. Dr. Andreas Wortmann, Dr. Evgeny Kusmenko und Dr. Judith Michael dafür, dass sie mir über die Jahre hinweg immer Mentoren waren und mich in allen Fragen des akademischen Arbeitens beraten haben. Ich danke David Schmalzing für die angenehme Büropartnerschaft und für die Pflege und Weiterentwicklung von MontiArc ohne das diese Arbeit nicht so möglich gewesen wäre. Simon Varga und Robert Eikermann danke ich dafür, dass sie mich als IT Admin aufgenommen haben. Insbesondere danke ich Simon Varga auch für die tolle Zusammenarbeit bei MontiGem und für die tollen Gespräche, wenn wir abends mal wieder als einige der letzten im Büro waren. Ich danke Arkadii Gerasimov, Lukas Netz, Galina Volkova und Kai Adam für die Hilfe bei der Benutzung von MontiGem. Bei Dr. Arvid Butting, Nico Jansen und Niklas Dienstknecht möchte ich mich für die Hilfe bei den MontiCore Versionsumzügen bedanken. Mein besonderer Dank gilt Sylvia Gunder und Sonja Müßigbrodt dafür, dass sie mich über die Jahre stets in allen organisatorischen Fragen unterstützt haben und jede noch so unüberwindbar scheinende administrative Hürde zu meistern wussten und dafür, dass sie mir bei der Bestellung von gefühlten drei Millionen Kleinteilen unterstützt haben, die ich für diese Arbeit und meine Projekte benötigte. Bei Deni Raco bedanke ich mich für die Organisation der unterhaltsamen Pokerabende. Desweiteren bedanke ich mich bei Daoud Ali, Vassily Aliseyko, Vincent Bertram, Miriam Boß, Jonas Böcker, Marita Breuer, Joel Charles, Manuela Dalibor, Florian Drux, Christoph Engels, Dr. Arne Haber, Malte Heithoff, Alexander Hellwig, Steffen Hillemacher, Hendrik Kausch, Dr. Marco Konersmann, Achim Lindt, Daniel Maibach, Dr. Matthias Markthaler, Joshua Mingers, Imke Nachmann, Mathias Pfeiffer, Nina Pichler, Manuel Pützer, Jonas Ritz, Dr. Christoph Schulze, Brian Sinkovec, Max

Stachon, Sebastian Stüber, Louis Wachtmeister, und Dr. Michael von Wenckstern. Bei meinen Hiwis Anno Kleiss, Julian Ruiz und Daniel von Mirbach und meinem Auszubildenden Julius Gummersbach bedanke ich mich dafür, dass sie mich bei der technischen Umsetzung dieser Arbeit unterstützt haben.

Außerdem möchte ich mich bei dem Team im Ford Research and Innovation Center bestehend aus Dr. Marcel Grein, Detlef Kuck, Nicole Eikelenberg, Jeroen Lem, Turgay Aslandere, Moritz Martinius, Alexandra Holz, Roy Hendrikx, Mark Gijbels, Abhinav Dhake und Walter Pijls bedanken durch die ich zum einen spannende Projekte bearbeiten durfte als auch wertvolle Einblicke in die Industrie erhalten habe. Ich danke Alice Minet vom Lehrstuhl für Marketing für die gute Zusammenarbeit bei diesen Projekten. Insbesondere danke ich auch den Mitarbeitern der DSA Daten- und Systemtechnik GmbH für die Unterstützung in technischen Fragen. Dr. Ansgar Schleicher danke ich für die gute Zusammenarbeit bei der Abhaltung der Vorlesung *Der digitale Lebenszyklus von Fahrzeugen als Teil des Internet of Things (IoT)*.

Mein Dank gilt außerdem meinen ehemaligen Betreuern Dr. Martin Serror, René Glebke und Dr. Mirko Stoffers am COMSYS dafür, dass sie mir die Grundlagen des wissenschaftlichen Arbeitens (und Schreibens!) beigebracht haben.

Außerdem danke ich meinen Freunden für die unterhaltsamen Abende durch die wir auch mal von Promotionen und Arbeit abschalten konnten. Von ganzem Herzen danke ich außerdem meinen Eltern und meinem Bruder dafür, dass sie mich nicht nur während der Promotion durch alle Phasen meines Lebens hinweg immer unterstützt haben. Ohne euch wäre mir diese Promotion nicht möglich gewesen.

Aachen, November 2022
Jörg Christian Kirchhof

Contents

I	Prologue	1
1	Introduction	3
1.1	Motivation	3
1.2	Goal, Approach, and Main Contributions	5
1.3	Thesis Organization	7
1.4	Publications	8
2	Background	13
2.1	Internet of Things (IoT)	13
2.2	Cloud Computing and Digital Twins in the Context of IoT	17
2.3	Model-Driven Software Engineering and Domain-Specific Languages	19
2.4	MontiCore	20
2.5	Software Architecture and Architecture Description Languages	23
2.6	MontiArc	25
3	Scope of the Thesis	29
3.1	Vision and Assumptions	29
3.2	Lifecycle and Development Process of IoT Applications	32
3.3	What Do IoT Projects Need?	35
3.4	Challenges	35
3.5	Research Questions	37
3.6	Requirements	39
3.7	What Is Out of Scope?	42
3.8	Method at a Glance	43
3.9	Running Use Case: Smart Home	48
II	The MontiThings Ecosystem for Model-Driven IoT Applications	49
4	C&C-based IoT Application Development	51
4.1	Research Questions	51
4.2	MontiThings Language	51
4.2.1	Component Definition and Instantiation	53

4.2.2	Type System	56
4.2.3	Timing	58
4.2.4	Behavior Description	59
4.2.5	OCL	63
4.2.6	Sensor and Actuator Access	66
4.2.7	Dynamic Reconfiguration	68
4.3	Language Integration	72
4.3.1	Integration With Class Diagrams	72
4.3.2	Configuration Language	73
4.3.3	Sequence Diagram Test Specification	78
4.4	Discussion	79
5	Code Generation	85
5.1	Methodology and Tool Infrastructure	85
5.2	Run-time Environment (RTE)	88
5.2.1	Components and Event-Handling	89
5.2.2	Ports and Communication Technologies / Protocols	90
5.3	Generated Code Structure	96
5.3.1	Architecture Partitioning and Setup Information Exchange	98
5.3.2	Generated CLIs	100
5.3.3	Generated Scripts and Compilation	102
5.3.4	Supporting Different Target Platforms	103
5.3.5	Test Case Generation	103
5.4	Discussion	104
6	Deployment and Integration of C&C-based IoT Applications	109
6.1	Research Questions	109
6.2	Development and Deployment Processes	110
6.3	Requirement-based Deployment	113
6.3.1	Deployment Workflow	113
6.3.2	Deployment System Overview	116
6.3.3	Prolog Code Generation	120
6.4	Feature-based Deployment	126
6.5	Model-driven App Store Concept	129
6.6	Integration with Model-driven Information Systems: Synthesizing Digital Twins	130
6.7	Discussion	137
7	Execution and Runtime Analysis of C&C-based IoT Applications	145
7.1	Research Questions	145
7.2	Methodical Considerations	146

7.3	Fault Tolerance	149
7.4	Tracing Behavior and Filtering Logs	151
7.5	Transformation-based Record and Replay	155
7.6	Discussion	159
III	Evaluation and Conclusion	165
8	Experiments	167
8.1	Case Study 1: Smart Home and Smart Hotel	167
8.2	Case Study 2: Fire Alarm Digital Twin	173
8.3	Case Study 3: HVAC Reproduction	176
8.4	Performance Evaluation: Transformation-based Replayer	180
8.5	Performance Evaluation: Log Tracing	182
8.6	Student Lab: Autonomous Driving	185
8.7	Student Lab: Fischertechnik	187
8.8	Discussion	189
9	Conclusion and Future Research Directions	191
	Bibliography	195
A	Acronyms	215
B	Selected Grammars from the MontiVerse	217
B.1	ArcBasis (MontiArc)	217
B.2	Class Diagrams	221
B.3	MCCCommonStatements	227
B.4	MCCCommonLiterals	229
B.5	OCL Expressions	235
B.6	Set Expressions	240
B.7	SI Units	243
C	MontiThings Grammars	253
C.1	Behavior	253
C.2	Error Handling	255
C.3	Set Definitions	256
C.4	MontiThings Main Grammar	257
C.5	Configuration	259
D	Open Source Software Used In RTE	263

E Models of the HVAC Case Study	265
F Diagram and Listing Tags	269
List of Definitions	271
List of Figures	273
Listings	279
List of Tables	281
Related Interesting Work from the SE Group, RWTH Aachen	283

Part I

Prologue

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) promises to turn our world into a substantially more digital one by connecting everyday objects with each other and with the Internet. Thereby, the IoT creates new opportunities in many domains, including smart homes, smart cities, health care, retail, agriculture, security and surveillance, logistics, and Industry 4.0 [DeF21, TM17a, TGPH20, MSPC12, AIM10]. By 2026, Ericsson expects 26.9 billion IoT connections [www20]. Due to their heterogeneity and size, the development of such systems is becoming increasingly complex.

Due to their lack of abstraction, traditional general-purpose languages are not well suited to meet the challenges of future IoT applications. The development of IoT is fundamentally different from the development of *traditional* applications [TM17a], *e.g.*, smartphone apps, because it forces developers to take new kinds of programming problems into account. This includes, for example, multi-device programming of decentralized devices [TM17a, MHF17], programming for heterogeneous target devices [MHF17, MSPC12], and self-adaptation of software [CS16, MSPC12]. The challenges and relevance of IoT development are underlined by the fact that all major cloud providers, *i.e.*, Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), started offering specialized IoT software as a service solutions in recent years. While some “past software engineering techniques can be harnessed and adapted to the challenges [...], new approaches to standard software engineering techniques are also needed” [LCFT17]. Most prominently, Google has discontinued *Android Things*, its attempt to create a general operating system for the IoT, after less than three years¹. Model-based development promises to make the complexity of IoT application development manageable [MHF17] by raising the level of abstraction. Consequently, the future of IoT is expected to be model-based [SKS18].

Related work suggests that model-based IoT applications shall be defined using component and connector (C&C) architecture description languages (ADLs), as

¹Ars Technica: “Google kills Android Things, its IoT OS, in January”, 17.12.2020. [Online]. Available: <https://arstechnica.com/gadgets/2020/12/google-kills-android-things-its-iot-os-in-january/> Last accessed: 11.01.2022

many popular approaches from both industry and academia use component-based architecture models. Examples include ThingML [HFMH16, MHF17], Ericsson’s Calvin [AP17, PA15, PA17], CapeCode [BJK⁺18], and Node-RED². Existing domain-specific languages (DSLs) for IoT applications, however, suffer from a lack of abstraction (*e.g.*, Eclipse Mita³), a limited focus on development that disregards challenges occurring later in the lifecycle of IoT applications (*e.g.*, CapeCode), or a lack of separation of concerns leading to models that are hard to understand for non-experts because they contain low-level code fragments (*e.g.*, ThingML). In general, “[m]odel-driven development remains mostly not adopted in IoT development at this stage” [DRF22].

As IoT applications are often based on a large number of devices that are sometimes operated under harsh environmental conditions, IoT applications are prone to various types of failures. These failures range from erroneous sensor readings [MNZC20, KMMN16, FG08] to unexpectedly failing devices [TM17a, MSPC12] to network failures that can disconnect groups of devices at once. Thus, reliability is an important aspect of developing IoT systems [MNZC20, HLR17, TM17a, Sta14, MSPC12] to enable them to cope with various types of failures. Related work on model-based IoT application development does not take this problem into account sufficiently and focuses mostly on superficial methods such as collecting logs (*e.g.*, [MF19]). Overall, “IoT system reliability is something that will need to be addressed comprehensively in order for the technology to fully mature” [MNZC20].

Additionally, there is a lack of research on IoT deployment. Currently, IoT applications are usually distributed as a bundle consisting of a piece of hardware and pre-installed software by the same vendor. Future IoT applications are expected to be distributable in app stores similar to today’s smartphone app stores [BSS⁺17, MM12], thus decoupling hardware and software vendors. Current deployment methods will no longer be sufficient as “IoT applications and services will have to be deployed over an existing IoT hardware infrastructure” [Zam17] that is not necessarily from the vendor that sells the software. Related academic work focuses mostly on rule-based approaches (*e.g.*, Calvin) or the technical aspects of the deployment, *i.e.*, copying software to the IoT devices (*e.g.*, GeneSIS [FN19, FNS⁺19, FNS⁺20]). Similarly, commercial products by cloud providers are usually based on grouping devices based on certain properties. Moreover, deployment also needs to take into account that new stakeholders, *e.g.*, the person managing the IoT devices, may want to influence the deployment of the software to fit their needs. [Zam17] outlines abstractions for IoT software, including stakeholders. Related work, however, does not yet take such new stakeholders into account. Overall, IoT deployment is considered to be “still in its infancy” [NFE⁺19] and new deployment methods are needed to one day reach the vision of an IoT app store.

²Node-RED project website. [Online]. Available: <https://nodered.org/> Last accessed: 11.01.2022

³Eclipse Mita project website. [Online]. Available: <https://www.eclipse.org/mita/> Last accessed 11.01.2022

1.2 Goal, Approach, and Main Contributions

Overall, this thesis contributes to answering the unanswered question of whether “model-driven engineering (MDE) [can] play a key role in the future of IoT” [BCPP20]. As discussed above, related work does not sufficiently address aspects such as reliability and is mostly focused on generating software in early development stages. Furthermore, the deployment of IoT applications in future IoT app stores is still an open challenge since stakeholders such as device owners are not adequately integrated into the deployment process. This leads us to the following question:

Main Research Question:

How to develop and deploy failure resilient model-driven IoT applications and analyze their generated behavior?

To answer this question, we present *MontiThings*, an ecosystem for model-driven IoT applications. Based on the existing approaches, the MontiThings ecosystem specifies an IoT-focused C&C ADL using the MontiCore [HKR21] language workbench. A code generator can generate C++ code from MontiThings models. This code can then be packaged as container images and be deployed to IoT devices based on sets of requirements that can be defined by both the application developers and the device owners. This includes, in particular, integration aspects to connect the IoT application to the hardware of the IoT devices, *e.g.*, sensors and actuators, and external services such as digital twins. In particular, we examine how MontiGem, a tool for the model-driven development of information systems, can be utilized to create digital twins of IoT components and how automatic synchronization can be established between the (generated) information system and the IoT devices. During run-time, the deployed application may interact with various services that provide, *e.g.*, failure tolerance or error analysis. Thereby, MontiThings aims at offering an ecosystem that covers the lifecycle of IoT applications starting from the first architecture concepts up to the eventual deployment of the application. At all stages of the process, MontiThings offers resiliency mechanisms that can help developers to specify reliable applications. As developers are usually not used to handling IoT-specific problems [TM17a, LCFT17], *e.g.*, device failures, this thesis especially examines reliability mechanisms that the code generator and deployment system can automatically add to the (generated) application without being explicitly specified by the developers.

Consequently, the main contributions of this thesis are:

- a (non-exhaustive) set of requirements of model-driven ecosystems for IoT applications in the context of the vision of creating future IoT app stores

- the MontiThings C&C ADL for creating IoT applications that offers an IoT-focused behavior language, OCL-based error-detection, and a strong type system incorporating international system of units (SI)
- a configuration language for integrating MontiThings with different hardware platforms
- a sequence diagram-based language for specifying tests of MontiThings models
- a code generator producing C++ code from MontiThings models and scripts to package and deploy the generated code
- an run-time environment (RTE) for IoT applications that supports different communication technologies and deployment styles
- an automated requirement-based deployment process for IoT applications based on existing continuous integration (CI) infrastructures such as GitLab and GitHub utilizing code generation and Prolog to take different stakeholders into account and propose changes if the requirements are unfulfillable
- a workflow for integrating MontiThings-based IoT applications with accompanying MontiGem-based information systems
- a tagging-based integration between MontiThings-based IoT applications and class diagrams used in MontiGem-based information systems that separates the business logic of both systems from integration aspects
- model-to-model transformations for synthesizing digital twin synchronization services using the tagging-based integration of MontiThings and class diagrams to keep MontiThings and MontiGem applications synchronized
- methods for automatically handling temporary or permanent failures of IoT devices and restoring the state of failed software components in case of permanent failures
- an analysis service for tracing the behavior of MontiThings applications at runtime and filtering relevant log messages from large lists of available log messages
- a service for recording the behavior of MontiThings applications including environmental influences such as sensor readings or network delays
- model-to-model transformations for reproducing the behavior of MontiThings applications including environmental influences
- a workflow for integrating the reproduction-based analysis of IoT applications into iterative development processes

- multiple case studies that evaluate MontiThings in multiple smart home scenarios
- performance measurements for the prototypical implementations of MontiThings' analysis services
- summaries of the experiences of using MontiThings in student labs to teach entry-level developers model-driven IoT development

1.3 Thesis Organization

The following chapters will present the MontiThings ecosystem for model-driven development and deployment of IoT applications. The chapters are structured as follows:

Chapter 2 presents background knowledge necessary for understanding this thesis such as an overview of the IoT, MontiCore, and MontiArc;

Chapter 3 gives an overview of the vision this thesis is based on, defines the scope of the thesis (and what is explicitly out of scope), explains research questions, and introduces the smart home as a running use case;

Chapter 4 presents the MontiThings language for specifying IoT applications and its integration with auxiliary languages, *i.e.*, a class diagram language, a configuration language, a test case definition language;

Chapter 5 explains the C++ code generator and run-time environment of MontiThings;

Chapter 6 introduces MontiThings' deployment algorithm based on the generation of Prolog code that is able of proposing to buy new hardware and relaxing requirements. Moreover, this chapter shows MontiThings' integration with digital twins synthesized from tagging models;

Chapter 7 describes MontiThings' error handling and analysis techniques at run-time;

Chapter 8 evaluates MontiThings using numerous case studies, performance measurements, and student labs;

Chapter 9 concludes this thesis and gives an outlook on future work.

Throughout this thesis, we will often refer back to the challenges, requirements, and research questions introduced in Chapter 3. In order not to affect the readability of the text too much, abbreviations are shown in parentheses and in bold font for this purpose (Example: **(TC3)**).

1.4 Publications

This thesis is the result of several years of research. Some of the results have therefore already been published in various contexts prior to this thesis. Accordingly, some of the results, figures, data and other content from this this thesis have been published at conferences and in journals or are currently in press or preparation. This section provides an overview of these publications.

- [BKK⁺22] discusses a concept of how to utilize model-driven development in future IoT app stores to create customizable applications without a tight coupling of hardware and software. The description of MontiThings' utilization of feature diagrams (Sec. 6.4) and the app store concept (Sec. 6.5) are based on this paper and the explanation of the hardware access (Sec. 4.2.6, Sec. 5.2.2) is an extended version of the description in this paper. Further, the explanation of class diagrams, object diagrams, and object constraint language (OCL) to check if an IoT device fulfills technical requirements (*cf.* Sec. 6.3.3) is based on this paper. The research problem and concepts were designed by the author of this thesis. The usage of feature diagrams was conceptualized in collaboration with Arvid Butting. Radoslav Orlov provided a prototypical implementation of the feature diagram integration as part of [Orl22]. Anno Kleiss provided a prototype implementation of the Prolog generator for checking which devices are able to execute which software. The case study is a joined effort of the authors.
- [KKR⁺22a] presents the requirements-based self-adaptive deployment of MontiThings. Unlike related work, our approach is also capable of making modification proposals in case the requirements are not fulfillable. Sec. 3.9, Sec. 6.3, Sec. 4.2.7, and Sec. 8.1 are mainly based on this paper. The research problem of offering modification proposals was defined in discussions between Bernhard Rumpe and the author of this thesis. The concepts were mainly developed by the author of this thesis and their prototypical implementation were mainly done by Philipp Schneider as part of [Sch21], the author of this thesis, and a student workshop supervised by the author of this thesis. The concept for dynamic architectures resulted from discussions between the author of this thesis, Bernhard Rumpe, and David Schmalzing. The smart hotel case study was planned by the author of this thesis and carried out by Philipp Schneider, Anno Kleiss, and the author of this thesis.
- [KMM⁺22] discusses a method for integrating web-based behavior tracing into behavior-focussed DSLs. A concrete implementation in MontiThings serves as an evaluation. Sec. 7.4 and Sec. 8.5 are mainly based on this paper. The main research problem and concept was developed by the author of this thesis and detailed in collaboration with Lukas Malcher. The generalization of the concepts beyond

the scope of MontiThings was developed in discussion between the author of this thesis, Andreas Wortmann, and Judith Michael. The prototypical implementation for MontiThings was realized by Lukas Malcher as part of [Mal21].

- [KKM⁺22] gives a short overview of MontiThings as a MontiCore-based C&C language. It contains a highly condensed overview of the contents of Chapter 4.
- [KRSW22] gives an overview of the MontiThings ecosystem as a whole, including the development and the deployment process. Especially, this publication focuses on how to handle error situations. Besides serving as a basis for Chapter 4, Chapter 5, and Chapter 6, especially Sec. 4.2.6, Sec. 4.3.1, and Sec. 7.3 are strongly based on this paper. The research problem, concepts, and discussion were mainly provided by the author of this thesis. Over the years, all authors have contributed to the design of MontiThings through numerous discussions. The case studies were designed by the author of this thesis. The author of this thesis carried out the modelling and deployment case studies, David Schmalzing carried out the scalability evaluation.
- [KMR21] describes a method for understanding (especially hardware-related) errors in MontiThings applications by recording data during the execution of the system, transforming the models to replay the recorded data, and then analyzing the application generated from the transformed models. Sec. 7.2, Sec. 7.5, Sec. 8.3, and Sec. 8.4 are mainly based on this paper. The research problem and general concept were designed by the author of this thesis. As part of his thesis [Mal21], Lukas Malcher provided a prototypical implementation. The concepts were detailed in a joint effort. The evaluation was designed in a joint effort and carried out by Lukas Malcher.
- [KMR⁺20b] presents a tagging-based method for synthesizing digital twins from C&C architecture models and class diagrams that describe the data structure of a generated information system. By applying model-to-model transformations we add the necessary infrastructure to the models that keep them synchronized. Sec. 6.6 and Sec. 8.2 are mainly based on this paper. The author of this thesis contributed the definition of the research problem, the main concepts, and design of the case study. Simon Varga conceptualized the data processing in MontiGem.

The author of this thesis is the main author of all aforementioned papers. Furthermore, the following publications have been published in the context of collaborations with other scientists, which, however, are not or only insignificantly related to the present work.

- [KKR⁺22b] compares two model-driven frameworks for machine learning, MontiAnna and ML².

- [HKK⁺22] extends the hardware emulator presented in [KKMR19] by a memory model to increase the accuracy of the emulation.
- [AKKR21] introduces an artifact model and a toolchain for incorporating machine learning models into C&C-based development processes. The artifact model was also conceptually applied to MontiThingsDL [Zha20]. The implementation was, however, based on EmbeddedMontiArc.
- [KNS⁺21] describes the results of the CrEst project⁴ on how to co-evolve artifacts in a product line.
- [KSGW20] analyzes the effects of different prioritization and cooperation mechanisms on media access control (MAC) protocols using network simulations. As a result, we provide guidelines for network protocol developers.
- [KMR20a] describes approaches to ensure software quality with a focus on projects in the energy sector.
- [KRSW20] introduces a method for differencing MontiArc C&C architectures based on their structures that connect the component instances.
- [KKRZ19] presents a simulation-as-a-service concept for model-driven applications with a focus on automated driving simulations.
- [KKMR19] proposes a hardware emulator for model-driven embedded applications.
- [SKS⁺17] presents a code-transparent network simulator for the wireless open access research platform (WARP). The simulation also allows WARP applications to be examined using traditional debugging tools such as `gdb`. As a result, multiple bugs in the WARP were found, reported, and fixed.

Lastly, it is worth noting that multiple bachelor and master theses have been carried out in the context of MontiThings. Each of them was supervised or co-supervised by the author of this thesis and hence the research problems and most of the concepts developed in these theses were created or strongly influenced by the author of the thesis at hand. Accordingly, parts of the results in this thesis were previously also described in these theses. For most of these theses, parts of the code created during these theses ended up in the present version of MontiThings.

- [Für20] provided the first prototype of MontiThings. This first version relied on C++ code within the models and a static deployment mechanism from a prior thesis. Most of these concepts were removed later. The general idea of modeling

⁴Project website. [Online]. <https://crest.in.tum.de/>. Last accessed: 13.10.2021

the hardware access within ports and having outsourcing hardware access to standalone applications was kept. These concepts were later extended by an overriding mechanism and reworked to make these ports transparent in the models.

- [Häu20] implemented a fault tolerance mechanism based on replaying messages. The general concept was already developed by the author of the thesis at hand before the start of the thesis. [Häu20] provided a scalable implementation for these concepts based on the replication of infrastructure elements, Apache Kafka, and a cloud provider.
- [Zha20] integrated Deep Learning based on MontiAnna [GKR19, KNP⁺19, KPRS19] components into MontiThings.
- [Kre20] provided an implementation for the digital twin synthesization described in [KMR⁺20b]. The concepts described in [KMR⁺20b] were already finished - but not published - before the start of the thesis presented in [Kre20]. Julian Krebber implemented a prototype and implemented the case study in Sec. 8.2 based on the electrical setup from [Für20].
- [Kle21] implemented a MontiCore-based variant of the OCL/P [Rum16, Rum17] and integrated it into MontiThings. Furthermore, an existing language component for SI units was integrated into MontiThings' type system.
- [Mal21] provided communication via the OpenDDS framework, record-and-replay of applications, and a log tracing mechanism. Parts of the results were also published in [KMR21] and [KMM⁺22]. The case study and performance measurements in Sec. 8.3, Sec. 8.4, and Sec. 8.5 were planned and analyzed in collaboration with Lukas Malcher. They were carried out by Lukas Malcher.
- [Sti21] enabled components to be controlled via MontiGem applications by integrating MQTT and GUI components for MontiThings components into MontiGem.
- [Sch21] rewrote the deployment manager to be more extendable, provided a web application for setting deployment rules, and extended the Prolog deployment generator. First versions of these tools were developed before the start of the thesis during a student lab supervised by the author of the thesis at hand. Parts of the results were published as part of [KKR⁺22a].
- [Sas22] examined methods of integrating multiple independent MontiThings applications with each other. Furthermore, the connection with external connectors was extended by offering a management system that coordinates the hardware access between multiple components that might want to access the same hardware resources.

- [Orl22] extended the deployment system with a feature diagram-based tool for selecting desired features from an end-user point of view. This further raised the level of abstraction compared to solely basing the deployment on the architecture models. Parts of the results are part of [BKK⁺22].
- [Rui22] extended the digital twin tagging concept from [Kre20] to enable a more customized synchronization and provided a method for integrating external (cloud) services into MontiThings based on OpenAPI specifications of the external services by generating components from OpenAPI specifications.

The author of this thesis also supervised numerous theses outside the context of MontiThings that are not described here because the results are not used in this thesis.

Chapter 2

Background

This thesis investigates the model-driven software engineering (MDSE) of IoT applications. This chapter introduces the necessary foundations and definitions for understanding this thesis. Sec. 2.1 generally introduces the IoT and the engineering challenges it poses. Since this thesis takes a model-driven approach, Sec. 2.3 briefly outlines the basics of MDE and domain-specific (modeling) languages. Sec. 2.4 introduces the language workbench *MontiCore* which is used to technically implement all languages presented in this thesis. Since the *MontiThings* language presented in this thesis is an architectural language, Sec. 2.5 explains the basics of software architectures and ADLs. Finally, Sec. 2.6 gives a brief overview of *MontiArc*, which serves as the foundation for *MontiThings*. For most of the topics discussed here, much more detailed introductions also exist, which are beyond the scope of this chapter. The respective sections provide references to further reading for interested readers.

2.1 Internet of Things (IoT)

Broadly speaking, the IoT describes the idea of connecting of everyday objects (“things”) to the Internet and thus form a network of interacting *things* that provides high-level functionalities. By equipping these interconnected everyday objects with sensors and actuators, it becomes possible to offer users services that react to and influence the system’s environment. Prominent application domains include smart home applications. For example, in a smart home, a rain sensor can detect that it is starting to rain and close the windows in response.

The application domains of the IoT are very broad and range from smart homes to industrial applications, e.g. in the area of health or agriculture, to infrastructure projects in the area of smart cities [PHPH19, GVM⁺17]. In addition, the IoT presents technical challenges in many areas (both software and hardware related), ranging from establishing connectivity to specifying application logic. The wide dispersion of applications and technical challenges makes it challenging to find a definition for the IoT that covers all relevant aspects. Accordingly, there is currently a lack of a common understanding of how exactly the IoT is defined [ASD19, RLC⁺20]. [AIM10] proposes to distinguish

between a “thing-oriented”, a “semantic-oriented”¹, and an “Internet-oriented” vision separating the different engineering aspects of designing IoT applications. Many attempts of defining the IoT also focus on certain technical aspects of the devices or applications. For example, [MSPC12] concentrates on the ability to identify things, connect them, and have them interact.

Since there is no universally accepted definition of IoT systems [DeF21], it is impossible to select one without possibly neglecting or misbalancing relevant aspects. This thesis is about software engineering of IoT systems. We, therefore, define the term “IoT system” from a software engineering perspective and select a set of characteristics mentioned in many other definitions that significantly influence software engineering.

Sensing Capabilities IoT systems have the ability to perceive properties of their environment through sensors. Sensors can sense a variety of different physical quantities and exist in different levels of complexity. Simple sensors convert only one measured value into a specific electrical voltage. More complex sensors, for example, record camera images and make them available in digital form.

Actuation Capabilities Actuators enable IoT systems to influence their environment. Similar to sensors, actuators exist in various degrees of complexity. In particular, from our software engineering point of view, we also consider hardware that presents information to the user of the system as actuators. This may include, for example, displays, LEDs, or speakers. Although these elements do not usually directly influence the environment, they can influence the behavior of users and thus indirectly influence the environment.

Distributed System While in some cases individual devices are also categorized as IoT, IoT systems often consist of multiple devices interacting with each other. Among other things, the interaction of multiple devices allows IoT systems to perceive and influence the environment also in a spatially distributed manner. Many use cases, *e.g.*, smart cities, can only be realized by such interaction of several devices.

Heterogeneity For many sensors and actuators, there are often only software libraries that support only certain platforms (*e.g.*, Arduino) and/or programming languages. This complicates combining multiple sensors and actuators in a system. Furthermore, the types of devices used in IoT projects range from microcontrollers with only a few kilobytes of memory (*e.g.*, ATmega328P) to cloud services with virtually unlimited resources [MSPC12].

Connectivity To interact with each other, devices in IoT systems must support some form of connectivity. It has become apparent that classic protocols such as TCP

¹*Semantic* in this context refers to “issues related to how to represent, store, interconnect, search, and organize information generated by the IoT” [AIM10]. This is in contrast to the understanding of semantics in the context of language engineering presented in [HR04].

or UDP are unsuitable for some IoT projects [SYDZ16], *e.g.*, when energy consumption constraints apply. To cope with the requirements of IoT devices, new communication protocols are needed, *e.g.*, Narrowband IoT or LoRaWAN. Such low-level communication aspects are out of the scope of this thesis. Nevertheless, the ability to communicate is an important characteristic of devices in IoT systems.

A sharp distinction between systems that may be categorized as IoT and those that may not is not possible. The characteristics mentioned here should help to classify a given system on the spectrum of possible systems, in order to be able to decide in the individual case whether one would like to classify the system under IoT. In doing so, we acknowledge that there is a “gray area” of systems about which no definite statement can be made.

Of course, in addition to the characterizing features of IoT systems, there are a variety of requirements and challenges. For example, the sheer mass of IoT connections—Ericsson expects 26.9 billion IoT connections in 2026 [www20]—imposes scalability requirements. A more detailed discussion of these challenges can be found, for example, in [AIM10, MSPC12]. Taivalsaari and Mikkonen provide a (non-exhaustive) list of seven aspects in which the software engineering of IoT applications differs from the software engineering of traditional applications [TM17a]:

1. “IoT devices are almost always part of a larger system”
2. “IoT systems never sleep”, *i.e.*, “IoT systems usually shouldn’t or can’t be shut down in their entirety”
3. “IoT systems are more like cattle than pets”, *i.e.*, they “must be managed en masse instead of receiving personal attention and care”
4. “IoT devices are often embedded in our surroundings such that they’re physically invisible and unreachable”
5. “IoT systems are highly heterogeneous”
6. “IoT systems tend to have weak connectivity, with intermittent and often unreliable network connections”
7. “IoT system topologies can be highly dynamic and ephemeral”

These aspects undoubtedly complicate the software development of IoT systems. Therefore, methods to systematically address the challenges arising from these aspects are needed.

The heterogeneous and dynamic nature of IoT systems means that the operation of an application no longer involves just the developer and the user. It is expected that future software will be more customizable to customer needs. From a software engineering

point of view, this increased customer influence results in new roles for the stakeholders involved in the development process. In this thesis, we will use adapted and renamed versions of the stakeholders defined in [Zam17]:

Definition 1 ((IoT) Developer). *(IoT) Developers are the product owners of an IoT system. They design (parts of) the system and its functionality or hardware and specify the requirements to the environment in which an application can be executed.*

This definition of IoT developers is based on the definition of “global managers” of [Zam17]. Note that the global manager in the definition of [Zam17] is seen mainly as a system owner in the sense of device owners². In this thesis, we understand global managers as hardware or software vendors. Thus, we do not consider them to have access to the infrastructure at runtime. When they sell their products through physical or virtual stores, they relinquish their direct control over the particular instances of their products. Accordingly, we use a different definition for global managers in this thesis.

Definition 2 (Device Owner). *Device owners “are the owners of, or delegates (permanent or temporary) given control over, a portion of the IoT system. They’re empowered to enforce local control and policies on that portion.” [Zam17]*

In [Zam17], this role is called “local manager”. They can choose where to place devices and are responsible for their maintenance. While they can choose which software should be executed on their IoT devices, they do not design the software themselves. Especially, they are not expected to be (software) engineers. They may, however, specify certain requirements or policies on the software to adapt it to their needs within the boundaries specified by the IoT developers.

Definition 3 (User). *“[U]sers are persons or groups that have limited access to the overall configuration of the IoT applications and services. That is, they can’t impose policies on the IoT but nevertheless are entitled to exploit its services.” [Zam17]*

²Original definition: “[...] global managers are the owners of an IoT system and infrastructure or are delegates empowered to exert control over and establish policies regarding the configuration and functioning of its applications and services.” [Zam17]

Users are the end-users of IoT systems. They interact directly with the IoT devices' physical or virtual user interfaces to achieve a goal. These interactions include, for example, pressing buttons, listening to speaker output, triggering a motion sensor by walking by, or using a mobile app to influence the system. In certain cases, users may also be device owners of a system. For example, if a fire alarm system is installed in a smart home, the residents may be both the device owners and users of their IoT devices. In contrast, if the same fire alarm system is installed in an office building, the janitor of the building might be the device owner and the employees might be the users.

2.2 Cloud Computing and Digital Twins in the Context of IoT

Today, many IoT applications are “cloud-centric” [TM17a]. Cloud providers, in general, provide numerous infrastructure and software solutions *as-a-service*, *i.e.*, without requiring maintenance from the users. These *as-a-service* solutions exist at different levels of abstraction, ranging from infrastructure such as storage and virtual machines to fully managed software solutions such as machine learning predictions based on models trained by the cloud provider to fulfill tasks such as face recognition. The level of abstraction of the service offering usually also determines the amount of maintenance the customers have to do themselves. For example, when using infrastructure-as-a-service virtual machines, where the content is managed by the user but the hardware the virtual machine runs on is maintained by the cloud provider. Most of these services feature a payment model, where the customers are billed based on their usage of the service. In extreme cases of serverless functions, this can go as far as billing milliseconds of CPU usage.

All major cloud providers, *i.e.*, AWS, Microsoft Azure, and Google Cloud Platform (GCP), also provide specialized IoT services. The most basic service offered by all of them³ consists of a registry for managing IoT devices and enabling them to securely connect to the cloud to send sensor data to the cloud. Besides these basic services, some cloud providers also offer solutions for certain IoT use cases, including the deployment of software (updates) or analyzing streams of events.

The general architecture of today's IoT applications utilizing the cloud looks as follows [TM17a] (Fig. 2.1): Very low-powered IoT devices that have no direct Internet connection connect to higher-powered IoT devices in their surrounding, the so-called *gateways*. In addition to sending their own data to the cloud, these gateways also forward the data of the low-powered devices to the cloud. In the cloud, this data is stored and further processed. The exact choice of storage and processing depends on the use case of the IoT application. For example, if the data is mostly stored for archival purposes, the cost can often be reduced by storing it in cheap storage with long access times. Processing the data can include, *e.g.*, visualizing it or classifying it using machine

³AWS and GCP call this service *IoT Core*, Azure calls it *IoT Hub*.

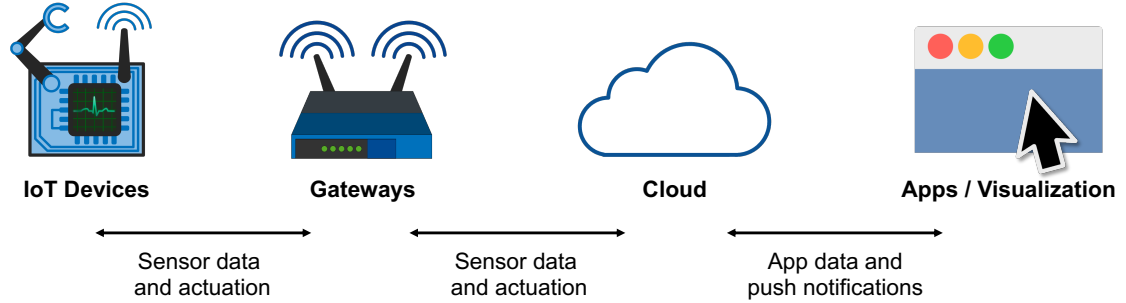


Figure 2.1: Common architecture of IoT applications as identified by [TM17a] (simplified and redrawn from [TM17a]).

learning. If necessary for the use case, apps and other user interfaces connect to the cloud instead of connecting directly to the IoT devices.

Apart from the basic connectivity, many cloud providers also offer more far-reaching IoT services. An important service for IoT applications are connections to *digital twins*. Unfortunately, there is no commonly agreed-on definition of digital twins [KMR⁺20b, EBC⁺22]. In this thesis, we will use the definition developed by our chair for the *Internet of Production*, an alliance of over 200 scientists from various domains:

Definition 4 (Digital Twin). “A *digital twin (DT)* of a system consists of a set of models of the system, a set of contextual data traces and/or their aggregation and abstraction collected from a system, and a set of services that allow using the data and models purposefully with respect to the original system.” [KMR⁺20b]⁴

In contrast, to many other definitions, this definition also takes model-driven development into account. From the point of view of cloud providers, digital twins often contain a synchronization aspect: The IoT devices in these cases have a virtual counterpart in the cloud that is synchronized with the real IoT devices. The digital representation then also acts as a stand-in for the real device while the real device is offline, *e.g.*, to save energy. Once the real device comes back online, it is synchronized with its digital representation.

Digital twins are, however, not limited to synchronization tasks as the above definition shows. Synchronization is only one of many popular services a digital twin can offer.

⁴While this exact wording was cited from this publication that the author of this thesis was involved in, the author of this thesis does not claim to have invented this definition. It was developed as part of the Internet of Production, as also mentioned in [KMR⁺20b]. Variants of this definition can be found in [BDH⁺20] and [EBC⁺22]. A similar definition by the Internet of Production for the closely related concept of “digital shadows” can be found in [BDJ⁺22].

Other services include for example simulation and analysis of the system for, *e.g.*, offering predictive maintenance.

2.3 Model-Driven Software Engineering and Domain-Specific Languages

Trends such as the IoT are presenting software development with new challenges. In addition, the development of more and more specialized software, *e.g.*, in the area of legal tech or healthcare, also requires developers to have extensive knowledge outside of software development. Overall, the increasing complexity of software development requires an increase in the level of abstraction.

Such an increase in the level of abstraction can be achieved by using *models* [HT06]. According to Stachowiak [Sta73] models have three main characteristics: *Representation*, *reduction*, and *pragmatism*. These three characteristics mean that a model is modeled from an original (representation), does not represent all the characteristics of the original (reduction), and that the model was created for some purpose (pragmatism). In MDSE processes, models are central development artifacts [Rum16, Rum17]. Especially, the models are used among other things to generate (parts of the) code of the software [Sel03]. This way, models increase the level of abstraction similar to how today's programming languages increase the level of abstraction compared to writing machine-specific instructions [Sel03].

To generate code from models, the models must be machine-readable. Therefore, models are created with modeling languages that specify the syntax and semantics of the models. The syntax does not necessarily have to be textual, but can also be graphical. Prominent examples of families of modeling languages are UML [Obj17] and SysML [Obj19]. UML and SysML define widely used model types in software development, such as class diagrams or activity diagrams.

Neither UML nor SysML are tailored to any particular domain and mainly address technical experts. Software developers are in many cases not domain experts of the domain for which they develop software. Domain experts, in turn, are rarely also software experts. This leads to a “conceptual gap” [FR07]. Models can enable domain experts to make their domain knowledge available through models in an abstract yet structured way, they can have more influence on the software without having to become software experts themselves. By using the models created by domain experts for code generation, domain experts without in-depth software knowledge get more involved in the software development process, including in terms of design. So-called DSLs are (modeling) languages tailored to a specific domain.

In order to facilitate use by domain experts, the syntax of DSLs is often based on the conventions of domain experts. For example, a language for implementing contractual requirements in the creation of television program schedules may borrow from

the natural language used in contract texts [DHH⁺20]. Another example that originated outside the software development community are guitar tabulators shown in Listing 2.1⁵.

1	e		0	----	1	-----	3	-----	3	12	-----	1	---	3	--	0	-----							
2	B		1	-1	---	1	-1	----	0	0	3	-----	9	9	10	12	---	1	---	0	--	1	-----	
3	G		0	--	2	--	2	--	2	----	0	-----	9	---	9	-----	2	---	0	--	0	-----		
4	D		2	---	2	3	---	3	----	0	-----	9	-----	3	---	0	--	2	-----					
5	A		3	-----	3	-----	2	-----	1	1	-----	3	---	2	--	3	-----							
6	E		-----	1	-----	3	-----	1	2	0	-----	1	---	3	-----									

Tab

Listing 2.1: A guitar tab specifying how to play music on a guitar.

Each line of the tab represents a string of the guitar with the letter at the start of the line indicating the tuning of the string. The numbers indicate the fret in which a string must be pressed. Guitarists use these tabs to tell each other how to play a particular song. In their simplest form, tabs are machine-readable and can be used, for example, to create playable music files. However, in addition to the simple form shown here, there are countless special notations for notating other playing techniques. This example illustrates how a DSL can be oriented to the respective domain or is even (co-)defined by the domain experts themselves.

A special form of languages are tagging languages. Tagging languages “logically [add] information to the tagged DSL model while technically keeping the artifacts separated” [GLRR15]. In other words, tagging models refer to elements of an already existing model and extend it with additional information. Tagging languages can be used for, *e.g.*, connecting models to different middlewares [HKKR19], adding communication information to models [DJK⁺19], or adding extra-functional properties to models [MRRW16].

2.4 MontiCore

MontiCore [HR17, HKR21]⁶ is a *language workbench*, *i.e.*, it can be used to define DSLs and create the necessary infrastructure to process models of the language. This section gives only a brief and superficial introduction to MontiCore. For a more detailed description of how it works, please refer to [HR17, HKR21]. The infrastructure provided by MontiCore contains, *e.g.*, parsers, abstract syntax trees (ASTs), and symbol tables.

Fig. 2.2 gives an overview of code generators developed using MontiCore. Overall, MontiCore first reads in a (set of) models and creates an AST for them. Control scripts defined using Maven or Gradle provide additional information to MontiCore, such as in which folders to search for models. The AST created from the models can optionally be transformed into a representation that is easier to use in the following steps. In the

⁵The syntax was taken from UltimateGuitar.com. [Online]. Last accessed: 23.05.2021. https://www.ultimate-guitar.com/lessons/for_beginners/guitar_tabs_template.html

⁶This section on MontiCore is largely based on [HR17, HKR21]. The information in this section and parts of the text and figures have been adapted or taken from there.

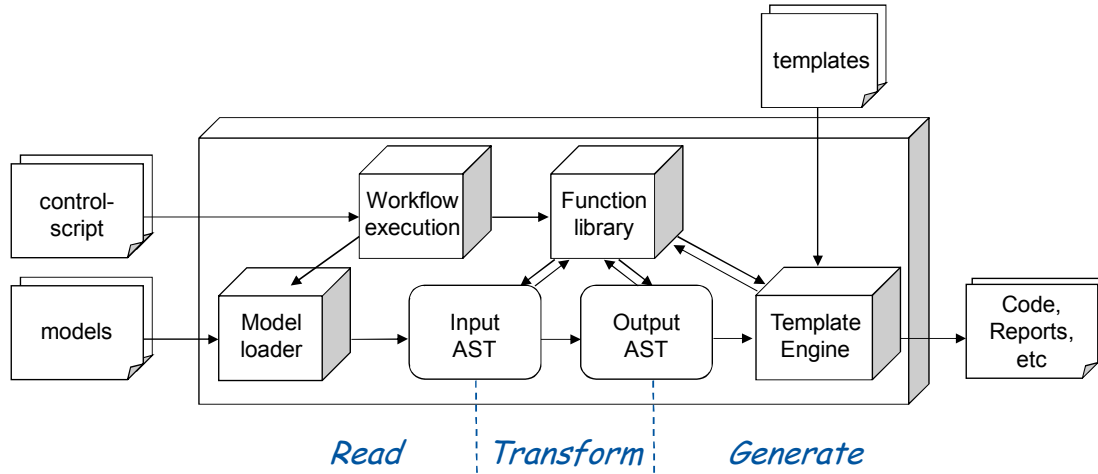


Figure 2.2: Architecture of (code) generators developed using MontiCore (adapted from [HR17, HKR21]).

last step, (code) templates can be used to create the desired output artifacts from the models. The remainder of this section goes into more detail about this process.

The key component of any language is the grammar. The grammar specifies the context-free syntax rules according to which the models of the language are formed. The syntax of the grammars itself is based on the extended Backus-Naur form (EBNF). Grammars mainly consist of terminals, *i.e.*, fixed character strings, and non-terminals, which are composed of other terminals and non-terminals. Similar to object-oriented languages, MontiCore offers interface non-terminals and allows extending non-terminals. Interface non-terminals can be used as placeholders that can be implemented by multiple non-terminals. Extending a non-terminal introduces an alternative for that non-terminal. Listing 2.2 demonstrates how to define the if-then-else construct known from many programming languages.

```

1 IfStatement implements MCStatement =
2   "if" "(" condition:Expression ")"
3     thenStatement:MCStatement
4     ("else" elseStatement:MCStatement)?
5   ;

```

MCG

Listing 2.2: A non-terminal defining an If-Then-Else statement (taken from MontiCore’s MCommonStatements).

IfStatement defines a non-terminal that implements the interface MCStatement. MCStatement is an interface non-terminal that acts as a placeholder for most other statements of the language. As the IfStatement implements MCStatement, it can

be used in all places, where an `MCStatement` can be present. The non-terminal definition starts with two terminals: `"if"` and `"("`. The fact that they are defined as two separate terminals allows the user to place an arbitrary number of spaces or line breaks between the terminals. This rule is enabled if a grammar extends `MCBasics`, which is the case for most grammars. The terminal for the opening parenthesis is followed by a reference to `Expression`. References to other non-terminals can be named, here using `condition:`, to resolve name clashes when the same non-terminal should be referenced multiple times within the same non-terminal definition. `Expression`, like `MCStatement`, is an interface non-terminal acting as a placeholder for possibly a large number of different expressions that could be used here in the model. After the terminal for the closing parenthesis, the `thenStatement` specifies the `MCStatement` acting as the *then* part of the if-then-else construct. Lastly, an optional *else* part concludes the non-terminal definition. The question mark declares the `else`-terminal and the `elseStatement` in the parenthesis is optional. For each non-terminal `X`, MontiCore generates a Java class called `ASTX`, where `X` is replaced by the non-terminal's name. This class offers the necessary methods to access the values of the parsed models.

As mentioned above, MontiCore's grammars are context-free. Context conditions (CoCos) enable further restricting the allowed models of the language. For every non-terminal `N` of grammar `G`, MontiCore generates a Java interface called `LASTXCOCO`, where `L` and `G` are replaced by the name of the non-terminal and grammar, respectively. These interfaces offer a `check` method that takes an object of the corresponding AST class as its only argument. MontiCore expects this method to be implemented by the language designers using hand-written Java code.

If model elements have a name that is referenced by other model elements, using the AST would be cumbersome. Symbol tables solve this problem by keeping a lookup table over AST objects. This allows the corresponding AST object to be found for a given name in a given scope. The details of this mechanism can be found in [HR17].

Overall, MontiCore generates a large amount of infrastructure for languages. However, since the generated code by its nature always looks very similar, there may be situations where the language developer is unsatisfied with the generated Java classes. To solve this problem, MontiCore uses the so-called *TOP mechanism*. The developer creates a class with the exact same name as the class that MontiCore would generate. MontiCore then discovers during the generation of the Java code that the class that should actually be generated already exists and generates a class with the postfix `TOP` instead. The handwritten class can then inherit from this `TOP`-class. This way it is possible for the developer to overwrite only the unwanted methods or add new methods without having to also have the code that can be generated by MontiCore in the handwritten class.

Once MontiCore has parsed the models, and optionally transformed the input AST, it can generate code, reports, or other artifacts from the models. To do so, the developers have to provide templates that specify how MontiCore's internal representation of the model can be transformed into the desired output artifacts. MontiCore uses Apache

Freemarker⁷ for this generation step. Developers can provide Freemarker templates to MontiCore that will be used to generate the output artifacts. In this regard, MontiCore offers a *template controller* infrastructure that can help developers to, *e.g.*, replace specific templates with other templates or provide arguments to templates.

2.5 Software Architecture and Architecture Description Languages

The more complex software systems are, the higher the potential savings that can be achieved by reusing already existing software. Parallel to the increase in the level of abstraction, new languages have also always increased the degree of reuse over the last decades. After procedural programming languages enabled the reuse of individual code blocks in the form of functions, object-oriented languages offer the possibility of bundling data structures and functionality into classes and thus making them reusable as a whole. With the steadily increasing complexity of software systems, the need for reuse is also increasing and the focus is gradually shifting towards architecture. On the architecture level, components, which usually consist of many classes, are wired together in a way that the desired functionality is achieved. According to [Sie04], the following definition of *software architecture* is widely accepted:

Definition 5 (Software Architecture). “*The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.*” [BCK98]⁸

In this regard, *component* shall informally be defined as follows:

Definition 6 (Component). “*A [component] is a physical encapsulation of related services according to a published specification.*” [Bro10]

In summary, architectures describe systems by interconnecting abstract functional blocks (the components). The description of these architectures can be done using models, which are created with so-called ADLs. A special style of ADLs are C&C ADLs.

⁷Project website. [Online]. Last checked: 23.05.2021. <https://freemarker.apache.org/>

⁸In later editions of the book, the authors chose a more abstract definition: “The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.” [BCK12]

In a broader sense, C&C is also a modelling paradigm that can be used independently of ADLs [Kus21]. C&C architectures describe systems using components that exchange data via ports connected through connectors. Hierarchically composed components can thus provide their functionality by reusing other components and connecting their ports with said connectors. Given that software projects and the expectations of their stakeholders are sometimes very different, “it is clear that an ideal and general purpose [ADL] is not likely to exist” [MLM⁺13]. Accordingly, over the years, a variety of ADLs have evolved, such as AADL [FGH06], Acme [GMW10], ROOM [SGME92, Sel96], and MontiArc [Hab16, HRR12]. More extensive surveys and comparisons of ADLs can be found, *e.g.*, in [MT00, Cle96].

With regard to the context of this thesis, there are also numerous domain-specific ADLs focused specifically on the development of IoT applications. Prominent examples include ThingML [HFMH16, MHF17], Calvin (including its extension Kappa) [AP17, PA15, PA17], FRASAD [NTBG15], and CapeCode [BJK⁺18]. As outlined in Sec. 2.1, the development of IoT systems presents developers with numerous tasks, such as accessing sensors or providing communication, that arise from the inherent characteristics of IoT systems. IoT-focused ADLs usually try to separate the different concerns of developing IoT applications and provide RTEs offering common functionality such as communication between components. Abstracting from these problems and handling them in a standardized way is especially important since average web developers are believed not to be “well equipped to cope with the challenges of IoT system development” [TM17a]. Some model-driven frameworks for the development of IoT applications also focus on particular challenges such as integrating machine learning (*ThingML+* [MRG18]), self-adaption (*MDE4IoT* [CS16] and *SysML4IoT* [HLR17]), or logging [MF19]. A more extensive survey on MDE in the context of IoT applications can be found in [WMW18].

Overall, there is an emerging trend in architecture development, encouraged by cloud providers, to make parts of a system available in an independently deployable form.

Definition 7 (Microservice). “*Microservices are small applications with a single responsibility that can be deployed, scaled, and tested independently*” [LSCPE18]

It is worth emphasizing that microservices need to be distinguished from components. Both components and microservices have the goal of making (partial) functionalities of a software maintainable and reusable independently of each other. Unlike microservices, however, components are not necessarily deployed in an independent manner. This independent deployability allows the microservices to scale. Cloud providers like AWS and Microsoft Azure offer their customers virtually unlimited resources—provided there is sufficient funding. By using microservices, much-requested application parts can be dynamically instantiated to match the current user traffic. This independent deployability

is usually achieved through containers. As Docker is arguably one of the most widely used container technologies and will also be used in this thesis, we adopt their definitions of *containers* and *container images*:

Definition 8 (Container). “A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.” [wwwa]

Definition 9 (Container Image). “A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.” [wwwa]

In other words, an image is a template that can be used to create the actual executed containers; similar to the relationship between classes and objects instantiated from them in object-oriented programming. Future IoT applications are expected to be built on such container technologies [TM17a]. In this regard, projects such as *Balena*⁹ (formerly *Resin*) already offer (Docker) container engines and other infrastructure tailored to the needs of IoT projects.

2.6 MontiArc

MontiArc¹⁰ is a MontiCore-based ADL for the simulation [Hab16] and verification [vW20, KPRR20] of distributed systems. MontiArc describes systems as C&C architectures and has been applied in numerous domains including robotics [ABH⁺17], automotive [BMR⁺17], and avionics [KRRS19]. At its core, MontiArc thus uses components whose ports are connected via connectors to specify the data flow of a system. MontiArc’s ports are typed and directed, *i.e.*, they are either incoming or outgoing. The types used by ports can either be primitive data types (such as Boolean, String, or Integer) or be more complex types defined in class diagrams. The necessary class diagrams for this can be modeled using MontiCore’s class diagrams for analysis (CD4A)

⁹Balena Website. [Online]. Last accessed: 03. June 2021. Available: <https://www.balena.io/>

¹⁰The MontiArc tool infrastructure [HRR12, RRW13, Hab16, Wor16, BKRW17a] has been described in many previously published articles. The description in this section is in part based on the descriptions in [KMR⁺20b, KRSW22].

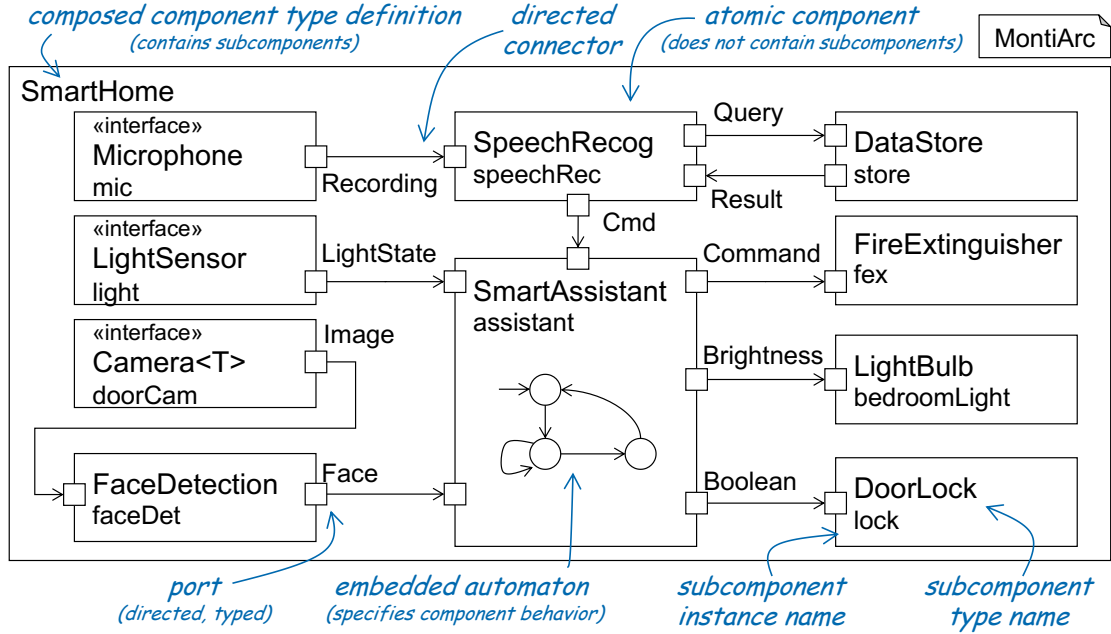


Figure 2.3: An example for MontiArc’s graphical syntax (taken from [KMR⁺20b]).

language¹¹. One of the main outputs of MontiArc is generated Java code that can be used to simulate the specified architecture.

Fig. 2.3 gives an overview of MontiArc’s graphical syntax taken from [KMR⁺20b]. This exemplary smart home component consists of a `SmartHome` component, that contains instances of other components as subcomponents. Components that define their behavior by instantiating and connecting subcomponents are called *composed components*. In contrast, *atomic components* do not contain subcomponents and use other means of defining their behavior, *e.g.*, automata. To avoid name conflicts, subcomponents can have instance names. If no explicit instance name is defined, the instance name is automatically set to the component’s type name with a lowercase first letter. Component types can also have generic parameters. For example, the `Camera` component has a generic parameter `T`. Similar to Java, generic parameters can be used to enable a more flexible reuse of components in other contexts. Interface components [Wor16] provide means of replacing the implementation of components. This is similar to the interfaces from Java that can be implemented by multiple classes. Shortly before code generation, interface components are replaced by regular components. The exact replacement is defined by *bindings*, *i.e.*, configuration files [Wor16]. This can be used, *e.g.*, to support

¹¹CD4Analysis Project website. [Online]. Last accessed: 06. June 2021. Available: <https://github.com/MontiCore/cd4analysis>.


```

1 component SmartHome {
2   Microphone mic;
3   SpeechRecog speechRec;
4   // ... more subcomponents ...
5
6   mic.record -> speechRec.voiceRecording;
7   // ... more connectors ...
8 }
9
10 component Microphone {
11   port out Recording record;
12
13   // ... behavior specification ...
14 }

```

MA

Listing 2.3: Excerpt of the textual representation of the MontiArc architecture in Fig. 2.3.

different technical target platforms [Wor16]. In addition to the graphical syntax, MontiArc also offers a textual syntax. In fact, the graphical syntax exists only for improving understandability but only the textual syntax is used to write the models read in by MontiArc. Listing 2.3 shows an excerpt of the textual representation¹² of the example from Fig. 2.3.

MontiArc’s semantics is based on the FOCUS calculus [BS01]. The FOCUS calculus [BS01, Bro10, RR11] understands components as stream processing functions.

Definition 10 (Stream). “A stream is a finite or infinite sequence of elements of a given set.” [Bro10].

Streams describe the communication history of components [Bro10]. In MontiArc this means that the history of messages exchanged between two ports, *i.e.*, connectors, can be described using streams [Hab16]. Streams exist in both untimed and timed variants [BS01, Bro10], as well as continuous and dense time [Bro01]. Timed streams include not only the messages exchanged but also *ticks* (denoted as “ $\sqrt{}$ ”). Ticks are special messages that represent time progress. A special case of timed streams are time-synchronous streams in which there is exactly one message between two ticks. The special message \perp denotes the absence of a message. The stream processing functions that MontiArc’s com-

¹²The concrete textual syntax has been changed since its original definition in [Hab16]. This thesis uses the (stable) syntax definition of MontiArc 7.

ponents represent map sets of (incoming) streams to sets of (outgoing) streams. More extensive introductions to the FOCUS calculus can be found in [BS01, RR11]. The advantage of basing MontiArc on this formal foundation is that it has enabled analyses such as semantic differencing [BKRW17b] and refinement analysis [RW18].

Over the past few years, numerous extensions for MontiArc have been developed, including an automaton behavior specification [Wor16, BKRW17a], support for dynamic reconfiguration [HKR⁺16], and variability [BEK⁺18]. A notable MontiArc-variant called EmbeddedMontiArc (EMA) focuses on modeling embedded systems [KRRvW18] and especially the simulation of cooperative driving [FIK⁺18, KKRZ19]. EMA also includes the necessary modeling techniques to define the behavior of components using deep learning [GKR19]. This thesis is mainly based on the standard MontiArc but borrows some concepts such as tagging ports with communication technology [HKKR19] from MontiArc’s extensions, *e.g.*, EmbeddedMontiArc [KRRvW18, KRSvW18, KPRS19], where appropriate.

Chapter 3

Scope of the Thesis

This chapter gives an overview of the scope of this thesis, *i.e.*, its assumptions, research questions, requirements, methodic, and more. We expect interested readers to come back later to this chapter to lookup certain details, *e.g.*, a certain assumption, while reading the remainder of this thesis. To make such lookups more efficient, this chapter defines identifiers (such as “**(RQ1)**”) that are used throughout this thesis to refer to parts of this chapter. Sec. 3.1, Sec. 3.2, and Sec. 3.8 give a high-level overview of this thesis.

3.1 Vision and Assumptions

Today, IoT applications often strongly couple hardware and software products [BKK⁺22]. Manufacturers usually sell hardware for specific software that is tailored to it. The hardware can then only be used with the hardware manufacturer’s software. Conversely, this also makes the hardware dependent on the software. Manufacturers are able to render hardware that has already been sold unusable by disabling associated software functionality. They thus have far-reaching leverage to persuade customers to pay money for the hardware they have already paid for, *e.g.*, in form of a subscription. In other cases, manufacturers also decide to make hardware unusable without any alternative options for the user because it no longer fits their own business strategies¹.

In such situations, users are at the mercy of individual manufacturers and functioning hardware becomes electronic waste, which is neither desirable from a sustainability nor a customer perspective. Like [AMMK19], for the future, we, therefore, envision that hardware and software can be sold more independently of each other. Installing new software on devices with operational hardware could avoid electronic waste for devices that would otherwise have to be recycled once the software becomes obsolete due to the tight coupling of hardware and software [DSF21]. We believe that a few platforms will emerge that enable IoT software to be sold in a kind of *app store* independent of the hardware. A similar trend has already occurred in the smartphone market, where

¹Klint Finley. *Nest’s Hub Shutdown Proves You’re Crazy to Buy Into the Internet of Things*. In *Wired.com*. [Online]. Last accessed: 11.06.2021. Available: <https://www.wired.com/2016/04/nests-hub-shutdown-proves-youre-crazy-buy-internet-things/>.

various hardware manufacturers sell smartphones for the Android operating system and software manufacturers that are independent of them then sell apps that are largely independent of the hardware [BKK⁺22]. First standardization efforts such as Apple HomeKit or Matter² indicate that a similar movement as in the smartphone market will also take place for the smart homes sector (as a special case of the IoT). The vision of such a marketplace is shared by, *e.g.*, [BSS⁺17], [MM12] and [AMMK19], which propose architectures for such a marketplace. Similar to the smartphone market, the authors of [BSS⁺17] expect multiple marketplaces to emerge.

Such a decoupling of hardware and software also has significant implications for software development. While software today is strongly tailored to specific hardware, a higher level of abstraction will be necessary in the future to achieve this decoupling. As stated in Chapter 2, model-based or -driven software development is a promising way to achieve such an abstraction. Unsurprisingly, there are already several model-based and -driven frameworks focused on IoT application development, *e.g.*, ThingML [HFMH16, MHF17], Calvin [AP17, PA15, PA17], and FRASAD [NTBG15]. However, these are usually limited to the specification of components, the generation of code, and the deployment of code. For these largely solved issues, this thesis will, thus, be inspired by existing research results. Challenges that arise later in the lifecycle of IoT applications (*cf.* Sec. 3.2), such as reliability in the context of unreliable hardware or the connection with digital twins, are mostly disregarded by existing work and left to the developer.

A key component of our vision is the integration of IoT applications into information systems. With the help of information systems, users can view data, be actively notified about particular data by the information system, and influence the (IoT) application. We follow a holistic model-driven approach, in which the information system is also developed using model-driven methodologies. However, this methodology for the development of information systems is not part of this work. We rely on the *MontiGem* framework for the model-driven development of information systems. *MontiGem* is further described, *e.g.*, in [AMN⁺20, GHK⁺20, ANV⁺18]. In this work, only the integration with *MontiGem* is discussed where applicable.

The main focus of this work is the (model-driven) development of IoT applications on the application layer. As outlined in Sec. 2.1, the development of IoT systems comes with a variety of challenges in a wide range of areas. Therefore, we will make assumptions in this work through which it is possible to abstract from some technical and network aspects of development for very resource-constrained systems. With regard to the rapidly increasing computing power of computers, we assume that these assumptions can be met by many IoT devices in the near future. Many popular single-board computers that are also frequently used in IoT applications, such as the Raspberry Pi, already meet

²Matter Project Website. [Online]. Last accessed: 11.06.2021. Available: <https://buildwithmatter.com/>

these requirements today. As a result of this vision, we formulate a set of technical assumptions (TAs):

- (TA1) Containerization.** *IoT devices shall have the capability to execute (Docker) containers.* [KKR⁺22a, BKK⁺22] Taivalsaari and Mikkonen [TM17a] expect future IoT applications to be built on container technologies. We think this vision is realistic, as the 2022 IoT & Edge Developer Survey [Ecl22] by the Eclipse foundation states that container images (49 %), virtual images (31 %), native binary (27 %), and script files (22 %) are the “top edge computing artifacts deployed for IoT solutions”, with a notable increase in the percentage of container images compared to 2021 (30 %). Because of the high adoption rate in cloud applications and the associated high chance of being widely used in future IoT applications as well, we chose Docker containers in this work. As mentioned in Sec. 2.5, companies like Balena already offer IoT-focused Docker engines for this purpose. This assumption helps us abstract away from the technical details of copying software to IoT devices when deploying applications. For research focused on more technical aspects of deploying IoT applications, we refer interested readers to the research of the GeneSIS project [FN19, FNS⁺19, FNS⁺20].
- (TA2) Linux.** *IoT devices shall have a Linux-based operating system.* [KKR⁺22a] With respect to assuming Docker support, we continue to assume that IoT devices use Linux as their operating system. According to the Eclipse foundation’s 2022 IoT & Edge Developer Survey [Ecl22], Linux is the most used operating system (43 %) for constrained devices, followed by FreeRTOS (22 %), not using an operating system (19 %), and Mbed OS (10 %). Considering that this thesis is essentially presenting a research prototype and that support for multiple operating systems is mainly interesting from a product perspective than from a research perspective, we believe this assumption is justified.
- (TA3) Internet access.** *IoT devices shall have Internet access.* [KKR⁺22a] In some IoT applications, not all IoT devices have direct Internet access. Instead, these devices are only capable of local communication, for example via Bluetooth. In these cases, so-called gateways are used that communicate locally with the IoT devices and enable the exchange of information between the devices and the Internet [TM17a, TM18]. Since this work does not focus on the network aspects of IoT applications, we abstract from this problem. We believe that this assumption is realistic as new network technologies such as LTE-M, NB-IoT, and 5G³ provide Internet access to more and more devices. This belief is shared by [TM17b], which states that “next generation network technologies [...] will allow direct, energy-efficient data transfer from devices to the cloud, thus bypassing the need for gateway devices”.

³A comparison of these technologies can be found, *e.g.*, in [KADAS19]

(TA4) C++11 Support. *IoT devices shall support the C++11 standard and its standard library.* The Eclipse foundation’s 2022 IoT & Edge Developer Survey [Ecl22] states that C/C++ are the most popular programming languages for constrained devices. Therefore, the tool presented here, MontiThings, also uses C++. As a compromise between a certain up-to-dateness and support for older devices, we have chosen the C++11 standard. In particular, we also assume that the standard library is supported. This is not always the case for very low-power devices, *e.g.*, Arduino Uno. In light of the fact that we are targeting future more powerful devices and assume support for container technologies, we consider this assumption to be justified. Especially for devices running Linux, this is normally the case.

Overall, these assumptions are comparable to the requirements for devices executing the AWS IoT Greengrass Core of AWS’s IoT Greengrass Service for deploying IoT applications to devices⁴.

3.2 Lifecycle and Development Process of IoT Applications

The lifecycle of IoT applications can be described using a variant of the MAPE-K [KC03] loop. The MAPE-K loop describes the autonomic system elements using the phases **monitor**, **analyze**, **plan**, and **execute**, which share common **knowledge** [KC03]. In [TGPH20], the IoT “data life cycle” is described using a similar loop with the phases “Capture, Communicate, Analyse and Act (C2A2)”. Capture and communicate refer to the generation and forwarding of (sensor) data. The analyze and act phases evaluate the data and initiate any necessary actions. In contrast to MAPE-K, however, the C2A2 loop has a more technical focus, where the authors also assign a layer of the architecture to each phase.

In contrast to this data-focused lifecycle, [ROL18] presents separate lifecycles for the IoT devices, services, and applications. Their device lifecycle consists of the following phases: (Re)construction, Production, Installation and commissioning, Update, Operation, Decommissioning [ROL18]. This lifecycle includes hardware-specific phases such as the production of the devices, which are not considered in this work. In comparison, their IoT services/application lifecycle takes a more software-oriented view, consisting of the following phases: (Re)construction, Deployment, Execution, Reconfiguration, and Termination [ROL18]. A whitepaper by Lantronix takes a similar but less detailed approach and identifies the following phases: Design, Deploy, Manage, Decommission [Lan]. In conclusion, many lifecycles for IoT systems separate development from deployment and include an execution phase and an end-of-life phase.

⁴Setting up AWS IoT Greengrass core devices. [Online]. Last accessed: 25.08.2021. Available: <https://docs.aws.amazon.com/greengrass/v2/developerguide/setting-up.html#greengrass-v2-requirements>

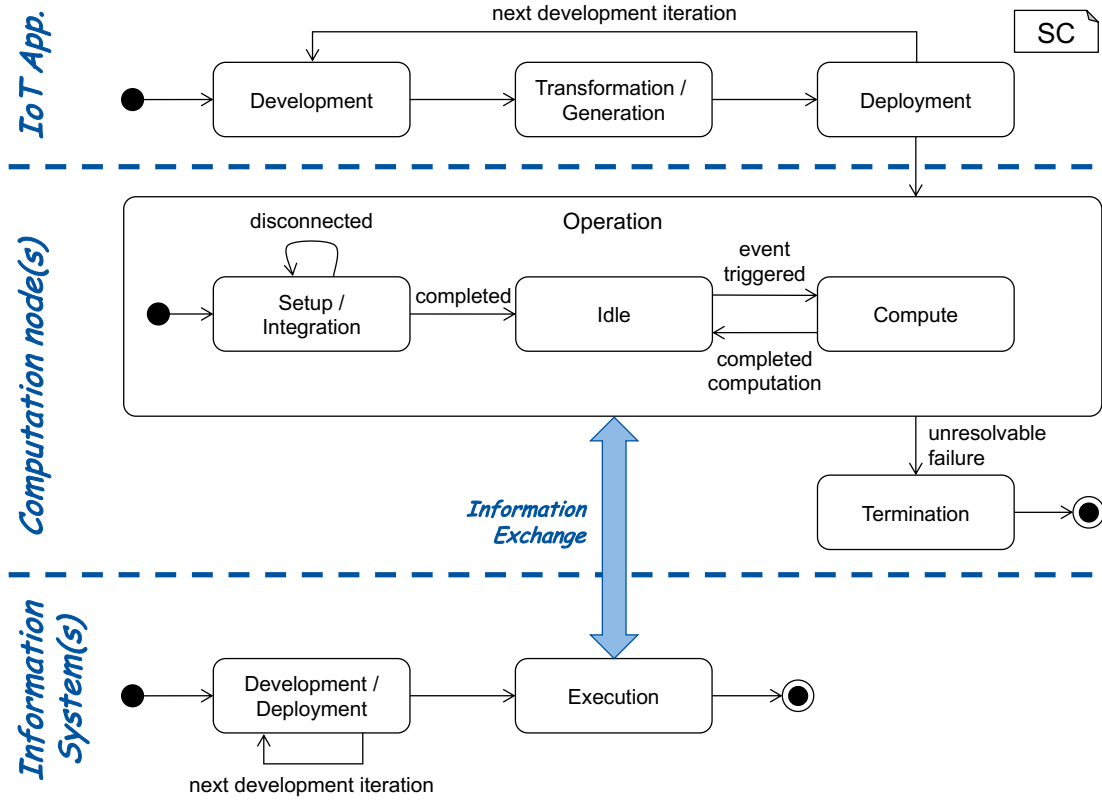


Figure 3.1: Lifecycle of model-driven IoT systems.

Fig. 3.1 presents the lifecycle for model-driven IoT systems which we are going to follow in this thesis. It is conceptually based on the already existing lifecycles above, but tailored to model-driven development. The lifecycle separates three different concerns: 1) the development and deployment of the IoT application 2) the operation of the IoT devices 3) the operation of related information systems.

The life of IoT applications starts with a development phase. During this phase, developers, after eliciting requirements in collaboration with customers, create models and code that specify the system. The models created in this phase can then be used in the next step to generate code. In some cases it makes sense to transform the models before generation, e.g. to add additional functionality or to combine different models. In the final step, the applications are deployed to the computation nodes. Computation nodes in this context refer to instances of (parts of) the generated code. Often, the generated code will be executed by IoT devices but IoT systems may also utilize, *e.g.*, cloud systems. This deployment is not necessarily performed by the same developers who created the application. If a deployment has special requirements, these can be

better specified by people who know the environment in which the application is to be executed better than the developers. For example, in smart building management, it may be desired not to provide smoke detector functionality in a particular room such as the kitchen to prevent false-positive alarms. This is in accordance with [Zam17], which distinguishes between *global manager* and *local manager*.

The computation nodes start their operation with a setup and integration phase. During this phase, primarily, connections to communication partners are established. In the following, the devices process incoming messages in an event-based manner: Initially, the devices are idle. Whenever an event is triggered, the device processes this event and switches back to idle mode. The assumption that devices use event-based processing is based on the fact that many popular model-driven frameworks such as ThingML [HFMH16, MHF17] and Calvin [AP17, PA15, PA17], as well as IoT-focused programming languages like Eclipse Mita [wwwb], use event-based processing. *Event* in this context is not further specified can refer to a variety of different situations including, *e.g.*, reception of a message, expiration of a (potentially periodic) timer, or a new sensor measurement.

Following the “IoT systems never sleep” characteristic from [TM17a] (*cf.* Sec. 2.1), individual computation nodes can reach a termination phase, but there is no such phase on the level of the IoT application as a whole. Instead, an IoT application will be implicitly terminated, once all its computing nodes are terminated. In our software-oriented lifecycle, the termination phase can be reached in cases of unresolvable failures. However, given the circumstances in which IoT devices are used, external influences can also remove computation nodes from the system. This includes, for example, when the battery of a battery-powered device can no longer supply power or (in extreme cases) when a device is physically destroyed. In these cases, however, the devices usually do not go through a coordinated termination phase but are just removed from the system. Similar to the separation between the lifecycle of the IoT devices and the IoT services in [ROL18], this lifecycle does not make any claims about the lifecycle of the IoT devices. Especially, the physical IoT devices may also be deployed before the software is deployed.

In parallel with the IoT application, our lifecycle includes the operation of one or more information systems. These information systems can, *e.g.*, be traditional web-based dashboards that present the IoT devices’ sensor data to users. The information system may also process data and may, especially, also include digital twins or control part(s) of the computation nodes. Their lifecycle is not discussed in detail in this thesis, as the (model-driven) development of information systems is not part of this thesis. It is, however, important to notice that the lifecycle of these information systems is independent of the lifecycle of the IoT application. This implies that digital parts of the IoT system may exist before the IoT devices are deployed, or that the information system continues to use the IoT systems data even after the last IoT device has already been removed from the system.

3.3 What Do IoT Projects Need?

According to an analysis of the 30 most-starred IoT projects on GitHub, “IoT projects [...] exhibit a more balanced distribution of primary languages, with the most popular languages being C, C++, Java, Python, and JavaScript” [CDRS20]. The 2020 IoT Developer Survey by the Eclipse Foundation supports this claiming that C, C++, Java, Python, and JavaScript “dominate the IoT space” [Ecl20], where C and C++ are popular on constrained devices while Java and Python are popular on gateway nodes and in the cloud. Further, the survey found that “HTTP/ HTTPS (51 %), MQTT (41 %), and TCP/IP (33 %)” are the leading communication protocols [Ecl20].

One of the implications of a 2013 survey among 48 practitioners [MLM⁺13] was that ADLs need to be both easy enough to understand so that they can be used for communication with stakeholders and formal enough to use them for automated tasks. Another implication was that practitioners “do not require technical features for verification, code generation or the like”. Based on the answers to different questions the authors state that “The need for code generation seems to be case- or project-specific”. Since we do not know all of the respondents’ specific answers, we can only speculate about the reasons for this perceived lack of need for code generation. The authors cite one respondent who states that “architecture description [is] on a too high level”. Based on this and the fact the respondents mostly used general-purpose ADLs like architecture analysis & design language (AADL) [FGH06] and ArchiMate [Lt04], we assume this perceived lack of need originates from a lack of experience with useful generators. The success of projects like MATLAB Simulink [Mat21] shows that there is a need for code generation. We believe the heterogeneous and communicative nature of IoT projects will increase the need for code generation in IoT projects.

Furthermore, a 2020 survey among 433 practitioners [RLC⁺20] found that “reliability, availability, performance, scalability, and security” are the most relevant quality attributes for IoT applications. Comparing this survey to the existing IoT ADLs like ThingML [HFMH16, MHF17] and Calvin [AP17, PA15, PA17], we especially see a need for further taking reliability into account when designing components. While formal ADLs like MontiArc often focus on analyses, reliability has not been a focus of IoT-focused ADLs.

3.4 Challenges

There are several challenges to consider when modeling and developing IoT systems. We distinguish here between technical challenges, *i.e.*, challenges that arise directly from the hardware used, and modeling challenges, *i.e.*, challenges that complicate the modeling of these systems. First, we address technical challenges:

- (TC1) **Heterogeneous platforms** [BKK⁺22, KKR⁺22a, KMR⁺20b]. The devices used in an IoT system are often very heterogeneous [AP17, HFMH16]. Some of the devices have only very limited computing power, while others offer almost unlimited performance as a cloud system. In addition, the available hardware also differs considerably. Not every sensor and every actuator is available on every device. And even between sensors of the same type from different manufacturers, for example two different weight sensors, there can be major differences. This heterogeneity complicates software development in that not every software component can be deployed on every device and the same software functionality may need to be adapted if it is to be executed by different devices.
- (TC2) **Intermittent connectivity** [KRSW22, KMR21]. IoT devices are very often operated via wireless networks. The 2022 IoT & Edge Developer Survey [Ecl22] by the Eclipse foundation names WiFi (36 %), Ethernet (29 %), Cellular (LTE, 4G, 5G, etc) (22 %), and Bluetooth/Bluetooth Smart (20 %) as most used connectivity protocols. In wireless networks, connectivity can be affected by a variety of factors. In addition to the conventional radio communication problems such as attenuation, reflection, refraction, or crosstalk, the mobility of IoT devices can also cause the network to be abandoned under certain circumstances. A simple example of this mobility issue is a car connected to the Internet driving through a dead zone. Consequently, “many devices may have intermittent connectivity and may thus be unreachable or offline for extended periods of time” [TM17b].
- (TC3) **Unreliable hardware** [KRSW22, KMR21]. IoT devices perceive and interact with their environment. The sensors and actuators required for this often show signs of wear over time. Therefore, sensors may provide incorrect values [MNZC20, KMMN16, FG08]. Errors can appear, *e.g.*, in the form of a constant offset, drift, temporary failure to provide values, trimmed values (outside a specified interval), outliers, or noise [JCO17]. In addition, IoT devices can also fail completely. Reasons for this can be a flat battery or an interrupted network connection. Since IoT systems often rely on the interaction of many devices, a permanent failure of the network connection effectively means that the device cannot continue to perform its service. In conclusion, it cannot be safely assumed that the installed hardware will always function correctly.

Besides these technical challenges, there are also some modeling challenges that affect the modeling of IoT systems:

- (MC1) **Separation of concerns** [KRSW22]. While a sensible separation of concerns is of course necessary in any software development project, this challenge becomes even more important in IoT projects. Typically, an IoT project involves stakeholders with different educational backgrounds and goals. Besides software developers, IoT projects often involve domain experts, user experience experts, and

hardware developers. Even within software development, individual developers have different areas of expertise. For example, some developers may be primarily concerned with the user interface, while others are involved with the low-level development of embedded systems. Because the various experts usually have only limited knowledge of each other’s areas, a sensible separation of concerns is necessary.

- (MC2) **Behavior specification.** IoT projects often use the same programming languages that are also widely used for classical software projects [TM17a] although they differ from classic projects in many aspects. Using languages that are not designed for use in the IoT context, consequently, sometimes results in hard-to-understand or unmaintainable code. As an example, Taivalsaari and Mikko-nen mention the “callback hell” that occurs when many asynchronous event-handling JavaScript calls are connected [TM17a]. The description of behavior, often driven by external events, timeouts, and error handling, requires new programming and modeling languages.
- (MC3) **Structural dynamics [KKR⁺22a].** In many cases, IoT systems are not designed (or sold) as complete systems. Instead, the system must be able to handle new hardware being added to the system at runtime. A prominent example of this are smart home systems where users rarely equip a house completely in one step. Instead, users are progressively buying new devices such as smart lights or speakers that expand the overall functionality of the smart home. In contrast, devices can of course also break and thus fall out of the system (*cf.* (TC3)). This means that at the time of initial deployment, the hardware of the system has not yet been finalized. When modeling IoT systems, these dynamics must be taken into account in the form of dynamic changes of the component structure and connectivity.

3.5 Research Questions

This section outlines the research questions addressed in this thesis. The first two research questions serve as a mantle for all other research questions in this thesis:

- (RQ1) *How to specify C&C-based IoT applications?*

This rather general question has already been addressed by various related works. However, these works usually focus only on a specific aspect (*e.g.*, Calvin [AP17, PA15, PA17] on deployment) or neglect later phases of the development cycle of IoT applications. This work, in contrast, provides a framework that addresses later phases of the lifecycle in particular. For the earlier development phases, this work builds heavily on existing insights from other work.

(RQ2) *How to deploy C&C-based IoT applications to heterogeneous targets?*

(RQ3) *How to connect C&C-based IoT applications to target devices?*

In order to bring software to the devices of an IoT system, it must first be decided which devices execute which parts of the software. Depending on the target system, the software must be adapted to the respective target system. The fact that there may be a discrepancy between the user's requirements and the available hardware also needs to be taken into account when deploying the software. It may also be necessary to make changes after the initial deployment, for example, when new devices are added to the system or devices are removed from the system. **(RQ2)** addresses these deployment challenges. Nevertheless, low-level technical challenges related to deployment (*e.g.*, copying scripts to IoT devices) are out of scope here **(TA1)**. In our vision, IoT devices will be sold in the future independent of the particular software components they execute. Accordingly, in a second step after deployment, software components must connect to the devices they are deployed on. In particular, it also implies that the specific sensors and actuators and the code that accesses them cannot be assumed to be known at design time. **(RQ3)** examines how a connection to such independent IoT devices can be achieved.

(RQ4) *How to reliably address malfunctioning hardware in C&C-based IoT applications?*

Since it cannot be assumed that the hardware will always function as expected **(TC3)**, an IoT application must be able to detect and deal with such deviations from the expected behavior. In naive approaches, excessive error handling leads to a situation where the business logic is difficult to follow [TM17a]. Therefore, a sensible separation of concerns must be ensured **(MC1)**.

(RQ5) *How to handle device failures in C&C-based IoT applications?*

In more severe cases of error, hardware may not only behave incorrectly, but devices may fail completely. However, the failure of individual IoT devices should not lead to the failure of an entire IoT system if the functionality can be provided by other devices. It must be taken into account that devices can fail either temporarily (*e.g.*, in the event of a temporary network failure) or permanently (*e.g.*, in the event of a hardware defect). This question deals with how these types of errors can be addressed in an automated way.

(RQ6) *How to integrate C&C-based IoT applications with accompanying model-driven information systems?*

(RQ7) *How to connect C&C-based IoT applications to their digital twins?*

IoT systems are often accompanied by information systems that are used, among other things, to display information about the system to users, or to let users influence the IoT system. If developers want to connect IoT devices with classic cloud systems, it is usually necessary to include a library from the cloud provider. At various points in the code, information is sent to the cloud or accepted by the cloud with this library. Due to the fact that C&C architectures use clearly defined communication interfaces, there is the potential to automate parts of this. **(RQ7)** explores how C&C-based IoT systems can be interconnected while reducing manual efforts. Digital twins are an increasingly important part of such information systems. In abstract terms, digital twins provide “an always in sync digital model” [TJSW18] of the real devices. Thus, if the state of the real system changes, the state of the digital twin changes likewise and vice versa.

(RQ8) *How to monitor and analyze the execution of C&C-based IoT applications?*

Over the last few years, iterative development processes have gained substantial popularity [HSG18]. When developing IoT systems iteratively, it is necessary to be able to observe the behavior of prototypes of the system to be developed. However, reproducing and simulating error situations under laboratory conditions is difficult. The reason for this is that IoT systems can be affected by a variety of external influences. Some of these, such as a network failure, are not necessarily visible in models that abstract from such details and are thus hard to detect. This question, therefore, investigates how (prototypes of) IoT systems can be observed and analyzed.

3.6 Requirements

Following the challenges in designing a model-driven IoT framework, this section names concrete requirements for designing model-driven IoT frameworks. For each requirement, we also provide the rationale for why the requirement exists.

- (R1) Integration of sensors/actuators [KRSW22].** Sensors and actuators are an integral part of IoT applications. Accordingly, a modeling framework for IoT applications must provide a way to incorporate such hardware.
- (R2) Executable behavior definition.** As with non-IoT applications, it must be possible to specify the behavior of the application. In the case of IoT applications, the behavior usually consists of the interaction of many different devices and components, which together result in the behavior of the application.
- (R3) Integration of previously existing as well as handwritten code [KRSW22].** In most cases, controlling sensors and actuators requires the use of a sensor-/actuator-specific library. While more primitive sensors may only apply a value to a digital

input pin, more integrated sensors can require compliance with complex communication protocols. The manufacturers usually offer libraries for accessing their hardware. To be practically usable, IoT frameworks must enable such libraries to be integrated into the applications. These libraries are often only made available in a specific programming language. To avoid being limited in the choice of sensors, the IoT framework must therefore offer the possibility of integrating libraries from different programming languages.

- (R4) **Separation of logical and technical architecture [KRSW22].** IoT frameworks should separate the logical aspects of the specification from the technical aspects (**MC1**). This increases the reusability of the models, as IoT applications are used on heterogeneous platforms (**TC1**).
- (R5) **Error handling [KRSW22].** Alas, sensors sometimes provide incorrect measurements (**TC3**). IoT frameworks should foster handling erroneous inputs and behavior. This reduces the risk of an error propagating through large parts of the system and then leading to errors in other parts of the system that are difficult to track and may be more serious.
- (R6) **Partitioning of software components.** Even though the logical architecture abstracts from the concrete IoT devices, an IoT application must ultimately be executed on IoT devices. Therefore, it is important that a code generator in a model-driven IoT framework does not produce a single monolithic executable, but rather a composite of multiple executables. While we assume in this thesis that the devices used have some performance capability (*cf.* (**TA1**), (**TA2**)), care must be taken not to split an application too small in order to reduce the overhead associated with the split.
- (R7) **Rule-based deployment [KKR⁺22a].** Before the application can be executed, the corresponding application parts must be deployed to the various target devices. Since IoT devices “are more like cattle that must be managed en masse instead [of] receiving personal attention and care” [TM17a], such a deployment cannot be assumed to be done manually. Instead, the system must be provided a set of rules specifying which parts of the software shall be executed by which devices (*cf.* [AP17]).
- (R8) **Reacting to deployment changes [KKR⁺22a].** In contrast to more traditional systems, *e.g.*, web servers, the hardware of IoT systems is often not final at the time of initial deployment. For example, consider a smart home application. Many users do not equip their entire home with IoT devices in a single purchase. Instead, only a handful of devices are installed at first and then later supplemented by more and more devices. To support such hardware growth, the IoT framework must be able to respond to a change in available hardware. Conversely, the equipment can

also fail. Again, the IoT framework must be able to handle the change in available hardware.

- (R9) Failure handling [KKR⁺22a, KRSW22].** If there are dependencies among the IoT devices, the failure of a device cannot simply be accepted. Instead, the IoT framework must offer appropriate strategies to adequately respond to failure. A distinction must be made here between temporary failures and permanent failures. Particularly in the case of mobile devices, temporary failures can also be caused by cellular dead spots that can cut off IoT devices from the network. Short-term temporary failures can usually be bridged by caching. In contrast, permanent failures may require other mechanisms.
- (R10) Automatic connection to digital twins [KMR⁺20b].** Digital twins have taken on an increasingly important role in IoT projects in recent years. Digital twins offer, in abstract terms, “an always in sync digital model” [TJSW18] of the real devices. Major cloud providers such as Microsoft Azure⁵ or Amazon’s AWS⁶ accordingly already provide means for creating such digital twins as part of their IoT suites. Model-driven IoT frameworks should, therefore, address the challenge of integrating the resulting applications with such digital twins.
- (R11) Tracing of system behavior [KMM⁺22].** Once an IoT application has been deployed to real devices outside of a controlled laboratory setup, it is often difficult to understand why it behaves as it does. One reason for this is the often large number of different sensor values and relationships between the application parts. A manual analysis of logs of the devices is possible but very time-consuming. Thus, IoT frameworks should facilitate the analysis of a running IoT system.
- (R12) Reproduction of system behavior [KMR21].** Another challenge in analyzing IoT systems is that problems encountered by real systems are sometimes difficult to replicate under laboratory conditions. Due to the often large number of sensors, it is difficult to recreate error situations. Therefore, to facilitate debugging, an IoT framework should provide ways to retrospectively reproduce the behavior of IoT applications on real devices.

⁵Azure Digital Twins. [Online]. Last accessed: 18.07.2021. Available: <https://azure.microsoft.com/en-us/services/digital-twins/>

⁶AWS IoT Device Shadow service. [Online]. Last accessed: 18.07.2021. Available: <https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html>

3.7 What Is Out of Scope?

As mentioned, the development of IoT systems involves many aspects of many different domains. It is impossible to cover all these aspects in a single thesis. This section clarifies which aspects of IoT systems are explicitly not the focus of this work.

Electronics design. Since IoT systems use sensors and actuators to interact with their environment, it is undeniable that the fabrication and wiring of these sensors can be part of the development of IoT systems. However, this work deals exclusively with the creation of software in the context of the development of IoT systems. Of course, connecting the software to the hardware takes on a crucial role in this thesis. For this, we use off-the-shelf hardware components. We often rely on the Grove platform⁷. The Grove platform offers hundreds of different sensors and actuators for the rapid prototyping of IoT systems. These sensors can be connected via a standard cable to, *e.g.*, a Raspberry Pi. Even if the resulting hardware prototypes are of course not ready for the market, this enables us to test the integration of various hardware components without having to design the hardware ourselves from scratch.

Network protocols. The MontiThings framework being developed in this work largely operates on the application layer of the ISO/OSI model [Tan11]. MontiThings components let the user abstract from network communications to focus entirely on application logic. Of course, it is necessary that data is actually transmitted over a network in the end. However, MontiThings does not provide its own protocols and procedures for this but instead uses widely used communication frameworks such as message queue telemetry transport (MQTT). One advantage of using well-known communication frameworks—besides reducing implementation workload—is that it makes MontiThings potentially easier to integrate with external ecosystems. *Note:* The author of this thesis was involved in communication-related research ([KSGW20, SKS⁺17]), but it is not used in this thesis.

Security. There is hardly any application that is not exposed to certain security risks. Attacks can take place at all levels of an application: from compromised passwords to transceivers that interfere with radio transmissions. Because of the wide range, it is almost impossible to claim that a system is “secure”. In particular, it is hardly possible to predict which new attack vectors will become relevant in the future. From a security perspective, the use of a model-driven platform brings both advantages and disadvantages. On the one hand, developers have to write less code manually. This results in fewer risks for manual implementation errors. However, if the platform itself has an error, in the

⁷Grove Project Website. [Online]. Last accessed: 25.07.2021. Available: <https://www.seeedstudio.com/grove.html>

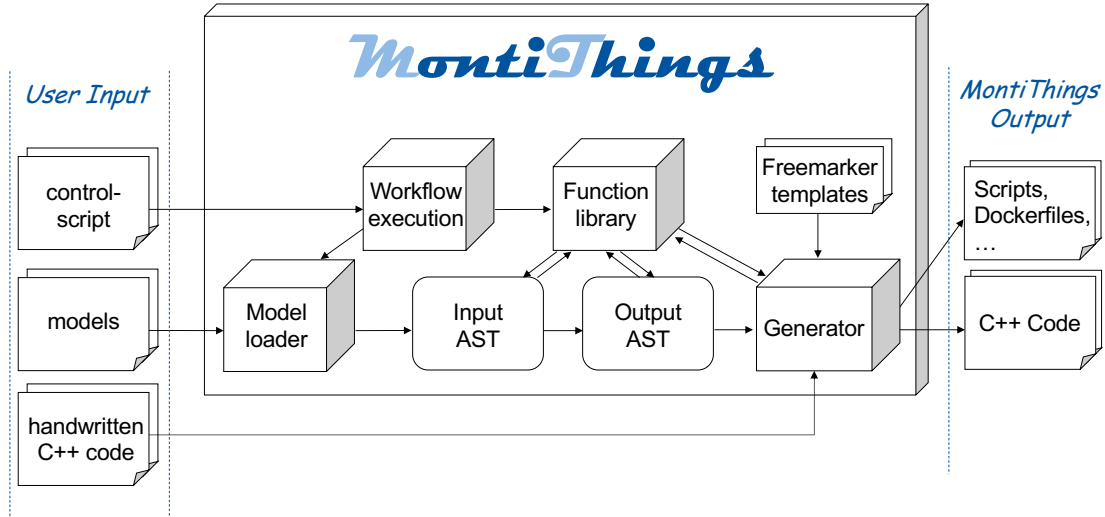


Figure 3.2: High-level overview of MontiThings' code generation. Figure conceptually based on [HR17, HKR21].

worst case all applications built on the platform are vulnerable. MontiThings is primarily designed for evaluation purposes in a scientific context. Accordingly, MontiThings relies on the devices used being secured according to the usual standards (firewall, strong Wi-Fi password, etc.). However, MontiThings itself does not take any additional security measures such as additional encryption of the data exchanged by components. We consider the use of standard communication protocols to be particularly advantageous because these frameworks, due to their widespread use, are much better examined for their security risks than MontiThings can be. It also allows MontiThings to benefit from any future security improvements to these protocols. Nevertheless, we clearly point out that MontiThings has not been subjected to any in-depth security audits. Should MontiThings be considered for large-scale commercial deployments, we recommend doing extensive security analyses prior to deployment.

3.8 Method at a Glance

This thesis presents the *MontiThings* framework for the model-driven development and deployment of IoT applications. The process of developing MontiThings applications consists of the following steps. Fig. 3.2 and Fig. 3.3 provide a high-level overview of the code generation process.

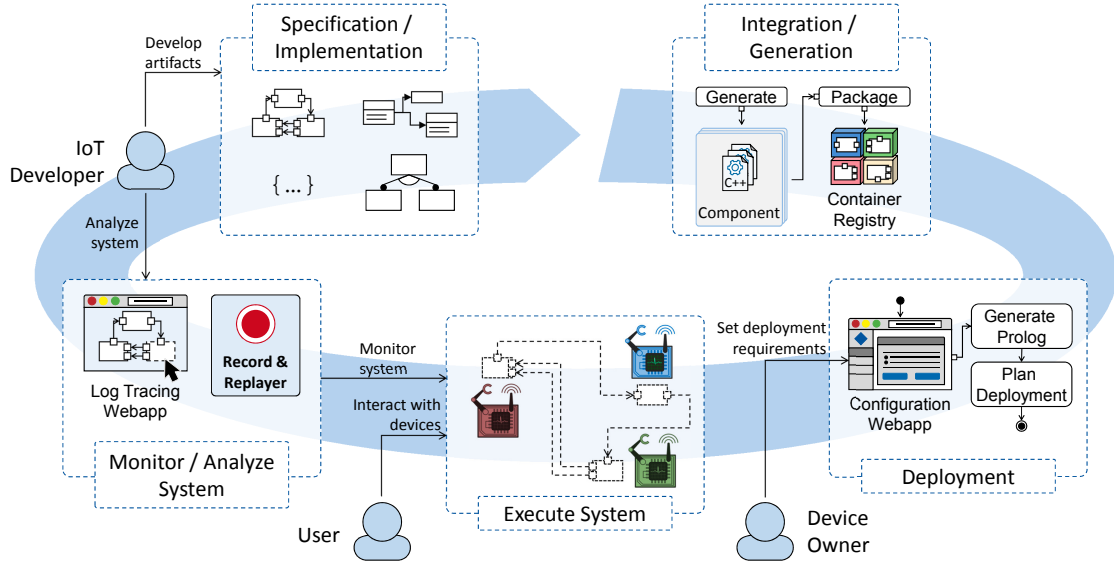


Figure 3.3: High-level overview of iterative development cycle using MontiThings.

1. Develop C&C models that describe the high-level logical architecture of the system. The components shall be independent of concrete hardware. MontiThings offers a MontiArc-based (*cf.* Sec. 2.6) C&C language for this purpose.
2. Connect the hardware-independent components created in the previous step to the hardware offered by the system. MontiThings offers mechanisms to fill ports with code that accesses hardware. Further, a configuration language allows configuring a component's ports for different platforms. For example, a Raspberry Pi might give access to a movement sensor via general-purpose input/output (GPIO) pins while an automotive system accessing a similar piece of hardware might use the controller area network (CAN) bus instead.
3. Configure the generator and generate C++ code using the C&C models and configuration. The generator configuration allows users to en- or disable certain features of the generator. For example, the user can decide to generate a single standalone application for easier testing on the developer's computer instead of generating a distributed application. After the generation, the generated code can be compiled and packaged into Docker images using the scripts provided by the generator.
4. The components of the IoT application communicate with several services during their execution. For example, a message broker like MQTT can be used to allow components to communicate and a deployment manager decides which device shall

execute which components. These services need to be started before the application is deployed.

5. A device owner can decide rules to adjust the deployment strategies to their needs. For example, a device owner might require a fire detection component to be executed in each room of a building. Once the device owner is satisfied with the deployment suggested by MontiThings, the MontiThings' deployment system transfers the Docker images to the devices and starts the application. The deployment system will adapt its decisions as required by the device owner's rules if devices fail or get added to the system.
6. During the execution of the system, developers can choose to interact with the system using services connected to the application. For example, they can use a log tracing tool (Sec. 7.4) to inspect the application and find the causes of errors.

IoT systems are often developed iteratively. Particularly in the early, experiment-heavy stages of development, some steps of the development cycle can be skipped. For example, automated deployment is more relevant in later stages of development and can be replaced by a manual deployment in early stages.

Based on the insights gained while developing MontiThings and multiple IoT applications using MontiThings, we advise IoT developers to consider the following recommendations when designing IoT frameworks and applications:

- *Clearly separate the execution of the hardware control from the execution of the application logic.* Such a separation offers the possibility to draw an abstraction layer between the application and the hardware. Model-driven frameworks without a clear separation, such as ThingML⁸, often pollute the models with hardware-specific logic such as refresh rates, analog-to-digital conversion, and the like. This hardware dependency makes the models less reusable, even if there are only small differences such as connecting the same sensor to a different digital input port of the device. Moreover, it simplifies the parallel development of hard- and software. If the hardware control software is separated from the application logic, it can be easily replaced by mocks. Thus, it is possible to develop software even before the hardware development is completed.
- *Design for the presence of hardware, but allow to reuse components in different contexts.* MontiThings puts hardware-specific code in ports instead of the components. If a port with hardware-specific behavior is connected to another port, this connection overrides using the hardware-specific code. This is similar to the inheritance mechanism of object-oriented programming, where a subclass can override

⁸Example for a hardware-specific model from the ThingML GitHub project. [Online]. Last accessed: 25.07.2021. Available: https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/samples/_arduino/LM335Sensor.thingml

the methods of its superclass. By allowing to override the use of hardware-specific code, components are easier to reuse. It also advocates testing components by creating wrapper components that contain mock components that mimic the interaction with the hardware.

- *Do not include handwritten code or technical details in models.* In an early version of MontiThings, we followed the steps of ThingML and allowed to include handwritten C++ code in models. ThingML uses custom annotations for this purpose such as `@c_header "#include <EEPROM.h>"` [KRSW22]. This problem of ThingML has also been identified in [DRF22] as *leaky abstractions*. An important disadvantage of this approach is that it makes the models less accessible to users without a deep level of understanding of both the generator and the generator’s target general purpose programming language (GPL). Instead, we advocate a clear separation of models and handwritten code. We recommend MontiCore’s TOP mechanism [HKR21] for this purpose. Similarly, other technical aspects such as IP addresses should not be part of the model. In an early version, MontiThings allowed to name IP addresses of communication partners in the models. This tightly bound the models to a single deployment. Instead, we advocate using a separate configuration that can be read in by the generated code shortly before executing an application or using mechanisms that do not require such configurations in the first place. In the case of IP addresses of communication partners, for example, this could be a central entity to which all components connect and which informs them about their communication partners.
- *Enable device owners to set deployment rules.* As mentioned above, [Zam17] separates *global*, *local* managers, and users. Each has different areas of expertise and wants to apply different policies. Similarly, MontiThings uses the distinction between IoT developers and device owners in deployment. The clear separation between hardware and software enables device owners to rules such as wanting to have a fire detector in each room of a smart home after MontiThings’ code generator has already been executed.
- *Design for unreliability.* IoT systems often rely on unreliable hardware (**TC3**). This requires designing model-driven IoT platforms to consider failures at each step of executing an application. This includes, *e.g.*, offering mechanisms to cope with erroneous sensor values or failing devices. Moreover, it requires a high degree of dynamism. Traditionally, “continuous” techniques such as *continuous integration* or *continuous deployment* are triggered by source code changes. In IoT systems, devices can fail and new devices get added to the system. These changes in the system’s hardware require the software to adapt itself triggered by hardware changes. Such continuous deployment has already been explored, *e.g.*, by GeneSIS [FNS⁺20].

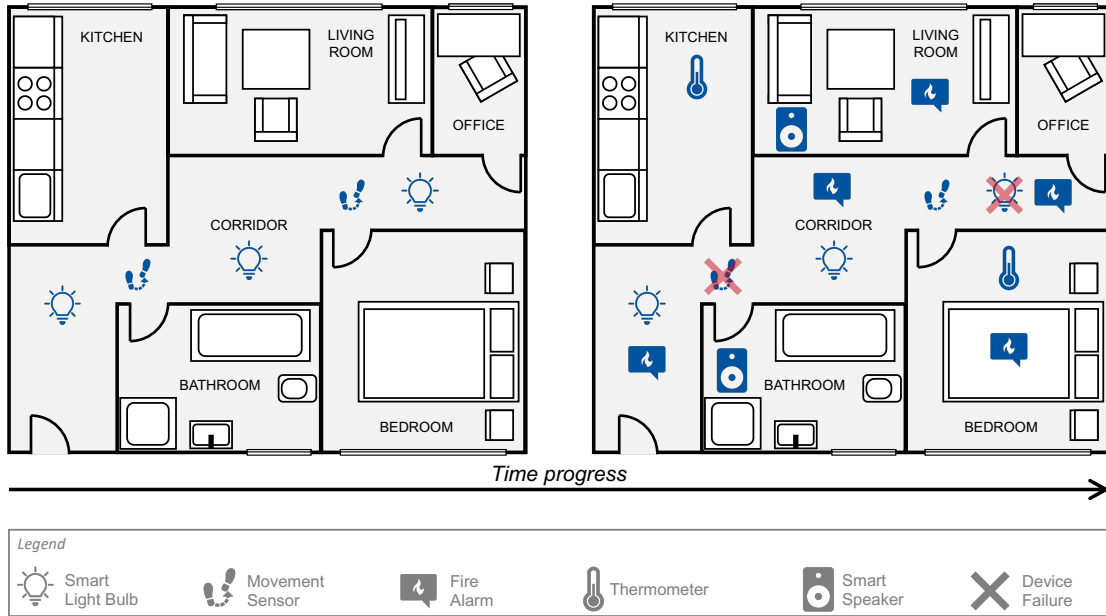


Figure 3.4: A smart home at different points of its lifetime. Some devices fail and new devices get added to the system. Figure taken from [KKR⁺22a].

MontiThings picks this idea up and extends it with a Prolog-based reasoner that decides which devices shall execute which pieces of software.

- *Design for incompleteness.* Many IoT systems are rolled out gradually. IoT systems often consist of a minimal configuration to which additional devices can be added. For example, in a smart home, initially, only the traditional lamps could be replaced by smart lights, and only later, given that the users are satisfied with the system, are other components such as motion sensors for automatic control of the lamps added. Designing for incompleteness requires that the system can react to the addition of new devices (**R8**). Besides integrating new devices into the system, IoT developers shall also consider how to integrate “things” that are currently not equipped with the system’s hardware. For example, in a fleet management system, users could decide to evaluate the system with only three of their twenty cars before they buy the hardware for all twenty cars. Developers should consider offering a method to manually manage the non-equipped vehicles, for example, through a smartphone app.

3.9 Running Use Case: Smart Home

Smart homes are amongst the most popular domains for IoT applications [Ecl20, RLC⁺20]. Although recent surveys suggest a slowly declining interest [Ecl20], it is still one of the most relevant domains. Compared to other domains like smart agriculture, the smart home requires less domain knowledge from the readers of this thesis to follow the examples. Hence, we will choose a smart home as our application domain whenever we demonstrate MontiThings in this thesis.

The smart home consists of IoT devices that the residents of the smart home place in their home to fulfill some overall functionality. For example, smart homes often contain lighting systems that control the home's light bulbs or smart speakers (*e.g.*, Amazon Echo, Apple Homepod) that enable users to control the smart home using voice commands. Today, these IoT devices often strongly couple hardware and software, *i.e.*, each piece of hardware is shipped with vendor-specific software. Besides supporting a number of standards proposed mostly by smartphone manufacturers (*e.g.*, Apple Home), there is often little to no interaction between the devices of different manufacturers. Standardization initiatives like *Matter* hope to change this in the future but are still in the early stages. As mentioned earlier, visions for future IoT applications include the distribution of IoT applications via app stores. Such distribution models will likely require the software to integrate with hardware from other vendors.

Unfortunately, many IoT devices are not built to last forever (**TC3**). Moreover, many users do not buy all hardware available for their home at once but instead start with only a few devices and then buy new devices later to augment their existing system. As a result of these two influences, the hardware available in a smart home often changes during the lifetime of the application. Fig. 3.4 gives an example of such a smart home where devices fail over the lifetime of the devices and new devices get added. The software needs to be able to adapt to such changes in the available hardware. In this example, a movement sensor and a light bulb failed. The system can try to fix this situation by, *e.g.*, redeploying parts of the software to other devices that provide hardware similar to the failed devices. If the application was stateful, the system can also try to recover the state of the failed software on the new device.

In less severe cases of failure, sensors might also not fail completely but output wrong values. Smart home systems need to detect such cases and try to limit their impact on the overall system. As smart homes are distributed systems, which are inherently hard to debug, it is crucial that the development tools for smart home developers, and IoT developers in general, offer appropriate error analysis functionalities.

In the following chapters, we will use numerous applications from the smart home domain to demonstrate MontiThings.

Part II

The MontiThings Ecosystem for Model-Driven IoT Applications

Chapter 4

C&C-based IoT Application Development

The core of MontiThings is a C&C architecture description language that describes the logical functionality of IoT applications. This chapter introduces the MontiThings language and its integration with associated languages such as class diagrams and the configuration language. Models written using the MontiThings language will be used in the next chapter to generate (distributed) IoT applications.

4.1 Research Questions

Overall, this chapter mostly addresses the general question of how to specify C&C-based IoT applications (**RQ1**). Multiple other languages for the development of IoT applications have already been proposed by related work. This includes both C&C-based ADLs, *e.g.*, ThingML [HFMH16, MHF17], Calvin (including its extension Kappa) [AP17, PA15, PA17], as well as IoT “programming” languages, *e.g.*, Eclipse Mita [wwwb]. Each of these languages has its individual strength and weaknesses. MontiThings reuses and adapts some of their concepts and extends them with new concepts to fix weaknesses not addressed by any of them. Overall, the MontiThings language aims to answer the research question:

How to design a modeling language for the specification of distributed IoT applications?

In the course of answering this question, modeling challenges (**MC1**) - (**MC3**) need to be addressed. As the main goal of MontiThings models is to be used as input for a code generator, this chapter also addresses hardware access (**RQ3**). As the target hardware is unreliable (**TC3**), MontiThings facilitates using error handling mechanisms. Thereby, this chapter contributes to handling malfunctioning (**RQ4**) and failing (**RQ5**) devices. Further, connecting to heterogeneous platforms (**TC1**) is addressed by the configuration language.

4.2 MontiThings Language

The MontiThings C&C ADL is based mainly on the MontiArc ADL (*cf.* Sec. 2.6) because MontiArc already offers a C&C ADL built on a solid semantic foundation and enables us

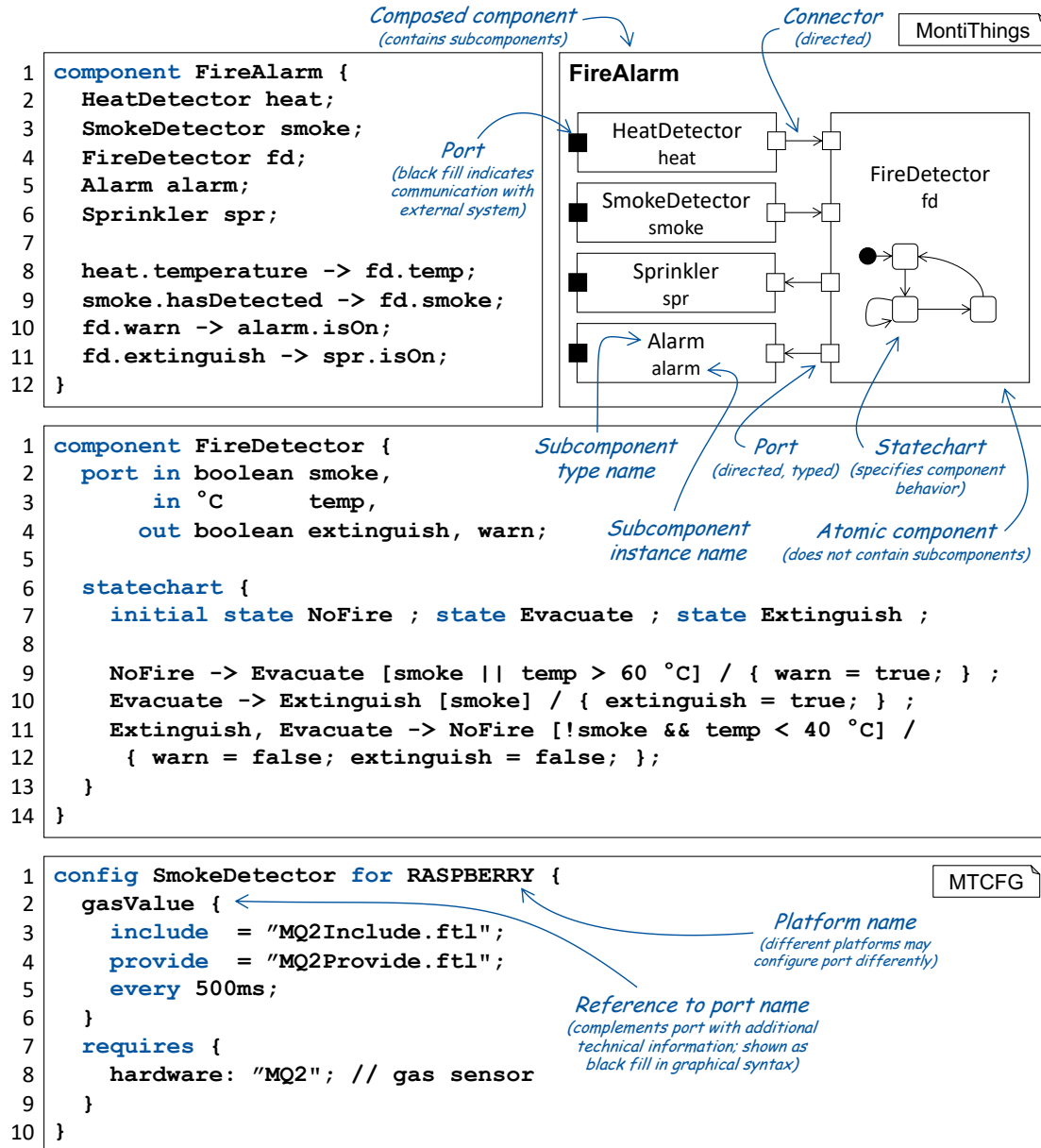


Figure 4.1: A teaser of the graphical and textual syntax of MontiThings. The constructs in this figure are explained throughout this chapter. Figure adapted from [KKR⁺22a].

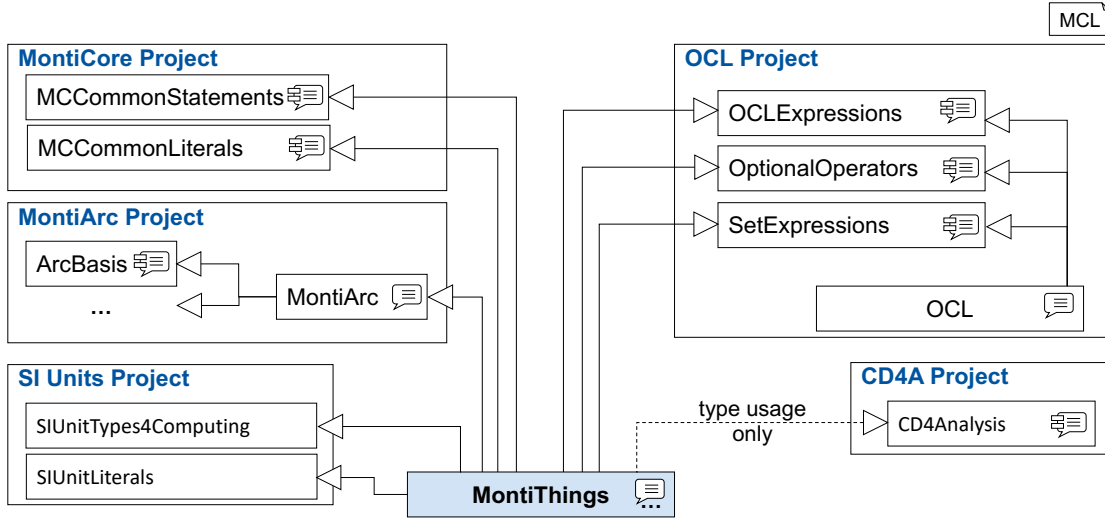


Figure 4.2: Overview of MontiThings' relation to languages of the MontiVerse. Figure taken from [KKM⁺22].

to reuse MontiCore's large language catalogue [KKM⁺22]. Consequently, MontiThings' concrete and abstract syntax are very similar to the concrete and abstract syntax of MontiArc. Extensions tailor MontiThings to its IoT use case. Fig. 4.1 demonstrates the MontiThings language. Additionally, MontiThings uses various other languages of the MontiVerse, *i.e.*, languages offered by the MontiCore project. Fig. 4.2 gives an overview of the relation between the MontiThings grammar and other grammars of the MontiVerse project. Transitive relations are omitted for readability. MontiCore's `MCCCommonStatements` and `MCCCommonLiterals` provide a lightweight Java-like language used by MontiThings as one way to specify behavior (Sec. 4.2.4). Besides, MontiThings can also use the MontiVerse's statechart language that is integrated into MontiArc. The `SIUnitTypes4Computing` and `SIUnitLiterals` extend MontiThings' type system with support for SI units. Like MontiArc, MontiThings can also use class diagrams modeled using the `CD4Analysis` language. Using parts of the OCL/P language, MontiThings facilitates specifying pre- and postconditions to catch errors, caused especially by malfunctioning hardware, as early as possible (**RQ4**). A selection of MontiVerse grammars used by MontiThings can be found in Appendix B. An overview of MontiThings' grammars can be found in Appendix C.

4.2.1 Component Definition and Instantiation

Components define the main building blocks of MontiThings. Components consist of an interface, parameters, behavior, pre- and postconditions, subcomponents, and a state.

```
1 component Sink {  
2   port in int value;  
3 }
```

MT

Listing 4.1: Definition of a component type Sink with a single incoming port of type int named value.

```
1 component Source (int startValue) {  
2   port out int value;  
3  
4   int lastValue = startValue;  
5 }
```

MT

Listing 4.2: Definition of a component type Source with a parameter of type int named startValue, a single outgoing port of type int named value, and a state variable of type int named lastValue.

The interface consists of incoming and outgoing ports defining the inputs and outputs of components. We make a further distinction between *atomic* components, which do not contain subcomponents, and *composed* components, which define their behavior by instantiating and connecting subcomponents [Hab16]¹. However, an interface description of a component does not reveal whether it is atomic. Atomicity is encapsulated and thus irrelevant from the users' perspective.

The example in Listing 4.1 shows the definition of an atomic component type Sink that consumes values of type int. The Sink contains a single incoming port for consuming values. Just like classes in object-oriented languages, components may define variables. To avoid a possible source of undefined behavior, variables are required to be initialized when they are defined. By defining a parameter in parenthesis after the component type's name, components can also accept arguments from the components that instantiate them. These parameters can be used like constants within the component. As shown in Listing 4.2, a parameter can be used to initialize a variable similar to how member variables in classes of object-oriented languages are often initialized using the arguments passed to the constructor.

Neither Source nor Sink define a behavior within the model. Their behavior is, thus, underspecified. At the language level, this is allowed. The code generator, however, will require developers to provide a behavior description for every component. If a component shall explicitly not have a behavior, the component needs to be declared as an *interface* component using the corresponding keyword. The idea of using behavior-

¹Some publications on MontiArc also call components with subcomponents *decomposed* components. Besides the name, there is no difference between *composed* and *decomposed* components.

```

1 interface component MathOperation {
2   port in int x;
3   port out int y;
4 }

```

MT

Listing 4.3: Definition of an interface component type `MathOperation`. Interface components have no behavior.

```

1 component Calc<T extends MathOperation> {
2   port in int x;
3   port out int y;
4
5   T t;
6
7   x -> t.x;
8   t.y -> y;
9 }

```

MT

Listing 4.4: Definition of a generic component type `Calc`. The argument passed to the type parameter `T` must conform to the interface of `MathOperation`.

less components for specification is also present in many other C&C languages. For example, Calvin [AP17, PA15, PA17] uses a similar concept which they call *shadow actor*. We adopt the definition from [Wor16]:

Definition 11 (Interface Component). “An interface component is a component without subcomponents, behavior model, or component behavior implementation reference.” [Wor16]

Interface components in MontiThings serve a similar purpose as interfaces in Java. Note, however, that [Wor16] used interface components to switch between implementations for different technical platforms. In contrast, all MontiThings components are platform-independent. Like in MontiArcAutomaton, MontiThings’ interface components are declared using the `interface` keyword, as shown in Listing 4.3.

The main purpose of interface components is to serve as placeholders in generic components, *i.e.*, components that accept other component types as arguments, as shown in Listing 4.4. Type parameters are defined in angle brackets. The name of the type parameter, `T` in this example, must be followed by the keyword `extends` and the name of an interface component type. On instantiation of this component, a CoCo checks that

```

1 component Example {
2   Source source (1);
3   Calc<Doubler> c;
4   Sink sink;
5
6   source.value -> c.x;
7   c.y -> sink.value;
8 }

```

MT

Listing 4.5: Definition of a composed component type Example.

the interface of the component type provided for T has the same ports as the interface component. Since the `Calc` component knows that T conforms to `MathOperation`, it can instantiate T and connect its ports. The instantiation is specified in line 5 of Listing 4.4. Instantiations start with the component type name followed by an instance name. The instance name can be used to refer to that component instance within the instantiating component. We call the instance *subcomponent* and the instantiating component the *enclosing component*:

Definition 12 (Subcomponent). *A subcomponent “instantiate[s] a component type definition as an element of another component type definition” [Hab16].*

Definition 13 (Enclosing Component). *The enclosing component (type) of a component instance is the component that contains this instance as one of its subcomponents.*

Composed components contain other (atomic or composed) components to define their own behavior. Using the arrow operator (\rightarrow), composed components can connect ports with each other (see lines 6-7 in Listing 4.5).

4.2.2 Type System

MontiThings’ type system differentiates component types from the types of variables and messages sent via ports. The components’ type system was already described in [Hab16] and in the previous section. MontThings’ type system for variables and messages builds upon the primitive types known from Java or C++. Table 4.1 summarizes these primitive types. Developers can define application specific data types using class diagrams. For

Data type	Range [Bre07]	Description
byte	$[-128, 127]$	8-bit signed integer
int	$[-2^{31}, 2^{31} - 1]$	32-bit signed integer
long	$[-2^{63}, 2^{63} - 1]$	64-bit signed integer
float	$[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$	32-bit IEEE 754 floating point number
double	$[-1.8 \times 10^{308}, 1.8 \times 10^{308}]$	64-bit IEEE 754 floating point number
boolean	{true, false}	Truth value, <i>i.e.</i> , either true or false
char	N/A	a single 16-bit Unicode character
String	N/A	text that can contain multiple characters

Table 4.1: Primitive data types of MontiThings.

```

1 component Source (m/s startSpeed) {
2   port out km/h value;
3
4   km/h lastValue = startSpeed;
5 }

```

MT

Listing 4.6: Definition of a component type Source that uses SI units.

this, MontiThings uses the CD4Analysis language. The integration with class diagrams is described in more detail in Sec. 4.3.1.

Additionally, MontiThings also extends the SI units language from the MontiVerse. Therefore, it is also possible to use SI units as normal types. For example, we can adapt our Source component from the previous section to use a velocity value.

The example also shows that SI values with different SI unit types can be combined in the same expression. Assigning startSpeed of type m/s to lastValue of type km/h could be a source of potential errors in traditional programming languages. If both values are implemented using the same primitive type, *e.g.*, int, developers might not be aware that they refer to different SI unit types and forget to convert them to the same SI unit. Since MontiThings is aware of the SI unit, it handles all necessary conversions automatically. The only requirement is that types of values that are to be calculated together must be compatible with each other, *i.e.*, there must be a formula for converting the types into each other. For example, adding m/s to km/h is possible, but adding m/s to dB is forbidden. This also holds for ports of components. As long as the types of the ports are compatible, MontiThings will automatically take care of the necessary conversions. In the generated code, SI units are treated as double values, unless specified otherwise in the model (by overriding its type using the kg/s<int> notation).

4.2.3 Timing


Because MontiThings generates code for real systems (as opposed to a simulation), MontiThings uses strongly causal event-based time semantics. Components execute their behavior whenever they receive a message on at least one of their ports. Developers can control this behavior by bundling ports so that messages on multiple ports are required in order to trigger a calculation. The execution of the behavior has run-to-completion semantics, *i.e.*, the execution of the previous behavior always needs to finish before the next execution may start. This is chosen to avoid hard-to-analyze concurrency problems for developers when messages are received on ports while the current execution is still running. *Strongly causal* means components may only depend on past messages but not on future messages [BS01, Hab16].

Conceptually, each MontiArc or MontiThings component can be specified by a stream processing function from the FOCUS calculus [BS01]. The FOCUS calculus defines three different timing modes: untimed, timed, and time-synchronous. MontiThings' timing is similar to the timed mode of FOCUS. Timed streams contain *ticks* that represent time progress [BS01]. FOCUS represents time progress with ticks because it is intended for simulation and the simulation controls the time progress. Since MontiThings generates code for real systems where time progress is real, ticks in MontiThings do not exist explicitly in implementation. The ticks exist only in the specification framework behind MontiThings and are only used to understand the semantics of MontiThings models. In this sense, each MontiThings port can be specified by a timed stream of messages. In the implementation, the ports only exchange messages as time progresses naturally. Conceptually, MontiThings uses ticks with a high enough frequency so that there is always at most one message between two ticks. Using the tm operator from [BS01], where $tm(s, j)$ is defined as the “time interval of s in which the j th message in s occurs” [BS01], we can denote this as:

$$\forall i, j \in \mathbb{N} : i < j \implies tm(s, i) < tm(s, j)$$

We also assume that the execution of a computational operation on real systems always has some delay. From a technical point of view, it is very unlikely, that a component receives two messages at exactly the same time at the same port. With simultaneous media accesses the messages on the medium usually interfere with each other. Therefore, medium access control protocols ensure that simultaneous access to a shared medium is prevented. However, especially in wireless communication, these protocols cannot prevent all simultaneous accesses, *e.g.*, in the hidden station problem [Tan11]. By using multiple processors (or processor cores) together with multiple Ethernet cables (for wired communication) or multiple antennas it is possible to receive two messages at exactly the same time. MontiThings, thus, needs to define how to handle the situation that two messages reach the same incoming port at exactly the same time.


```
1 component LowPassFilter (int threshold, int defaultValue) {
2   port in int givenValue;
3   port out int filteredValue;
4
5   behavior givenValue {
6     if (givenValue > threshold) filteredValue = defaultValue;
7     else filteredValue = givenValue;
8   }
9 }
```



Listing 4.7: Definition of a component type `LowPassFilter` that replaces messages higher than a threshold with a default value.

This problem does not exist in `MontiArc` because developers are only allowed to connect a single connector to each incoming port. If developers want to connect multiple outgoing ports to the same incoming port, they, thus, have to define a new component with multiple incoming ports that merges the messages into a single stream. `MontiThings` allows multiple connectors to be connected to an incoming port. This can lead to situations where multiple messages reach the port at exactly the same time. In this case, we assume that a merger component exists that resolves this conflict. This component has an incoming port for each of the connectors connected to the incoming port in question and one outgoing port that outputs a stream that merges the streams from all of its incoming ports. Its behavior, *i.e.*, the strategy by which the streams will be merged is underspecified. Note that this merger component exists in the conceptual framework behind `MontiThings` to model the merging technically realized by the infrastructure. In practice, it is up to the communication library to decide how to sort simultaneous messages. Messages that cannot be processed immediately because other message are still being processed are buffered until the component finishes processing the messages that were scheduled for earlier processing.

4.2.4 Behavior Description

`MontiThings` offers four ways of describing the behavior of a component:

1. By instantiating and connecting subcomponents
2. Using a lightweight IoT-focused Java-like language
3. Using Statecharts (SCs)
4. Using handwritten C++ code²

²This uses the generation gap pattern [Fow10]

```
1 component Source {  
2   port out int value;  
3  
4   int lastValue = 0;  
5  
6   every 1s {  
7     value = lastValue++;  
8     after 500ms {  
9       log ("Source: " + value);  
10    }  
11  }  
12 }
```

MT

Listing 4.8: Definition of a component `Source` with a periodic behavior.

The first method was already shown in Listing 4.5. The lightweight Java-like behavior description language uses MontiCore’s `MCCCommonStatements`. This Java-like language provides Java’s most important programming constructs such as `if` conditions, loops, or function calls. The example in Listing 4.7 defines the behavior of a component using this language.

Between the `behavior` keyword in line 5 and the opening curly brace, developers may reference an arbitrary number of comma-separated incoming ports. These groups of ports are very similar to `ParameterSets` from unified modeling language (UML) activity diagrams [Obj17, clause 16.3.4.5]. The behavior will be executed when a message can be consumed on each of the referenced ports. If no port is referenced, the behavior will be executed when a message arrives on any port. Within the behavior, developers may only access the ports which were mentioned after the `behavior` keyword. Outgoing ports are write-only. As soon as a value is written to an outgoing port the message will be sent immediately without waiting for the behavior block to finish executing. Therefore, `x=1; x=1` sends two messages and, thus, differs from `x=1`. It is possible to define multiple behaviors for multiple ports, but the groups of ports mentioned after the behavior keyword may not be subsets of one another.

A second form enables developers to trigger a cyclic execution of a component. For this, developers use the `every` keyword followed by a time duration and a block of code that shall be executed periodically (Listing 4.8). This concept is also present in Eclipse Mita [wwwb], an IoT-focused C-like programming language. Given that IoT devices may have resource-constraint hardware and behavior blocks may contain complex code, the execution may finish after the specified interval. In this case, MontiThings will log a warning and continue with the next execution directly after the preceding execution finishes.

```

1 component Example {
2   Loop loop;
3   loop.output -> loop.input;
4 }
5
6 component Loop {
7   port in int input;
8   port out int output;
9
10  init {
11    output = 0;
12  }
13
14  behavior input {
15    log("Input: " + input);
16    after 1s {
17      output = input + 1;
18    }
19  }
20 }

```

MT

Listing 4.9: A component that sends messages to itself in an endless loop.

Using the `after` keyword, a block of code can be delayed. In contrast to calling a `delay` function which effectively puts the program to sleep, as known from, *e.g.*, Arduino, the `after` block delays a block of code asynchronously. This means that the code block of the `after` statement is executed after the specified delay, but code below the block of the `after` statement is executed immediately. The time until the execution of the `after` block is always measured from the time the execution reaches the `after` statement, as opposed to the time the behavior was started. For example, `log("a"); after 500ms {log("b");} after 200ms {log("c");} log("d");` would print `adcb`.

There is third a variant of `behavior` called `init`. The `init` behavior is executed only on the first message on the referenced port(s). If the `init` behavior does not reference any ports, it is executed immediately when starting the component before processing any message. This is similar to the `setup()` function from Arduino. The idea behind this is to enable developers to initialize sensors or actuators that might be connected to the component. For example, if you consider a weight sensor, these weight sensors often have no clearly defined zero. Therefore, many scales automatically set the current weight as zero when being turned on. This is called the *tare* weight. Using the `init` such initialization functionalities can also be achieved in MontiThings. It is also possible to combine `init`, `behavior`, and `every` in the same component. For example, developers can create an endless loop if a component sends messages to itself, as shown in Listing 4.9.

```
1 component LightBulb {  
2   port in String command;  
3   port out boolean lightSwitch;  
4  
5   statechart {  
6     initial state Off ;  
7     state On ;  
8  
9     Off -> On [command == "on"] / {  
10      lightSwitch = true;  
11    };  
12    On -> Off [command == "off"] / {  
13      lightSwitch = false;  
14    };  
15  }  
16 }
```

MT

Listing 4.10: A component for a smart light bulb that turns on or off based on voice commands from the user.

In time-synchronous MontiArc, such self-loops are forbidden because components could send messages to themselves that need to be processed in the same time slice as the message that triggered the computation. More generally, feedback loops require a delay in the loop [RR11]. As MontiThings does not allow components to output messages in the same time slice that triggered the computation, such self-loops are automatically delayed, because MontiThings conceptually always ensures a delay. This delay is not an artificially added delay, but exists naturally because the execution of computing operations always requires processing time. Therefore, components cannot instantaneously use their own output as input in a feedback loop.

The third option to define the behavior of components is by using statecharts. For this purpose, MontiThings uses the MontiVerse's statechart language. In the example in Listing 4.10, a (very simple) smart light bulb is controlled via voice commands from the user. How the voice commands are converted into text is irrelevant for this example, but usually, a different component would make use of a third-party service like Microsoft Azure's *Speech to Text* or AWS's *Amazon Transcribe* service for this purpose. Initially, the light is off. If the voice command is "on" the light bulb is turned on by sending a message on the `lightSwitch` port. Similarly, if the voice command is "off" the light bulb is turned off. If the voice command refers to the current state, the component ignores the command.

```

1 component RunningSum {
2   port in int input;
3   port out int result;
4
5   int cumulativeSum = 0;
6
7   pre input >= input@pre;
8   catch { input = 0; }
9
10  behavior input {
11    cumulativeSum += input;
12    result = cumulativeSum;
13  }
14
15  post cumulativeSum == cumulativeSum@pre + input;
16 }

```

MT

Listing 4.11: A component that calculates a running sum and for monotonically increasing values.

4.2.5 OCL

As MontiThings is an IoT-focused language, there are usually lots of error checks required to get the application to execute reliably (**TC3**). Studies suggest that a higher assertion density in code leads to lowered fault density [KNB06, CDO⁺15]³. Nevertheless, an analysis of GitHub projects showed that assertions are used “often sparingly” [CDO⁺15]. Thus, MontiThings aims to make it easy to specify assertions in components. To do so, MontiThings also incorporates expressions from a programming-focused adaption of the OCL [Obj14] called OCL/P [Rum17]. These expressions enable developers to define pre- and postconditions for the behavior without cluttering the business logic inside the behavior blocks. Components evaluate preconditions after receiving a message on a port and before executing the corresponding behavior for this message. Postconditions are evaluated after requesting to send a message on an ongoing port and the message is sent only if the postcondition evaluates to true. Additionally, postconditions are also evaluated after the component’s behavior finishes to ensure the component is in a valid state before processing the next message.

Preconditions enable developers to detect hardware errors before they are processed further by the system and corresponding subsequent errors make the actual cause of the error more difficult to detect. Similarly, developers can also use it to safeguard compo-

³A study of industrial telecom systems could not link the number of assertions and defects [CHS⁺17]. The expressiveness of this result is, however, limited by the fact that only two industrial telecom systems were analyzed.

```
1 component FilterPrimes {  
2   port in int i;  
3   port out int o;  
4  
5   behavior i {  
6     if (!exists x in {2 .. i-1}: i % x == 0) {  
7       o = i;  
8     }  
9   }  
10 }
```

MT

Listing 4.12: A component that takes an integer and only forwards it if it is a prime number.

nents against inputs that do not conform to the component’s specification. Preconditions thus take on a role similar to that of context conditions in language development: While the data type of a port assures the basic structure of the data, preconditions can specify additional conditions, such as that the stream of messages on an incoming port consists of monotonically increasing values (line 7 in Listing 4.11). Such conditions about the sequence of the data cannot be specified by the data type alone. Postconditions, on the other hand, can be used similarly to `assert` statements of C/C++ and ensure that the stream of outgoing messages meets the component’s specification. Thereby, postconditions can prevent error propagation in case of implementation errors. If pre- and postconditions depend on values from the past, they will only be evaluated if all these values are present. A condition like the example above for checking monotonically increasing values, which refers to a value from the previous execution, would thus be considered the first time during the second execution.

When a pre- or postcondition is violated, MontiThings offers two possible coping strategies. First, developers can specify a `catch`-block after the pre- or postcondition (line 8 in Listing 4.11). Similar to a `catch` statement in Java, this block of code will be executed if the pre- or postcondition is violated. Conceptually, conditions with a `catch` statement act as components upstream (*i.e.*, before the incoming ports) or downstream (*i.e.*, behind the outgoing ports) of the actual component that only forward messages if the conditions are not violated. Developers can use this block to specify how the component shall handle the situation, *e.g.*, by using a sensible default value instead of the actual value. If no `catch`-block is provided, MontiThings will stop the corresponding component and log the current state and the violated condition. This type of error behavior is called *fail fast* [Sho04]. By immediately stopping the component if it deviates from its specification, MontiThings prevents error propagation that would lead to even harder to detect errors later or in other components.

The example in Listing 4.11 shows how pre- and postconditions can ensure that a `RunningSum` component only runs on monotonically increasing values. The `pre` statement specifies that the message on the `input` port has to be at least the value of the previous message on the port (line 7 in Listing 4.11). The `@pre` keyword also refers to the state before the execution of the most recent computation (line 15 in Listing 4.11). If used on a port in a precondition, it accordingly refers to the message processed by the previous computation. In case the input message is not at least as high as the previous input, the component will overwrite the input message with zero, thereby leaving the state unchanged. The postcondition (line 15 in Listing 4.11) ensures that the state of the component changed according to the specification during the computation, *i.e.*, that the `cumulativeSum` variable is increased by the value of the input message.

For IoT systems, programming with time is very important. Therefore, MontiThings offers an extension to the `@pre` notation from OCL. The `@ago` can be attached to ports and variables to refer to their value a certain period of time ago. For example, `cumulativeSum@ago(2s)` would refer to the value the `cumulativeSum` variable had two seconds ago. For memory management reasons (*cf.* Sec. 5.3), only constant time durations can be used in `@ago` expressions. As components are strongly causal and, thus, cannot look into the future, negative time values in the `@ago` expression are not allowed.

Aside from pre- and postcondition, OCL expressions can also be used like normal boolean expressions in other behavior blocks (Listing 4.12). This can be useful especially for OCL expressions using sets because it enables developers to efficiently write up the intent of calculations that would otherwise be implemented in loops. For example, this can be used to specify a component that filters prime numbers, *i.e.*, discards all incoming messages that are not prime numbers, and forwards the messages that refer to prime numbers. Using OCL, the fact that a number is a prime number can be directly specified by using the fact that for each prime number, there exists no number between two and the number itself minus one that divides it without a remainder.

The last option to specify component behavior is to use handwritten C++ code. Using MontiCore’s TOP mechanism [HKR21], users can override any code file generated by MontiThings. Therefore, it is also possible to provide the implementation for a component using handwritten C++ code. Developers who want to provide such C++ code need to provide a file with the name `CompImpl.cpp` where `Comp` is replaced by the component’s name. In this file, they implement a `compute()` method provided by generated `CompImplTOP` class. In this method, developers can access the component’s interface and state. We recommend only using handwritten C++ code if the behavior language cannot be used. For example, if a component wants to access a third-party service like a cloud service provided by, *e.g.*, Microsoft Azure or AWS. Such cloud services usually provide dedicated (C++) APIs that cannot be accessed in MontiThings’ behavior language. As the API is inaccessible in MontiThings, it may be necessary to access these services by using handwritten C++ code.

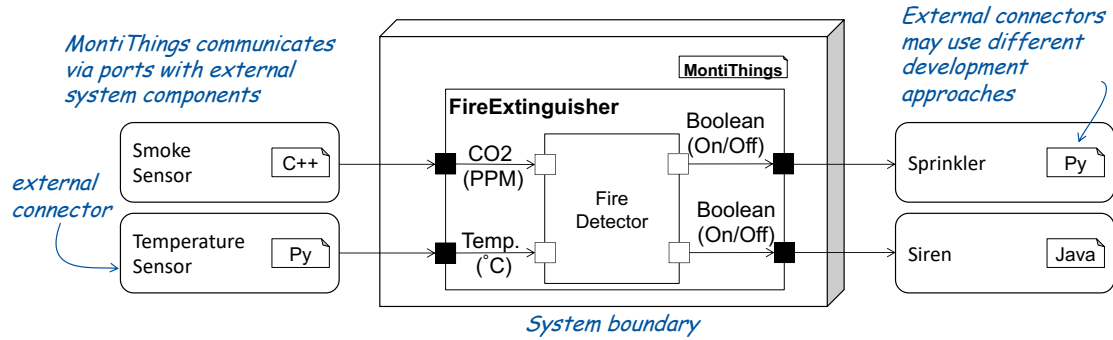


Figure 4.3: MontiThings' system boundary is at the hardware access. External services can be used to access sensors, actuators, and other hardware via ports.⁴

4.2.6 Sensor and Actuator Access

An important part of IoT applications is the integration with sensors and actuators, or more generally, hardware. IoT devices are very heterogeneous (**TC1**) and accessing the same kind of sensor, *e.g.*, a temperature sensor, may need to be implemented differently for different platforms. MontiThings' vision is to decouple the development of hardware and software (Sec. 3.1). The software components generated using MontiThings are to be deployed on general-purpose hardware that is not necessarily provided by the same vendor as the software. Therefore, MontiThings' decouples the hardware access from the specification of the software. This general idea of decoupling external software from the IoT application is in line with the approach proposed in [AMMK19] that, however, is not related to a modeling language.

Fig. 4.3 shows how MontiThings components may connect to external hardware such as sensors and actuators. As MontiThings' system boundary lies at the hardware access of a component, this communication with external services is done via ports (shown with black filling in Fig. 4.3). MontiThings expects that the devices on which the generated components are deployed will offer small services, called *external connectors*, via which components can access the hardware. Conceptually, these external connectors have the role of drivers. Using a configuration language (Sec. 4.3.2), developers can further specify what this connection looks like.

Compared to other IoT languages like ThingML [HFMH16, MHF17] and Calvin [AP17, PA15, PA17], this approach provides the advantage, that the external connectors can be implemented in different languages than MontiThings' target language C++. From our experience with dozens of sensors and actuators for the Raspberry Pi platform, most of them provide either drivers for Python or C/C++. Without

⁴This figure has been used as part of various lectures of our chair, *e.g.*, the *Model-based Systems Engineering* lecture.

this mechanism, the Python-based drivers would require developers to either adapt them to C/C++ or to use a different piece of hardware. By enabling the integration of software written in different languages, MontiThings offers more flexibility in the choice of hardware. CapeCode’s [BJK⁺18] accessor pattern is similar in that it tries to integrate third-party software into the architecture. The accessor pattern uses a special component that has the purpose of accessing the hardware and making it available to the rest of the architecture. MontiThings on the other hand uses ports instead of components to access hardware. By using ports instead of components, the hardware access happens via clearly defined interfaces and can be checked, *e.g.*, using pre- and postconditions as described in Sec. 4.2.5.

In many applications, however, the hardware access does not only happen at the outermost component but also at components deeper in the component hierarchy. Directly accessing the hardware there with an unconnected port would, however, be in violation of FOCUS. Therefore, MontiThings offers an automatic transformation that can add and connect unconnected ports to the interfaces of the components that instantiate them. Using this mechanism, developers can use hardware access in components deep within the hierarchy, while developers who want to use FOCUS-based verification tools like MontiBelle [KPRR20] can still get a consistent architecture.

It is important to note here that while we draw ports that could access sensors with a black filling in the graphical models, these ports are indistinguishable from normal ports in the textual model. Indicating in the model that a port access hardware would limit the composability of this component. This is in contrast to approaches like ROOM [SGME92, Sel96] and UML modeling and analysis of real-time and embedded systems (MARTE) [Sel08] that distinguish between ports that connect components and ports that connect to underlying infrastructure⁵. Instead, MontiThings uses a priority mechanism similar to inheritance from object-oriented languages. If a port is unconnected and developers provide a configuration that enables it to access hardware, this port will access said hardware. If instead, the port is connected to another port in the architecture, the port is treated as a regular port while ignoring all hardware-related configurations. Conceptually, this means that the component that instantiates a sub-component has the power to override its subcomponent’s decision to access hardware by connecting the port to another port.

Especially, this also enables developers to use components that are supposed to access hardware in test environments where the component is connected to components that provide test data or validate outputs of the system under test. Also, this makes it easy to use mock components instead of real hardware in the early stages of the development where the hardware might not even have been developed. Later once the real hardware

⁵In ROOM’s and MARTE’s layered architecture *service access points* and *service provision point* connect different layers of the architecture.

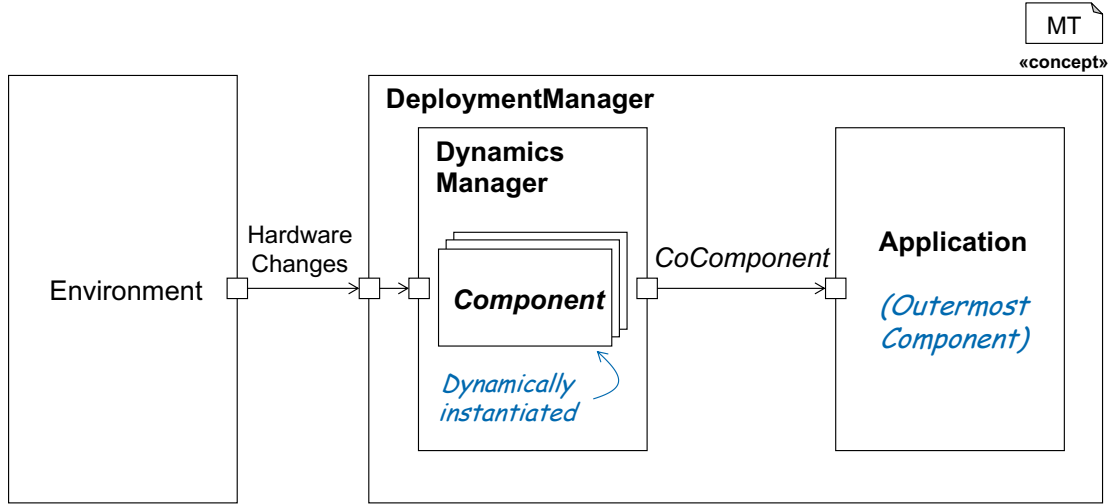


Figure 4.4: Conceptual overview of MontiThings’ dynamics. The environment informs the deployment manager about hardware changes. As a result, the dynamics manager may choose to instantiate components. Their interface is given to the outermost component of the application modeled by the developers. Figure adapted from [KKR⁺22a].

exists, these mock components can be removed. Compared to other IoT languages, MontiThings components are, thus, more flexible to reuse.

4.2.7 Dynamic Reconfiguration

IoT systems are often dynamic (**MC3**), *i.e.*, need to handle component instances joining and leaving during the runtime of the system. A special characteristic is that the system itself does not decide when dynamic changes occur. Instead, the changes are triggered by hardware being added to the system or hardware failing or being removed deliberately. The system thus needs to be able to react to new software components being instantiated or removed as a result of hardware changes. How the system determines which software components to instantiate based on the available hardware is further discussed in Chapter 6.

Conceptually, MontiThings handles such dynamics by enabling components to exchange their interfaces via ports and connect to or disconnect from interfaces received via ports. Intuitively, the concept behind this is similar to two people exchanging business cards containing their contact information. Components do not always have to give out their entire interface, just as business cards do not necessarily contain all possible ways to reach a person (for example omitting the private cell phone number). By implementing an interface component that references only parts of the component’s interface,

developers can specify the ports to be exchanged. Once one receives the business card of the other, the person receiving the business card can contact the person to whom the business card belongs. Consequently, the components whose interface a component receives do not become subcomponents of this component, even if it connects to it. The component that receives (parts of) an interface of another component has no possibility to delete this component. Also, the component that receives the interface of another component has no guarantees that this component will stay available forever. If a dynamically instantiated component becomes unavailable this can lead to two situations: If the removed component is connected to an incoming port of another component, this component will no longer receive messages on this port. If the removed component is connected to an outgoing port of another component, this component can still send messages on its outgoing port but they will not be received by the removed component. This is equivalent to connecting the port to an empty “Sink” component that discards all incoming messages.

Fig. 4.4 gives a conceptual overview of how MontiThings implements this concept. The Application, *i.e.*, the outermost component provided by the developers is conceptually instantiated by a DeploymentManager component. This DeploymentManager gets informed about hardware changes by the Environment. Hardware changes include all modifications to the technical infrastructure, *e.g.*, adding devices, removing devices, or failing sensors. Based on this information, the DynamicsManager may choose to instantiate components or remove component instances it previously instantiated.

The DynamicsManager is the only component that can instantiate components. The reason for this is that MontiThings’ goal is to react to hardware changes that can not be prescribed by the software. Accordingly, the MontiThings language does not offer means to instantiate components (no `new` operator). As the DynamicsManager is the only component that is explicitly allowed to instantiate components, its code is handwritten so that it can instantiate components despite the language not offering a way to do so. Chapter 6 describes in more detail how the Deployment- and DynamicsManager decide which components to instantiate. To summarize, the global and local managers define technical requirements of components and additional rules they want the deployment to fulfill. The dynamics manager then uses a Prolog-based code generator to find suitable deployments or make suggestions if no deployment can fulfill all rules.

For each of the instantiated components, the DynamicsManager provides the interface of the instantiated component (its “business card”) to the application (CoComponent in Fig. 4.4). How this interface is integrated into the component hierarchy by connecting to it is determined by the application, *i.e.*, by the specification provided by the developers. For the dynamic reconfiguration, developers are allowed to also add a behavior to composed components. The behavior of composed components may also use the syntax of connectors (`->`) as a statement to connect to the received interfaces. Similarly, a disconnect statement (`-/>`) enables developers to remove the connection to an interface if the application is informed that the component no longer exists. Thereby, developers

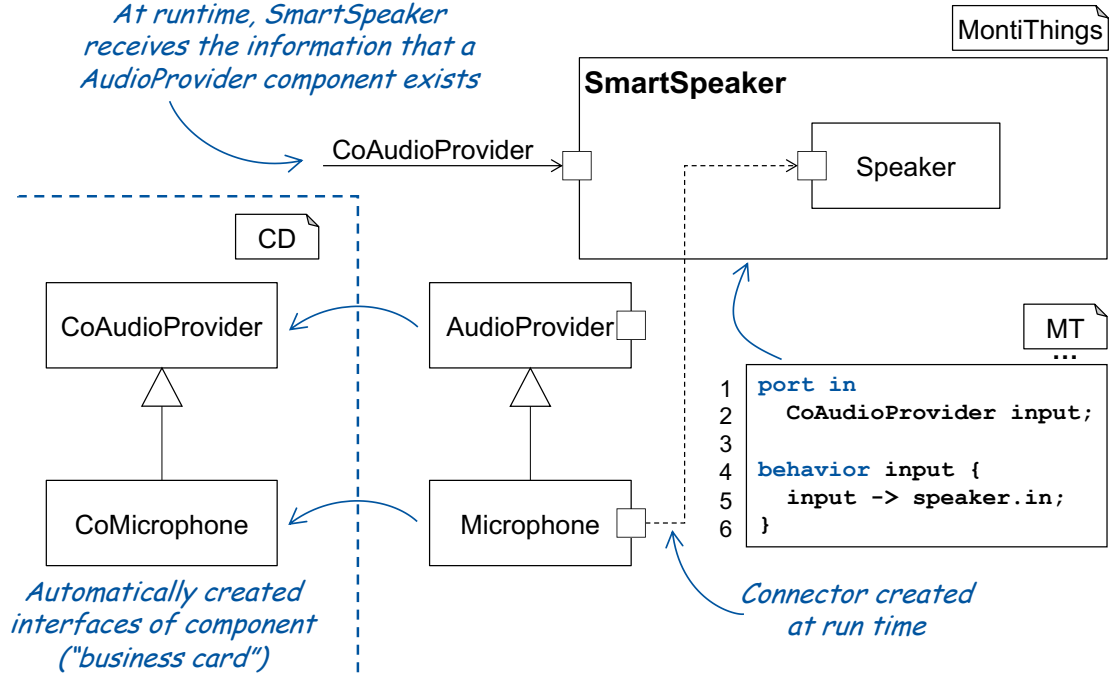


Figure 4.5: Example for using dynamic reconfiguration: A smart speaker gets connected to a microphone at runtime. Figure conceptually based on [KKR⁺22a].

can reduce wasting network resources by stopping to send messages to components that no longer exist. To prevent developers from introducing a behavior to composed components that is unrelated to reconfiguration, the behavior blocks of composed components are not allowed to send messages on the outgoing ports of the composed component or send messages to subcomponents. The only exception to this rule is that composed components are allowed to forward the received component interfaces to their subcomponents.

Fig. 4.5 gives an example of this dynamic reconfiguration using a smart speaker, *e.g.*, Apple Homepod or Amazon Alexa. In this example, the microphone is only added dynamically at runtime, so that it can also be provided, *e.g.*, by the microphone of a nearby smartphone. The SmartSpeaker component only knows that it wants to connect to an abstract AudioProvider component and accepts a CoAudioProvider object accordingly (lines 1-2 in the code excerpt in Fig. 4.5). The microphone component that is to be dynamically connected at runtime extends this AudioProvider component. Accordingly, CoMicrophone also extends CoAudioProvider and is thus accepted by SmartSpeaker. If the SmartSpeaker component receives a CoAudioProvider

object on its input port, it creates a new connector to its speaker subcomponent (lines 4-6 in the code excerpt in Fig. 4.5).

Dynamic reconfiguration is also considered by other IoT-focused C&C ADLs. Calvin [AP17, PA15, PA17] uses a similar mechanism for dynamics where components are instantiated based on rules. Unlike MontiThings, however, Calvin does not explicitly inform components about the appearance or disappearance of components. Instead, they only start sending messages on the same channels that might already exist and be also in use by other copies of the same component instance. Effectively, this restricts Calvin to allowing multiple instances of components within the predefined architecture. MontiThings offers a more expressive form of dynamics where components can also create new connections at runtime and composed components are aware of their subcomponents' new connectors by explicitly creating them.

CapeCode [BJK⁺18] offers a concept called *accessors*. Accessors act as proxies for hardware accesses or, more general, external services that shall be integrated into the C&C architecture. Accessors can also be sent via ports, including their source code. Thereby, receiving components are able to execute the received code. In contrast, MontiThings does not allow components to modify their behavior specification at runtime. Instantiating components in MontiThings is only needed to adapt to changes to the underlying hardware infrastructure. MontiThings' mechanism of only connecting to the new software components works similarly to using a received accessor as the behavior specification of an existing component. However, by not modifying the behavior of existing components, MontiThings makes it explicit that the connection to this new component is volatile because the hardware that triggered the new behavior is not guaranteed to exist forever. Since CapeCode uses JavaScript, it can also add completely new behavior at runtime. While technically MontiThings' implementation would allow instantiating components of component types unknown at design time⁶, MontiThings does not allow this. We think adding behavior specifications that were unknown at design time can make the overall business logic harder to understand and in some cases introduce security issues.

MDE4IoT [CS16] is a language-independent framework that considers dynamics with a focus on self-adaption. Since MDE4IoT is language-independent, it provides no specific mechanisms for dynamics. Its street light example treats dynamics only in the sense of a re-allocation of failed software components to new hardware components.

⁶For example, when using MQTT as a message broker (*cf.* Chapter 5), this could be achieved by instantiating new a Docker container containing software that first connects to the MQTT broker and then starts sending MontiThings-compatible messages.

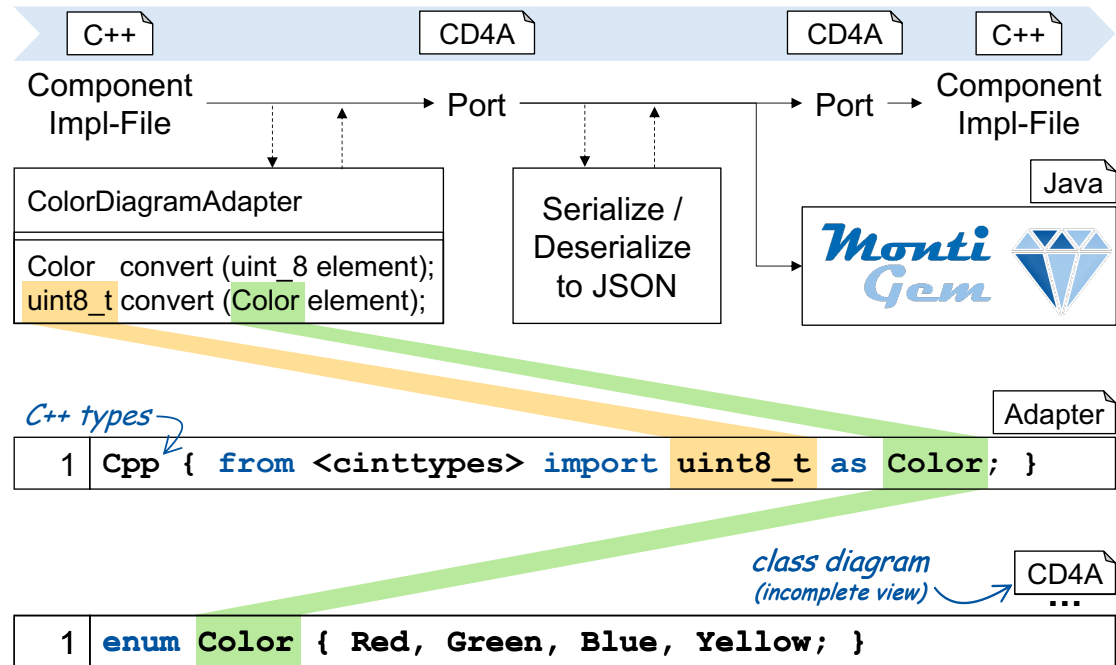


Figure 4.6: Adapters can specify that class diagram types shall be converted to a type of a target language to facilitate the integration with hand-written code. Figure taken from [KRSW22].

4.3 Language Integration

The MontiThings ecosystem consists of more languages than the ADL. This section describes their purpose and integration with the MontiThings ADL. Another standalone language for integrating MontiThings with digital twins is described in Sec. 6.6.

4.3.1 Integration With Class Diagrams

Class diagrams enable developers to specify the data types components exchange via ports. For this purpose, MontiThings uses the CD4Code language, a variant of the CD4Analysis language of the MontiVerse. As MontiCore’s symbol tables can be serialized into a standardized JSON format, MontiThings and the CD4Code project are independent of each other. MontiThings imports the serialized symbol tables from CD4Code models and thus makes the types defined by CD4Code models available to MontiThings’ type system. To not require developers to work with tools of different languages and know the order in which to process models, MontiThings’ generator and command line tool can also automatically serialize CD4Code files.

However, in some situations it is not appropriate to require developers to use generated classes, even if they can be customized using the TOP mechanism. Libraries of the target language (C++) build on types supplied by the language in addition to self-defined types. If developers want to use libraries, they must convert the generated classes thus into other types. To systematize this process MontiThings offers an adapter language. The principle of this language is shown in Fig. 4.6. The adapter language is a tagging language (*cf.* Sec. 2.3) that can be used to assign types of a target language to types defined in class diagrams. One can specify for a class diagram type which type of the target language it corresponds to and which package / header is necessary to import this type. These statements of the adapter language are based on the import statements of Python. For instance, the example in Fig. 4.6 specifies that the enum `Color` in the target language C++ could also correspond to the type `uint8_t` and that to use `uint8_t` the `<cstdint>` header must be imported.

For each class diagram that has an adapter, the generator creates an adapter class (here: `ColorDiagramAdapter`) that declares two `convert` methods for each adapted type. One of the methods converts from the class diagram type to the target language type, the other provides the reverse direction. These methods must be implemented by the developer via TOP mechanism. The generator then uses these methods to provide the converted types to the developer of the handwritten code in addition to the class diagram types, and to receive the converted types from the developer. For example, using the adapter in Fig. 4.6, each port of type `Color` could also be assigned a value of type `uint8_t`. MontiThings then takes care of converting the objects back to class diagram types before forwarding them to other model elements (*e.g.*, ports) or even other systems such as MontiGem.

4.3.2 Configuration Language

The main purpose of MontiThings is to create distributed IoT applications. Generating code for such applications may, however, require the generator to adapt the components to different hardware platforms. As such technical information is independent of the business logic and data flow described in the C&C architecture, such information can be added to components via a configuration language. This configuration language is implemented as a tagging language for the ADL.

The configuration language has the following capabilities:

1. Tag ports of the component with Freemarker templates. These templates contain C++ code that defines how to access hardware.
2. Tag ports with an MQTT topic. As an alternative to Freemarker templates, this topic will be used to communicate with an external connector that provides hardware access (Sec. 4.2.6).

3. Tag ports with their pull frequency. Some hardware may provide continuous inputs, *e.g.*, a voltage level. This frequency defines at which rate to discretize the values of these ports and create an event.
4. Tag components with their technical requirements. If components require external hardware, *e.g.*, a specific sensor, not every component can be deployed to every device. Chapter 6 further discusses how MontiThings uses this information to calculate which component shall be deployed to which device.
5. Prevent the automatic splitting of composed components into different binary partitions. Normally, MontiThings will generate code so that all component instances are executed in single binaries (*cf.* Chapter 5), as this gives maximum flexibility for deployment. This behavior can be undesirable for some components. This is the case, *e.g.*, if all subcomponents of a component are required to be executed on the same device (*e.g.*, all robot arms of a robot shall be deployed to the same device) or if deploying the subcomponents independently would create unnecessary communication overhead (*e.g.*, if a subcomponent only adds two values).

These functionalities may differ for different hardware platforms. For example, if a component is deployed to a device with a VL53L0X laser distance sensor⁷ accessed via the I²C bus, it may require different code than the same component being deployed to a device with an ultrasonic distance sensor accessed via a digital port. By enabling developers to configure components for different platforms, MontiThings accounts for this heterogeneity (**TC1**).

The configuration of Freemarker templates provides five different hook points⁸ for every port: *include*, *init*, *body*, *provide*, and *consume*. The *include* hook point can be used to specify a template that consists of C++ *include* statements for including the necessary headers needed by the implementation of the hardware access. MontiThings generator will include these headers at the beginning of all files that use the other four hook points. The code in the *init* hook point will be executed at the start of the application. This is offered because some sensors and actuators require certain setups to be performed before they can be accessed or make useful measurements. For example, the WiringPi Library⁹, a popular library for accessing the GPIO pins of Raspberry Pis, requires the function `wiringPiSetup()` to be called before the library can be used. The *provide* hook point needs to be filled with the code for providing a message to the architecture, *e.g.*, a sensor value. To do so, developers are required to call the `this->setNextValue(myValue)` exactly once in the template, where `myValue` refers to

⁷VL53L0X data sheet. [Online]. Available: <https://www.st.com/resource/en/datasheet/vl53l0x.pdf> Last accessed: 29.01.2022

⁸More information on MontiCore's hook point mechanism for Freemarker templates can be found in [HKR21].

⁹Wiring Pi Library. [Online]. Available: <http://wiringpi.com/>. Last accessed: 07.11.2021


```

1 config Scale for RASPBERRY {
2   weightLeft {
3     include = "HX711Include.ftl";
4     provide = "HX711Provide.ftl" (3, 4);
5     every 500ms;
6   }
7
8   weightRight {
9     include = "HX711Include.ftl";
10    provide = "HX711Provide.ftl" (5, 6);
11    every 500ms;
12  }
13
14  display {
15    mqtt = "oled-message";
16  }
17
18  separate none;
19
20  requires {
21    hardware: "HX711"; // weight sensor
22    hx711_count: 2;
23    hardware: "SSD1315"; // oled display
24  }
25 }

```

MTCFG

Listing 4.13: Configuration file for a scale component. Two weight sensors provide values via Freemarker templates on the weightLeft and weightRight ports. An OLED display receives messages via an MQTT topic.

the new value. Conversely, the *consume* hook point needs to be filled with the code for processing a message given by the architecture, *e.g.*, a command to an actuator. In this template, developers can access a variable called `nextVal` to access the message to be processed. The *body* hook point enables developers to specify code that will be placed in the body of hardware accessing classes. This can be beneficial for providing low-level helper functions such as bit shifts, that are only relevant to the hardware access and, thus, shall not become part of the architecture models that specify the business logic (MC1).

Listing 4.13 gives an example for configuring a component that implements a kitchen scale using MontiThings' configuration language. The corresponding component contains two incoming ports (`weightLeft` and `weightRight`) and one outgoing port (`display`). The two incoming ports shall be connected to HX711 load cell amplifiers, which we can consider as weight sensors for this example. The outgoing port shall be

connected to an OLED display that shows the current weight placed on the scale. The first line specifies that this configuration of the `Scale` component applies to the RASPBERRY platform, *i.e.*, Raspberry Pis. Configurations for other platforms would likely include different templates tailored to the specific platform. Lines 2–12 specify accessing the load cells. One important detail to notice here is that the `provide` template is given two arguments. This enables developers to reuse templates for different hardware configurations. In this case, we have two instances of the same load cell. Both of them, in general, can be accessed using the same code. However, they will necessarily be connected to different GPIO pins. Using the arguments, developers can reuse the same code template and set the pin numbers to the ones provided to the template. MontiThings uses MontiCore’s *GlobalExtensionManagement* to set global variables the corresponding values which developers can retrieve inside the template. Moreover, the two incoming ports get assigned a pull frequency of 500 ms in lines 5 and 11. This causes MontiThings to request a new value every 500 ms using the code in the *provide* template.

The display in lines 14–16 is configured to use an MQTT topic. Whenever the component sends a new message on this port, a message will be published to the specified topic of an MQTT broker running locally on the device that executes the component. External connectors (*cf.* Sec. 4.2.6) can use this message to further access the requested hardware. In this example, this message would include a message to be displayed on the OLED display. Since it makes no sense to place the two weight sensors on different devices, line 18 specifies that all subcomponents of the `Scale` component need to be deployed to the same device. Technically, this causes MontiThings to create a single binary for the `Scale` component and all of its subcomponents.

Lastly, lines 20–24 specify the technical requirements of the `Scale` component. Devices executing this component are required to have both HX711 and SSD1315 devices because that is the hardware the code templates are written against. Furthermore, there need to be at least two HX711 sensors present on the device. The configuration language does not restrict developers in the naming of further technical requirements. For example, developers could also choose to specify `color: "green"` to only deploy the software to green devices. Whichever requirements are specified here, however, need to be fulfilled by the devices in their configuration (Chapter 6).

Configuration mechanisms like the one presented in this section can also be found in many other ADLs: MontiArcAutomaton [Wor16] uses interface components for tailoring components to different platforms. The architecture uses only platform-independent interface components. Using a *binding*, *i.e.*, a mapping from platform-independent interface components to platform-dependent components, the interface components are replaced shortly before generating code. A downside of this approach is that it may lead to a high degree of code duplication and, thus, low maintainability if platform-specific components only differ in a few details. ThingML mostly allows to mix platform-specific details and platform-independent logic using annotations for different target

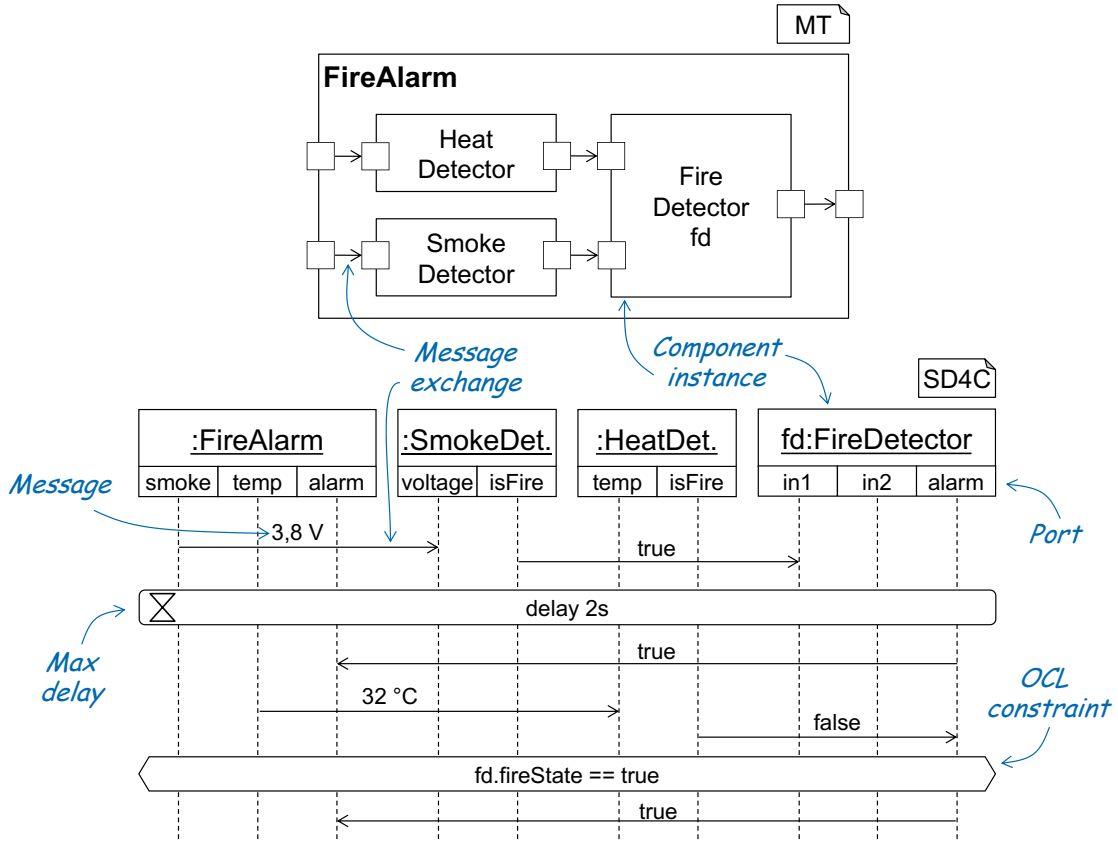


Figure 4.7: Sequence diagram specifying a white box test of a fire detector. The graphical syntax of placing ports below components is taken from [HNPR13].

languages (e.g., `@c_byte_size "1" @c_type "uint8_t"`¹⁰) and platforms (e.g., `@pim "Arduino" @platform "arduino"`¹¹). Such annotations, however, make the models harder to understand for non-experts. ThingML *configurations* provide means for instantiating and connecting components for different use cases. Eclipse Mita offers a platform description language¹². This language describes the capabilities of a certain platform. Developers are expected to provide an appropriate generator for each such

¹⁰Full code example: <https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/hardware/arduino/arduino.thingml>. Last accessed: 04.11.2021.

¹¹Full code example: https://github.com/TelluIoT/ThingML/blob/master/org.thingml.samples/src/main/thingml/hardware/arduino/_arduino/arduino.thingml. Last accessed: 04.11.2021.

¹²Eclipse Mita Documentation. [Online]. Available: <https://www.eclipse.org/mita/platforms/integratorsguide/>. Last accessed: 04.11.2021

```
1 testdiagram ShouldAlarmOnHighSmoke for FireAlarm {
2   -> inPort : 3.8V;
3   inPort -> smokeDetector.voltage : 3.8V;
4   smokeDetector.isFire -> fd.in1 : true;
5
6   delay 2s;
7
8   fd.alarm -> alarm : true;
9   temp -> heatDetector.temp : 32°C;
10  heatDetector.isFire -> fd.alarm : false;
11
12  assert fd.fireState == true;
13
14  fd.alarm -> alarm : true;
15 }
```

Listing 4.14: Textual representation of the test specification from Fig. 4.7.

platform. Compared to this approach, MontiThings is more easily extendible, as developers can provide their own implementation for specific sensors without having to adapt a generator.

4.3.3 Sequence Diagram Test Specification

An important aspect of creating reliable applications is testing them before they are released. MontiArc already provides a stream test language for black-box testing of components [Hab16]. However, for white-box tests, MontiArc requires users to write test code against the generated code [Hab16]. MontiThings fills this gap with its *SD4ComponentTesting* language, an extension of the MontiVerse’s sequence diagram language¹³. The idea of using sequence diagrams as test specifications has also been briefly proposed in [HNPR13]. The SD4ComponentTesting language extends this idea, *e.g.*, with delays and a more expressive type system including SI units. The generation of test code from these models is described in Sec. 5.3.5.

Fig. 4.7 shows a graphical example of how the SD4ComponentTesting, SD4C for short, can be used to specify a test scenario. Each component instance is represented by a box at the top of the diagram. Below these boxes, smaller boxes are placed for the ports of these components. Message exchange can be specified between the ports by means of a horizontal arrow. The message content is written to the arrow. Message exchange between ports that are not connected in the architecture model cannot be fulfilled by the architecture and is therefore prohibited by a CoCo. A rounded box with

¹³Sequence Diagram Language. [Online]. Available: <https://github.com/monticore/sequence-diagram>. Last accessed: 07.11.2021

an hourglass denotes a delay. The test thus expects all message exchanges before the delay to be finished after the delay. Additionally, diamond-shaped boxes can be used to denote assertions using (OCL) expressions. In this example, a smoke sensor is read out with a value of 3.8 V. The smoke detector derives from this value that there is a fire and informs the smoke detector that forwards this information after at most 2 s to its enclosing component. Shortly after that, a heat detector reports a temperature of 32 °C. However, since the smoke detector reported a fire, the state of the fire detector remains unchanged, which is assured by an assertion. The fire detector then reaffirms its enclosing component that there still is a fire.

The textual representation of this example is shown in Listing 4.14. The textual specification first defines a name for the test specification and the component to which this test specification applies. Message exchanges are denoted by an arrow (\rightarrow). The sending port is in front of the arrow, one or more comma-separated receiving ports are behind it. The message content is at the end of a statement after a colon. Incoming ports of the component to which the test applies may receive values without an explicit sender. In this case, the messages are test input provided by the test environment. Similarly, outgoing ports of the component for which the test is specified do not need to specify a receiver. The test environment will then ensure that the port gets the specified value. Delay statements (line 6) can specify a delay using SI units. Assert statements (line 12) can specify assertions using (OCL) expressions. For the test case to be fulfilled, all message exchanges need to be executed in the order specified by the test case specification. If a message exchange is either not executed or with a different value than specified, the test case is considered a failure. Also, if an assertion is violated, the test case fails.

4.4 Discussion

MontiThings features a clear separation of concerns. This is especially visible when separating technical aspects of IoT applications from the business logic of the application. For example, the ThingML language uses annotations in the models to control certain aspects of the code generation for its target languages. This includes annotations like `@c_type "const char*"` that control how the generator should map model elements to types of the target language [KRSW22]. In contrast, MontiThings models do not contain any C++ code.

In early versions of MontiThings [Für20], MontiThings also contained technical information in the models. This included, *e.g.*, communication specific information such as IP addresses of communication partners. This, however, made the models dependent on a specific target infrastructure. Therefore, we decided against using such information in the models. Instead, it is up to the code generator and the run-time environment to establish communication between the devices.

To still be able to specify some technical information about the components, MontiThings uses a configuration language, *i.e.*, a tagging language, that is separate from the modeling language of the components (see Sec. 4.3.2). This configuration language also does not include technical code, *e.g.*, C++, but only enables developers to specify *which* code shall be used. This information can be customized for different platforms. Besides not polluting the component models with technical information, this method has mainly two advantages: 1. It enables technical developers to implement technical aspects of the code, *i.e.*, sensor drivers, largely independent of the component. For example, they can also implement the drivers without knowing already which components' ports will use the driver. Similarly, the component can also be developed without the need for the drivers to already exist. 2. By not being able to directly access the component from the driver, we prevent developers from writing component-dependent drivers. This facilitates reusing the technical code across different components. Similarly, it also facilitates reusing the components in different contexts. As the technical code is based on C++, it is possible to reuse already existing drivers. Often, the Freemarker templates used by MontiThings only contain wrappers for already existing driver libraries by hardware vendors. The Freemarker templates then only specify how to read out or write single values. The reusability of components is further facilitated by the inheritance-like override mechanism for ports that are tagged with Freemarker templates (*cf.* Sec. 4.2.6).

Tagging ports of components to connect them to different middlewares has already been proposed by [HKKR19]. Their work specifies to which topics a certain port shall connect. Similar to our approach, component models are agnostic of the middleware. Their work did, however, not specify user-provided Freemarker templates. Therefore, users are limited to the middlewares provided out of the box or need to extend the generator themselves. Similarly, developers who use the IoT language Eclipse Mita need to write a new generator when they want to include a new platform. By enabling users to tag model elements with Freemarker templates, users can effectively extend the generator without having to modify the generator's code or, in extreme cases, write their own generator. This is, naturally, limited by the hook points offered by MontiThings. Adding new hook points requires modifying the generator.

Overall, MontiThings enables connecting to external software using every language that supports MQTT and can format its messages in the JSON format MontiThings uses to serialize messages. Besides C++ (via Freemarker templates), MontiThings also offers an adapter for Python. By supporting both C++ and Python, MontiThings already supports large parts of the available IoT sensor and actuator drivers with only minimal efforts for writing a wrapper. By making the connectors to such sensors not only independent from the component on a code level but also on a binary level, these external connectors can be started and stopped independently of the components. This supports our vision of an IoT app store, where each IoT device provides access to certain hardware capabilities out-of-the-box that IoT applications can connect to. In other words, the separation between components and technical code can go as far as letting other vendors

provide the connectors to the hardware. Using MQTT as a shared protocol, neither the configuration files nor the generator need to know the programming language the external connectors are programmed in. The disadvantage of this level of flexibility is that fewer checks can be performed at generation time. For example, if the format of the messages sent by the external connector does not match the format expected by the component (*e.g.*, integer vs. string content), the component will detect this kind of misconfiguration only at runtime and inform the developers of the component.

Besides automatically detecting misconfigurations at runtime, MontiThings also offers mechanisms to enable developers to specify how to detect programming or modeling errors at runtime using pre- and postconditions using OCL expressions. While this enables MontiThings to detect low-level errors, there are also types of errors MontiThings cannot detect [KRSW22]. MontiThings does not feature an in-depth anomaly detection as desired by future IoT systems [MNZC20]. Such anomaly detections go beyond the scope of this thesis but could be added as a service (*cf.* step 6 in Sec. 3.8). Especially, MontiThings offers no mechanisms for detecting anomalies that might be caused by security issues (*cf.* Sec. 3.7). Furthermore, while MontiThings enables developers to inspect the data exchanged between external connectors and the components, it does not enable developers to detect problems with the hardware itself without polluting the business logic as components are abstracted from the devices they are executed on. For example, developers can not check if the central processing unit (CPU) load is high or the storage is filling up. Also, developers cannot detect problems that go beyond the scope of a single component. Overall, we recommend that error handling should be done at different stages of the development. While simple error checks like MontiThings' pre- and postconditions can be within the model, other problems such as device failures can be handled better at different stages of the development/deployment. Some techniques for this will be presented in the following chapters.

Since MontiThings components are not aware of the IoT device they are executed on, the MontiThings language does not offer mechanisms for controlling the device. This includes, *e.g.*, setting the device to sleep mode. For dependent on battery power (that do not use any other energy harvesting techniques), going into a sleep mode is crucial for reaching a long battery life. Of course, IoT devices in MontiThings applications can go to sleep modes. For example, the DSA vehicle connectivity gateway (VCG), which is one of MontiThings' target platforms, can go into sleep mode after a specified number of seconds if it is not connected to an external power supply. This behavior is similar to uninterruptible power supply (UPS) shields offered for the Raspberry Pi¹⁴. MontiThings components can bridge such outages (the mechanisms for bridging temporary failures are discussed in the following chapters). However, one potential extension of the language would enable modelers to take energy into consideration. For example, the components

¹⁴Joy-IT StromPi 3 product website. [Online]. Available: <https://joy-it.net/en/products/rb-strompi3>. Last accessed: 09.01.2022

could inform the IoT device that executes them when they are in a *good* situation to go into sleep mode or when to avoid going into sleep mode. Further priority mechanisms could enable developers to specify how important a component is for the overall system. Managing energy is, however, currently out of the scope of MontiThings. Such mechanisms ideally would not be strict but enable the IoT devices to make informed decisions based on both the components' information and the devices' state, *e.g.*, left battery power. In combination with managing the deployment (*cf.* Chapter 6), such mechanisms could then be used to implement functionalities similar to predictive maintenance. For example, an IoT device could inform the system that it wants a different device to execute one of its components because the component is important and the device expects to run out of power soon.

Table 4.2 compares MontiThings' ADL to other IoT-focused languages. It uses the following criteria:

Language Type classifies what type of language is examined. Language Families like UML contain multiple languages.

Base Language If the language is directly based on another language or is strongly influenced by other languages, this language is indicated here.

Target/HWC Language Tells to which language models of the examined language can be compiled. This is usually also the language in which hand-written code can be provided (if hand-written code can be provided at all).

Type System examines whether the language uses a static or dynamic type system. Static type systems check types at compile time, dynamic type systems do so at run time.

Language Features Checks whether languages offer 1. OCL expressions, 2. SI units in the type system, 3. SCs, and 4. exception handling.

Compared to other IoT languages like ThingML, Calvin, or Eclipse Mita, MontiThings provides a powerful type system and behavior specification. By including SI units in the type system and automatically converting between them, MontiThings makes the intent of SI unit types more explicit. In contrast, other languages usually specify SI types using their encoding format or value, *e.g.*, `string` or `double`. By making SI units part of the type system, MontiThings can detect errors at design time that other languages cannot find, *e.g.*, an illegal conversion between *km/h* and *kg*. By including the OCL expressions and set expressions, MontiThings' behavior language goes beyond the capabilities of many other (programming or modeling) languages. This expressiveness of MontiThings is mainly made possible by the integration of languages of the MontiVerse

Table 4.2: Overview of related IoT modeling methods. ● = fulfilled, ◐ = partly fulfilled, ○ = not fulfilled.

IoT Language	Language Type	Base Language	Target/HWC Language	Type System	Language Features			
					OCL	SI Units	SCs	Excep. Hand.
Arduino	Prog. Lang.	C/C++	C++	Static	○	○	○	●
AutoIoT [NCM ⁺ 20]	JSON	JSON	Python	Dynamic	○	○	○	○
CapeCode [BJK ⁺ 18]	C&C ADL	Ptolemy II [Pto14]	JavaScript	Static ¹	○	◐ ²	●	● ³
Ericsson Calvin [AP17]	C&C ADL	N/A	Python	Dynamic	○	○	○	○
Eclipse Mita	Prog. Lang.	JavaScript, Typescript, Java, Swift, Go	C	Static	○	○	○	●
FRASAD [NTBG15]	Rules DSL	N/A	C	unclear ⁴	○	○	○	○
MDE4IoT [CS16]	UML	(f)UML/ALF/MARTE	C++ [CCS15]	Static	○	● ⁵	● ⁵	●
Node-RED	C&C ADL	N/A	JavaScript	Dynamic	○	○	○	●
SysML4IoT [CPD16]	Lang. Fam.	SysML	N/A	Static	○	○	● ⁶	●
STS4IoT [PBS ⁺ 22]	UML Prof.	UML	C ⁷	Static	◐ ⁸	○	○	○
ThingML [HFMH16, MHF17]	C&C ADL	N/A	Java, JavaScript, C, Go	Static	○	○	●	○
MontiThings (this thesis)	C&C ADL	MontiArc	C++ ⁹	Static	●	●	●	◐ ¹⁰

¹ The authors of Ptolemy II consider it to be statically typed, because “types are checked just prior to execution of a model” [Pto14].

² Units are part of the expression system. Using the `InUnitsOf` actor, developers can convert between units [Pto14].

³ Uses an implicit port named *error*

⁴ The generated C code is statically typed. It is unclear what type system the “rule interpreter” uses.

⁵ Because MDE4IoT enables developers to use MARTE.

⁶ Because SysML offers SI units and statecharts

⁷ Only for devices using the “Simple Embedded Operating System” by LIMOS

⁸ Not part of the code generator. Code for OCL expressions has to be hand-written.

⁹ For external connectors, every language that support MQTT can be integrated. MontiThings explicitly supports Python for external connectors.

¹⁰ MontiThings has pre- and postconditions that can be handled if violated. Moreover, developers can use try/catch statements handwritten C++ code. However, MontiThings statement language does not offer exception handling to encourage developers to separate error handling from business logic.

language library¹⁵. By offering a wide variety of behavior definition methods ranging from high-level statecharts to low-level C++ code, developers can choose the appropriate level of abstraction for each component.

Overall, the expressiveness of the MontiThings language enables the language to detect many errors at design time. MontiThings also enforces several rules that are mainly usage conventions of the language. For example, MontiThings enforces that each component is defined in its own file¹⁶. The grammar of the MontiThings language could, of course, be easily extended to allow multiple component definitions in the same file but we think this would lead to harder-to-maintain projects as components because components cannot be found by their file name and once a component shall be reused, it has to be moved to its own file anyways to avoid copy-paste reuse. The disadvantage, however, is that MontiThings can feel frustrating for new developers. Compared to languages such as Python, that do not check many errors at design time, *e.g.*, not checking illegal type conversions, developers are likely to face a large number of error messages when first starting to work with MontiThings which can feel frustrating. Also, combining multiple languages (MontiThings components, class diagrams, configurations) can feel overwhelming at first. As a result, compared to Python or Arduino, writing “quick and dirty” applications is harder. We argue that the steep learning curve is justified by the long-term advantages of adhering to good practices.

Overall, MontiThings shows what a modeling language for IoT applications can look like. Compared to existing languages, MontiThings offers a stronger type system, different levels of abstraction for behavior specification, and a clear separation of concerns. Concepts like the inheritance-like override mechanism for ports also facilitate reusability. The following chapter presents the technical aspects of the MontiThings language, *i.e.*, its code generation capabilities.

¹⁵This includes both language components that were (co-)developed by the author of this thesis, *e.g.*, OCL, and languages that the author of this thesis was not involved in and that were developed independently of MontiThings, *e.g.*, SI units.

¹⁶As pointed out by [JBD21], using file-level modularity instead of a single file can improve tool performance (*e.g.*, saving and loading operations), supports collaboration between modelers, and “facilitate[s] MDD-specific activities such as model comparison and incremental code generation” [JBD21].

Chapter 5

Code Generation

The development of IoT systems involves many repetitive and error-prone programming tasks. This includes, for example, establishing communication between IoT devices. MontiThings can free developers from many such tasks by generating code that can be used to execute the models specified using the MontiThings language. This chapter presents the code generator, the RTE of the generated applications, and the structure of the generated artifacts in more detail. Unlike MontiArc, MontiThings does not generate code for simulations but prototypes of distributed applications that can be executed on real IoT devices. Such distributed applications consist of multiple binaries that may be deployed to different devices (Chapter 6) and interact with each other via some form of network. For this purpose, MontiThings generates C++ code and scripts to cross-compile the code and package it as container images to make it easily deployable.

5.1 Methodology and Tool Infrastructure

MontiThings uses code generation to create concrete prototypes of IoT applications from models. The models operate at a substantially higher level of abstraction than general-purpose programming languages such as C++, Java or Python. By raising the level of abstraction, the code generator aims at supporting developers by relieving them of repetitive implementation tasks. As described in the previous chapter, these models take into account different aspects, from the description of the business logic (MontiThings models), to the data structures (class diagrams), to associated tagging languages that supplement the models with additional information such as synchronization specifications for digital twins (*cf.* Sec. 6.6).

Fig. 5.1 gives an overview of MontiThings' model-driven methodology for creating distributed applications. At design time, developers create various platform-independent artifacts. These mainly include the MontiThings architecture models explained in the previous chapter. These MontiThings models can be integrated with other models (*cf.* Sec. 4.3): Class diagrams can be used for specifying the data types exchanged by ports. Tagging models can be used for further adapting the generated code to the developers' needs on a model-specific level. For example, tagging can be used to instruct pre-generation model-to-model transformations to enrich the models with additional func-

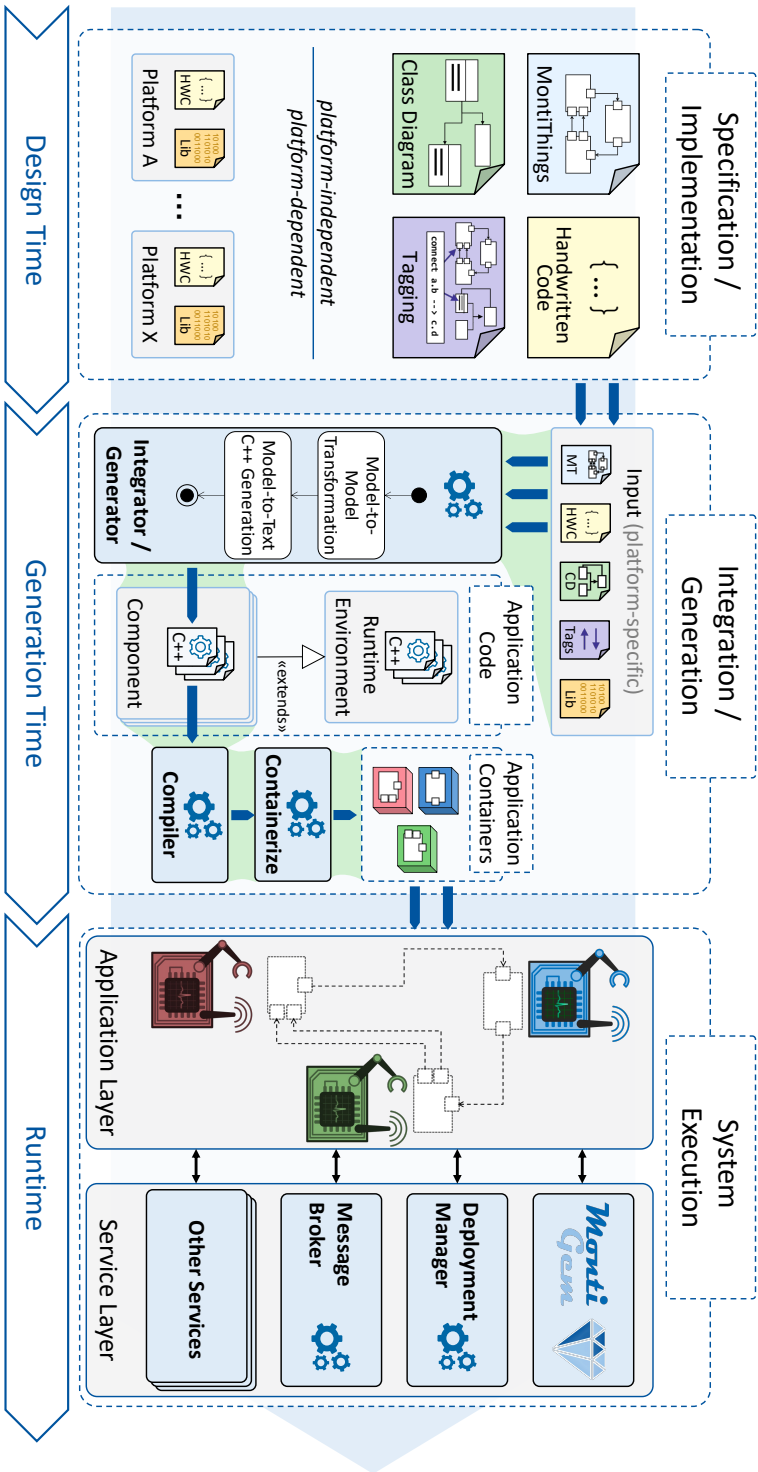


Figure 5.1. Methodology of developing IoT applications using MontiThings. Figure taken from [KRSW'22].

tionality, *e.g.*, model elements can be created that automatically synchronize with a digital twin.

To customize generation for specific target platforms, developers can also create platform-specific artifacts at design time. If developers target specific target platforms, the code for interacting with the target platform may need to be very specific for said target platform. The specifics of the target platforms cannot be predicted by the generator. Therefore, developers can provide the generator with libraries and handwritten code for each target platform. For example, on a Raspberry Pi the GPIO pins might be accessed using the Arduino-like WiringPi library while when using the DSA VCG¹, developers use the diagnostic data cache (DDC)² instead.

All of these models and code files are provided as input to the generator at generation time (top center of Fig. 5.1). Depending on its configuration, the code generator can apply model-to-model transformations before generating C++ code from the models. Such transformations can, *e.g.*, be used to create variants of the architecture that can be used to replay a recorded run of the application (*cf.* Sec. 7.5). Using a set of Apache Freemarker templates, the code generator executes a model-to-text C++ generation. For each component, multiple C++ files are generated that build upon the RTE's code. This generation and the RTE are described in more detail in the following sections. For ease of use, MontiThings also offers build scripts for this purpose. The binaries created from the code can also be packaged in Docker images to make the deployment easier. This step can also automatically be performed by the scripts generated by MontiThings. The resulting application containers can optionally be uploaded to a standard Docker image registry to make them available to the IoT devices. The Docker containers instantiated from these images will then communicate via one of the supported network protocols. The default network protocol is MQTT.

To make the compilation easier for developers, MontiThings also provides the toolchains for compiling the code as two Docker images, `montithings/mtcmake` and `montithings/mtcmakedds`. The latter is significantly larger and shall only be used when the generator is configured to use DDS for communication. Using these images, developers can use MontiThings with Docker being the only dependency to the developer's computer.

At runtime, the IoT devices may communicate with various services. For example, the Deployment Manager decides which devices shall instantiate which components (Chapter 6). Other services include, *e.g.*, a synchronization with MontiGem-based digital twins Sec. 6.6, a recording module used for creating replays of the architecture execution (Sec. 7.5), or a live debugger (Sec. 7.4).

¹DSA VCG product website. [Online]. Available: <https://www.dsa.de/en/solutions/products/vehicle-connectivity-gateway/>. Last accessed: 11.11.2021

²The DDC is, essentially, a key-value store that can be used to access various hardware functionalities and configurations.

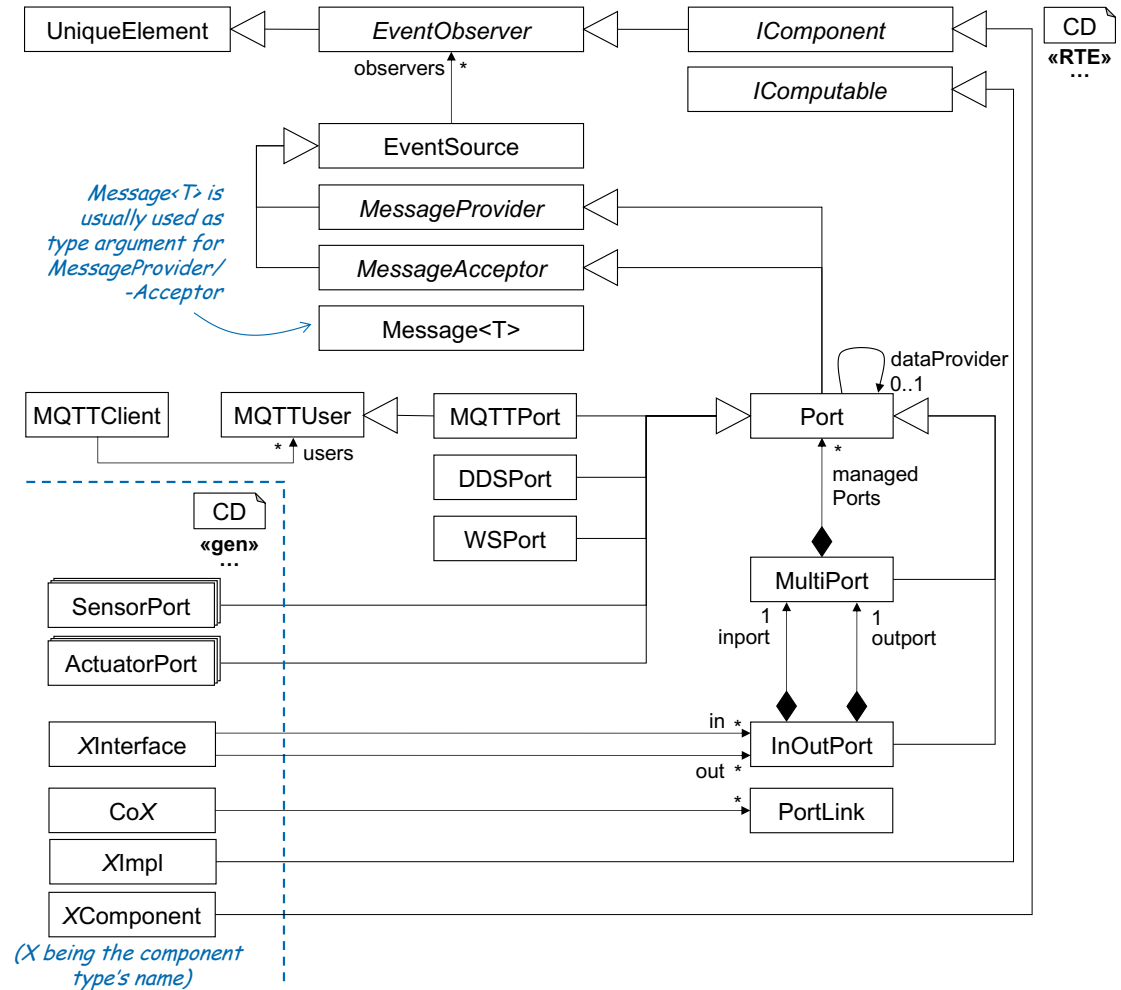


Figure 5.2: Overview of MontiThings' RTE. Figure is an extended version of the overview from [KRSW22].

5.2 Run-time Environment (RTE)

The runtime environment provides functionalities that are the same for all target platforms. This includes in particular implementations for the basic functions of each component, such as communication via ports. Fig. 5.2 gives an overview of MontiThings' RTE. The **UniqueElement**³ provides a C++ equivalent of Java's Object base class.

³To make this chapter easier to read and to use as a reference manual for the RTE, class names are printed in bold at the place where they are explained in the text.

It contains a single protected attribute `uuid` and a public getter for this attribute. Thereby, it assigns an identity to all classes that inherit from it.

5.2.1 Components and Event-Handling

IComponent provides four methods that shall be implemented by the specific class generated for each component type: `setup`, `init`, `start`, and `compute`. The `setup` method is called before the component can be executed. It is responsible for setting up connectors of subcomponents, configuring the network connections of ports (if necessary), and announcing to other components that a new component instance will join the architecture. The last part is necessary because the order in which the binaries are started on the different devices cannot be guaranteed in a distributed application without much overhead. If a composed component is started before (some of) its subcomponents are started, the composed component will in response to this announcement of a new subcomponent need to ensure the connectors of the new subcomponent get established on the network. The `init` method is called before the component can start its normal event-based operation. It can be used for sending initial messages (Sec. 4.2.4). The `start` method is called to start threads with which the component will execute cyclic behavior (*e.g.*, from every blocks). The `compute` method of a component is the entry point for executing the behavior when a new message is received. It first ensures that no two behaviors are being executed concurrently within a component, to avoid race conditions in the behavior blocks. Then, it checks that all preconditions are met and saves the current state of the component so that it can be referred to by `@pre` expressions (Sec. 4.2.5). It then calls the behavior implemented by the corresponding `IComputable` subclass.

The **IComputable** contains two methods: `getInitialValues` and `compute`. The `getInitialValues` method can be used to send out initial messages on the component's outgoing ports. The `compute` method contains the event-based behavior of the component that will be executed in response to incoming messages. It can be considered an application of the strategy pattern [GHJV95], where the generator implements this interface in a class that contains the actual behavior of the component.

The internal event-based control flow of every component is using the observer pattern [GHJV95] implemented by the `EventSource`, the `EventObserver`, and their subclasses. The **EventSource** offers two methods `attach` and `detach`, each of which takes an `EventObserver` object as an argument. These methods are used to add and remove objects from the set of observers that shall be notified when an event happens. The `notify` method triggers calling the observers. Optionally, the `notify` method takes an argument that suppresses informing a specific observer. This can be used to avoid notifying an observer that created the event itself. The **EventObserver** only contains a single untyped method `onEvent` that is called by the `notify` method of an `EventSource`. Classes implementing the `EventObserver` interface need to imple-

ment this method to specify how to react to events. For example, components will call their compute method when they get notified about a new message.

The RTE contains two EventSources: The `MessageProvider` and `MessageAcceptor`. The **MessageAcceptor** provides a method `setNextValue` to accept a new message. Setting a new message will trigger an event in all implementations of this class. Like its counterpart, the **MessageProvider** can be used to access received messages. Its `hasValue` method is used for checking if the `MessageProvider` has an unprocessed message. The `getCurrentValue` will return the next message (or an empty optional if there is no message) and remove that message from the queue of unprocessed messages. `MessageProviders` keep track of which receiver already processed which messages. This enables `MessageAcceptors` instantiated on devices with different computing powers to process messages at different speeds (**TC1**) and to bridge outages of components (**TC2**). The **Message** class is a wrapper for messages exchanged by `MessageProviders` and `MessageProviders`. It enables the serialization of the message into a JSON format that can be exchanged via a network.

5.2.2 Ports and Communication Technologies / Protocols

The base class for every port is the **Port** class. Ports inherit the functionality to send and receive messages from the `MessageAcceptor` and `MessageProvider` classes and the functionality to create events from the `EventSource` class⁴. Besides, each `Port` may also have a `dataProvider`. The `dataProvider` is another port that provides messages to this port. Since the device executing a component instance may fail at any time, both the incoming port and the outgoing port of each connector have a message buffer. The buffer in the outgoing ports is needed to bridge outages of the receiving port. The buffer in the incoming ports is needed to buffer messages while the processing of the previous messages is still running. As the outgoing and incoming ports may be located on different devices, this functionality cannot be implemented using a single buffer. The `updateMessageSource` method of each port is used to transfer the messages from the `dataProvider` to the port calling this method. This data transfer is triggered by the event that a new message is available. While a component is offline, it may miss the event that its data provider has new messages available. Therefore, a port will also query its data provider, *e.g.*, when it has no messages available but is requested by the architecture to provide the current message. Without querying the data provider, the `hasValue` and `getCurrentValue` would in some cases incorrectly answer that no message is available because the data transfer from the `dataProvider` to the queried port has not yet been executed.

⁴Components only observe the events of their incoming ports to trigger their behavior. Nevertheless, it makes sense that no distinction is made here between incoming and outgoing ports because in other contexts the events of outgoing ports may also be relevant. For example, in the case of test cases, spy objects may also be interested in the events of outgoing ports (*cf.* Sec. 5.3.5).

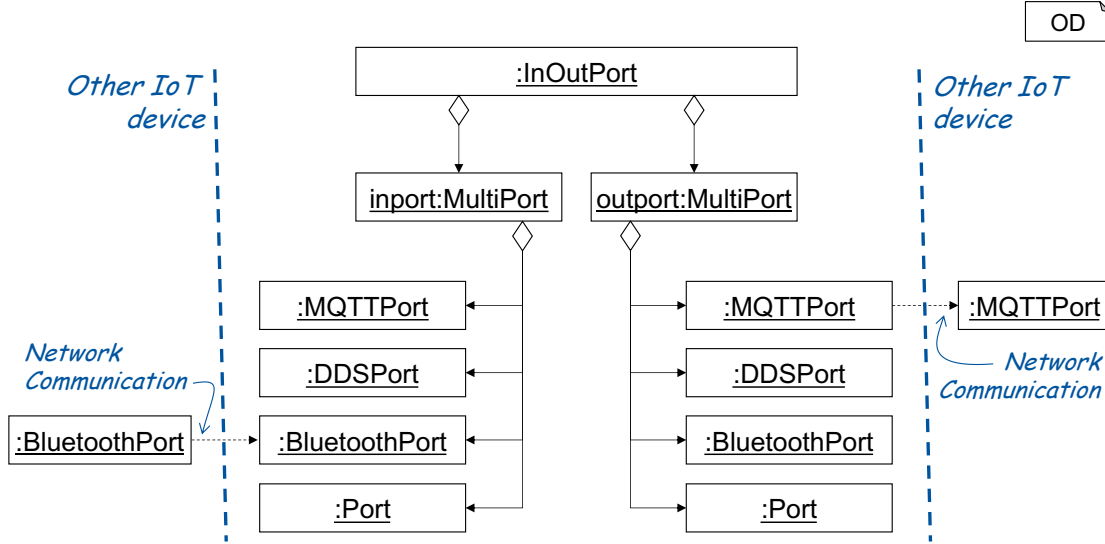


Figure 5.3: InOutPorts forward messages and can translate between different communication technologies. In this case a message is received via Bluetooth and forwarded via MQTT. MultiPorts defer the decision which communication technology to use to runtime by providing multiple alternative implementations for the same port. The BluetoothPort is only shown for better understanding and not part of the RTE.

The `Port` class is the basic extension point for implementing different communication technologies or for communicating with external connectors (Fig. 4.3). MontiThings RTE itself offers three main communication technologies implemented by subclassing the `Port` class: The **MQTTPort** for communication using MQTT, the **DDSPort** for using the OpenDDS implementation of the OMG data distribution service (DDS) [Obj15] standard, and the **WSPort** for communication using web sockets. Fig. 5.2 shows exemplary for MQTT how technology-dependent subclasses of the `Port` class act as a facade [GHJV95] to hide away the more complex access to the different technologies from the rest of the RTE. Here, the **MQTTUser** and **MQTTClient** provide wrappers for initializing and accessing an MQTT communication library. Similarly, the generated can contain different port types for accessing sensors and actuators, called **SensorPort** and **ActuatorPort** in Fig. 5.2, using the developer-provided code templates.

A **MultiPort** is a wrapper for multiple ports acting as one port. Its purpose is to enable multiple ports to connect to the same port using different communication technologies (Fig. 5.3). This means that it is not necessary to determine at design time which communication technology is available at runtime or offers the best communication conditions. If several communication technologies are supported by the hardware,

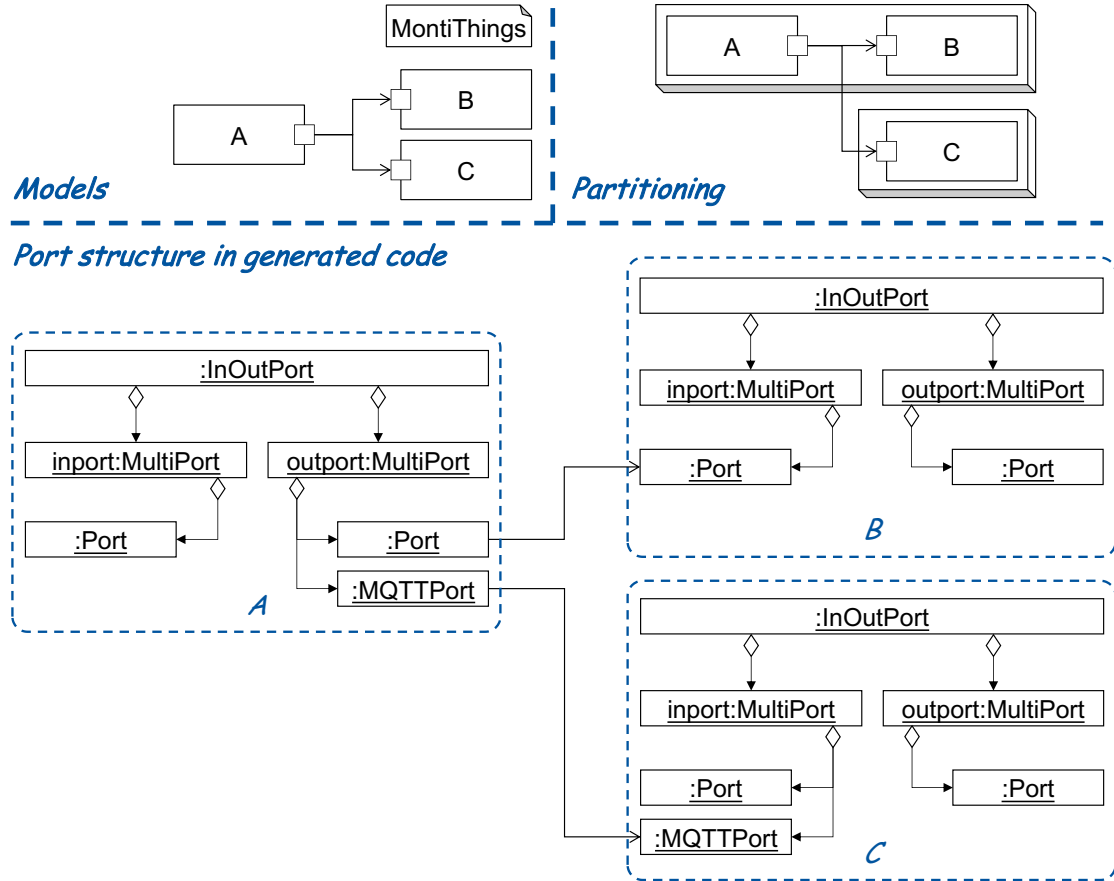


Figure 5.4: Example for the object structure created when connecting ports.

an automatic fallback mechanism could also prevent failures⁵. For example, in the event of a Bluetooth channel interference, the system could automatically switch to cellular communication (which, unlike Bluetooth, uses guaranteed frequency ranges). For example, the same port could be connected to a port of a component within the same binary and to the port of a different component instance executed on a different device. Sending messages between the first pair of ports can be implemented most efficiently by just copying the messages in RAM without involving any networks communication. The second pair of ports, however, needs to serialize the message and send it via some kind of network protocol and technology, *e.g.*, MQTT over Ethernet. Since different communication technologies are realized using different subclasses of the `Port` class, `MultiPort` bundles multiple ports in a single class. Besides two additional methods

⁵Note that such an extension is possible but not implemented by MontiThings.

for adding and removing ports to the set of managed ports, the `MultiPort` offers the same interface as a regular port. Semantically, a `MultiPort` has data available if any of its managed ports has data available. If data is sent using a `MultiPort`, it will be sent on all managed ports.

The **InOutPort** enables components to receive and forward data. It consists of two `MultiPorts`, one representing the input and one representing the output. This is necessary for composed components that forward data to their subcomponents using ports. In each of the two `MultiPorts`, the `InOutPort` automatically adds an instance of the normal `Port` class so that internal communication is always possible. `InOutPorts` use `MultiPorts` to support increased flexibility in choosing communication technologies and even allowing to switch communication technologies at runtime. This way a composed component can act as an adapter between communication technologies (Fig. 5.3), *e.g.*, forward messages received via an `MQTTPort` to a component using a `DDSPort`. While ports allow only one data provider, the `InOutPort` also uses a `MultiPort` on the input side. This is done to enable a seamless transition between communication technologies at runtime in case components fail. For example, the component containing the data-providing port may first be deployed to a device that uses DDS for communication. Later, this device fails (**TC3**) and the deployment manager needs to restore this component to a different device using a different communication protocol, *e.g.*, MQTT. In this scenario, the `MultiPort` on the input side can offer different technology-specific instances of subclasses of the `Port` class to enable a seamless transition between two communication technologies. In principle, switching between communication technologies could also be realized using only a single `Port` instance that is replaced as the communication technology changes. However, if the component has not yet processed all messages in the buffer of that port when the communication technology switch happens, this will lead to data loss. MontiThings' solution with using multiple input ports, even though only one is active at each given point in time, prevents such data losses.

Fig. 5.4 shows this structure with an example. Component A sends data to both B and C. We assume that partitioning (*cf.* Sec. 5.3.1) combines A and B in the same binary and C in a separate binary. Since there is a connector between A and C in this case that crosses the boundary of two binaries, A has an instance of both the normal port and the `MQTTPort` in its outgoing `MultiPort`. This enables A to communicate directly with B internally as well as with C via MQTT. Using `MQTTPorts` exclusively would create unnecessary communication overhead between A and B. If A now receives a message from the component on the (normal) port of its `inport`, this is forwarded to both ports in the output via the `InOutPort`. The `Port` and the `MQTTPort` instances of the output then forward the messages to B and C. Thus, A exchanged messages with B and C using different communication techniques. There, the incoming messages are received by the respective ports in the `inports` and forwarded to the (normal) port of their `outports`. These ports in the `outports` generate events that cause B and C to process the messages. If instead we assume that all components are partitioned into

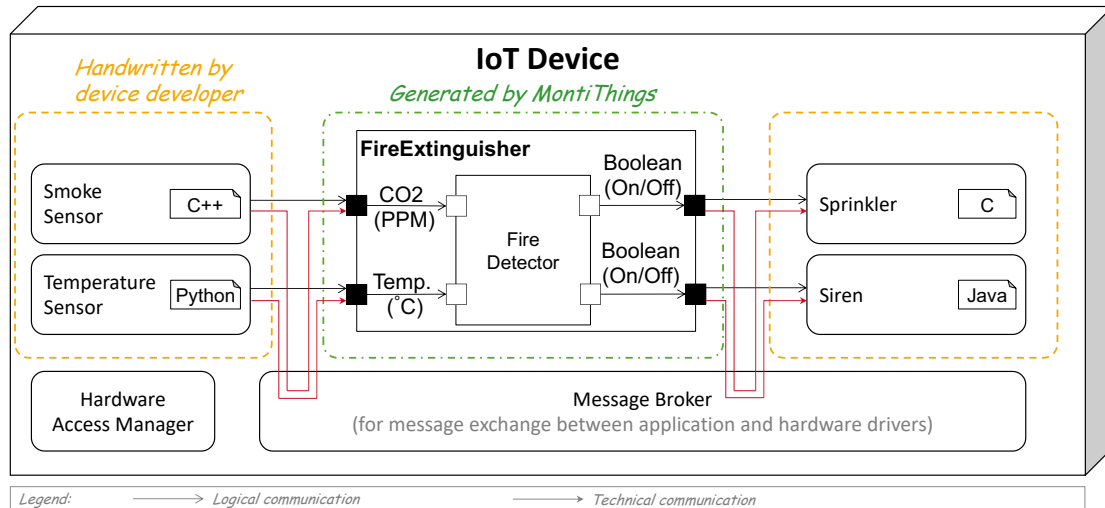


Figure 5.5: Communication with independently executed external ports uses local message broker (*cf.* Fig. 4.3). The hardware access manager connects the architecture to external ports. Figure taken from [BKK⁺22].

the same binary, the `MQTTPorts` would not exist. In this case, both B and C would be connected to the same instance of the (normal) port of A's `outport`. The port therefore keeps track of which of its recipients have already received which messages. If one of the receiving components is not able to receive incoming messages from A immediately, it would receive the next message as soon as it is ready again, regardless of which messages have already been received by other components that receive messages from the same outgoing port of A.

In the case of external ports that are not generated directly into the component via the Freemarker templates (*cf.* Sec. 4.3.2) but are executed independently on the IoT devices, communication is carried out via a local message broker. Fig. 5.5 depicts this situation. The component in this case uses regular `MQTTPorts` for communication with the external ports. A hardware access manager assigns ports to the component on demand to match its requirements. The hardware access manager can ensure that no multiple assignment takes place. For example, this prevents the same actuator from receiving commands from several components without the knowledge of the individual components. In the case of sensors, on the other hand, it can make sense to have several components process the values of the same sensor if no other sensor is available. For example, it may make sense to have multiple components process the values from a single GPS receiver while in the case of a bathroom scale with four weight sensors, the measurements from all sensors should be used instead of using sensors multiple times. Once the assignment of external ports to ports of the architecture is done, communication takes place without involving

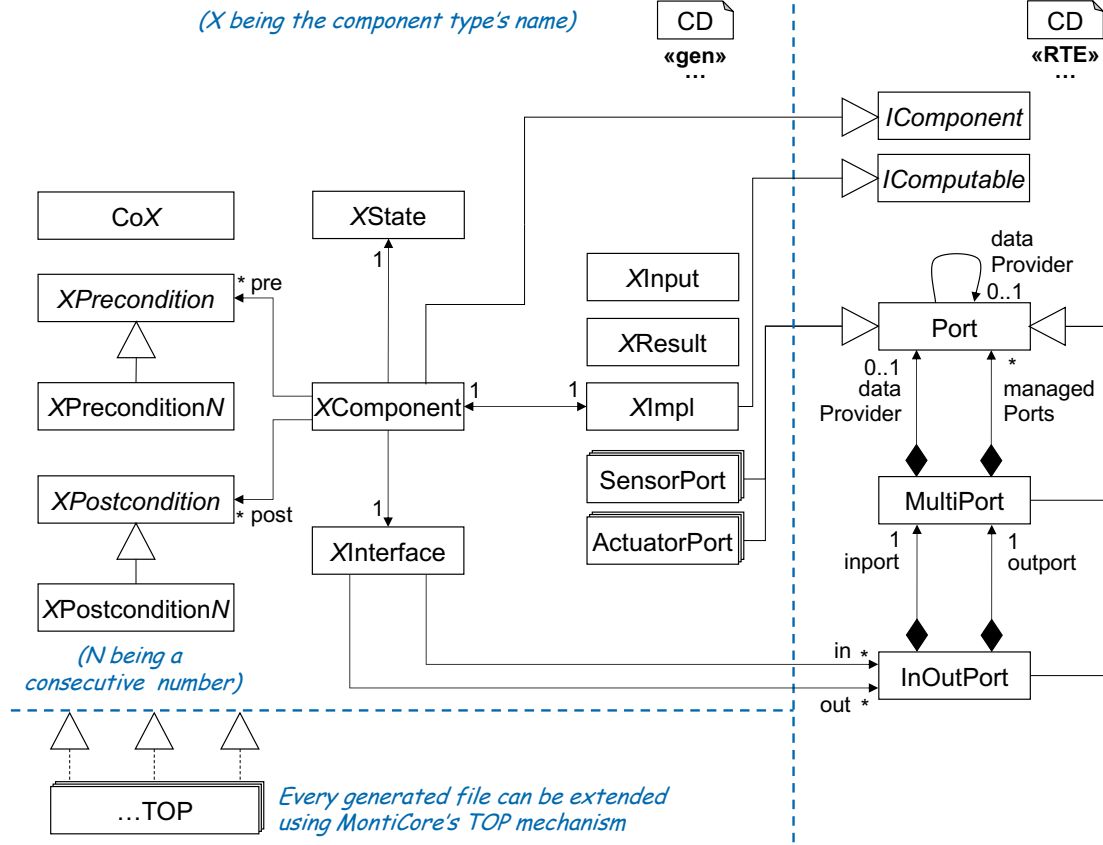


Figure 5.6: Overview of the C++ code MontiThings generates from architecture models.
Figure is an extended version from [KRSW22].

the hardware access manager. The components send heartbeat messages to the hardware access manager at regular intervals to prove that they are still using the hardware. If a component fails and accordingly does not send any more heartbeat messages, the hardware access manager has the possibility to reassign the access to the hardware. This way, no hardware remains unused forever because a component has failed.

The **PortLink** class is generated for each port but is not directly linked to a port. It contains the necessary information to connect to a port using a specific communication technology (Sec. 4.2.7). It can be subclassed to include the information needed by future communication technologies. In its current implementation, it contains a topic name. This topic can then be used to subscribe to the messages of the port using, *e.g.*, MQTT. This class will be used by the types automatically created for each component (CoX) as will be explained in Sec. 5.3.

5.3 Generated Code Structure

Of course, not all functionalities of a component can be covered by using the identical code of the RTE. The information given to MontiThings about the models requires code that fits the individual models. This includes, for example, the behavior of the respective component. MontiThings' Code Generator therefore allows to generate code customized to the models, which in turn builds on the RTE. Fig. 5.6 gives an overview of the code MontiThings generates from the architecture models. For each component, MontiThings generates a class that is named like the component followed by the word component (**XComponent** in Fig. 5.6). It implements the `IComponent` interface described in the previous section. This class is responsible for initializing the component and for coordinating between the other classes generated for that component.

Each component also has a state object (**XState**). The state class is responsible for managing all variables of a component. This is useful for implementing `@pre` expressions; before a computation starts, the generated `compute` method will save the current state. Furthermore, the state can also be serialized, stored, and, when necessary, restored. If `@ago` expressions are used, the state class automatically stores all changes to all variables referenced by these `@ago` expressions. If a statement refers to a previous value, MontiThings retrieves the value with the latest timestamp that is at least as old as requested by the `@ago` statement. For example, if a variable changed one, two, and three seconds ago and the value from 1.5s is requested, MontiThings will choose the value from two seconds ago. It does so because the value from one second ago is not older than the requested period of time and the value from three seconds ago is older than the value with the timestamp from two seconds ago. By using this strategy and only storing variables when their value changes, MontiThings uses the minimal amount of memory required to be able to resolve these `@ago` expressions. Values that are older than the highest `@ago` statement are discarded to save memory. By encapsulating the component's variables into a single class, MontiThings ensures that the component's variables are only accessed through their getters and setters and, thus, modifications to the variables can be tracked to reference them in `@ago` expressions.

Similarly, the **XInput** and **XResult** classes wrap a set of incoming and outgoing messages. It is, however, important to notice that while these classes are similar to the classes defined in `MontiArcAutomaton`, they serve a different purpose in MontiThings. In `MontiArcAutomaton`, the implementation of a component cannot directly access ports but only process messages by accessing an `XInput` object and send messages by returning an `XResult` object. This is possible, as `MontiArcAutomaton` uses a time-synchronous semantics, where every component may only process and send at most one message per port. In contrast, MontiThings uses event-based semantics, where components may send multiple messages on the same port in the same computation. Therefore, MontiThings components may directly access the components interface. The `XInput` and `XResult` classes exist in MontiThings so that a set of messages can be easily passed to pre- and

postconditions. Whenever the code generator encounters a statement that would result in sending a message on an outgoing port, it will generate three statements: 1. Create a result object with the messages that shall be sent, 2. Check the postconditions using said result object, and 3. Actually send the messages if the postconditions are fulfilled. This way, it is possible to ensure that the postconditions are fulfilled without actually passing messages to a port that would not fulfill the postconditions.

The components refer to pre- and postconditions using the **XPrecondition** and **XPostcondition** interfaces. The difference between the pre- and postcondition classes is that only the postconditions' methods get passed a result object because preconditions are checked before executing the computation of a component, *i.e.*, before any message can be sent. Overall, the pre- and postconditions offer six methods. 1. `check` returns a boolean that indicates whether the condition is violated or not 2. `toString` returns a pretty-printed string representation of the pre- or postcondition 3. `logError` creates an error message telling the user which condition failed and under which conditions, *i.e.*, it also logs the current state, incoming messages, and, in case of a postcondition, outgoing messages 4. `isCaught` return a boolean indicating whether the pre- or postcondition include a `catch` block that could resolve violated pre- or postconditions 5. `resolve` tries to resolve a violated pre- or postcondition by executing the code generated from the `catch` block of that condition 6. `apply` uses all of the above methods to apply a pre- or postcondition by checking if it is violated and if so trying to resolve it or logging an error. Specific pre- and postconditions are implemented using the **XPreconditionN** and **XPostconditionN** classes (N being a consecutive number). They extend the general pre- and postcondition classes and implement the `isCaught`, `check` and `resolve` methods according to the specific condition.

The actual behavior of a component is implemented using the **XImpl** class. This class implements the `IComputable` interface described in the previous section. If a component has a behavior that is specified in the model, *i.e.*, uses MontiThings' Java-like behavior language or statecharts (Sec. 4.2.4), then the code generator will fill the `compute` method with C++ code according to that specification. For the Java-like behavior language, this means essentially pretty-printing the code while replacing certain constructs. During this pretty-printing process, MontiThings will translate certain constructs from the behavior language that do not normally exist in Java or C++. When referring to time, MontiThings replaces all references to SI units with uses of the `chrono` library from the C++ standard library. Thereby, MontiThings models can use different time units in, *e.g.*, `@ago` statements, while the `chrono` library takes care of performing the necessary conversions. For OCL expressions, MontiThings mostly utilizes lambda expressions to realize more complex expressions such as `exists`, `forall`, or `let-in`. Since lambda expressions are normal expressions in C++, this makes it possible to pretty-print the lambda expression in all places where the OCL expression is used. Logical implication and equivalence are translated to their representations using \neg, \wedge, \vee that have corresponding operators in C++ (`!`, `&&`, `||`). For set expressions,

MontiThings mostly converts the set expressions into an equivalent expression without actually building the set. For example, checking whether $a \in \{1 : 100000\}$ (written as `a in {1..100000}`) can be solved in $\mathcal{O}(1)$, if it is implemented as `if (a >= 1 && a <= 100000)`. In contrast, creating a set with 100000 entries (or, in general, n entries) and then checking if it contains a would be in $\mathcal{O}(n)$. Unions and intersections can be interpreted as `||` and `&&` in these if statements. References to variables and ports are replaced by calls to the appropriate getter and setter methods.

The ports of the component are grouped by the **XInterface** class. For ports that access a sensor or actuator, MontiThings will generate classes shown as **SensorPort** and **ActuatorPort** port in Fig. 5.6. These ports create threads that will execute the code given by the developer-provided Freemarker templates as described in Sec. 4.3.2.

To support dynamic reconfiguration, MontiThings generates a type that can be considered the “business card” of the component, *i.e.*, it includes all information necessary to connect to a component instance (Sec. 4.2.7). This class is called `Co` followed by the name of the (interface) component type (shown as `CoX` in Fig. 5.6). It contains port link objects for all ports of the component. If the component implements interfaces, the `CoX` will inherit from the `CoX` class of these interfaces.

5.3.1 Architecture Partitioning and Setup Information Exchange

Naturally, a distributed system requires multiple systems each executing parts of the system’s software to interact. Therefore, MontiThings is able to generate multiple binaries from the C&C models provided by the developers. This process of deciding which components shall be executed in independent binaries is called *partitioning* in MontiThings. By default, MontiThings will generate one binary per component to give the device owner maximum flexibility in later deploying the application. Using the configuration language, this partitioning can be restricted by selecting components that shall not be split Sec. 4.3.2. Overall, MontiThings will create a standalone binary for each component except for those that are both exclusively used as subcomponents of unsplit components and can also not be used by any connector that uses a dynamically received component interface. If a component could be used dynamically anywhere, we need to have a binary for that component to be able to instantiate it dynamically.

For the support of different processor architecture, the code generated by MontiThings can be cross-compiled using, *e.g.*, DockCross⁶ for cross-compiling the binaries only or Docker Buildx⁷ for creating multi-platform images. For more specialized platforms, *e.g.*, the DSA VCG, it is of course also possible to use the individual cross-compiling toolchains of the vendor. For this, MontiThings requires the toolchain to be CMake

⁶DockCross GitHub project. [Online]. Available: <https://github.com/dockcross/dockcross>. Last accessed: 27.11.2021

⁷Docker Buildx Documentation. [Online]. Available: <https://docs.docker.com/buildx/working-with-buildx>. Last accessed: 27.11.2021

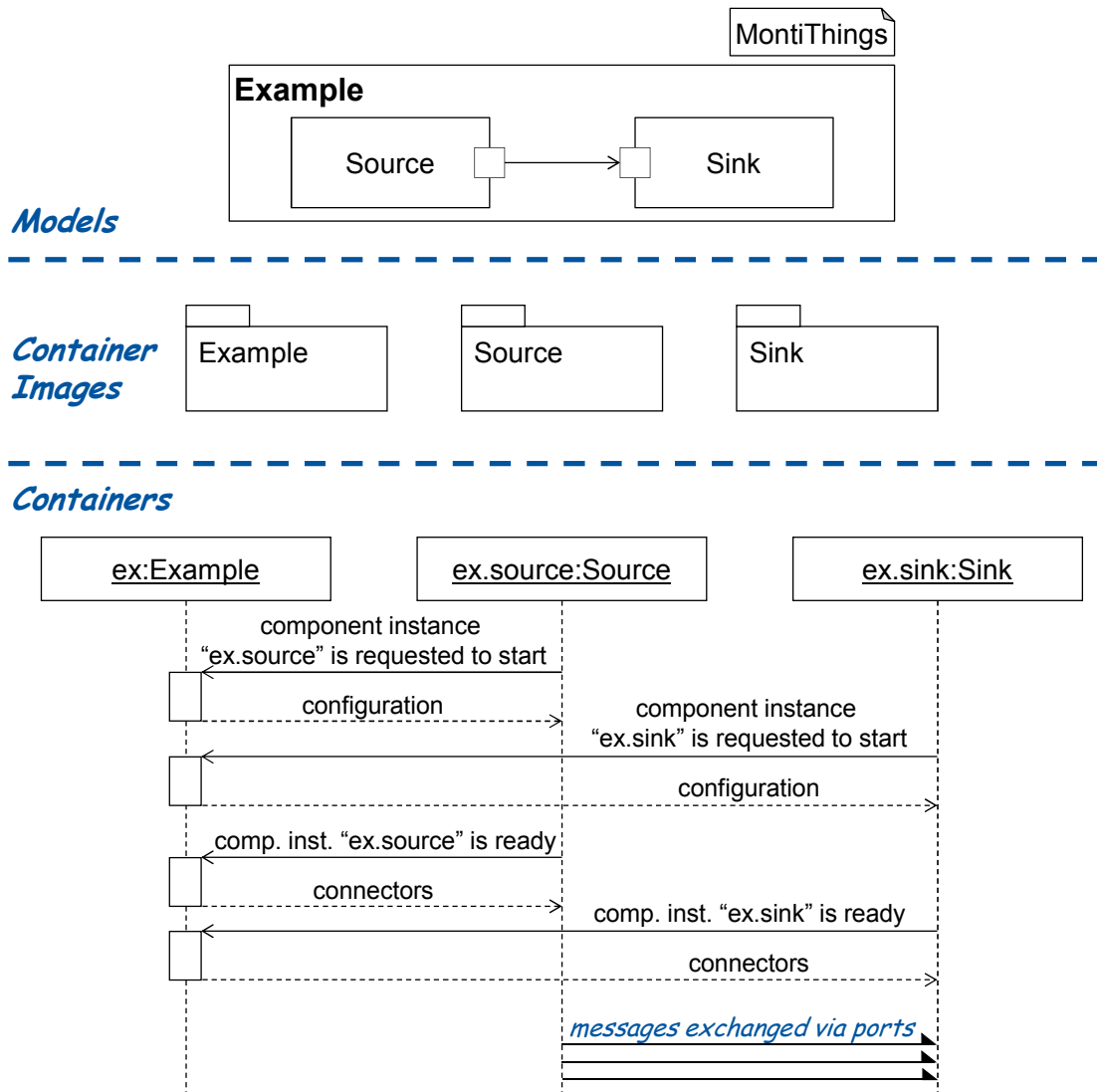


Figure 5.7: Partitioning an architecture into different container images and setting up the component instances started from these container images.

compatible and support the C++11 standard library (**TA4**). Any of MontiThings external dependencies can be cross-compiled from their sources using the Conan package manager⁸. For this, MontiThings generates a `conanfile.txt` based on the selected generator configuration.

⁸Conan project website. [Online]. Available: <https://conan.io/>. Last accessed: 27.11.2021

Fig. 5.7 shows an example of how an architecture is partitioned and how the resulting containers are connected to each other. Three independent container images are created from the three component types Example, Source and Sink. At runtime, these container images are started as containers, with the outermost component being given the instance name `ex` and the subcomponents being assigned names derived from it. When a component instance is started, it announces itself to the other components and thereby asks its enclosing component for a configuration. Announcing the request to start a component is a broadcast. Since the components run on independent devices it is not guaranteed that the subcomponents are started after their enclosing components. If the enclosing component is started after one or more of its subcomponents, the subcomponents receive the announcement of their enclosing component and repeat their own announcement. This enables the enclosing component that was not ready at the time of the initial announcement to respond. The configuration with which the enclosing component answers contains the component parameters of the subcomponent instance and, if required, conversion factors for ports that use SI units. The conversion factors are necessary when two ports are connected whose SI unit types are compatible but different, *e.g.*, m/s and km/h. As soon as the setup of the component is started (setup method from the RTE, *cf.* Sec. 5.2) with this configuration, the component requests its connectors from the enclosing component. Once the connectors are set up, the subcomponents can communicate directly via ports with each other.

5.3.2 Generated CLIs

Some parameters of a component shall not be fixed at design or generation time. Therefore, MontiThings generates a command line interface (CLI) for each generated component. The CLI can be used to pass arguments to a component instance when starting it. The CLI's parameters can also be passed when starting a Docker container. This enables the automated deployment manager to adapt component instances to its needs (*cf.* Chapter 6).

The generated CLI consists of the following parameters:

n, name This string parameter sets the instance name of the component. It is the only required parameter. If a component has subcomponents, their instance names will be derived from this argument, *i.e.*, appended with dots between the name components. For example, the subcomponent `einstein` of a component `newton` will be called `newton.einstein`.

muteTimestamps A switch to suppress timestamps in log files. The primary use case for this is creating log files that can be compared against an expected output in test cases. Having timestamps in such test cases would create different log files for the same execution and thus require additional effort for removing the time stamps in test cases.

muteRecorder Suppresses all log messages from the optional recording module (*cf.* Sec. 7.5). This switch is only available if the recording module was enabled in the generator configuration.

monochrome Normally, MontiThings log files will be colored using PS1. In some cases, this may be unwanted. This switch can make all log messages monochrome.

printConnectStr If enabled, the log file will contain a message that can be sent via a port to dynamically connect to or disconnect from the component (Sec. 4.2.7). If a component implements interface components, this parameter will also be available for each implemented interface component by appending the name of the generated type to the `printConnectStr` switch. For ease of use during testing, by default the printed string will be in a format that can be copied and pasted in terminal windows, *i.e.*, using escaped quotation marks.

pretty If enabled, all `printConnectStr` messages will be pretty-printed. This makes them easier to read as a human but harder to reuse as a computer.

Based on the selected communication technology, further communication technology-dependent can be available. For MQTT, the following parameters will be added.

muteMQTT If enabled, this switch will suppress all log messages from MQTT ports, *e.g.*, notifications of connecting to a broker, or subscribing to a topic.

brokerHostname This parameter can be used to specify the hostname of the MQTT broker to which the component shall connect. By default, this will be `localhost`. For a distributed deployment, this argument shall be filled with the IP or domain name system (DNS) name of an MQTT broker, *e.g.*, `broker.hivemq.com`.

brokerPort Sets the (network) port on which to connect to the MQTT broker. By default, this parameter is set to the standard MQTT port: 1883.

localHostname This parameter can be used to override `localhost` in the generated code. In some instances, the `localhost` DNS name will not be available. For example, when executing the component in a Docker container, users may need to use `host.docker.internal` instead of `localhost`.

When communicating using DDS, the following CLI parameters are added:

muteDDS If enabled, this switch will suppress all log messages from DDS ports.

DCPSConfigFile Can be used to override the DDS configuration file. By default, `dcp-sconfig.ini` is used. This (generated) configuration file contains information such as the utilized transport layer protocol (TCP).

DCPSInfoRepo Can be used to set the hostname and port of the DCPSInfoRepo. The DCPSInfoRepo is a (centralized) discovery service that the devices use to find each other. For details, refer to the OpenDDS developer's guide⁹.

5.3.3 Generated Scripts and Compilation

The generated code can be compiled using standard C++11 compilers (**TA4**). For ease of use, MontiThings also generates various scripts that can be used to build, package, and execute the generated C++ code using CMake and Ninja as build tools. The `build.sh` script calls CMake and Ninja to compile the generated sources. Optionally, it can be used with the Conan package manager to also receive and, if necessary, build all libraries MontiThings' generated code relies on. As mentioned above, if the binary is supposed to be executed on a different target system, it may be required to call the `build.sh` script using DockCross to cross-compile the code. After building the source code, the binaries can be executed. In the case of a distributed application (partitioning mode not set to `off`), this means that the user has to start a potentially large number of binaries. To simplify this, MontiThings offers a `run.sh` script that will automatically start all components. Their output will be redirected to a file named after like the component's instance name with the file extension `.log`. Thereby, users can use standard (Linux) command line tools like `tail`, `head`, or `cat` to inspect the output of the components. To stop the components, the `kill.sh` script can be used.

When using Docker, the binaries need to also be packaged into container images. For this purpose, MontiThings generates `dockerBuild.sh`, `dockerRun.sh`, `dockerStop.sh`, and `dockerKill.sh` scripts that behave like the non-Docker variants of these scripts. The `dockerStop.sh` is a variant of the `dockerKill.sh` script that gracefully stops the containers instead of killing them. As the container identifiers are not known at generation time, the `dockerStop.sh` and `dockerKill.sh` scripts are not generated by the Freemarker generator. Instead, the Freemarker generates (Bash) code into the `dockerRun.sh` script that generates the `dockerStop.sh` and `dockerKill.sh` scripts whenever the `dockerRun.sh` script is executed.

To build the images, MontiThings offers images including the complete compiler toolchain and all C++ dependencies (called `montithings/mtcmake`). When using DDS for communication, a separate base image is used (called `montithings/mtcmakedds`). The reason for using a separate image is that the DDS image is significantly larger. Thus by providing a separate image, the required download size can often be decreased. These base images are provided for both x86 and ARM processor architectures. This way, the docker images can even be created on machines where MontiThings

⁹OpenDDS Developer's Guide. [Online]. Available: <https://download.objectcomputing.com/OpenDDS/OpenDDS-latest.pdf>. Last accessed: 28.11.2021

was not installed, *e.g.*, in CI pipelines. As recommended by Docker as a best practice¹⁰, MontiThings generated Dockerfile use a multi-stage build. During the first stage, the generated code is compiled using the `mtcmake` image. In the following stages, Linux Alpine-based containers are created for each component type. In each of the generated images, the `libgcc` and `libstdc++` are installed (**TA4**). If MQTT is used, additionally `mosquitto-libs++` is installed. Lastly, the compiled binaries are added to the image. By adding the containers after installing the libraries, the Docker runtime can cache the first layers of the image between all images. This reduces the disk space required to execute an additional component. In environments where the images are supposed to be deployed to a heterogeneous infrastructure (**TC1**), the generated Docker file shall be built as a multi-arch image using `docker buildx` and all required target platforms.

5.3.4 Supporting Different Target Platforms

IoT devices can be very different from one another (**TC1**). One aspect of this is that not all target devices may be capable of executing the same software. For the low-level hardware-accessing code, this can be managed by providing different Freemarker templates for different platforms (*cf.* Sec. 4.3.2). Regarding the operating system, MontiThings will adapt its build scripts to use, *e.g.*, certain cross-compilers, for different platforms. In general, the overall support for the operating system has to be provided by extending the generator so that it adapts the build scripts to this platform, *i.e.*, setting the appropriate CMake toolchain and linking libraries differently if necessary, while support for application-specific hardware has to be provided by the developers. This is different from Eclipse Mita [wwwb] where accessing the platforms' hardware is also part of the generator. As IoT projects are characterized by using sensors and other hardware, we considered this approach to be too inflexible.

5.3.5 Test Case Generation

As explained in Sec. 4.3.3, MontiThings includes a test case generator. The generated test code is based on of the most popular C++ test frameworks: GoogleTest. In the constructor, the test class instantiates the classes generated for the component under test. Using its generated getter methods, it further creates variables that are used by the rest of the generated test code to conveniently access subcomponents, `Impl` classes, and `State` classes.

The test cases follow the *given-when-then* pattern [Fow19]¹¹. Thus, each every test case starts by setting up the environment, followed by phases for acting on input and

¹⁰Best practices for writing Dockerfiles. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. Last accessed: 28.11.2021

¹¹Robert C. Martin's "Clean Code" calls this pattern the "build-operate-check" pattern [Mar08]. Sometimes it is also called "arrange-act-assert" or "setup-exercise-verify" [Fow19].

evaluating whether the output matches the expectations. There are multiple when- and then-phases if the component is supposed to process multiple inputs.

Given For all ports (including incoming ports!), the generator creates a spy class. The task of the spy class is to record all messages of a port to make them available to the test case. For this, the spy class implements the `EventObserver` interface from the RTE. Using this interface, it subscribes to all events from its accompanying port.

When For every message that is given to the component under test, *i.e.*, for each connection of the sequence diagram that does not have a source, the generator sends a message to the outermost component. All further communication between subcomponents then has to be handled by that component.

Then If a connection in the sequence diagram has both a source and a target, the test case generator will generate an assertion that accesses the spy objects of the ports to check that the requested message is both sent and received. Unlike the simulation-based `MontiArc`, `MontiThings` may replace connections using communication technologies such as MQTT. By checking both the dispatch of the message as well as the reception at the incoming port, the generated test cases can detect communication-related errors, *e.g.*, components that are not correctly connected to the MQTT broker. If a connection in the sequence diagram has no target, only the dispatch of the message is asserted. For assert statements within the sequence diagram, the test case generator reuses the pretty-printer from `MontiThings`' Java-like behavior language. It was, however, slightly adapted to use the spy objects instead of the actual port when accessing messages of a port. Delays are implemented directly by adding a delay to the test case. As the test case does not control the time as a simulator does in `MontiArc`, `MontiThings`' test cases cannot skip delays and only adjust the global clock.

5.4 Discussion

Using Docker containers for deploying individual components matches the development concepts of C&C architectures. As each component is a black box, it can be deployed independently of other components. In fact, using containers reduces the risk for certain bad practices like implementing certain aspects of “dirty components” [Hab16], *i.e.*, components that open communication channels to other components without using ports, *e.g.*, by accessing a shared file. As containers run isolated from each other, the components cannot create shared files or open (network) ports that are accessed by other components. In many cases, such interactions also would not make sense as developers do not know at design time which devices a component will be deployed to. This prevention mechanism is, however, superficial and can be bypassed, *e.g.*, by placing shared

files on an external storage provider such as an AWS S3 blob storage. As already shown by projects like balena (and the balenaEngine), Docker containers can also be used in real-world projects, although the container engine ideally needs to implement some IoT-specific techniques to, *e.g.*, cope with intermittent connectivity¹². The size of the produced Docker images could further be reduced by using distroless images. However, since this approach makes the images harder to debug due to the missing tools, while only providing a few megabytes advantage, we decided against using a distroless image as a base image.

A disadvantage of Docker containers is that they require a certain amount of hardware resources. Thus, targeting very low-powered devices like the Arduino Uno or an Espressif ESP32 is not possible with Docker containers. Overall, low-powered devices, alas, lack common standards that would make it possible to run the same software on most low-powered devices. An additional problem in supporting such low-powered devices is that their compilers often do not support the full C++ standard library. Additionally, their program memory is often very limited, so loading multiple libraries is sometimes a problem. As a result, the generator is based on a number of technical assumptions (Sec. 3.1) to limit the development effort.

An important aspect of implementing the communication was moving away from IP-based identifiers of communication partners and towards topic-based identifiers. Especially, MontiThings uses MQTT and DDS. The underlying communication protocols, of course, can still handle communication using IP-based protocols. Managing the raw IP addresses, however, introduces too much complexity in the generator. Especially, in IoT use cases the IP address may often not stay the same over extended periods of time. Reasons for this include, for example, deploying an IoT device in a cellular network where multiple devices may share the same IP address and the IP address may also be changed by the network provider. MontiThings is tested against multiple MQTT brokers (Eclipse Mosquitto, HiveMQ, and EMQX). Related work found EMQX to provide a higher throughput than HiveMQ and HiveMQ for high reliability, *i.e.*, less message loss [KGR20]. Since both EMQX and HiveMQ can be horizontally scaled [KGR20], MontiThings by default uses HiveMQ as a central broker and Eclipse Mosquitto as a local broker because of its lightweight implementation. However, these choices are not binding on developers.

While MontiThings offers both centralized communication via an MQTT broker and peer-to-peer communication via DDS, we, in general, recommend centralized communication. Other IoT languages, such as Calvin, use peer-to-peer communication exclusively. This is often accompanied by a registry of devices and their IP addresses to enable the

¹²This and other problems addressed by balenaEngine to meet IoT requirements can be found on the project website. [Online]. Available: <https://www.balena.io/engine/> Last accessed: 09.01.2022

device to find each other¹³. When using DDS, MontiThings uses a similar registry (provided by the OpenDDS library; called *DCPSInfoRepo*). Such registries are needed to go beyond the scope of a single broadcast domain, *i.e.*, often subnet in the case of using raw IP addresses. Overall, we made the experience that centralized communication has fewer problems than peer-to-peer-based communication for IoT applications. Besides having a central access point for all devices, it can also avoid problems with firewalls and network address translation (NAT) gateways. For example, opening a connection to a device in a mobile network may be impossible even when knowing the IP address because the cellular provider may use said firewalls and NAT in their network which prevents such connections. If, however, the connection is established by the IoT device itself, *e.g.*, by connecting to a central broker, such communication is usually not a problem. In general, we recommend using MQTT over (Open)DDS for IoT projects. The OpenDDS library is much more complex compared to MQTT. This includes both programming and compiling the code. Since MQTT is more widely supported, it is also easier to integrate MQTT ports with other technologies.

Overall, the generator's configurations make the generator very flexible. The partitioning modes (Sec. 5.3.1) are especially useful during the development of components. Even if a component is intended to be deployed in a distributed fashion, preventing the partitioning (and thus much of the communication) can help to find errors, as it is easier to debug a single application than a distributed application consisting of multiple independent binaries. The disadvantage of the high number of configurations is that they make templates more complex by introducing if-statements in the templates that only include certain code for certain generator configurations. While this complexity can be hidden using hook points, hook points make it harder to understand exactly which templates are included by a certain template. Compared to early versions of the generator, the maintainability of the templates increased notably by standardizing the folder structure and file names and reducing the size of templates. Our structure requires a single folder for each generated file that is named after the generated file. For generated C++ files, this folder contains a template for the header (.h-file), the implementation (.cpp-file), and the content of the implementation. The content of the implementation is separated from the implementation file so that it can be added to the header file for generic classes. A *methods* subfolder contains a single template for each generated method. Moreover, a folder of templates may offer a *hooks* folder containing templates that are intended to be included by other templates. For example, such hooks could define include statements for the generated class or instantiating a member variable using the generated class. By using such hook templates, small changes such as adding a

¹³For example, Calvin uses its own registry based on a distributed hash table. Calvin Registry Documentation. [Online]. Available: <https://github.com/EricssonResearch/calvin-base/wiki/Configuration#registry>. Last accessed: 08.01.2022

parameter to the constructor or changing the class's name do not require modifying all templates that use the generated class.

Unlike `MontiArc`, `MontiThings` generates code that is supposed to be executed as a distributed application on sometimes unreliable devices. Therefore, compared to `MontiArc`'s simulation-focused code, the RTE classes used to provide ports differ noticeably from the structure in [Hab16, Wor16] because they need to take various communication aspects into account. The port structure of the RTE has the advantage of making it easy to support different communication technologies and access to hardware. By allowing multiple port instances to be included in `MultiPorts`, it became possible to also accept multiple (communication) technologies and translate between them. The `InOutPort` that combines incoming and outgoing ports is `MontiThings`' counterpart of `MontiArc`'s `IForwardPort` [Hab16], that handles such translations. The `MessageProvider` and `MessageAcceptor` of the RTE are, essentially, renamed variants of the `IOutPort` and `IInPort` interfaces from [Hab16]. One reason to differentiate between outgoing and incoming ports in [Hab16] was also to save unnecessary objects, *e.g.*, buffers. In `MontiThings`, however, these buffers are necessary to protect against the (temporary) failure of communication partners at various points. Since `MontiArc` generates a simulation unlike `MontiThings`, such failure situations do not occur there. Conceptually, the structure of the generated code is similar to the structure of the generated code from `MontiArcAutomaton` [Wor16]. The `IComponent` and `IComputable` interfaces¹⁴ were taken over from [Wor16].

An open challenge is still the integration of cloud technologies in the generator. While `MontiThings` components can, of course, be deployed to virtual machines in the cloud¹⁵, such deployments do not take full advantage of the capabilities of cloud providers. By generating serverless functions, it would be possible to also use a pay-per-use billing method for the generated code. The main reason why `MontiThings` does not support this is a lack of standardization between cloud providers. Furthermore, the libraries offered by cloud providers to access their services are not guaranteed to be stable. Moreover, it would be desirable to integrate `MontiThings` with more external services, *e.g.*, for machine learning using pre-trained models offered by a cloud provider. In general, we see interesting opportunities in the combination of code generation with infrastructure as code languages, *e.g.*, `Terraform`¹⁶.

¹⁴There are actually no interfaces in C++. Interfaces are implemented as classes containing pure virtual methods. For easier comprehensibility, we will call them interfaces even though they are technically realized as abstract classes.

¹⁵In this regard, `MontiThings` also offers an installer for Amazon Linux so that it can be deployed to EC2 instances easily

¹⁶`Terraform` website. [Online]. Available: <https://www.terraform.io/> Last accessed: 09.01.2022

Chapter 6

Deployment and Integration of C&C-based IoT Applications

IoT applications are often distributed across multiple devices. The distribution process, including the decision-making of which device is supposed to execute which software, is called *deployment*. IoT applications have to handle heterogeneous devices (**TC1**) and unreliable hardware (**TC3**) which are usually not a consideration when deploying software to traditional server or cloud environments. Thus, the process of deploying IoT applications comes with different requirements and challenges. This chapter discusses how to solve these deployment challenges. Furthermore, IoT applications may not be limited to low-powered devices attached to “things”, but include information systems and digital twins that enable the interaction with users (*cf.* Sec. 3.2). Accordingly, this chapter also describes how IoT applications and model-driven information systems can be connected (semi-)automatically.

6.1 Research Questions

This chapter mainly answers the question of how to deploy C&C-based IoT applications (**RQ2**). Since IoT devices are often prone to failure (**TC3**), part of this is that the deployment needs to adapt to failing devices (**RQ5**). Compared to traditional deployment approaches that model the deployment of the software for each device individually, IoT devices “must be managed en masse instead [of] receiving personal attention and care” [TM17a]. Previous works such as Calvin [AP17, PA15, PA17] suggest that a rule-based approach is beneficial for deploying IoT systems. MontiThings builds upon these results and further examines deploying IoT software based on rules (**R7**). Especially, MontiThings also examines handling unfulfillable rules by proposing changes to rules and infrastructure to device owners.

The connection to information systems (**RQ6**) and, especially, their digital twins (**RQ7**) is also examined. This connection between IoT systems and their digital twins has not received a large focus in research. Many commercial cloud providers offer solutions for building digital twins of IoT systems, *e.g.*, (Microsoft) Azure Digital Twins, (Amazon) AWS Device Shadows, or the Arduino IoT Cloud. These solutions require developers

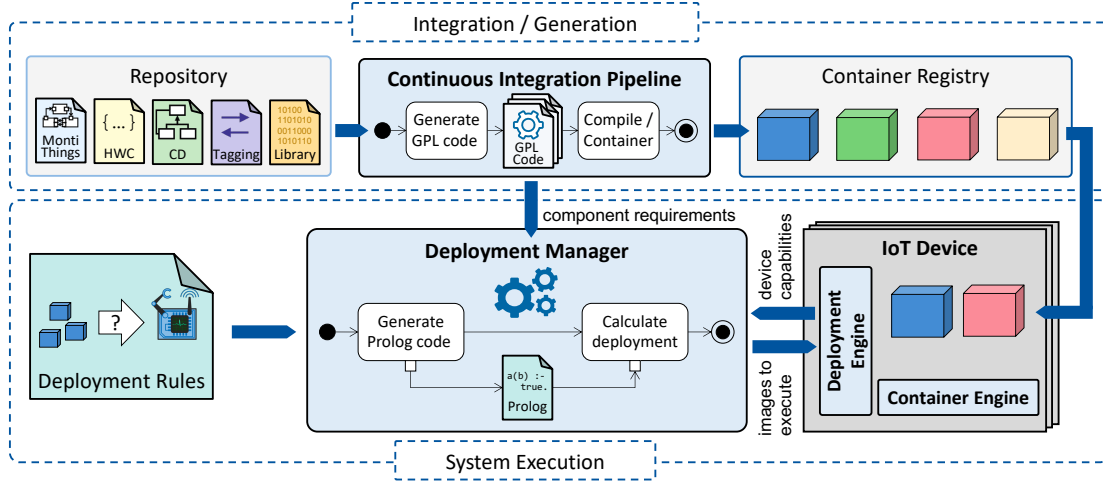


Figure 6.1: High-level overview of MontiThings' deployment process. Figure taken from [KRSW22].

to manually adapt their code to include the digital twin framework. In this thesis, we examine how to leverage model-driven engineering to automatically create connections to information systems and digital twins [KMR⁺20b]. Automating this process removes the chance for developers to make mistakes, *e.g.*, when needing to set the same MQTT topics in multiple files across multiple applications.

6.2 Development and Deployment Processes

MontiThings' deployment process is built around deploying containers including parts of the software. This decision was made because future IoT applications are expected to use a “universal, containerized application-deployment-and-execution model” [TM17a]. Businesses like Balena with their IoT-focused (Docker) container engine balenaEngine show that deploying (Docker) containers to IoT devices is a viable approach. Furthermore, the components of C&C ADLs are independent black boxes. Accordingly, deploying components using containers is a natural choice, as the container images can also be viewed as independent deployment units. By focussing on containers, the results of our deployment research also become better transferrable to other languages as the deployment can be applied largely independently of the MontiThings language.

Fig. 6.1 shows an overview of the deployment workflow. In the upper part of the figure, the various models of the application are first pushed into an online repository such as GitHub or GitLab. This triggers a CI pipeline that generates code from these models, compiles the code, and packages it in (Docker) container images. These images can be stored in a container registry. Major platforms like GitHub or GitLab also provide

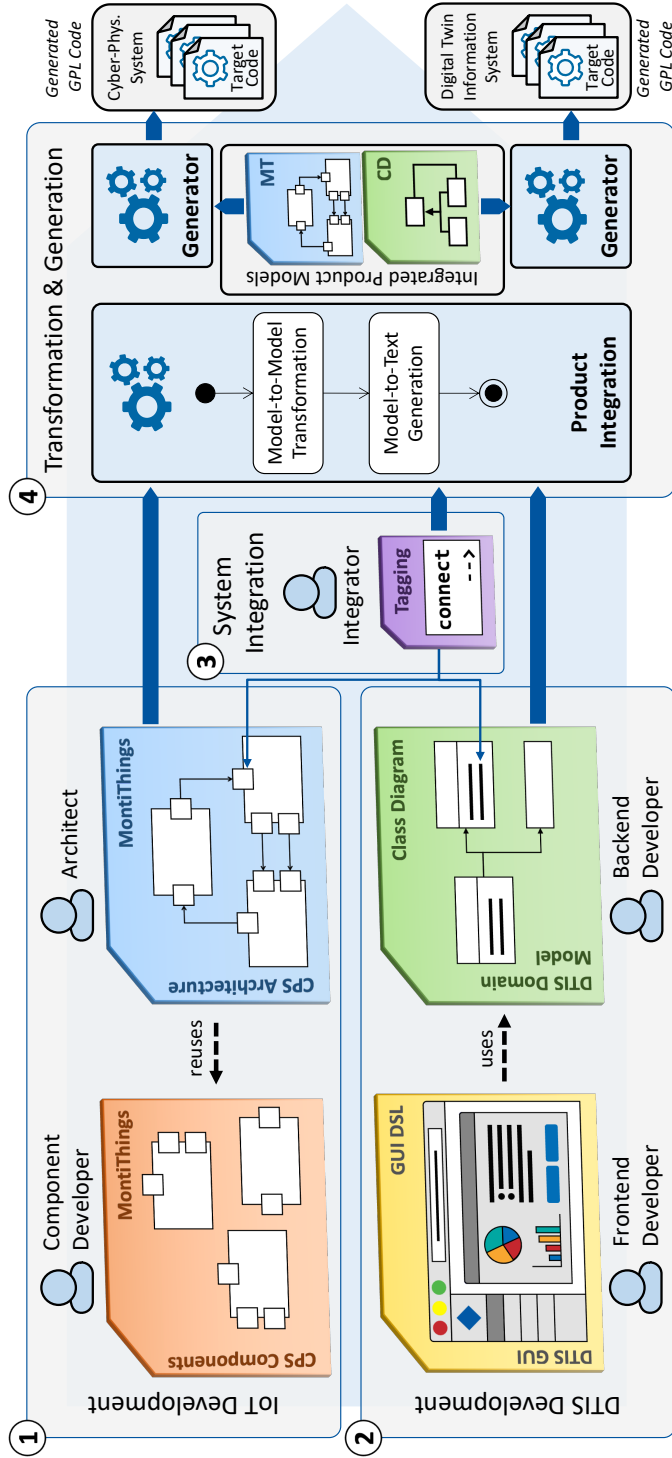


Figure 6.2: Workflow for the development of integrated IoT systems and information systems. Both the IoT system and the digital twin information system (DTIS) are developed using model-driven development. Figure adapted from [KMR⁺20b].

such registries together with the code repositories. Another result of the CI pipeline is also that the code generator creates an overview of the technical requirements of the individual container images. In MontiThings, these can easily be extracted from the configuration language (Sec. 4.3.2).

The lower part of the image shows how these images are then deployed to the IoT devices. First, device owners define a set of local rules. These rules set their preferences of which devices shall (not) execute which parts of the software. For example, one rule could be never to deploy camera recording software in the bedroom. A deployment manager acquires these requirements together with information about the infrastructure, *i.e.*, which devices are available with which capabilities, and the requirements of the components. This information is used to generate Prolog code that calculates which devices are supposed to execute which software. The deployment manager utilizes Prolog for calculating deployments because Prolog’s backtracking-based evaluation strategy makes it possible to find alternative solutions in case some solutions are rejected by the device owners. After the deployment is calculated successfully, the deployment manager sends commands to the IoT devices that tell them which containers to execute. The IoT devices then download them from the container registry. Sec. 6.3 discusses this workflow in more detail.

While the workflow for the deployment is largely independent of the MontiThings language, integrating IoT applications with information systems requires changes to the behavior of both systems and is, thus, tighter integrated with the languages used for modeling these systems. Fig. 6.2 gives an overview of the workflow for creating an IoT application that is integrated with a DTIS [KMR⁺20b]. A DTIS is an information system that manages a digital twin and provides graphical user interfaces (GUIs) to its users. These GUIs enable the users to monitor and control the digital twin. Our workflow for creating an IoT system and integrating it with a DTIS starts by first developing both systems. While it is not required to do so, the development of the IoT application (①) starts by developing a set of MontiThings components independent of a specific IoT application. These can then be used by an architect, whom we also refer to as IoT developers, to create the architecture of an application. The DTIS development (②) using the MontiGem toolchain for enterprise information systems consists of creating class diagrams to model the data structure and GUI models for specifying the user interface. MontiGem uses the class diagrams to generate database schemas and various Java classes that facilitate access to the databases created from this schema. Moreover, MontiGem uses the GUI models to generate a user interface. For this, MontiGem can take advantage of numerous off-the-shelf GUI components such as tables, pie charts, or bar charts.

After both systems are modeled, an integrator who knows both systems uses a tagging language to connect them (③). This tagging language connects ports from the MontiThings models to attributes of the class diagrams of the DTIS. Using this information, model-to-model transformations for both systems can add the necessary infrastructure

to synchronize the IoT application and the DTIS with each other (④). After applying the transformations, the code generators can produce GPL code as usual. This process of integrating IoT applications is described in more detail in Sec. 6.6.

6.3 Requirement-based Deployment

As explained in Sec. 6.1, our deployment method is based on the idea that IoT should not be managed individually. Therefore, we base the deployment on the requirements-based deployment Calvin [AP17, PA15, PA17]. In Calvin, each component may have a set of requirements. Adversely, every device has a set of properties¹, *i.e.*, hardware capabilities or location. By matching the requirements against the device properties, Calvin decides whether a device shall execute a specific component. The idea of letting device manufacturers define the capabilities of their devices is also used by cloud providers. For example, Microsoft Azure’s *IoT Plug and Play*² uses a JavaScript object notation (JSON)-based “Digital Twins Definition Language” for this purpose.

MontiThings extends this approach and in doing so tries to solve a number of open challenges of this approach:

1. There is a lack of separation of concerns in this approach (**MC1**). The IoT developers who specify the software components should not be required to have knowledge about the infrastructure their application is deployed to. The device owner likely has a number of requirements that cannot be specified by the IoT developers due to their lack of knowledge about the infrastructure.
2. Since no device has global knowledge about the system in Calvin, the decision of whether to deploy a component to a device cannot refer to the global knowledge about the deployment. For example, it is not possible to specify that a maximum of (or at least) five devices shall execute a component since no device knows all other devices. This limits the expressiveness of the requirements.
3. In cases where requirements cannot be fulfilled by the system, the system should ideally be able to propose modifications. Of course, such modification proposals have to include human consent before being applied.

6.3.1 Deployment Workflow

The workflow for developing and deploying the software of IoT applications is outlined in Fig. 6.3. At design time, IoT developers create a set of (MontiThings) components and

¹The authors of Calvin refer to them as “capabilities” [AP17]. However, since the capabilities also include the location or name of a device, we refer to them as properties.

²What is IoT Plug and Play? [Online]. Available: <https://docs.microsoft.com/en-us/azure/iot-develop/overview-iot-plugin-and-play> Last accessed: 15.01.2022

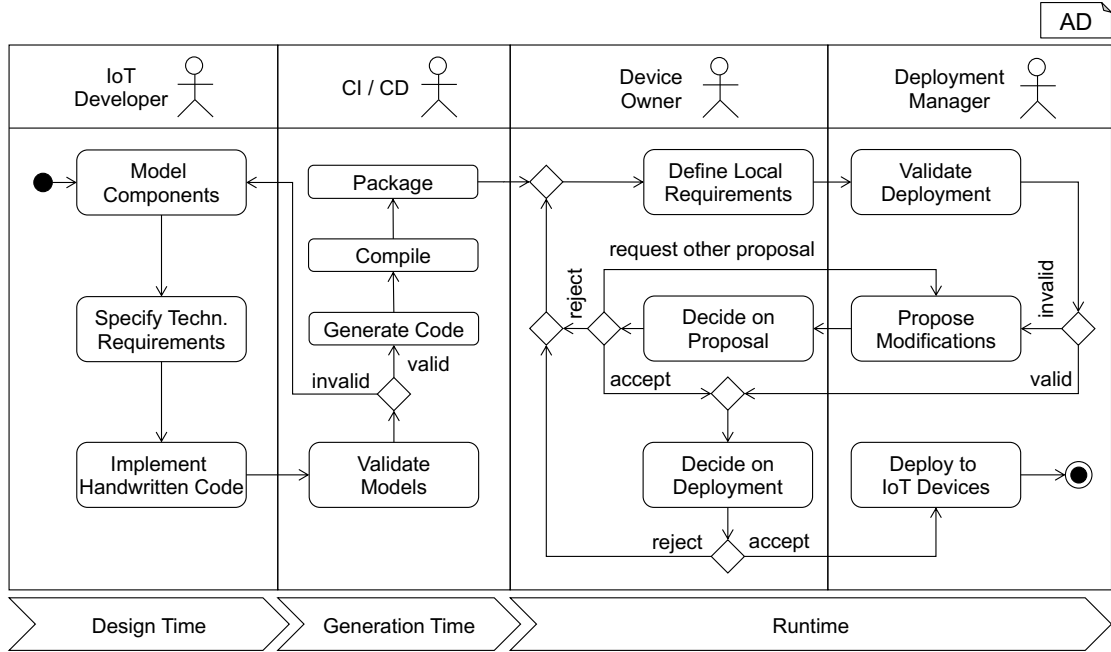


Figure 6.3: Workflow of developing and deploying software for IoT applications. Figure taken from [KKR⁺22a].

connect them in an architecture model. Components may also be tagged with technical requirements.

Definition 14 (Technical Requirement). *A technical requirement is a key-value pair of a property type and a property value. Each device can either fulfill a technical requirement or not.*

For example, a technical requirement would be that a device possesses a “DHT22” (value) “sensor” (type) for measuring temperature. This makes the requirements flexible enough that they can be used as an extension point for more complex ontologies like the one proposed in [CAF21]. Such ontologies are, however, often project-specific. Thus, MontiThings does not assume the usage of one specific ontology but instead leaves it to the IoT developer to decide if an ontology (and which ontology) may be beneficial in their specific projects. Additionally, components may be accompanied by handwritten code, as explained in Sec. 4.2.4.

Once the architecture is modeled, the artifacts get passed to a CI / continuous deployment (CD) pipeline that validates the models. In case the models are invalid, the

pipeline will result in an error message and the IoT developer has to fix the models. If they are valid, the pipeline triggers the code generation, compiles the code, and packages the binaries in (Docker) container images that can be provided in a container registry (*cf.* Chapter 5). If the application was to be deployed via an app store (*cf.* Chapter 3), these container images and technical requirements would be required information to deploy the applications.

Next, the device owner may define a set of *local requirements*. Local requirements differ from technical requirements in that they do not make statements about the technical properties of a device. In other words, technical requirements define which devices *can* execute a component, and local requirements define which devices *should* execute a component. As device owners have in-depth knowledge about the infrastructure they manage, they can make decisions about the infrastructure that the IoT developer cannot take. For example, a device owner could define a rule that prevents camera applications to be deployed in the bedroom. Besides such privacy-related requirements that restrict the deployment, device owners can also request the deployment of certain components in specific locations. For example, device owners could require a fire alarm to be deployed in every bedroom of a smart home to comply with their country's laws.

Definition 15 (Local Requirement). *A local requirement is a predicate logic formula over the sets of components, clients, and locations.*

Accordingly, the above example could also be expressed as

$$\forall \text{Room } r \ \exists \text{Device } d \ \exists \text{FireAlarmComp } c \ [B(r) \rightarrow [L(d, r) \wedge D(c, d)]],^3$$

where r is a room, d is a device, c is a fire alarm component, B is a unary predicate telling whether a room is a bedroom, L is a binary predicate determining whether a device is in a certain room, and D is a binary relation telling whether a component is deployed onto a device. As this example demonstrates, this notation may require a lot of relations (and mathematical syntax) to be understood by the user. Since device owners are not expected to have a background in mathematical logic, we decided to restrict the types of local requirements to a set of commonly used requirement types and make them easy to input via a web frontend. In addition, this limitation has considerably reduced the implementation effort for our prototype. This is in contrast to IoT deployment approaches like the one presented in [SDF⁺20], that directly specify similar constraints using a mathematical notation. Our deployment algorithm supports the following four kinds of local requirements:

³We use $\forall X x$ as a shorthand notation for $\forall x X(x)$ (and similarly $\exists Y y$ for $\exists y Y(y)$).

1. A component shall (not) be deployed at a specific location,
2. A location requires a (minimum, maximum, or exact) number of components to be deployed there,
3. A component requires a certain number of components (optionally in a similar location, *i.e.*, the same room, floor, or building).
4. Two components may not be deployed to the same device, and

Locations in these local requirements can of course still be used with quantors, *i.e.*, *exists* (\exists) and *for all* (\forall). This removes the need for device owners to express requirements that apply to all or any room, floor, or building more directly compared to setting up rules for each individual room, floor, or building.

Defining these local requirements is a very dynamic process dependent on the infrastructure. The current state of each device, *e.g.*, the availability of certain sensors, may not be known to the device owner, since IoT devices are managed “en masse” [TM17a]. Unlike when specifying models where their validity depends only on information expected to be possessed by the modeler, we expect a higher chance of making errors when defining such local requirements. Thus, MontiThings provides an interactive web application for defining these local requirements⁴.

After specifying the local requirements, the deployment manager tries to find a deployment fulfilling all requirements. To do so, the deployment manager utilizes a Prolog code generator that is explained in Sec. 6.3.3. If all requirements can be fulfilled, the software can be deployed immediately. In the contrary case that the local requirements cannot be fulfilled, the deployment manager can propose a set of modifications to the device owner. Such proposals can include modifications to the local requirements, *i.e.*, relaxing the requirements, or the purchase and installation of new hardware. The device owners then have to decide if they want to accept the proposals. If so, the proposed modifications are applied and the software can be deployed. If, however, the device owner does not want to accept the proposal, there are two options: The device owners either request a different proposal or they reject the deployment completely and cannot deploy the software.

6.3.2 Deployment System Overview

Fig. 6.4 gives an overview of the components of MontiThings’ deployment system. As already described in Sec. 6.2, the IoT developers upload their code to an online **model/code repository** such as GitHub or GitLab. This triggers a **CI / CD pipeline** that generates code from these models and packages them as (Docker) container images. Ideally, these images are built as multi-arch images so that they can be used on devices with different

⁴Screenshots can be found in Sec. 8.1.

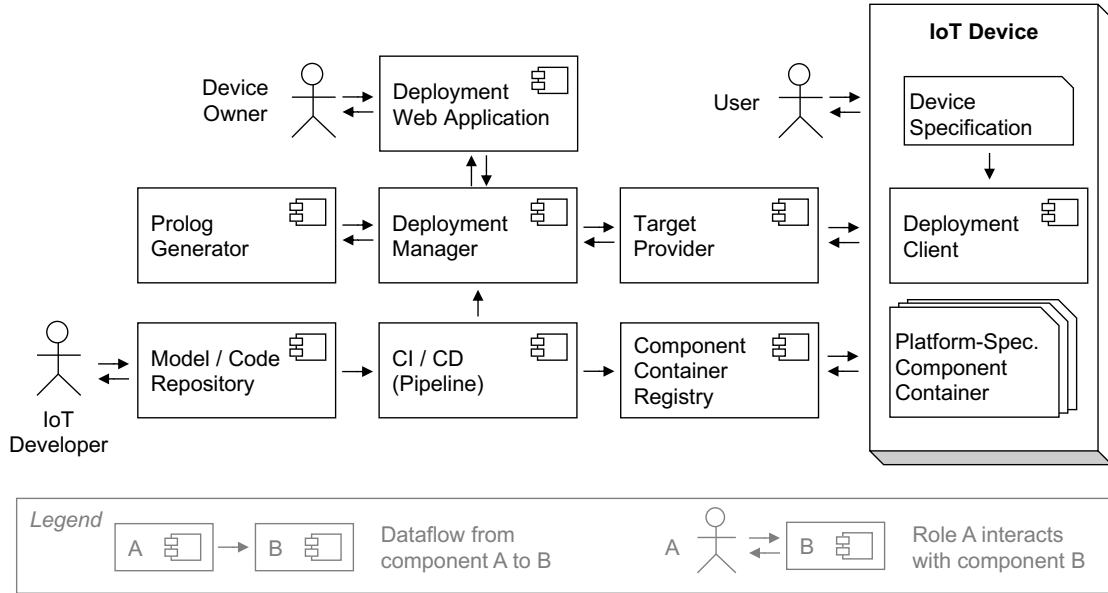


Figure 6.4: Overview of the components of the deployment system. Figure taken from [KKR⁺22a].

architectures ((TC1), cf. Sec. 5.3.3). These images are then uploaded to a **component container registry**. IoT devices can later send requests to the container registry to pull these containers from the registry. This part of the workflow is similar to normal GitOps workflows for the continuous deployment of (microservice) applications. Keeping the part of the development which involves the IoT developers, *i.e.*, the developers, similar to their existing workflow, the workflow likely becomes easier for them to adopt.

Device owners specify their requirements using a **deployment web application**. Moreover, they can manage their IoT devices and issue requests to the deployment manager. They can request to validate a deployment, accept or reject a modification proposal, deploy an application (after successful validation only), or stop an already deployed application. The **deployment manager** is the central component for coordinating the communication between the device owner (via the web application), the Prolog generator that calculates the deployments and modification proposals, and the target providers that abstract from the communication with the actual IoT devices. The deployment manager works in an event-based mode of operation, *i.e.*, calculating new deployments is triggered either by receiving a request from the web application or by getting notified about infrastructure changes by one of the target providers. The **Prolog generator** provides two services to the deployment manager. The first service takes a set of facts from the deployment manager and generates a Prolog file from them that encodes these facts as Prolog facts. The Prolog generator is described in more detail in Sec. 6.3.3. The sec-

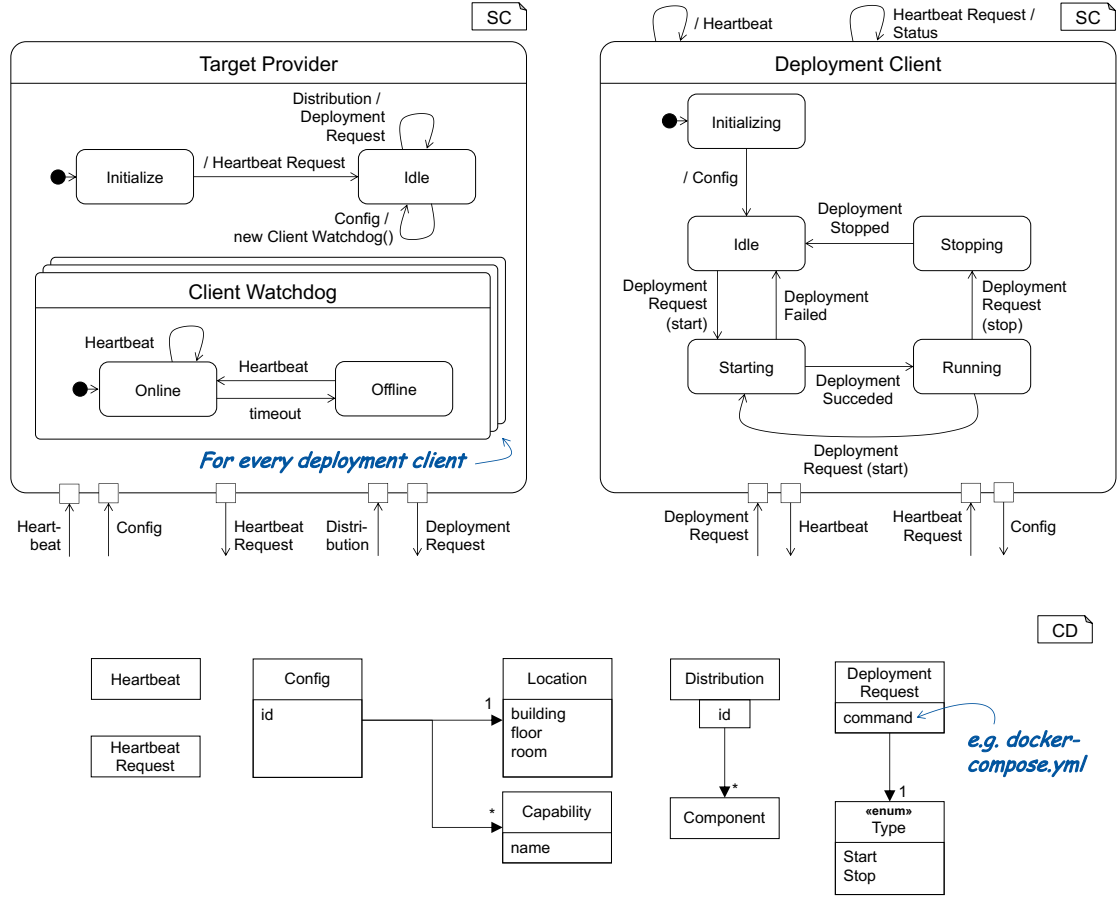


Figure 6.5: Interaction between the target providers and deployment clients. Figure taken from [KKR⁺22a].

ond service takes a set of requirements and generates Prolog queries from them that can be used to calculate which devices are supposed to execute which (MontiThings) components or request a modification if that is impossible. The deployment manager uses these two services to generate code and evaluates the queries of this code to calculate a valid deployment or modification proposals. If the result is a set of modification proposals, they will be forwarded to the web application and presented to the device owner. If a valid deployment is supposed to be deployed to the IoT devices the deployment manager informs the target providers to execute the deployment.

Target providers abstract from the actual communication with the IoT devices. Each IoT device executes a **deployment client** for this purpose that will interact with one target provider to execute its requests. This interaction is detailed in Fig. 6.5. The

target provider starts in an initialization state where it sets up connections to, *e.g.*, an MQTT broker that enables it to communicate with the deployment manager. When all connections are initialized, target providers request a heartbeat message from all of their clients and go into an idle state. In this idle state, the target provider waits for new clients to register and for the deployment manager to send new distributions that the target provider is supposed to apply. Clients register with a target provider by sending their configuration to the target provider. The config contains properties of the client, including a unique identifier, the client's location, and the capabilities of its device, *i.e.*, available hardware or software. The clients read this information from a **device specification** file that is located on each device. Once a client registers with a target provider, the target provider starts tracking its state. As long as the client sends regular heartbeat messages to the target provider (by default: at least every 20s), the target provider considers the client online. If the client did not send a heartbeat for the timeout period, the client is considered offline. Offline clients are ignored for deployments.

The deployment manager may request a deployment by sending a distribution to the target providers. Distributions are multimaps that map each client to a set of components it is supposed to execute. Upon receiving such a distribution, the target provider sends deployment requests to each client. These deployment requests contain the command the client is supposed to execute to deploy the components. For example, a deployment request may include a (generated) `docker-compose.yml` file.

The **deployment client** has five states:

1. In the *initialization* state, the client sets up its network connection to the target provider. Once the connection is established, it registers at the target provider by sending it its config. The config is sent as a retained (MQTT) message to handle situations where the client goes online before the target provider.
2. After the initialization, the client enters an *idle* state. In this state, it will wait for the target provider to send its deployment requests.
3. As soon as the client receives a deployment request, it enters a *starting* phase. In this phase, the necessary container images are downloaded and then started. If the deployment fails, *e.g.*, because the component is not available for the client's system architecture, it will reenter the idle state. If the deployment succeeds, it enters the running state.
4. In the *running* state, the components are executed as requested by the target provider. This state is only left if the target provider requests to either execute different components or stop the deployment. In the former case, the client goes back into the *starting* state with the new deployment request. In the latter case, the client enters the stopping state.

5. In the *stopping* state, the container images are stopped. After this is completed, the client reenters the idle state.

During all of these states, the clients will send heartbeat messages to the target provider to indicate they are still available even if they do not execute any components currently. Also, the clients will answer any requests for heartbeat messages with a heartbeat message. Note that the states of the clients are very similar to the states of Docker containers. This is because the main task of the clients is to manage the execution of (Docker) containers.

Overall, the target provider/deployment client structure is applied to four different technical systems: 1. Docker Compose, 2. Kubernetes, 3. GeneSIS [FN19, FNS⁺19, FNS⁺20], a research project funded by the European Commission, and 4. Microsoft Azure IoT Hub. For Docker Compose, we implemented our own client using Python that handles the communication with the target provider and calls `docker compose` using the `docker-compose.yml` files it receives from the target provider. For Kubernetes, the configuration is attached to the devices using *Labels*, *i.e.*, key-value pairs managed by Kubernetes. While Kubernetes (*i.e.*, k8s) normally incurs considerable overhead, there are low-powered IoT-focused implementations such as k3s, MicroK8s, or KubeEdge for using Kubernetes on IoT devices [Kay20, XSXH18]. For GeneSIS, we used their web interface for registering the devices. However, we cannot utilize their Docker executor and instead have to use the SSH executor. Using the `device_type` attribute, we tag the devices with the information normally provided via the device specification file. The reason for not using the Docker executor despite deploying Docker containers is that the Docker executor has several restrictions such as not allowing to access private container registries. For non-public projects, which we expect the most commercial applications of future IoT app stores to be, GeneSIS' Docker executor is, thus, currently not suitable for use without modifications. These limitations may, however, be removed in future iterations of the project, as they are not related to their concept. For the Microsoft Azure IoT Hub, we collect the devices from the IoT hub. The devices are tagged with their sensors and capabilities. For each device, we generate a “deployment manifest”⁵ and apply it using the IoT Hub. This manifest specifies which components a device is supposed to execute.

6.3.3 Prolog Code Generation

For the algorithm that calculates which device is supposed to execute which components, we use Prolog. Prolog was chosen because of its evaluation strategy: Since Prolog uses backtracking, it can not only decide whether a certain distribution of components to

⁵Microsoft Azure Docs: “Learn how to deploy modules and establish routes in IoT Edge” [Online]. Available: <https://docs.microsoft.com/en-us/azure/iot-edge/module-composition?view=iotedge-2020-11> Last accessed: 14.02.2022

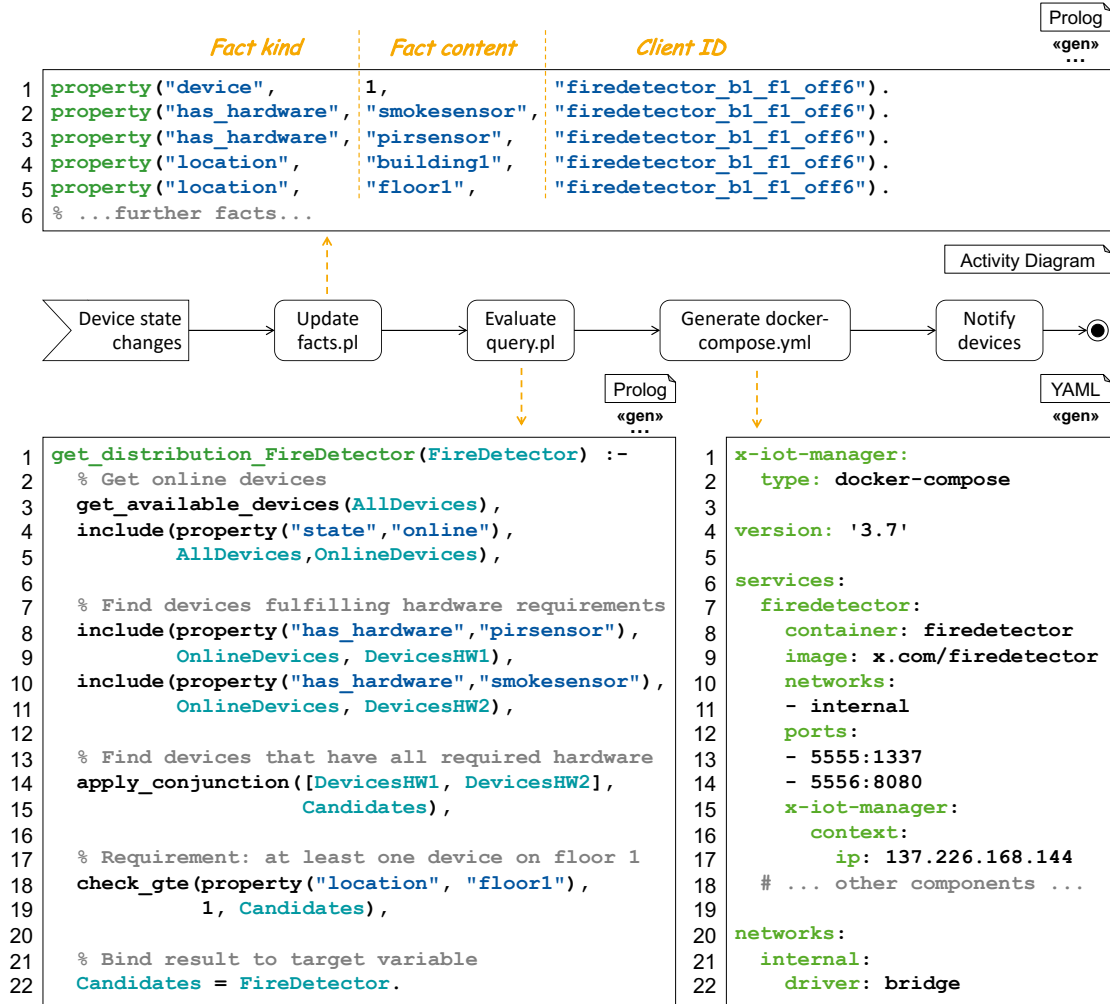
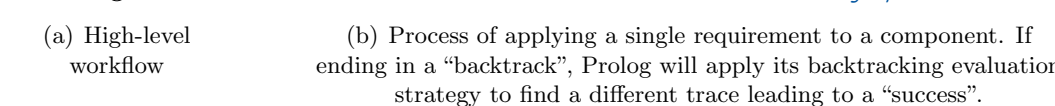


Figure 6.6: Workflow how the deployment manager utilizes Prolog and Docker Compose for deploying software to an IoT device. Figure adapted from [KRSW22].

devices fulfills all requirements, it will not stop to search for a solution once it encounters a statement that evaluates to false. Instead, it will continue to search for other solutions to fill all of its free variables with values that make the query evaluate to true. Thus, Prolog is the ideal choice for making modification proposals in the Process of finding out which device shall execute which components.

Before starting the Prolog generator, the deployment manager will apply two transformations to remove all quantors from requirements. Each exists-quantor is removed by replacing it with an anonymous variable. Thereby, Prolog is allowed to set these



After these transformations, the generator can be used to generate Prolog code. As already mentioned in Sec. 6.3.2, the Prolog generator offers to generate two files: The

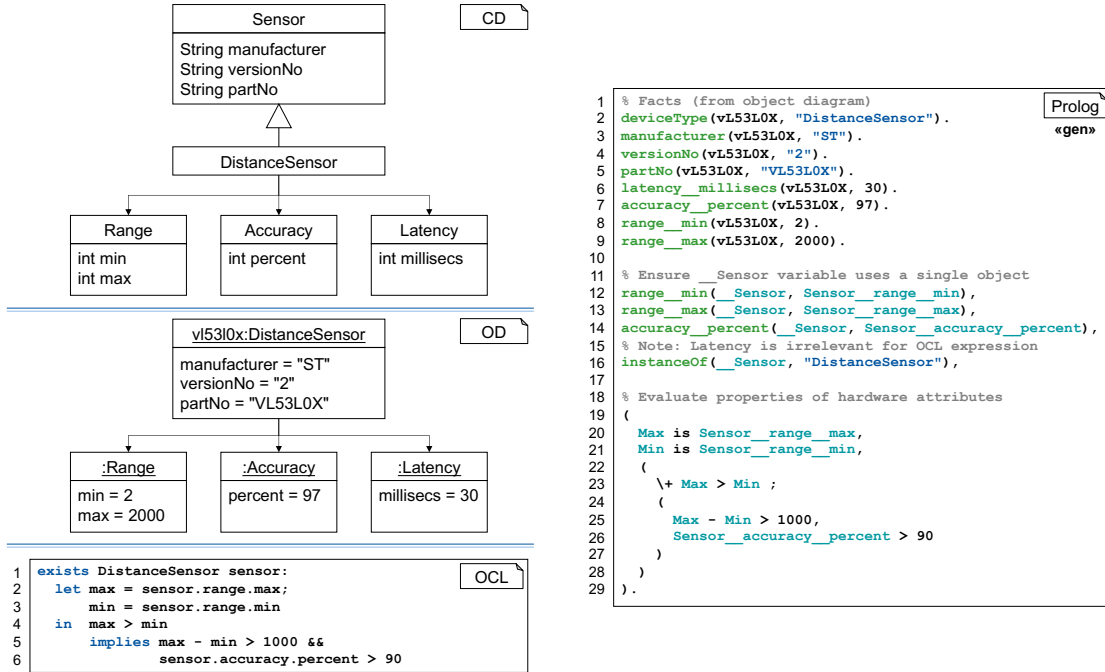


Figure 6.8: Example of IoT app store-based specification of technical requirements. The app store provides a hardware ontology as class diagram. Device developers associate each IoT with an object diagram that specifies its hardware. IoT application developers define the components' requirements using OCL. Prolog code generated from the object diagrams and OCL expressions checks whether a device can execute a component. Figure adapted from [BKK⁺22].

first file contains facts about the infrastructure, the second file contains the queries that calculate the deployment. The queries depend on the facts from the first file. Fig. 6.6 shows how the deployment manager uses both of these files to generate code and then calculate a deployment.

For generating the facts, Prolog takes the configurations and current states of the components as reported by the target providers as input. They are converted directly into Prolog facts, using a Prolog fact called `property`. A property consists of a fact kind, a fact content, and a client identifier (*cf.* Fig. 6.6). If a client is able to execute a component, the fact kinds and contents need to include all key-value pairs required by technical requirements.

For generating the queries that calculate which clients are supposed to execute a component, Prolog takes all components with their requirements as input. By referencing the `property` facts from the other generated Prolog file, Prolog can base its decisions on the current information about the infrastructure. By separating the facts from the

requirements the overhead for regenerating Prolog code can be reduced. If the information about the available clients is updated by the clients, only the facts need to be regenerated. If requirements are modified by the device owner, only the queries need to be regenerated. To prevent the Prolog algorithm from making unnecessarily moving components to other devices, *i.e.*, modifying the distribution of components to clients when it is not required, the algorithm can be initialized with an existing distribution as a starting point. In this case, it will only modify the distribution when the existing distribution does not fulfill the requirements. Fig. 6.7 details how Prolog generates queries from the information about the components. In essence, Prolog generates a query for each component. On a high level, this query works by first filtering all devices for the devices that are considered online by the target providers (Fig. 6.7(a)). Then, the generated code further filters this set of devices for the devices that fulfill the non-negotiable, *i.e.*, technical, requirements. Using this set of devices that can technically execute the component, Prolog then tries to find a set of devices that fulfills all local requirements.

For IoT app stores, this process can be improved using class diagrams, object diagrams, and OCL [BKK⁺22]. Fig. 6.8 gives an example of this. A class diagram provided by the IoT app store serves as a hardware ontology that describes what kind of properties which piece of hardware has. In the example, a sensor has a manufacturer, a version number, and a part number. Further, distance sensors are sensors that also have a range, accuracy and latency. For each specific IoT device, the device developer provides an object diagram in accordance with the app store’s class diagram. In commercial implementations of app stores, this could be required by device certifications, as they already exist for currently sold devices, *e.g.*, “Works with Apple HomeKit”. The object diagram specifies the concrete hardware of the device, *e.g.*, sensors or actuators. The IoT application developer specifies the technical requirements of each component using OCL expressions. These expressions define which hardware a device needs to possess to be able to execute the component. Using OCL these requirements can be more complex than the requirements referring to property statements. Similar to the property-based approach, the Prolog generator takes both the object diagram and the OCL expressions and converts them into facts and queries. The object diagram is converted into Prolog facts. As Prolog is not object-oriented, the object diagrams are flattened in this process. The OCL expressions are translated to Prolog using a pretty-printer based on the *Visitor* pattern [GHJV95] as provided by MontiCore [HKR21] that traverses the AST of the expression. While doing so, the pretty-printer replaces operators and constructs from the OCL using equivalent Prolog constructs, *e.g.*, replacing `A implies B` with `\+ A ; B`. For variable assignments as in `exists`-expressions, the generator creates free Prolog variables and ensures that all attributes of the object represented by this variable belong to the same variable. Using the `instanceOf` expression together with the `device-Type` fact, the generated code ensures that the inheritance hierarchy of the original class diagram is kept, *i.e.*, a free variable representing an object may also match a subclass of the requested class as per Liskov’s substitution principle [Lis87]. In case the device is

able to execute the software component, Prolog is able to find a solution to the query. Otherwise, Prolog will fail to find a solution. While this approach is more complicated than the property-based approach, it further decouples device and software developers and enables greater flexibility in choosing hardware, *i.e.*, enables developers to specify requirements to the hardware without *naming* the required hardware. In practice, the app store also needs to enforce standards, *e.g.*, on the drivers of the hardware to enable software components to access the hardware in a standardized way.

The process of trying to find a set of devices that fulfill all local requirements is shown in Fig. 6.7(b). For each local requirement, Prolog first checks if the requirement is already fulfilled by the set of clients that Prolog currently plans to execute the component on (①). If this is the case, Prolog can stop any further evaluation and continue with the next requirement. In case the requirement is not fulfilled, Prolog differentiates between cases where there are too many instances of the component planned and cases where Prolog planned too few instances to fulfill the requirement (②). If there are too few, Prolog proposes to buy new hardware that fulfills the requirements (③). In case the device owner discards this proposal, Prolog will check if the violated requirement can be further relaxed (⑥) and if so propose to relax the requirement (⑦). If Prolog planned too many components in step ②, Prolog tries to remove some of the components and tests if this leads to a violation of the previously validated requirements. At first sight, this procedure might seem unnecessary, as there is a separate query for each component type. However, since local requirements may also specify, *e.g.*, incompatibilities preventing two components from being executed by the same client, removing one particular instance of a component and adding an instance on a different client may enable Prolog to fulfill the requirement. In case Prolog cannot fulfill one of the previously checked requirements, Prolog will continue to apply its backtracking strategy to find a different set of component instances that could be dropped (⑤; Yes-path). If Prolog does not succeed by removing component instances (⑤; No-path), it will continue by trying to relax requirements as explained above (⑥, ⑦).

The strategy for relaxing requirements of different forms is set by the code generator. In general, applying a modification works by allowing Prolog to accept that requirement is not fulfilled (line 4 of the Pseudocode in Fig. 6.7(b)), but expecting Prolog to accept a proposed change in return for not fulfilling the requirement (line 5) and keeping track of the used proposals (line 6). This can be continued for as many requirements as needed. It is also possible to completely reject a requirement by using `True` instead of a proposal. The strategy set by the code generator is to generally count away from the target number of component instances requested by the requirement. For requirements that request “at least n ” instances, this means proposing $n - 1, n - 2, \dots, 0$ component instances. For requirements that request “at most n ” instances, this means proposing $n + 1, n + 2, \dots, |\mathbb{C}|$ component instances, where \mathbb{C} is the set of compatible and available clients. In this prototypical implementation, our Prolog generator does not account for instantiating a component multiple times on the same client.

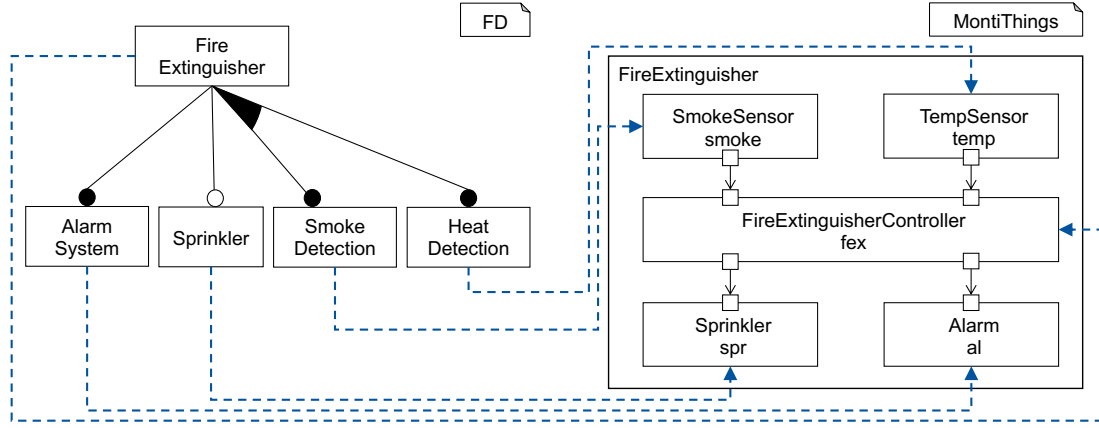


Figure 6.9: A fire extinguisher application for a smart home. IoT developers tag features with the component instances that implement them. Tagging is shown using the dashed arrows. Device owners can choose a feature configuration without knowing which components are required to implement the feature. Figure taken from [BKK⁺22].

To increase usability and not flood the device owner with a lot of proposals that lead to a failure later in the process even if accepted, Prolog will always assume the device owner accepts all proposals. Then, after Prolog found a set of proposals that could be used to create a valid deployment, the web application will present all of the necessary proposals to the device owner as a set of proposals. This is the reason why Prolog keeps track of the used proposals (line 6 in Fig. 6.7(b)). The device owner can then choose to accept the set of proposals or reject it. In case the device owner rejects the proposals, this will trigger the backtracking indicated by the “failure”-final nodes in Fig. 6.7(b). Prolog will then continue to propose different sets of proposals as already explained in Fig. 6.3.

6.4 Feature-based Deployment

Up to this point, the deployment process expected the device owner to have extensive knowledge of the software to be deployed. By utilizing the feature diagram language from the MontiVerse in combination with a tagging, it is possible to raise the level of abstraction for the device owner. For this, one needs to consider the MontiThings architecture a 150% model. 150% models contain all variants of a system within a single model. This approach of combining architecture models with feature diagrams has already been used successfully in the automotive domain [GHK⁺08, GKPR08].

The IoT developers, who have the knowledge of what distinct features their system shall offer, tag each feature with the components required to execute this feature. By

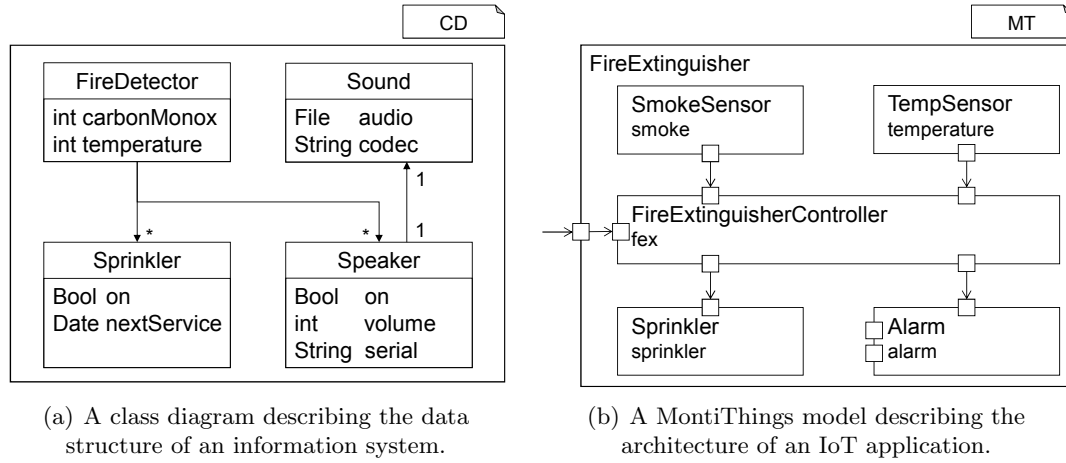


Figure 6.10: Models of a fire extinguishing application. The information system and the IoT system can be connected by connecting attributes of the class diagram to ports within the architecture. Figure taken from [KMR⁺20b].

selecting a set of features in a *feature configuration*, device owners implicitly decide which components are needed to execute the system that offers all of the features they request. They do so without knowing the underlying tagging model, thus operating a higher level of abstraction. Fig. 6.9 shows an example of how a feature diagram can be used in combination with such a tagging model to connect a feature diagram to a MontiThings architecture.

This feature configuration can be used to generate a set of local requirements. These local requirements, as explained in Sec. 6.3, can then be used to generate Prolog code that checks if the architecture can be deployed to the available set of devices. In a similar manner to how the Prolog generator makes modification proposals, it is also possible to find, *e.g.*, the maximum configuration that could be deployed to guide the device owner. For this, the feature diagram tool from the MontiVerse’s feature diagram language can be used to first calculate a sorted list of the configurations by their size (disregarding available devices). Note that the valid feature configurations form only a partial order, *i.e.*, there may be multiple configurations of the same size. The generated Prolog code then checks, if (one of) the largest configuration can be deployed using the available device. If not, it will continue by checking the next smaller configuration in the list. Once a deployable configuration is found, this can be communicated to the user. Similarly, by generating local rules from the feature configurations, it can also be possible to tell the user which additional hardware would be necessary to make a certain feature deployable.

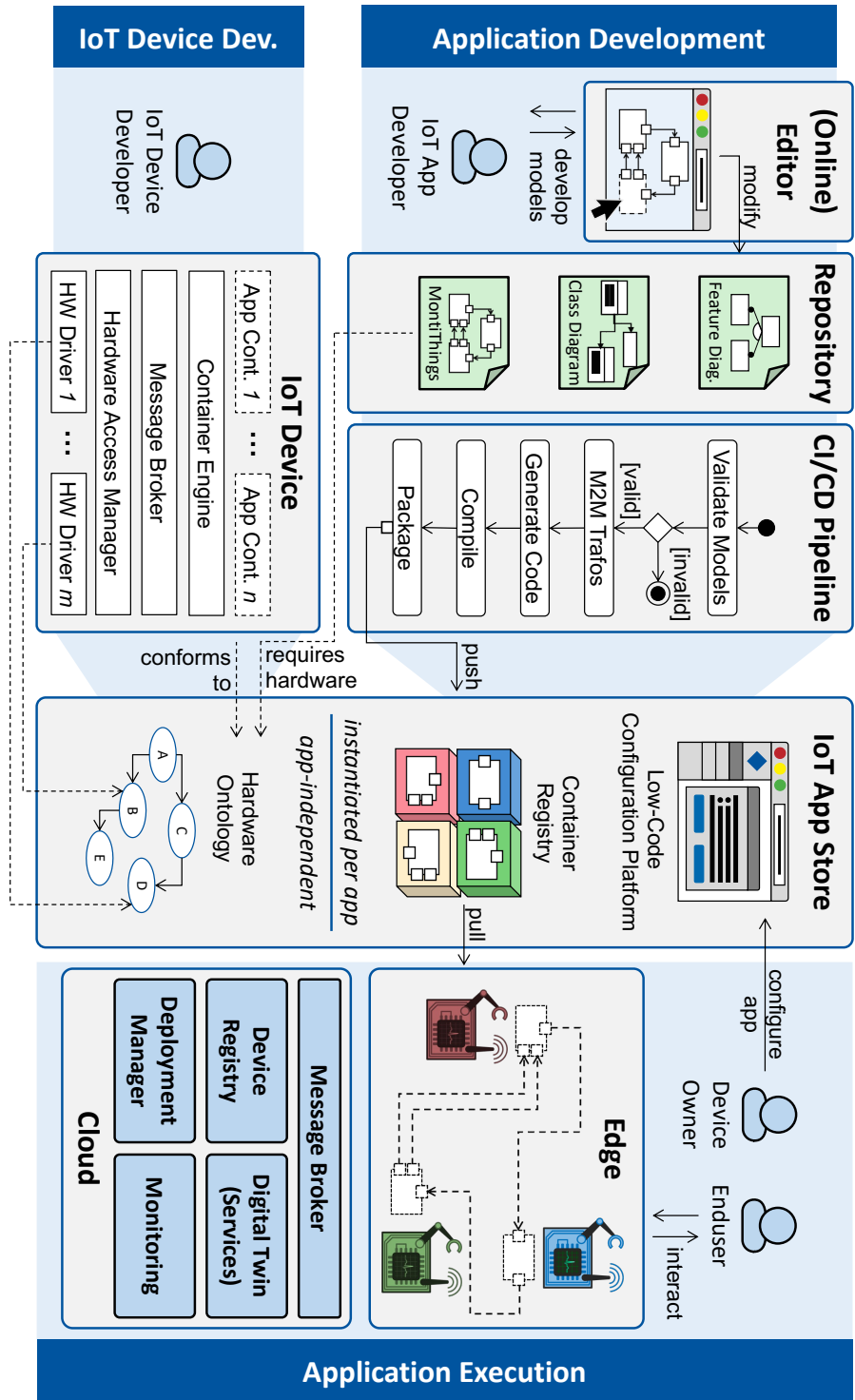


Figure 6.11: Overview of the app store concept. Application development and device development are decoupled from each other. By specifying a hardware ontology, the app store ensures compatibility between software and hardware. Figure taken from [BKK⁺22].

6.5 Model-driven App Store Concept

The development techniques shown in the previous sections and chapters converge into an app store concept that decouples hardware and software development. Fig. 6.11 provides an overview of this concept.

Application development is carried out by the IoT app developers, who use MontiThings to specify an application and upload it to a repository. In particular, they also specify the technical requirements of the components (*cf.* Sec. 6.3.3) and, with the help of a feature diagram, the high-level features of the application (*cf.* Sec. 6.4). The specification of the technical requirements of the components takes place in the form of OCL expressions (*cf.* Sec. 6.3.3). The specification of high-level features is optional and only serves to facilitate the configuration of the application. Code generators use the uploaded models to create a set of container images that are uploaded to a registry provided by the app store. From there, the IoT devices can download them again when the application is deployed. In addition to the image registry, the app store offers the web application with which device owners can configure their IoT applications (*cf.* Sec. 6.3.2).

In addition, the app store provides a hardware ontology in the form of a class diagram. This class diagram specifies which hardware is supported by the app store. In particular, the hardware can be grouped by subclassing so that, for example, a DHT22 sensor represents a special kind of temperature sensor, which in turn represents a special type of sensor. This class diagram provides a common basis for the development of hardware and software. IoT app developers write the OCL expressions that specify the technical requirements of their components against this class diagram. Similarly, IoT device developers specify the properties of their devices based on this class diagram. For this purpose, they create an object diagram for each IoT device that represents its properties. In particular, a certain flexibility can be maintained by this approach. For example, an IoT application developer can issue a request that requires a temperature sensor that can detect temperature values between 5 °C and 30 °C without requiring a specific sensor.

Matching the properties specified in the object diagram, the IoT device developers develop hardware drivers. These drivers connect to the Hardware Access Manager and enable the components of the application to access the sensors and actuators of the device (*cf.* Sec. 5.2.2). When a container is deployed, it requests access to the required sensors and actuators via the Hardware Access Manager. In addition to a message broker, via which drivers and components communicate with each other, the IoT devices contain a container engine with the help of which the containers of the IoT application are executed.

From the perspective of device owners and end users, the procedure is no different from the procedure described in the previous sections: The device owners use a web application to specify the deployment of their IoT applications there, whereupon a deployment is calculated using generated Prolog and counterproposals are made if necessary.

6.6 Integration with Model-driven Information Systems: Synthesizing Digital Twins

As explained in Fig. 6.1, some IoT applications are required to be connected to a digital twin. This section discusses how model-driven IoT applications and model-driven information systems can be developed in an integrated fashion to synthesize such digital twins using a tagging language that connects their models (Sec. 6.6). The resulting system keeps the IoT application and the information system synchronized.

As described in Sec. 6.2, our process for developing an IoT application that is integrated with a MontiGem information system via a digital twin starts by first developing both systems. The resulting class diagrams and MontiThings architectures can then be connected using a tagging language. Fig. 6.10 gives a motivating example using a fire extinguisher application from a smart home. MontiGem uses the class diagram (Fig. 6.10(a)) to generate a MySQL database schema. This application is modeled to be similar to Google's *Nest Protect*⁶ fire alarm.

Now that the models are developed, the task of the integrator is to identify which ports of the MontiThings architecture shall be synchronized to which attributes of the class diagram. Our mechanism synchronizes ports with attributes of the class diagram because ports and attributes are the model elements that hold the actual data that is used within the system. Such a connection can be either one-directional or bi-directional, *i.e.*, only inform one of the two systems about the data of the other or enable both systems to synchronize to changes of the other system. For example, the integrator wants to synchronize the `carbonMonox` attribute of the `FireDetector` class in the class diagram to the outgoing port. Thus, whenever a message is sent via the (outgoing) port of the `SmokeSensor` component, the information system needs to be informed so that it can update its database value for the `carbonMonox` attribute in the class diagram.

This digital twin synchronization mechanism can also be used in combination with underspecification. In the fire extinguishing example, the `Alarm` component has two ports that are not connected. These two ports are thought for telling the `Alarm` component at which volume to playback the alarm sound and which sound file to play as an alarm. The latter is necessary for more modern fire alarms like the Google Nest Protect that not only can play uncomfortably loud siren sounds but also, *e.g.*, read out text messages in case a fire is detected. Under normal circumstances, these two unconnected ports would never receive any value. Thus, this model would be considered invalid by a context condition. If, however, a connection to a digital twin is defined on these two ports, this situation can be corrected. In this situation, values to these ports could also be provided by sending messages from the information system to the unconnected port.

⁶Nest Protect Product Website. [Online]. Available: https://store.google.com/product/nest_protect_2nd_gen_specs Last checked: 12.12.2021

Tagging Model

```

1 // Objects of the Sound and Speaker classes serve as
2 // digital twins for CPS devices that use the value of
3 // Speaker.serial as identifier
4 identify Sound by attr Speaker.serial
5 identify Speaker by attr Speaker.serial
6
7 // Automatically create and link a digital twin when
8 // a device first connects to the IS
9 auto identify FireDetector
10
11 // Send data from the CPS architecture to the IS
12 connect port smoke.value
13 --> attr FireDetector.carbonMonox
14
15 connect port temperature.value
16 --> attr FireDetector.temperature
17
18 // Send data from the IS to the CPS architecture
19 connect attr Sprinkler.on --> port sprinkler.on
20 connect attr Speaker.on --> port alarm.on
21 connect attr Speaker.volume --> port alarm.volume
22
23 connect attr Speaker.sound.audio --> port alarm.sound

```

class from domain model *class from domain model. Same as before or reachable via to-1-association.* *attribute of the class before the dot. Stores device identifier.*

class name from domain model.

architecture component instance *outgoing port* *"Use messages from temperatureSensor's port 'value' to update the temperature attribute of the FireDetector class"*

class name from domain model *attribute name*

"If 'Speaker.volume' changes, inject the new value into alarm's 'volume' port"

class name from domain model *(association +) attribute* *component instance* *port*

Figure 6.12: Example of the tagging language that connects the class diagram and the MontiThings model from Fig. 6.10. Figure taken from [KMR⁺20b].

The case of connecting to incoming ports can also be applied to already connected incoming ports. For example, the integrator could also decide to synchronize the `on` attribute of the `Speaker` class with the incoming port of the `alarm` component. In this case, whenever the `on` attribute is changed in the database the architecture needs to be informed about this change by sending a message to this port. This synchronization enables users of the information system to trigger a test alarm by setting the `on` attribute in the database. Then the `alarm` component would receive the value set by the user and turn on or off the alarm accordingly. This case, however, requires more synchronization logic to resolve conflicts between values set by the information system and the components that might send values to that port.

Overall, this leaves us with three cases to handle digital twin synthesization:

1. Sending values from an outgoing port to the information system,
2. Receiving values from the information system at an unconnected incoming port (underspecification), and
3. Receiving values from the information system at an already connected incoming port.

Fig. 6.12 shows an example of tagging language for connecting the IoT system and the information system. Besides making the connections between the ports and attributes, an important task of the tagging model is to specify to which IoT device instances of a class shall be attributed. This is important if a system consists of more than one instance of a component/class. In this case, the information system needs to know to which device it needs to send data in case it is updated. Vice versa, the information system needs to know which database entry to update if a message is received from an IoT device. For this purpose, the tagging language offers the `identify` keyword. This keyword can be used in two ways: In the manual case, the integrator may specify that a device identifies itself using an attribute from the class diagram when sending an update for an object. This attribute can belong to the same class as the updated object or come from an object that is connected to the updated object using a *to-one*, *i.e.*, a unique, association.

This manual identification is shown in lines 4 (to-one association) and 5 (same class), respectively. The second method of matching component instances to objects is to use an automatic connection. In this case, the digital twin will automatically create a new object of the according class in the database once an IoT device first communicates with the information system. When automatically creating a new object, the information system will identify the IoT device using an implicitly created attribute and a unique identifier of the IoT client included in all messages from the IoT device to the information system, *e.g.*, its MAC address (*cf.* the `id` attribute in the config of Fig. 6.5). At first glance, the mode for manual identification of IoT devices may seem unnecessary. It is, however,

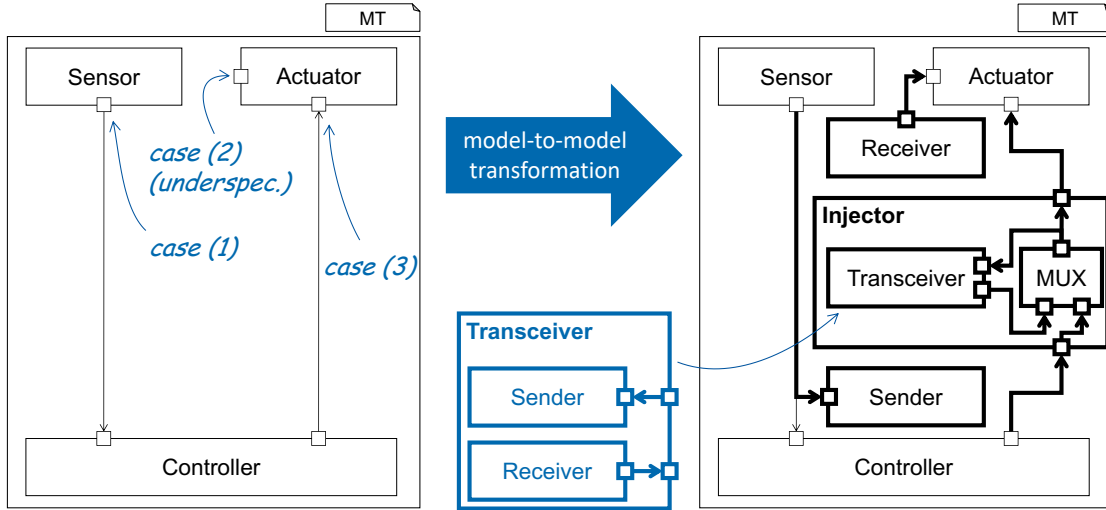


Figure 6.13: Model-to-model transformations for keeping adding synchronization elements to MontiThings models. Elements created by model-to-model transformations are shown in bold. Figure adapted from [KMR⁺20b].

important to note that the automatic mode prevents the users of the information system to add IoT devices to the system before they first connect to the system. In some scenarios, users might, however, want to create the digital twin before the devices are connected or even produced. These cases are covered by manual identification.

Lines 9 - 13 show how to send data from the IoT system to the information system by connecting an outgoing port of the MontiThings architecture to an attribute of the class diagram. In this example, the `carbonMonox` and `temperature` attributes from the `FireDetector` class will be updated using the outgoing ports of the `smoke` and `temperature` components. The inverse case of updating the IoT system using data from the information system is shown in lines 14 - 18. Here, the `sprinkler` and `alarm` components are updated using the attributes of the `Sprinkler`, `Speaker`, and `Sound` classes. Note that attribute can also use the attributes of objects associated with it using a *to-one* association.

Model-to-model transformations use the information in this tagging model to extend both the class diagram and the MontiThings model with elements that keep the data synchronized between the information system and the IoT system generated from these models. Fig. 6.13 shows the three model-to-model transformations that are used to extend the MontiThings models. These three transformations each address one of the cases listed above. If an outgoing port is tagged to be synchronized to a database entry of the information system, the transformation instantiates a new **Sender** component to the architecture. This **Sender** component has a single input port that has the same

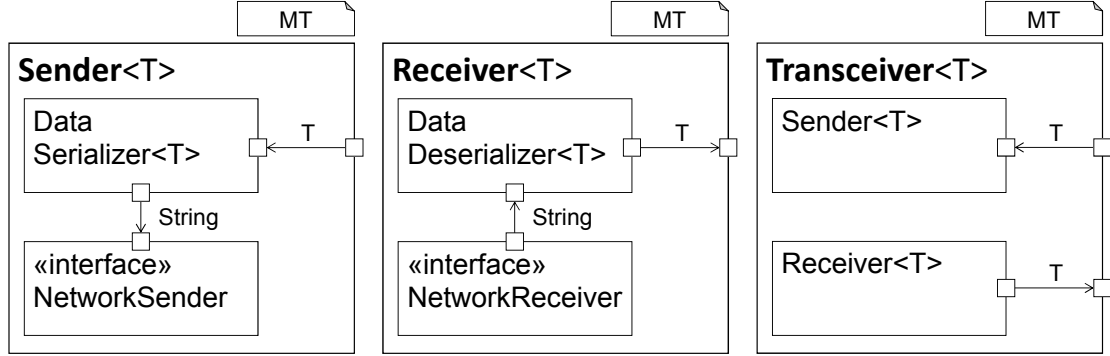


Figure 6.14: Generic components for exchanging data with the information system. Generated elements are shown in bold. Figure adapted from [KMR⁺20b].

type as the tagged outgoing port. For this purpose, the Sender component has a generic type parameter that is used to specify the type of the port. The port of the Sender component is connected to the tagged port. Internally, whenever the Sender component receives a message from the tagged port, it serializes the message into a JSON representation and then sends the serialized message to the information system (Fig. 6.14).

The second case of tagging an unconnected incoming port is handled inversely to the first case: The model-to-model transformation creates a **Receiver** component. Internally, this component is connected to the information system and may receive messages from the information system (Fig. 6.14). The information system will do so when the entry in the database that corresponds to the tagged port is updated. When the Receiver component receives a message from the information system, it deserializes it and forwards it on its outgoing port. The outgoing port's type is determined by the generic type parameter of the Receiver component. The model-to-model transformation connects this outgoing port to the tagged (unconnected) port. Thus, whenever the according database entry is updated, the tagged port will receive a message via the Receiver component.

The third case of tagging an already connected incoming port is more intricate to solve than the other two cases. In this case, the transformation needs to ensure that both the component that was already connected to the tagged port and the information system can send messages to the tagged port. The transformation replaces the already existing connector with an **Injector** component that has an incoming and an outgoing port. Their types are determined by the generic parameter of the component. The incoming port of the Injector component is connected to the port the tagged port was connected to before replacing the connector. The outgoing port is connected to the tagged port.

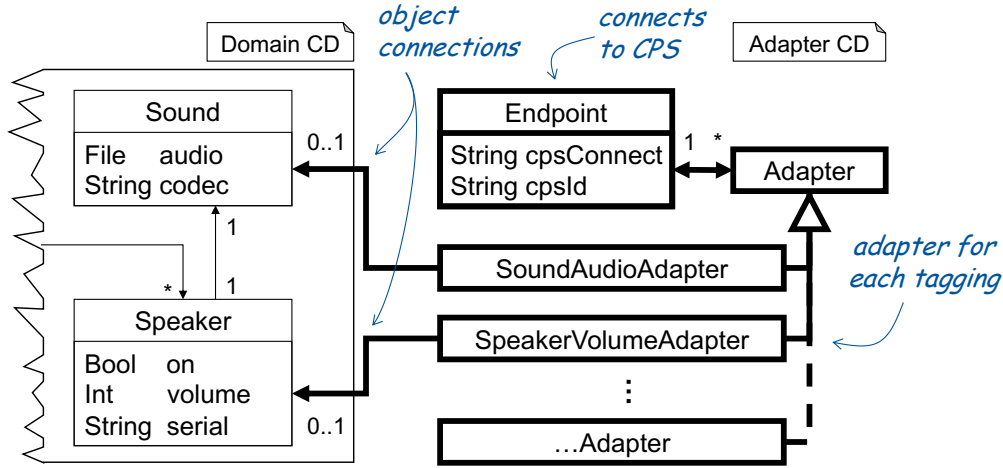


Figure 6.15: Extension of the information system’s class diagrams to synchronize with the IoT system. Elements created by model-to-model transformations are shown in bold. Figure taken from [KMR⁺20b].

Internally, the **Injector** component consists of a **Transceiver** component, that combines a **Sender** and a **Receiver** component, and a multiplexer (MUX) component. The **Transceiver** ensures that messages coming from the port the tagged port was already connected to are forwarded to the information system. It also receives messages from the information system that are forwarded to the tagged port. Having both the information system and a component of the IoT system act as a data provider for the tagged port can, however, lead to various problems. The most serious problem arises when the information system and the component disagree on which message to send to the tagged port. In this case, they may immediately override the decision of each other by sending another message. This problem is, for example, visible in the Arduino IoT Cloud. The Arduino IoT cloud enables users to set variables of the Arduino using a web interface. However, if the Arduino itself also continuously updates the variable’s value, any value set by the user in the web interface may be immediately overwritten. This effectively makes the web interface useless in these cases. To solve this situation, MontiThings uses the MUX component. The MUX component gives the web interfaces messages precedence over the messages coming from the other components of the architecture. The information system can cancel its priority by sending an empty message. Since the messages are still sent to the information system, the information system is aware of the mismatch between the user-set value and the values sent by the IoT system to the tagged port. This allows the information system to present this information in an appropriate way to the user and give the user control on how to resolve the conflict. Overall, we expect giving the user the control a better solution than automatically overwriting user-set values without

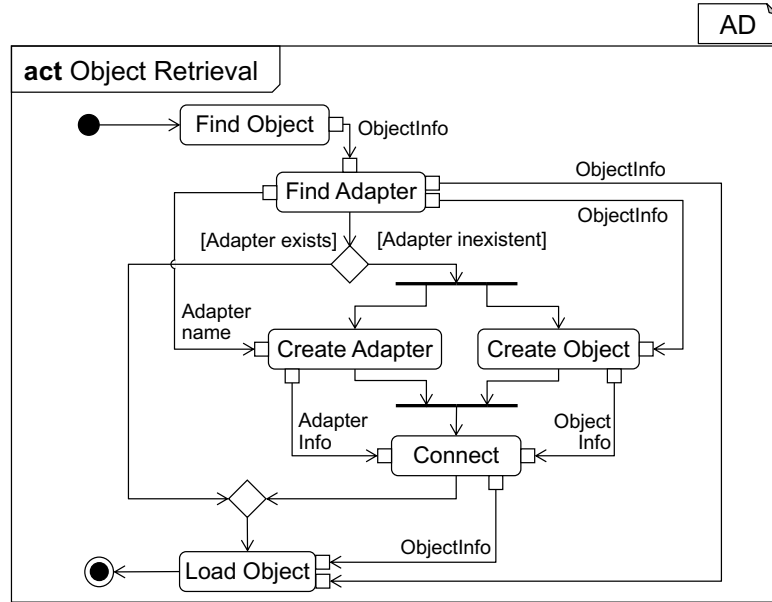


Figure 6.16: Process of retrieving an object from the information system’s database in response to receiving a message from the IoT system. Figure taken from [KMR⁺20b].

notice. In our fire alarm example, this behavior can for example be used to trigger a test alarm. Even though the sensors would indicate to turn the alarm off immediately, the user can force the alarm to stay on for a defined period of time and then choose to revert to automatic alarm triggering.

Now that we have shown the adaptations to the MontiThings models, we discuss the adaptations to the information system. Fig. 6.15 shows the adaptations to the information system’s class diagram. The *domain class diagram* refers to the classes specified manually by the developers, the *adapter class diagram* contains the classes created automatically base on the tagging. The **Endpoint** class keeps track of (network) connections to IoT systems. Its `cpsConnect` attribute connects the information that tells the information system how to send data to the corresponding IoT device. In the simplest case, this might be a socket, *i.e.*, an Internet Protocol (IP) address and port number. Other implementations may, however, choose to set this variable differently according to their communication protocols, *e.g.*, to an MQTT topic. The `cpsId` attribute contains the identifier sent by the IoT system with each message (*cf.* Fig. 6.12 lines 1-8), *e.g.*, its MAC address or serial number.

At first sight, the `cpsConnect` and `cpsId` attributes may seem redundant as the `cpsConnect` attribute in some cases can also be used to identify the device. However, it

is necessary to have both the `cpsConnect` and `cpsId` attributes. First of all, the value of the `cpsConnect` may be unknown at design time. This is the case if, for example, the IoT devices use dynamically assigned IP addresses. Secondly, the `cpsConnect` attribute may not necessarily be unique. For example, if multiple IoT devices are deployed behind a NAT gateway, multiple devices might share the same IP address. Moreover, the `cpsConnect` attribute may change over the IoT system's lifetime. For example, if the IoT device is for example connected using a cellular network it usually does not have a static IP. If the same device connects again using a different `cpsConnect` information but the same `cpsId`, the information system could use this information to update the `cpsConnect` attribute.

Besides handling the network connection, the `Endpoint` class also keeps track of the instances of the **Adapter** class associated with the IoT device. Each instance of an adapter can be connected to an object from the domain class diagram, and therefore, an entry in the database. The generator creates subclasses of the adapter class for all tagged attributes. Using the adapter, it is known which database exactly is associated with a specific port in case there are multiple instances of a class in the domain class diagram. The adapter also creates a message and sends it to the corresponding IoT device when the object is updated. For establishing a connection to the IoT device, it uses the `Endpoint`.

The activity diagram in Fig. 6.16 shows the process of accessing the database using the adapters. After retrieving the info which object shall be retrieved from the database, the information system searches for a corresponding adapter. In case the adapter exists, it can be used to load the object, which ends the object retrieval. In case the adapter does not exist, this means that the information system is asked to resolve an object that doesn't yet exist in the database. This situation is caused if an attribute is tagged to be identified automatically and a previously unseen port tries to connect to the information system. In this case, the information system creates both the object in the database and the adapter and connects them with each other. The newly created adapter can then be used to retrieve the object from the database.

6.7 Discussion

Overall, this chapter showed how model-driven development can support the deployment (**RQ2**) and integration with digital twins (**RQ7**) of IoT applications. Especially, we showed how code generation and Prolog can be combined for making modification proposals. Compared to the other solutions MontiThings offers the most end-to-end solution for model-driven IoT application development and deployment. By utilizing Prolog, MontiThings is the only system that can offer modification proposals for deployment rules. Using feature diagrams, this deployment procedure can be further abstracted to hide architecture-level knowledge from device owners who might not be technical ex-

perts but still want to deploy a certain set of features (**MC1**). Since IoT devices need to be managed “en masse” [TM17a], these are important steps in making large IoT deployments more user-friendly.

Table 6.1 gives an overview of how MontiThings compares to other solutions from both academia and industry. It evaluates related work based on the following criteria:

Modeling Language Does the tool offer a modeling language for specifying the behavior of the IoT application?

Code Generator Does the tool offer a code generator for its modeling language? Fulfilling this category naturally requires that the *Modeling Language* category is also fulfilled.

Automated Deployment Does the tool offer an automated deployment, *i.e.*, tools to automatically transfer software to IoT devices and an algorithm for deciding which devices shall execute which software?

Dynamic Self-Adaptation Is the tool able to adapt the deployment automatically to changes in the infrastructure, *i.e.*, new or failing devices? This can also be fulfilled independently of the dynamic deployment if the tool offers means for the developers to specify how to react in case of an infrastructure modification.

Rule-based Deployment Does the tool enable developers to specify rules or requirements that specify which devices shall execute which software? Note that this can only be fulfilled by tools that offer an automated deployment.

Modification Proposals (Rules) Is the tool capable of offering modification proposals to the rules of a rule-based deployment if not all rules can be fulfilled? This category can only be fulfilled by tools that offer a rule-based deployment.

Modification Proposals (Infrastructure) Is the tool capable of proposing modifications to the infrastructure that would allow fulfilling all rules in case not all user-defined rules can be fulfilled. This, of course, requires that the tool offers a rule-based deployment.

Automated Digital Twin Does the tool automatically provide the necessary software to keep the IoT devices and their accompanying information system synchronized?

Without question, the offerings from commercial cloud providers like AWS or Microsoft Azure are more mature than MontiThings’ implementation, a research prototype, can be with the limited time and resources of a dissertation. Especially, all cloud providers provide security concepts that are largely out of the scope of this thesis (*cf.* Sec. 3.7). They typically work by offering *connection strings* for each IoT device, *i.e.*, generating a string for each device that acts as an application programming interface (API) key

Table 6.1: Overview of related IoT modeling and deployment approaches. ● = fulfilled, ◐ = partly fulfilled, ○ = not fulfilled. Table and annotations largely taken from [KKR⁺22a].

IoT Tool	Modeling Language	Code Generator	Automated Deployment	Dyn. Self-Adaptation	Rule-based Deployment	Modification Proposals		Automated Digital Twin
						Rules	Infrastr.	
Arduino IoT Cloud	○	○	○	○	○	○	○	●
AWS Greengrass	○	○	●	◐ ¹	●	○	○	○ ²
Azure IoT	○	○	●	◐ ¹	●	○	○	○ ²
balenaCloud	○	○	●	◐ ¹	○	○	○	○
CapeCode [BJK ⁺ 18]	●	●	○ ³	● ⁴	○	○	○	○
Ericsson Calvin [AP17]	●	◐ ⁵	●	●	●	○	○	○
DIANE [VSID15]	●	○	●	◐	●	○	○	○
Distributed Node-RED [GBLL15]	●	●	○	○	○	○	○	○
Eclipse Mita	●	●	○	○	○	○	○	○
Foggy [YMLL17]	○	○	●	◐ ¹	●	○	○	○
GeneSIS [FNS ⁺ 20]	○ ⁶	○ ⁶	●	◐ ²	◐ ⁷	○	○	○
Google Cloud Platform	○	○	○	○	○	○	○	○
JFrog Connect	○	○	●	○	●	○	○	○
MARTE + CONTREP [MVH17]	●	●	○	○	○	○	○	○
MDE4IoT [CS16]	● ⁸	●	●	◐ ²	◐ ⁸	○	○	○
OpenTOSCA [BEK ⁺ 16]	●	●	●	○	●	○	○	○
ThingML [HFMH16, MHF17]	●	●	○	◐ ¹⁰	○	○	○	○
MontiThings (this thesis)	●	●	●	●	●	●	●	●

¹ Only deployment is adapted. Integrating dynamically instantiated components into the system is up to the developers.

² Both Azure and AWS offer digital twin services. However, they require the developers to manually react to updates or provide updates, *e.g.*, by subscribing to a set of MQTT topics used by the cloud provider.

³ “An accessor can be dynamically downloaded and instantiated” [BJK⁺18], but CapeCode does not include an automatic deployment algorithm.

⁴ Because accessors can take accessors as input and instantiate them.

⁵ Calvin can generate JSON from CalvinScript. The actors are implemented using handwritten Python code.

⁶ The GeneSIS modeling language defines deployment/orchestration. It does not define application logic.

⁷ GeneSIS can ensure devices have certain capabilities (regarding security and hardware). It does, however, not have a reasoner for defining requirements similar to our *local requirements*.

⁸ Does not use a custom IoT-focused language but UML profiles and action language for foundational UML (ALF) [CCS15, CS16].

⁹ Uses MARTE to specify the requirements of the devices. It does, however, not have the means for specifying requirements about the system as a whole like our *local requirements*.

¹⁰ ThingML has “dynamic sessions” [MHF17] for handling joining or leaving nodes. It does, however, not actively instantiate nodes to use available devices. ThingML can be used dynamically using GeneSIS: “ThingML code can be dynamically migrated from one device and platform” [FNS⁺19].

authenticating the IoT device. Additionally, they usually offer means to encrypt data. Moreover, some of them provide support for a wide variety of hardware platforms (**TC1**). For example, Amazon’s AWS-focused implementation of FreeRTOS offers support for a large number of low-powered devices, and balena offers images for its balenaOS for a large number of higher-powered devices. While MontiThings offers the necessary scripts for creating multi-arch container images, the support for different platforms of the commercial tools is arguably better than MontiThings’. As the main purpose of the reference implementation of MontiThings is to act as a tool for research, we have to make some assumptions about the IoT devices that are used as deployment targets (*cf.* Sec. 3.1).

Concerning the deployment itself, MontiThings offers a more expressive means for defining deployment rules. Cloud providers usually either provide no rule system at all, *e.g.*, balena, or only a grouping mechanism that enables deploying a particular software to a certain set of devices. These groups can also be dynamic⁷, *i.e.*, define membership of the group based on attributes of the device. This is similar to the matching of requirements and capabilities done by MontiThings. However, MontiThings deployment rules are far more expressive and enable, *e.g.*, expressing that certain pieces of software are incompatible with each other or that a piece of software requires a different piece of software to be executed by a device in the same room. MontiThings DevOps that uses a repository, CI pipeline, and container registry is aligned with those of the major vendors, like GitHub or GitLab. This is similar to balenaCloud which uses the same pipeline for preparing the deployment of software to IoT devices. Compared to major cloud providers and MontiThings, balena’s deployment strategy is, however, relatively limited. Usually, balena expects all IoT devices to execute the same (set of) containers. This can somewhat be circumvented by creating multiple repositories where each contains the software for one group of devices. It is, however, considerably less expressive than MontiThings.

A special case of cloud providers is the Arduino IoT cloud. Their offer is custom-tailored to devices supporting the IoT platform and having internet access. For these devices, *e.g.*, the Arduino Nano 33 IoT, the Arduino IoT Cloud offers an online editor to write Arduino *sketches*, *i.e.*, the small script-like C++ programs executed by Arduino devices. As part of this, developers can add simple GUI elements, such as sliders, to primitives. After deploying the sketch to the Arduino, these GUI elements can show the current value of the variable and can also be used to set the value. However, unlike MontiThings, the Arduino IoT cloud provides no mechanisms to prevent the Arduino from immediately overwriting user-set values. Hence, the possibilities to actually influence the system are limited if the sketch was not developed with digital twins in mind.

Besides the commercial approaches, numerous academic DSLs for IoT applications have been developed over the past decade. The most widely known are probably ThingML and Calvin. ThingML itself does not offer a deployment mechanism. Calvin

⁷AWS Dynamic Thing Groups Documentation. [Online]. Available: <https://docs.aws.amazon.com/iot/latest/developerguide/dynamic-thing-groups.html> Last accessed: 26.12.2021

offers a requirement-capability-based deployment algorithm like MontiThings. Unlike MontiThings, Calvin uses a distributed deployment approach where no device knows the whole network of devices. As no device has global knowledge about the state of the deployment, their deployment is naturally limited to rules that each device can evaluate for itself. For example, devices can evaluate a rule that states that a certain component shall be deployed to devices in a certain room by comparing their own location against this room. Rules like deploying two components in the same room or deploying a certain number of component instances, however, cannot be expressed as no device has the necessary knowledge to evaluate such rules. As MontiThings uses a centralized deployment, its deployment rules are more expressive. Moreover, MontiThings offers modification proposals that are not provided by Calvin. The price for this centralized deployment is that developers need to provide some infrastructure such as a server to run the deployment web application. Using infrastructure as code scripts, like those provided by Terraform, comes down to developers needing access to a cloud provider. Since all major cloud providers provide high availability, especially compared to IoT devices, we do not consider using a centralized infrastructure a major drawback.

GeneSIS is another academic deployment approach. GeneSIS provides a modeling language for specifying the deployment. Compared to MontiThings they focus more on low-powered devices, *e.g.*, devices that are not able to execute containers. They do, however, not provide a reasoner to calculate which devices shall execute what software. Instead, their algorithm focuses on reaching a state specified by the deployment models. Since GeneSIS does not provide a reasoner, we considered MontiThings' deployment algorithm a natural addition to GeneSIS. As described in Sec. 6.3.2, we thus provided a target provider implementation for GeneSIS that deploys MontiThings components using GeneSIS.

Overall, MontiThings provides a deployment algorithm that is tailored to the highly dynamic nature of networks of IoT devices. Using its requirement-based deployment, it is possible to also cover different infrastructures of different instantiations of the same IoT application. This makes MontiThings' deployment algorithm also especially useful for future app stores (*cf.* Sec. 3.1) of IoT applications, where different users likely have very different infrastructures. It is capable of adapting to changing infrastructures, *i.e.*, failing and new devices (**RQ5**), and, thus, also of moving to different infrastructures (*cf.* Sec. 8.1). In some IoT cases, however, this algorithm is not the best choice. If the infrastructure is very static and usually set up by experts that do a precise analysis of which device shall execute which software, *i.e.*, in Industry 4.0 scenarios, developers likely will prefer a manual deployment strategy.

By utilizing feature diagrams, MontiThings can raise the level of abstraction to avoid the need for device owners to know the details of the software architecture. Furthermore, by treating the architecture as a 150 % model, it becomes possible to provide software alternatives for unavailable hardware. For example, if smart-home-owners want to track the contents of their refrigerator but do not own a smart fridge, one alternative would

be to provide an app that enables them to manually track the contents of their fridge. A drawback of the feature diagram approach is, however, that the evaluation strategy to find, *e.g.*, the largest set of deployable features is inefficient because it may have to test a large number of feature configurations before finding a valid solution.

A further drawback of the Prolog-based deployment calculation is that the results of the deployment are sometimes not what a human would expect from the deployment. For example, if a requirement asks to deploy a particular component on *at least* five devices, Prolog will stop after deploying the software to *exactly* five devices unless other requirements require a higher number. Human operators might, however, expect their components to be deployed to as many devices as possible but only wish to be notified when the algorithm cannot find at least five devices to deploy the software to. To cover such cases, we introduced quantors (*EVERY* and *ANY*) to enable a more natural specification of rules.

Moreover, Prolog’s evaluation strategy would normally propose a large number of very similar modification proposals. For example, if the device owner rejects the proposal to relax rule *A*, Prolog might still propose to relax rules *A* and *B*. MontiThings can filter modification proposals that are supersets of already rejected modification proposals. However, this only hides them in the GUI but cannot change Prolog’s evaluation strategy. Thus, the calculation of the next proposal might take some time depending on the number of rules to be evaluated.

Currently, MontiThings does not include the workload of the IoT devices. In some cases, devices may have requirements to get guarantees to use a certain amount of resources exclusively. For example, users might not want fire detector devices to execute software that could increase their workload so high that they can no longer execute their main functionality of detecting fires. While it is possible to solve this to some degree using MontiThings’ incompatibility requirements⁸, other approaches like MARTE can handle such precise allocations better than MontiThings. Future work thus should address the question of how to combine approaches that can provide resource guarantees while still being able to handle the large and highly dynamic infrastructures of IoT systems.

Regarding the model-driven development of digital twins, MontiThings answers the questions of how to automatically connect IoT devices to accompanying information systems (**RQ6**) by synthesizing digital twins (**RQ7**). Compared to other commercial and academic approaches, our method separates the connection to the twin from the development of the IoT system’s business logic (**MC1**). As the information system and the IoT system are built independently of one another in our methodology, an explicit integration step is needed. This integration step requires expertise from both the information system developers and the IoT system developers. This makes this step non-trivial for complex systems. Compared to other solutions from, *e.g.*, cloud providers,

⁸To do so, device owners would specify that the fire detector is incompatible with every other component. Therefore, it would never share a device with other components.

it is, however, abstracted to a modeling level instead of having to consider the digital twin connection during the development of the business logic. By using a tagging model that is aware of both the data structure and the IoT application, hard-to-find configuration errors such as inconsistent MQTT topic names can be avoided.

Since the technical realization of our information systems is built on MontiGem which is a Java application, our solution naturally does not scale as well as the IoT data ingress solutions from cloud providers that can usually handle thousands of messages per second. Since our concept is, however, not bound to MontiGem, future realizations could transfer our concept to connect to the IoT solutions of cloud providers instead of MontiGem.

Chapter 7

Execution and Runtime Analysis of C&C-based IoT Applications

The previous chapters mainly focused on design-time aspects and the initial setup and integration of IoT applications. In contrast, this chapter focuses on the execution of IoT applications. As IoT devices often use unreliable hardware (**TC3**), IoT applications may be impacted by various error situations ranging from unreliable sensor outputs to failing devices. This chapter presents methods to analyze and handle such error situations.

7.1 Research Questions

Having to cope with unreliable hardware is a major challenge when developing IoT systems (**TC3**). Not taking failures into account when developing an IoT system may lead to unreliable systems and, thus, unexpected behavior. This chapter addresses this challenge on multiple levels: The most obvious (and severe) kind of failure is often the complete failure of a device. This problem is addressed by (**RQ5**) and (**R9**). Previous chapters already showed how to handle failing devices from a deployment perspective, *i.e.*, by detecting the failure via missing heartbeat messages and then rerunning deployment algorithms to find an alternative deployment. The device that replaces the failed device has however none of the data that the failed device had. This problem is addressed in Sec. 7.3.

Another class of errors is hardware components that do not behave as expected. Handling malfunctioning hardware is addressed by (**RQ4**). Sec. 4.2.5 already showed how incorporating the OCL in the language can offer mechanisms to specify, and thus detect, deviations from the expected behavior. Given the complexity of IoT systems with a multitude of sensors, actuators, unreliable network connections, and other components that may fail, it is, however, unlikely that developers foresee all possible error situations at design time. Thus, methods for analyzing errors within a running system are needed. This is addressed by (**RQ8**). Especially, we present two methods for effectively tracing the system's behavior (Sec. 7.4, (**R11**)) and reproducing the behavior in a controlled environment without the IoT devices (Sec. 7.5, (**R12**)).

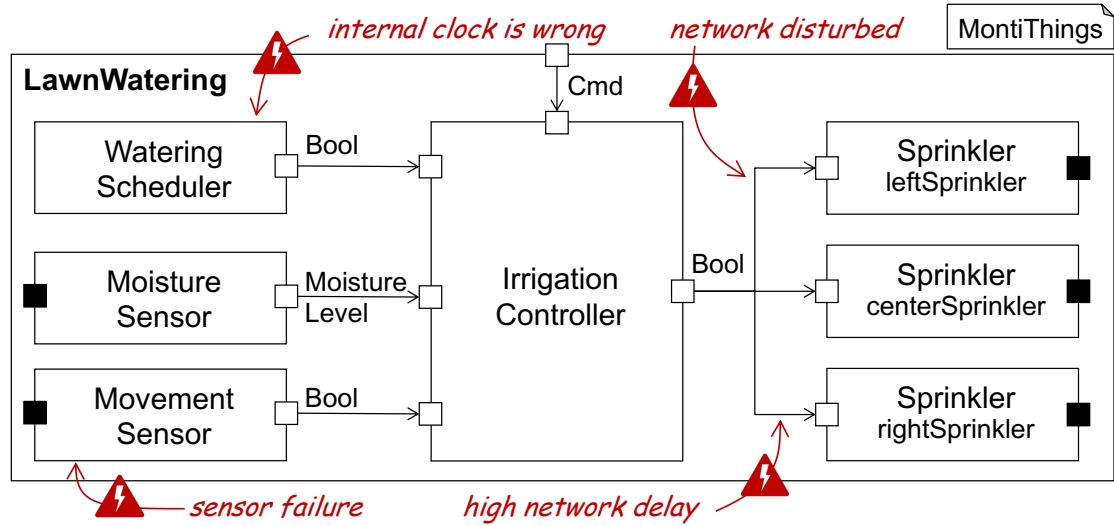


Figure 7.1: Motivating example for MontiThings’ error analysis. Figure taken from [KMR21].

7.2 Methodical Considerations

As shown in Fig. 5.1, the executed MontiThings application may rely on various services that are provided in a service layer separate from the actual application. Conceptually, these services can be considered part of the run-time environment. They are, however, executed on centralized infrastructure instead of IoT devices to make them independent of the failure of individual devices and utilize the higher network bandwidth, storage, and computing power of servers or cloud systems.

Each of the methods presented in this chapter constitutes such a service. The generator is aware of each of these services. Thus, for each service, the generator can be configured to either enable or disable the service. If enabled, the generator includes the service-specific templates in the generation and, in some cases, executes some additional preprocessing steps such as model-to-model transformations.

From the developers’ point of view, they develop their models like they would when not using any of the services presented in this chapter. Fig. 7.1 motivates that even models that contain no inherent errors can lead to unexpected behavior when deployed on IoT devices. In this example, the lawn watering application of a smart home behaves unexpectedly due to various errors that are out of control of the model, *e.g.*, failed sensors producing wrong values, disturbed network connections, or incorrectly set clocks. If not prevented by additional checks, the errors produced by such unreliable hardware can easily propagate to other parts of the system. When the error finally becomes apparent

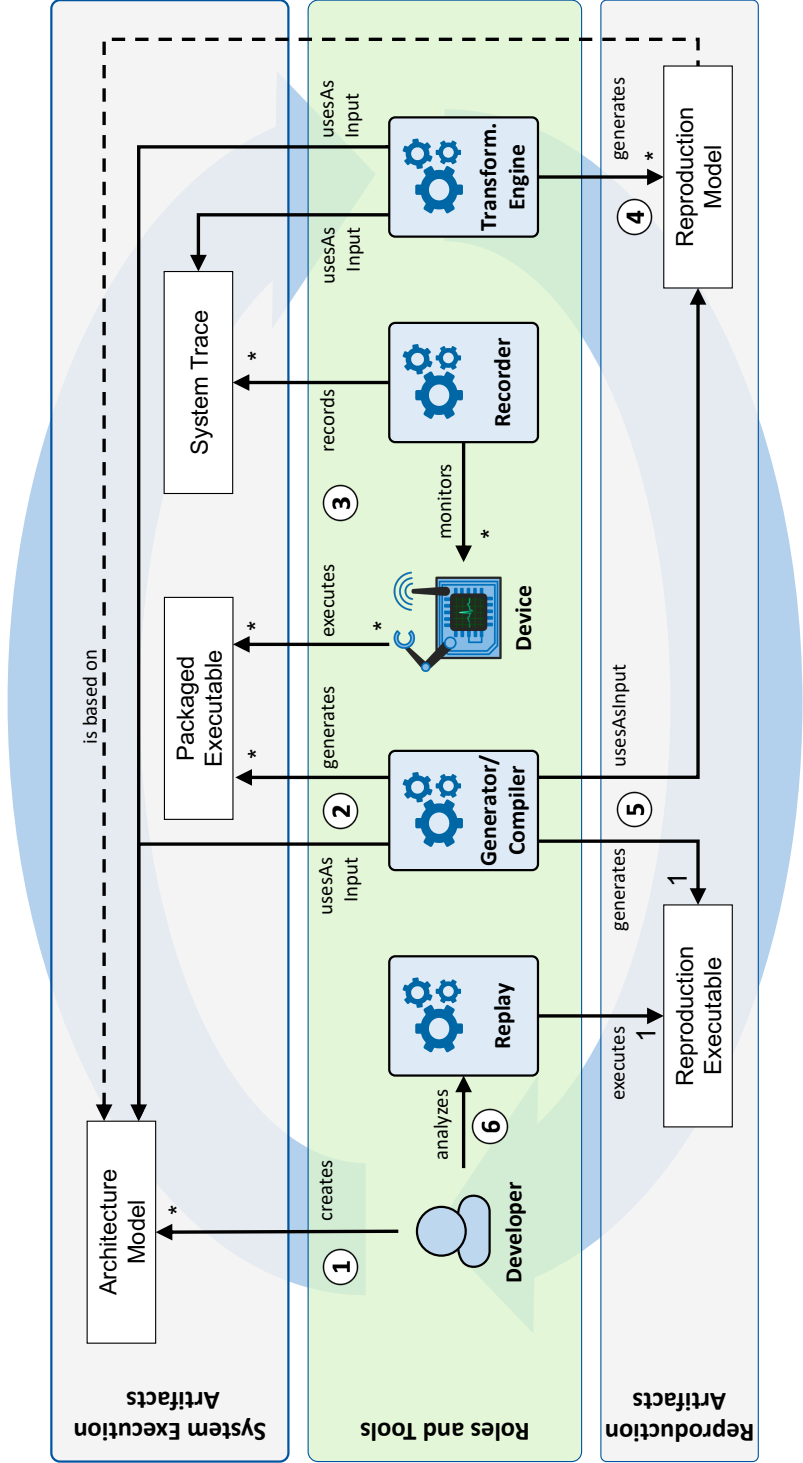


Figure 7.2: Overview of MontiThings' transformation-based replay. Figure taken from [KMR21].

to the end-user, the cause of the error may be hard to track down. Thus, developers need methods to inspect IoT systems.

Traditionally, developers need to inspect a large number of logs collected from each device, compare these logs to their models and code to then come to an answer as to why the system behaved unexpectedly. Related work on C&C IoT applications proposed automatically generating such logs by tagging the elements in the model that shall be logged and then centrally collecting the logs [MF19]. However, the sheer amount of logs generated by IoT systems can make this process very time-consuming. Moreover, since developers have to manually decide what to log at design time, the logs may also lack the necessary information for finding the cause of the error.

To mitigate this problem, our method for understanding the behavior of C&C applications filters the logs to the relevant information needed to trace a particular state of the application [KMM⁺22]. To remove irrelevant log messages, the tracing tool takes the architecture models into account, as the architecture contains the information to which other components a component may pass data. Secondly, we offer a method for recording the system's behavior and replaying it in a controlled environment [KMR21]. This gives developers the possibility to, *e.g.*, set breakpoints using the analysis tools they are used to from the development of non-distributed systems.

For the developers, activating most of these services works by setting a switch in the generator configuration. As described above, this will cause the generator to add the necessary code to the generated code that enables the IoT application to communicate with the services. The services themselves are standalone applications that can be started independently, *e.g.*, using Docker containers. For tracing the application while it's being executed, developers then only have to connect to the web application provided by the service. In the case of reproducing the behavior, two generation steps are necessary. Fig. 7.2 shows the process for reproducing the behavior using our method. It consists of the following steps:

- Step ①:** The developers write the MontiThings architecture models like they normally would when not intending to reproduce the behavior.
- Step ②:** Next the developers generate (C++) code from the models using MontiThings' code generator. Developers configure the generator to activate the recording module. The generator will inject a recording module into each port that connects the ports to a central recording module.
- Step ③:** The IoT devices execute the generated code. During this execution, a central recording module captures the initial state of the components and all following communication between the components. This includes not only the exchanged messages but also metadata such as the delay between sending and receiving a message. The recording module saves the recording data into *system traces*, *i.e.*, a JSON file containing the information about the data exchange.

- Step ④:** After recording the necessary data, the models are adapted using a set of model-to-model transformations (*cf.* Sec. 7.5). As a result, a *reproduction model* that can be used to reproduce the recorded system’s behavior is created. To do so, the recorded messages, sensor data, and the like are reproduced by components generated from the system traces and added using the model-to-model transformations.
- Step ⑤:** Using the code generator, developers can generate (C++) code from the reproduction model. This time, the generator is configured for reproduction, *i.e.*, it does not unnecessarily add the recording modules. The generated code is not dependent on any sensors, actuators, or networks. It is used to create a *reproduction executable*.
- Step ⑥:** The developers can execute and analyze the reproduction executable on their own computer. They can use standard analysis tools like gdb, Valgrind, or others to inspect the application’s behavior.

7.3 Fault Tolerance

It is important to differentiate between temporary and permanent failures when trying to mitigate failures (**R9**). Temporary failures are usually caused by IoT devices entering battery-saving mode or by a lost network connection. For example, if IoT devices are connected to the Internet using a cellular connection, they might lose this connection when entering a dead zone. MontiThings bridges such temporary failures by using a central and highly available message broker, *i.e.*, an MQTT broker. Such message brokers can cache messages while the intended receiver is not disconnected from the network. Thereby, communication partners do not need to know if their communication partners are available. The IoT devices only need to know that they can reach the central message broker. Once the temporarily disconnected IoT device comes back online, it can receive the messages sent to it while it was offline. This caching mechanism is also known as the *device shadow pattern* [RBF⁺16]. This strategy is also used by AWS as part of their *device shadow* service.

For brokerless communication, *e.g.*, using DDS, this strategy, however, cannot be applied. Therefore, MontiThings ports store the information which communication partner has already received which messages. This enables the IoT devices to perform local caching for offline communication partners. Once the communication partner comes back online and requests the next message from the port, the port does not return the newest message but the oldest message that the respective communication partner has not yet processed. By using this form of caching, the ports effectively implement a distributed variant of the above device shadow pattern.

In more severe cases of failure, IoT devices may be unable to come back online. This usually happens if the IoT device gets (partly) destroyed or its memory gets corrupted.

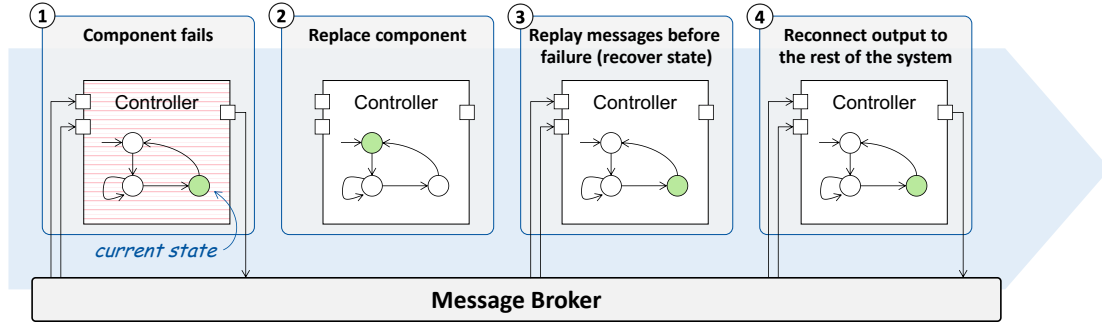


Figure 7.3: MontiThings' failure recovery process. If a component fails, the system replays messages received by the failed component to restore the state of the failed component. Figure taken from [KRSW22].

For example, if Raspberry Pis do not use a UPS, they are often not shut down correctly before losing their power supply. This may corrupt the Linux system on their SD card and prevent it from booting in the future. As described in Chapter 6, our deployment manager can automatically detect such permanent failures and trigger a redeployment. Without further steps, the component that replaces the failed component does, however, always start from a clean state. Any data that the failed component stored is lost. While this is not a problem for stateless components, losing the state of stateful components may impact the overall function of the IoT application.

To mitigate this problem, MontiThings offers a failure recovery service. This service listens to all messages that components exchange via their ports and stores them. Since the state of components only depends on the messages they exchange via their ports, the stored messages can be used to restore the state of a component. When stateful components are started, they will first request the failure recovery service to start a failure recovery. The component only starts from a clean state if the failure recovery service does not have data about the component. Fig. 7.3 shows the workflow of this failure recovery procedure.

- Step ①:** First, an already existing component instance fails (permanently). The reason for the failure is irrelevant for the failure recovery. Before failing, the component had a certain state (marked with green filling).
- Step ②:** After detecting the failure, the deployment manager replaces the failed component with a new component instance on a different device. Right after the instantiation, the new component is in its initial state, which might be different from the state of the failed component.
- Step ③:** The failure recovery service replays all messages that the failed component received before the failure. This restores the state of the failed component. To

not confuse other components with the messages the new component tries to send by calling its `compute` method during the failure recovery, the failure recovery happens before the outgoing ports of the component are connected. Thus, the component simply discards all outgoing messages during the failure recovery.

Step ④: After the state has been restored, the outgoing ports of the component get connected and the component starts processing *regular* messages that might have been sent by other components since the failure.

This procedure has two major drawbacks: The first drawback is that it requires the new component to process all messages the failed component received since it was started. Thus, this procedure has a complexity of $\mathcal{O}(n)$, where n represents the number of messages. Given the usually limited computing power of IoT devices, this makes the failure recovery very slow. Secondly, the `compute` function of the component might depend on non-deterministic factors such as drawing a random number. In these cases, replaying old messages might lead to a different state.

To solve these two problems, components also explicitly serialize their current state and send it to the failure recovery service. If components save their state every m messages (m being a constant number), the complexity of the replay is reduced to $\mathcal{O}(1)$, since only the (constant number of) messages the component received since last saving its state have to be replayed. Besides reducing the computational complexity, storing the state explicitly likewise also reduces the memory needed by the failure recovery service to $\mathcal{O}(1)$, since all messages prior to the latest saved state can be discarded. Furthermore, by saving the state explicitly, the component also saves all prior non-deterministic influences on the current state.

7.4 Tracing Behavior and Filtering Logs

Analyzing a large amount of very detailed jobs can be a very time-consuming task for developers who want to find out why their application behaved in a certain way. To mitigate this situation, MontiThings offers a web-based tracing service. This service connects to IoT devices and collects their logs. To help find the cause of each particular behavior in large logs, the service can filter the log messages based on its knowledge about the architecture.

When enabling the log tracing tool in the generator, the generator adds recording modules to each component that collect the logs of that component. Also, it generates the necessary code to enable the components to interact with the tracing service to process the developer's requests. Conceptually, the main task of the log tracing service is to present a filtered version of the log messages that shows only the relevant information for what state of the application the user is currently interested in.

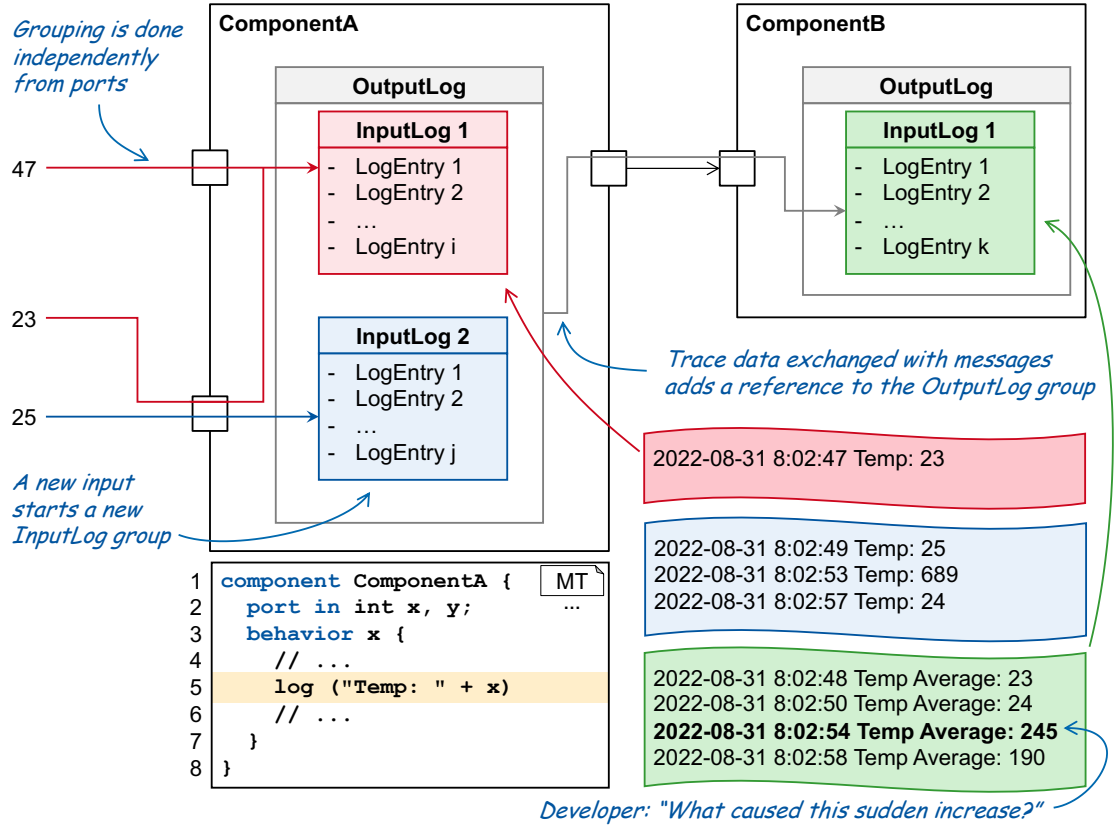


Figure 7.4: Concept of bundling logs for tracing. Figure taken from [KMM⁺22] and based on [Mal21].

To do so, the log tracing service groups log messages in a way that it can set log messages in relation to each other. This is based on two facts about connectors: If a component provides messages to another component, it may influence its log messages. If, however, two components cannot send messages to each other—not even indirectly via other components—they cannot influence each others' state.

Fig. 7.4 shows how the tracing tool creates two kinds of groups of log messages: InputLogs and OutputLogs. InputLogs group the log messages happening between processing two sets of incoming messages. In contrast, OutputLogs group the messages between two messages sent on outgoing ports. More precisely, OutputLogs do not refer to the actual log messages but to the InputLog groups that contain them. An InputLog is created each time the component processes a new message on one of its ports. If multiple messages are processed simultaneously, they belong to the same InputLog. Both InputLogs and OutputLogs contain lists of logs or groups of logs.

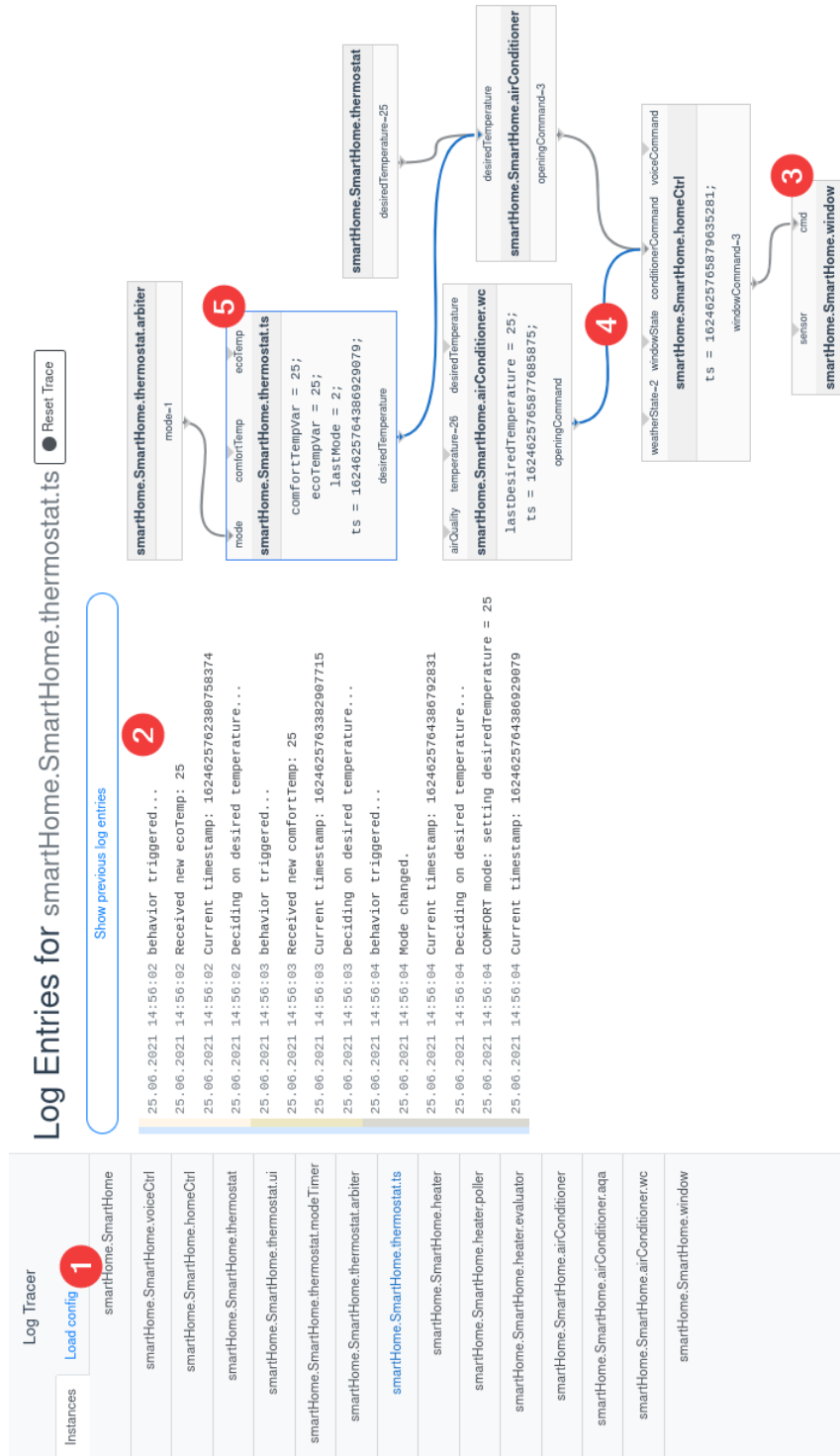


Figure 7.5: GUI of the web-based log tracing tool. Figure taken from [Mal21].

To further relate the log groups to each other, each log group has a unique identifier. When a component sends a message on one of its ports, it includes the identifier of the respective OutputLog. Thereby, the receiving component can add this identifier as a predecessor to the InputLog created for this log message. Note that the actual log messages are not exchanged. The logs are not merged again until later in the log tracing service. This is done to save network bandwidth by preventing the IoT devices from exchanging (and locally storing) logs that might never be requested.

As discussed earlier, the log messages created by the developers might not contain all the necessary information to inspect the state of a component. To solve this problem, the components do not only store their log messages but also their internal state and the content of incoming and outgoing messages. This enables developers to inspect even information that they did not actively log during the execution.

Fig. 7.5 shows the GUI of the log tracing service. The GUI consists of a sidebar that contains the fully-qualified names of all component instances (①). If the user clicks on one of the component instances, the log tracing service connects to the respective component instance and requests its logs. The logs are then presented in section ② of the GUI. The two-column colored bar on the left of the log message indicates which logs belong together. The left part of the bar indicates messages that belong to the same output message, the right part does the same for incoming messages. If the user clicks on one of the log messages, the tracing service generates the little component-like image in the right section of the GUI (③-⑤).

Conceptually, this image can be thought of as the equivalent of an object diagram when compared to a class diagram. This image shows the current state of the architecture at the time of the log message. Initially, this image only shows the currently selected component instance (③) and its direct predecessors, *i.e.*, the component instances it directly receives messages from. By clicking on the predecessor, the logging service will show the log messages that belonged to the corresponding OutputLog. In the image, it will also show the predecessor(s) of the clicked predecessor. If the clicked predecessor was a composed component, it will also reveal its subcomponents by clicking on the component (④). The currently selected component instance is shown with a blue border (⑤). In some cases, our logic for filtering logs might be considered insufficient by the developer. To solve this, we allow the developers to optionally also show log messages that the tracing service classified as unrelated.

The general idea of instrumenting models with monitoring or logging functionality has already been proposed by, *e.g.*, [HBJD20]. However, they do not focus on relating the logs and filtering them to make them more understandable.


```

1  template <typename A>
2  A nd(A value)
3  {
4      if (isRecording)
5      {
6          storage[index] = value;
7      }
8      else if (isReplaying)
9      {
10         value = storage[index].get<A>();
11     }
12
13     index++;
14     return value;
15 }

```

Usage example:

```

1  time_t timer = nd(time(NULL));

```

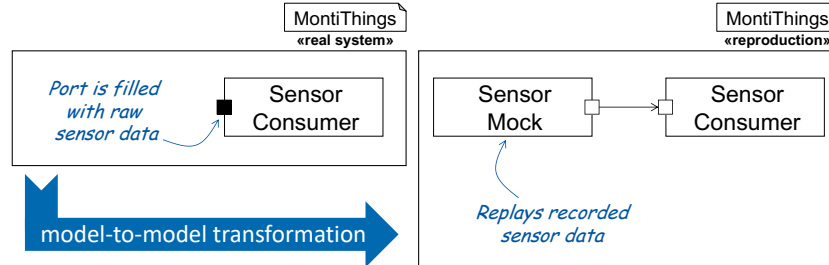
Figure 7.6: Handling non-determinism using the `nd` function. The results of non-deterministic function calls are stored while recording the system’s executions. If the system is replayed the actual results of the function call get replaced by the recorded results. Figure taken from [KMR21].

7.5 Transformation-based Record and Replay

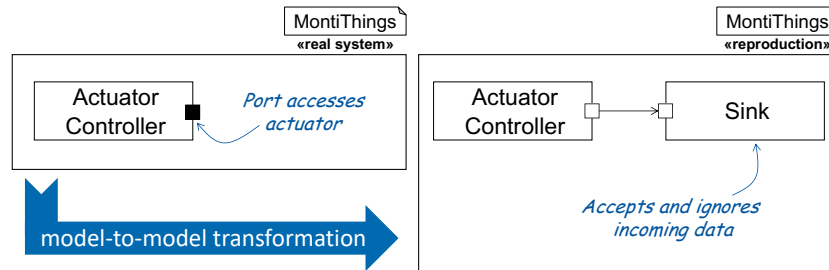
As explained above, IoT applications may fail even if the models do not contain any inherent errors. Such failures can be caused by hardware (**TC3**) or network issues (**TC2**), *e.g.*, a sensor that produces wrong values. If not detected and handled by assertions, such errors can lead to hard-to-trace errors in other parts of the application.

To understand such errors that are not directly visible in the model, MontiThings offers a record-and-replay approach. Recording and replaying system executions is a well-known approach for debugging distributed systems [LLPZ07, KSJ00, GASS06]. Using model-driven development, MontiThings can (partly) automate adding the necessary modules to the models. Thus, MontiThings keeps the business logic and the record-and-replay related model elements and code mostly independent (**MC1**). The only exception to this is explicitly marking non-deterministic function calls.

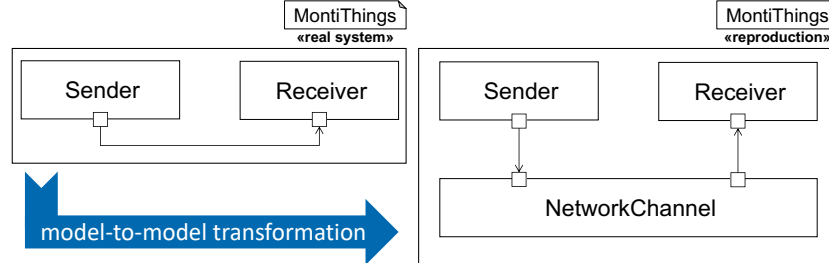
The general workflow of recording and reproducing the behavior of IoT applications has already been described in Sec. 7.2. If the code generator is set to recording mode, it adds a recording module into each port. These modules inform the central recording service about message exchanges, *i.e.*, sent or received messages. Additionally, the recording



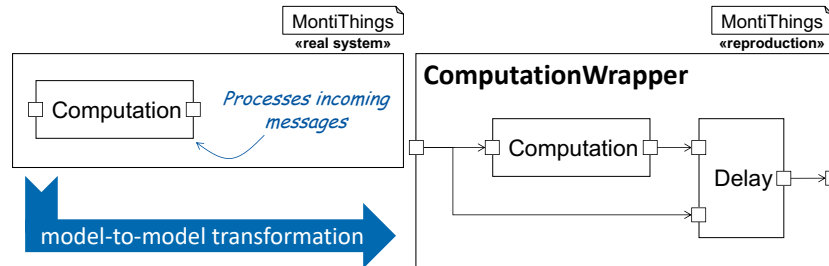
(a) Sensor data is provided via a mock component that replays the recorded sensor data.



(b) Requests to actuators are forwarded to a Sink that discards all messages.



(c) Connectors are replaced by a NetworkChannel component that replays network properties such as transmission delay derived from the recorded metadata.



(d) Atomic components are wrapped in a new component. This component contains a delay that reproduces the computation delay on the device where it was originally executed.

Figure 7.7: Model-to-model transformations for transforming the original models into the reproduction model. Figure taken from [KMR21].

service records metadata such as the time between sending and receiving a message. To be able to reproduce the behavior of the applications correctly, the recording service also needs to be aware of non-deterministic effects on the execution. If non-deterministic effects were not recorded, the reproduction might behave differently than the original system. For example, if the original system draws a random number during its execution and the actually drawn number is not recorded, the reproduction might draw a different random number.

We consider anything as non-deterministic that might lead to a different result if executed at a later time or by a different device. This includes, for example, reading a file from the file system, using random numbers, and accessing the current timestamp. To make the recorder aware of such non-deterministic function calls, developers have to mark them by wrapping them using the `nd` function (Fig. 7.6). The `nd` function is a generic function that is aware of the fact that the system is recorded or replayed. If the system is currently in recording mode, the `nd` function stores the results of the function call it wraps and then returns the result of the wrapped function call. If, in contrast, the system is in replay mode, the `nd` function accesses the previously recorded values and returns the recorded value instead of the actual result of the function call.

Another important non-deterministic influence on the IoT application is the interaction with sensors and actuators. In MontiThings, this interaction happens via ports. Unlike non-deterministic function calls in handwritten code, these ports provide a clear interface that is known on the model level. Thus, the recording and reproduction of the usage of these ports can be handled automatically without requiring manual markers, like the `nd` function, from the developers. For recording their values, ports that interact with the environment are recorded using the same recording module that is also injected by the generator in all ports that are used for the communication between two ports. When the IoT application shall be replayed, the models are first transformed using model-to-model transformations (step ④ of Fig. 7.2). Fig. 7.7 gives an overview of these transformations.

If a port is used to receive values from a sensor, or, more generally speaking, an external connector (*cf.* Sec. 4.2.6), the transformation adds a new mock component to the system that replays the values collected while recording the system (Fig. 7.7(a)). For this, the transformation uses the system traces recorded of the system that contain all sensor values. To do so, the transformation can utilize MontiThings' priority mechanism (*cf.* Sec. 4.2.6). By connecting a component to the port, that port automatically takes precedence over the values that might possibly be provided by a real sensor on the system that executes the reproduction. If, in contrast, the port is used to communicate with an actuator, the port is connected to a mock component that just discards all incoming messages (Fig. 7.7(b)). If an actuator has an influence on the environment that is relevant to the IoT application, this influence is captured by sensors. For example, if a radiator heats a room, the IoT application only becomes aware of this effect if it uses a temperature sensor to measure the temperature in that room. In case the actuator has

an effect on the devices that execute the IoT application, *e.g.*, by destroying one of the devices, this effect will be captured by recording the (in this case infinite) network delay. Thus, the actual effects of controlling an actuator can be ignored in the reproduction.

The transformations also utilize the metadata captured by the recorder, *i.e.*, the timestamps between receiving and sending a message. For reproducing transmission delays, message losses, and other effects of the network, the transformations replace connectors between two components with a network channel component that replays these effects (Fig. 7.7(c)). For this, it uses the delays between sending and receiving a message. The component added by the transformation forwards all messages it receives on its incoming port on its outgoing port after a delay. The delay is determined by the recorded system traces. To factor out the delay for the transmission between the recording module on the IoT devices and the central recorder, the recorded uses the round-trip time that it can measure from sending and receiving acknowledgments for the messages exchanged with the recording modules. Thereby, the clock of the recorder can also act as a single point of truth instead of having to rely on the IoT devices' clocks to be perfectly synchronized.

The time needed for executing the compute function of atomic components is calculated similarly to the network delay. To calculate the delay, the transformation uses the time between receiving a message on a port and sending a message on that same component. Using this delay information, the transformation wraps the original component in a new composed component that also includes a Delay component (Fig. 7.7(d)). This Delay component adds a delay if the computation was executed too faster on the developers' computer than on the IoT device. Note that this uses the assumption that the developer's computer does not execute the component slower than the original IoT device. Since the wrapper component has the same interface as the original atomic component, it can replace the original component in the models.

Since all components are independently executed in the reproduction, there can also be non-deterministic effects caused by the operating system's scheduler: If two messages arrive (almost) simultaneously at a component, the scheduler might choose to change the order of the computation. For an accurate reproduction, changing the order of messages may, however, introduce various problems leading to a different execution. To mitigate this problem, the transformations use what we call *determinism spacings (DSs)*. Essentially, DSs take events that are very close together and increase the temporal distance between them by introducing an artificial delay. This delay is not carried over to subsequent events but only affects the events that are close together. By introducing this delay, the transformation discourages the operating system's scheduler from swapping the order of the events. The exact value of the necessary DS depends on the specific computer and operating system. In our evaluations, we could see that even a DS of 5 ms can drastically improve the accuracy of the order of the replayed events. Of course, adding DSs makes the timing of the reproduction less accurate. Overall, DSs trade a slightly less temporal reproduction for a strongly more accurate reproduction of the order of events.

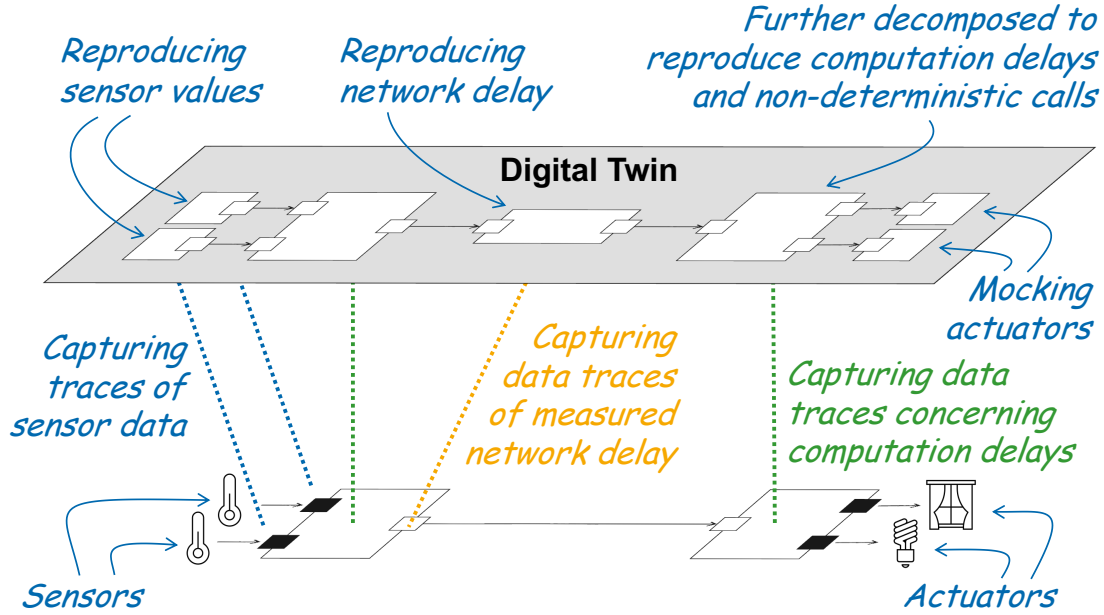


Figure 7.8: Relation between the IoT system and the digital twin used for the replay.
Figure taken from [KMR21].

Fig. 7.8 shows the relationship between the original system and the reproduction. Our reproduction consists of a set of MontiThings C&C models of the system, a set of data traces used for applying the transformations, and the services to reproduce the system’s behavior in order to analyze it. Thus, the reproduction of the original execution constitutes a digital twin.

7.6 Discussion

This chapter presented various ways for analyzing and handling errors in the execution of IoT applications. These approaches relied on recording data about the system during the execution and then using this data to provide services. Thereby, the approaches in this chapter are closely related to the development of digital twins that also rely on digital shadows, *i.e.*, data about the system, to provide additional services. In particular, we used the recorded data to restore the state of failed components, inspect currently running components, and reproduce the behavior of IoT systems.

By restoring the state of components, in combination with automatically redeploying them (*cf.* Chapter 6), we addressed research question **(RQ5)**. Our failure handling procedure can solve the problem of restoring the state in $\mathcal{O}(1)$ if the state is also regularly stored [KRSW22]. The advantage of replaying messages in addition to only restoring a

state is that also the state changes since the last backup can be restored. Technically, our service is, however, a research prototype that does not scale well. In [Häu20], we examined in a bachelor thesis how the architecture of the failure handling could be improved to be more scalable. One of the results was that the MQTT broker can be a bottleneck if it is not horizontally scalable. For commercial implementations, it could also be worth examining the combination of our failure handling with the automatic failover strategies of cloud providers. For example, by setting up an auto-scaling group¹, virtual machines can automatically be replaced if they become unresponsive. Thereby, a completely automatic failover strategy could be achieved for clients that do not require access to sensors, actuators, or other location-bound hardware:

1. An IoT client fails permanently
2. The health check of the cloud provider notices the failure and replaces the virtual machine
3. The deployment manager also notices the failure. It redeploys the software to the new virtual machine that replaces the failed machine
4. Our failure handling service restores the state of the failed machine

Our failover strategy does, however, rely on the assumption, that failures are not caused by the business logic, *e.g.*, hardware failures or a corrupted file system. If the fault is instead caused by the business logic, the failure handling service will restore the component that replaces the failed component into the same state, which is likely to result in a fault again shortly thereafter. To handle such problems, it would be possible to not restore the component's state if restoring it has failed a certain number of times. We did not implement such a policy as this could have an impact on the business logic, as the communication partners of the failed component may not expect it to lose its state without notice. Handling such situations within the component would also be possible but violate **(MC1)**.

This chapter also presented tools for analyzing the execution of IoT systems that address **(RQ8)** and (partly) also **(RQ4)**. By not only collecting the logs of IoT devices but filtering them to analyze the causes of particular log messages, we reduce the number of log messages a developer has to inspect and, thus, the time it takes to analyze logs. Storing the log messages on the IoT devices is, however, not an approach that is viable for long-running executions, because IoT devices usually only have a very limited amount of storage available. A more feasible approach for a commercial implementation would be to introduce some form of Linux's `logrotate`², where compressed old logs are regularly uploaded to an online blob storage instead of being stored locally.

¹AWS Auto Scaling groups documentation. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html> Last accessed: 30.12.2021

²`logrotate(8)` - Linux man page. [Online]. Available: <https://linux.die.net/man/8/logrotate>. Last accessed: 30.12.2021

Likewise, the record and replay approach is only feasible for recording a limited amount of time. Even though the system traces are not stored on the IoT devices for extended periods of time, the developers still have to analyze the reproduction. Debugging the reproduction inherently gets more time-consuming as the duration of the reproduction is increased. To mitigate this situation, the recoding service can also be dynamically attached and detached from the system. As the recoding service captures the initial state of the components upon being attached to the system, it can also reproduce such partly executions. Such a strategy, however, only works if the developers already know when to expect a failure. If, for example, the sprinklers of a lawn irrigation system always wrongly turn on every Monday at 8 a.m., it might be possible to attach a recorder during this time. If errors happen at random times, it is necessary to record the system for extended periods of time. Dynamically attaching the recorder does, however, of course not provide any explanations as to how the IoT system got into the state it was in when the recording started.

An advantage of our record-and-replay approach is that it includes all components of the original system. This enables developers to ask *what if?*-questions of the system within certain boundaries [KMR21]. In other words, they are enabled to change details of their implementation and then execute the system under the same conditions as the conditions under which the original system failed. This is, of course, only possible within the limits of the recoded data. For example, changing the interface of a component to use an additional sensor is not possible as there is no recorded data for the additional sensor. Moreover, the more the components in the reproduction deviate from the original components, the less valid the reproduction becomes. For example, if a change in the model leads to a radiator being turned on correctly, this will not be reflected by the (recorded) sensor data of a temperature sensor close to the radiator. If developers are aware of these limitations, being able to ask such *what if?*-questions can still facilitate analyzing the system.

The *probe effect* [Gai86] threatens the validity of the reproduction [KMR21]. In other words, by observing the system, the recording modules might influence the system's behavior. For example, by causing additional load on the network, delays might be increased when using the recording module. This effect can make the reproduced system behave differently than the original system.

Furthermore, the reproduction can also be distorted if the assumption that the developer's computer executes the components faster than the original IoT device is not met [KMR21]. In this case, the additional delay might cause other parts of the system to behave differently from their original behavior, *e.g.*, if a component measures the time between receiving two messages. To measure accurate delay times, the record-and-replay tool is only implemented using DDS and not using MQTT. The reason for this is that DDS uses peer-to-peer-based communication whereas MQTT uses a central broker. Any round-trip-time measurements would, thus, be influenced by the delay induced by the message broker and the network delay between the components, recording service,

and message broker. Furthermore, as visible from two theses [Häu20, Mal21], we advise turning off Nagle’s algorithm for network measurements. Since Nagle’s algorithm may delay the transmission of small packets, it may also delay sending acknowledgment messages between the recording modules and the recording service. This distorts the delay measurements used for reproducing the network delay.

As mentioned above, MontiThings is also not the first tool to use a record-and-replay approach. Instead, this approach has been chosen specifically because related work suggested it [LLPZ07, KSJ00, GASS06]. One of the most notable implementations of the record-and-replay approach is ROSbags³. ROSbags provides record-and-replay functionality for the robot operating system (ROS) [Kou16, QCG⁺09]. Their approach mainly differs from our approach by not being model-driven. Technically, their implementation is also based on DDS but does not include the network delay in the calculation [KMR21]. While this is usually not a problem for robots that are operated in a shared network, this can be a problem for IoT systems that often have to work with unreliable (sometimes cellular) network connections that can lead to variable network delays [KMR21].

Other IoT modeling languages, such as ThingML [HFMH16, MHF17] or Calvin [AP17, PA15, PA17], usually do not provide comparable failover and analysis tools. ThingML provides a tagging language for selecting model elements for which logs shall be produced and collected [MF19]. In contrast to MontiThings’ tracing tool, ThingML does, however, not include a way of effectively presenting these logs to developers. As ThingML’s logging is also based on a C&C modeling language, we think it could be extended by our approach. More generally, since our analysis tools only rely on the most common modeling elements of C&C languages, *i.e.*, ports and components, and many IoT languages are C&C languages, we think our approach is well transferrable to many other tools. The technical realization of course needs to be adapted to their individual programming languages and communication protocols.

It is worth mentioning, that there are still open questions with regard to the analysis of IoT systems at runtime. For example, digital twins of the IoT system could also be utilized to execute simulations to make predictions about the future of the system. One special case of this is predictive maintenance. If, for example, the frequency at which sensors produce unreliable values increases, this might indicate that the sensor should be replaced. For this, more advanced anomaly detection algorithms than what MontiThings offers might be needed [MNZC20], since MontiThings mechanisms are mostly focused on single components. By examining the messages of all components in the system, more advanced algorithms might be able to predict failures that cannot be detected with the limited black-box point of view of a component. Furthermore, while MontiThings is able to handle failures, it does not offer means to analyze the effects of a certain component failing. MontiThings can, as explained in this thesis, handle the failure of components

³Project website. [Online]. Available: <https://github.com/ros2/rosbag2>. Last accessed: 30.12.2021

and could, of course, also dry run a deployment on a simulated infrastructure. It does, however, not analyze the effects the failure might have on the business logic. If the effects are severe and the system is safety-critical, *e.g.*, the traffic lights of a smart city, developers might need more advanced failure handling mechanisms that offer some kind of redundancy for a more seamless failover.

Part III

Evaluation and Conclusion

Chapter 8

Experiments

In this chapter, MontiThings is evaluated in case studies and performance evaluations. Additionally, we describe our experiences from using MontiThings in student labs with student developers that (mostly) did not use MontiThings before the lab. The case studies are built around the running example of a smart home application and show various aspects of MontiThings discussed throughout this thesis. In contrast, the student labs also show MontiThings being applied to other use cases, *i.e.*, industrial IoT and autonomous driving. Furthermore, by inspecting the artifacts created by the students, we gain information on how inexperienced developers use MontiThings.

8.1 Case Study 1: Smart Home and Smart Hotel

This section demonstrates MontiThings deployment capabilities using a smart home and smart hotel case study. The smart hotel use case is inspired by the running example in [Zam17]. After modeling a smart home application and deploying it to a smart home, the residents of the smart home decide to visit a hotel. As hotel guests, they also act as (temporary) device owners and thus may deploy their smart home application to the hotel's infrastructure.

Fig. 8.1 shows the architecture of the smart home application. Overall, the application consists of a TV, an heating, ventilation, and air conditioning (HVAC), a smart assistant (such as Amazon Alexa or Apple Siri), an radio-frequency identification (RFID)-based door lock, and smart light bulbs. As implementing a large number of hardware drivers is not the focus of this case study, the ports that access external connectors only exist configuration-wise, *i.e.*, the project does not include code to control, for example, an actual TV. Instead, the components contain code that mimics the behavior of these hardware components, *e.g.*, by generating temperature values instead of reading out an actual sensor. A case study with real sensors and actuators can be found in Sec. 8.2.

According to our development process (*cf.* Fig. 5.1, Fig. 6.4), the development starts with the IoT developer creating the models and accompanying handwritten code. These artifacts are uploaded to a central repository. For this case study, we chose GitLab as our central repository, since GitLab offers a Git repository, a CI pipeline, and a container

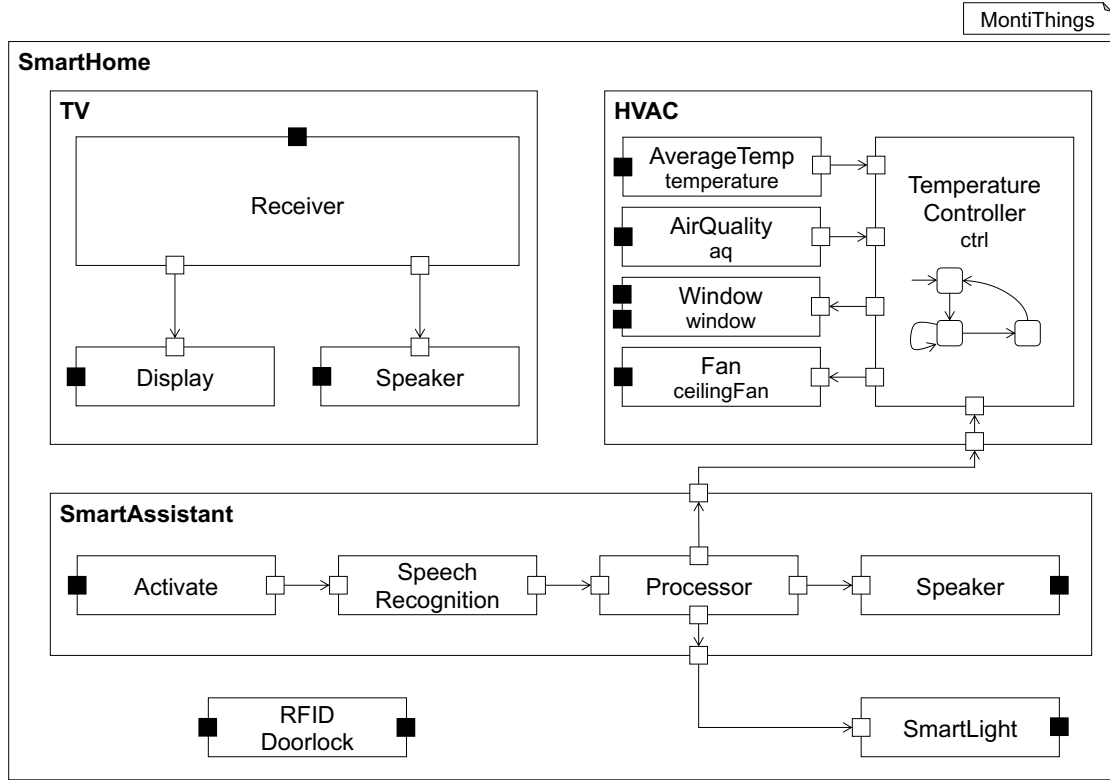


Figure 8.1: MontiThings architecture of the Smart Home / Hotel application. Figure taken from [KKR⁺22a].

registry together. It is, however, not required to choose GitLab. Many vendors, *e.g.*, GitHub, provide a similar bundle of services.

Pushing the artifacts to the GitLab repository triggers GitLab’s CI pipeline. This pipeline uses the models to generate code using MontiThings’ code generator. The generated code is then cross-compiled and packaged in a Docker image. For cross-compilation, we used Docker buildx as described in Sec. 5.3.4. More specifically, the CI pipeline is configured to cross-compile for linux/amd64, linux/arm64, linux/arm/v7, and linux/arm/v6. The resulting multi-arch container image is then pushed to the container registry of the GitLab project. Now that the images are available in the image registry, the IoT devices are able to download and execute the images.

We used a total of ten Raspberry Pi 4 Model B devices in this case study. Seven of these devices were deployed in Aachen, Germany. These devices represent the smart

8.1 CASE STUDY 1: SMART HOME AND SMART HOTEL

Device	Main Purpose	Capabilities	Location	
			Room	Building
Pi1	TV	speaker, display, tvReceiver	Living Room	Smart Home, Aachen, Germany
Pi2	HVAC	sensorAirQuality, actuatorWindow, actuatorFan	Living Room	
Pi3	Smart Assistant	microphone, speaker	Bedroom	
Pi4	Bedroom Light	light	Bedroom	
Pi5	Bathroom Light	light	Bathroom	
Pi6	Livingroom Controller	sensorRFID, light, actuatorLock, sensorTemperature	Living Room	
Pi7	Soundbar	speaker	Living Room	
Pi8	Hotelroom Controller	light, sensorRFID, actuatorLock	Bedroom	Smart Hotel, Stuttgart, Germany
Pi9	TV	speaker, display, microphone, tvReceiver	Bedroom	
Pi10	HVAC	sensorTemperature, sensorAirQuality, actuatorWindow, actuatorFan	Bathroom	

Table 8.1: Overview of the devices used in the case study. All devices are Raspberry Pi 4 Model B.

home. The other three devices were deployed in Stuttgart, Germany¹. They represent the smart hotel. All of the devices executed the IoT client software for docker-compose deployments. An overview of the configuration of all IoT devices is given in 8.1.

After logging into the deployment web application, the device owner registers the target provider for the devices in Aachen. The devices then show up on the device management page. Moreover, the device owner registers the application to be deployed by uploading the `deployment-info.json` file generated by the CI pipeline. This file tells the deployment web application which Docker images belong to which MontiThings component and which technical requirements each of them has. For example, the Speaker component can only be deployed to devices with the speaker capabil-

¹The devices in Stuttgart were provided by the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart.

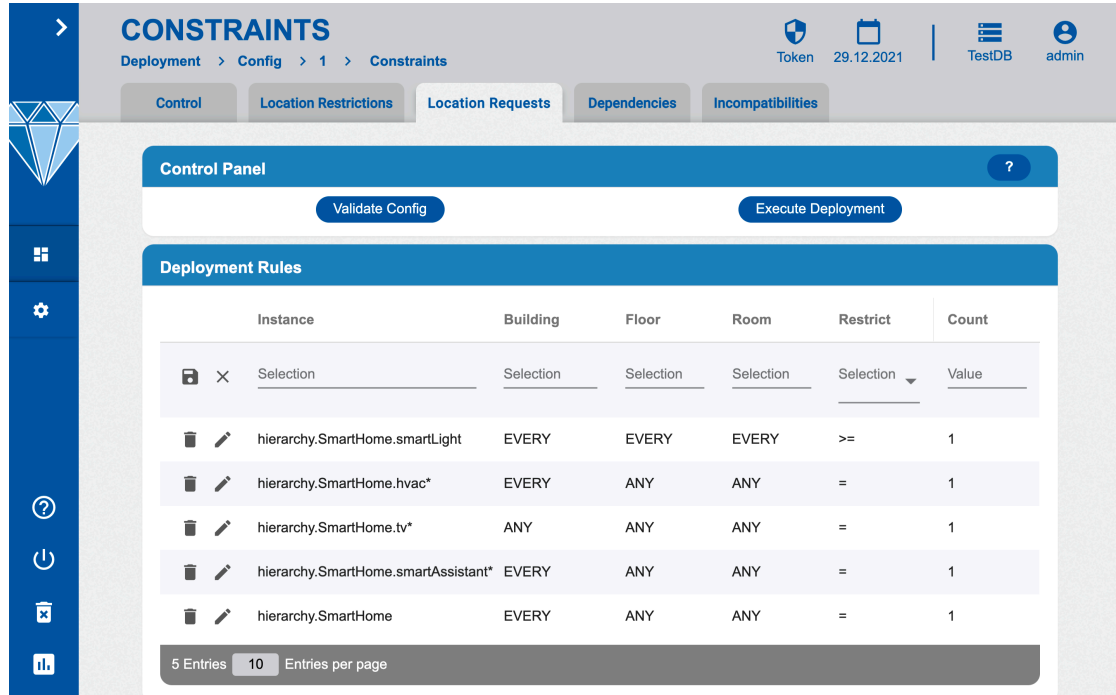


Figure 8.2: Screenshot of the deployment web application for entering location requirements. Labels in the screenshot were translated from the original German web application to English. Unnecessary website elements were removed to save space. Figure taken from [KKR⁺22a].

ity. Now the device owners can set local requirements that express their wishes for the deployment. The device owner chooses the following rules:

- Deploy at least one smart light to every building, floor, and room;
- Deploy exactly one HVAC (including all of its subcomponents) to every building in any room and floor;
- Deploy exactly one TV (including all of its subcomponents) to any building, floor, and room;
- Deploy a smart assistant (including all of its subcomponents) to every building in any room and floor;
- Deploy exactly one SmartHome component (the outermost component) per building.

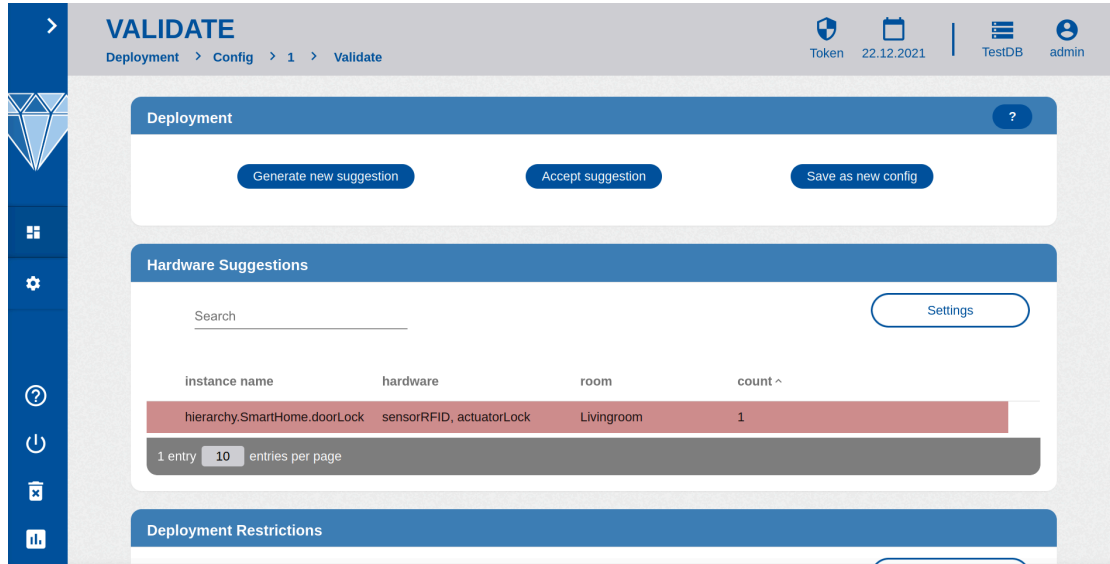


Figure 8.3: Screenshot of the deployment web application suggesting to buy a new device with a *sensorRFID* and *actuatorLock* capability and placing it in the living room. Labels in the screenshot were translated from the original German web application to English. Figure taken from [KKR⁺22a].

Fig. 8.2 shows a screenshot of the MontiGem web application the device owner used to input these rules. The five tabs except the *Control* tab represent the four types of local requirements from Sec. 6.3.1. Implicitly, the deployment web application also adds rules that make each composed component depend on its subcomponents, *i.e.*, the deployment manager is not allowed to deploy only some of the subcomponents of a subcomponent.

Next, the deployment can be validated. Given the devices in Aachen, the deployment would be satisfiable. To force the deployment manager to make modification proposals, we deactivate device Pi6 by stopping its client software. Pi6 is the only device in Aachen that provides the capabilities for deploying the RFID door lock. Thus, it is not possible to deploy the application without Pi6. The target provider notices the change and marks the device as offline. Since it is offline, it can no longer be used to deploy software.

The device owner now chooses to validate the deployment. This causes the deployment manager to generate Prolog code from the requirements. Additionally, it generates code from the information about the IoT clients as reported by the target provider. The generated code is then executed and its results are parsed by the deployment manager. The deployment web application informs the device owner that the validation failed but that there are modification proposals that could fix the problem. The device owner is then led to a page that shows the proposals (Fig. 8.3). As Pi6 is missing, the deployment manager suggests buying a new IoT device that has the *sensorRFID* and *actuatorLock*

capabilities and deploying it in the Livingroom. The device owner follows this advice by starting the IoT client software on Pi6. The deployment is now valid and can be deployed to the IoT devices.

Requesting the deployment web application to deploy the application causes the deployment manager to forward the calculated distribution, *i.e.*, the mapping which client shall execute which containers, to the target providers. The target provider generates a `docker-compose.yml` for each IoT client and forwards them to the IoT clients. The clients use the `docker-compose` file to pull the necessary containers and start executing them. This completes the deployment.

The algorithm chose to deploy the components as follows:

- Pi1** runs the `display` and `receiver` subcomponents of the TV;
- Pi2** executes the `aq`, `window`, and `ceilingFan` subcomponents of the HVAC;
- Pi3** runs the `speaker` and `activate` subcomponents of the smart assistant;
- Pi4, Pi5** each execute a smart light component;
- Pi6** the RFID door lock and the `temperature` subcomponent of the HVAC. Additionally, it executes all components without technical requirements;
- Pi7** runs the `speaker` subcomponent of the TV.

IoT devices are, alas, prone to failure (**TC3**). To simulate the failure of a device, we now stop the IoT client software on Pi7, the soundbar. The target provider detects that Pi7 is no longer sending heartbeat messages. Therefore, it marks Pi7 as offline and informs the deployment manager about the change. The deployment manager uses the Prolog generator to regenerate the Prolog code that contains the facts about the system. Executing the generated Prolog code leads to the decision to redeploy the TV's `speaker` subcomponent to Pi1. It informs the target provider that executes the change. This whole process happens without any user interaction.

Now, the smart home resident decides to visit a hotel. As outlined in [Zam17], the hotel guest could be considered the device owner in this scenario. During the journey, the smart home resident disables the smart home software in his home to save energy. This is done by stopping the deployment and removing the target provider.

The hotel is, however, not as well-equipped as the smart home. Accordingly, the deployment cannot be fulfilled because it is not possible to fulfill the rule that a smart light shall be deployed to every room since the hotel's bathroom does not contain a smart light bulb. First, the deployment manager suggests buying a new smart light and putting it into the bathroom. As the device owners' stay at the hotel is only temporary, the device owner does not want to buy hardware for the hotel. The device owner rejects the proposal. The deployment manager then calculates a different proposal. This time it

suggests relaxing the rule to put a smart light in every room. The device owner accepts the proposal and the following distribution gets deployed:

Pi8 executed the RFID door lock and a smart light;

Pi9 ran the TV's subcomponents and the smart assistant's speaker and activate subcomponent;

Pi10 executed the HVAC and its subcomponents and all other components without technical dependencies.

Even though the hotel room does not have dedicated hardware to execute a smart assistant, the hotel room's TV can take over this task because it has a microphone and a speaker. Deploying the software from his smart home enables the device owner to skip the check-in process in the hotel. As the door lock of the hotel room already runs the device owner's software, the RFID card from the smart home can also be used in the hotel.

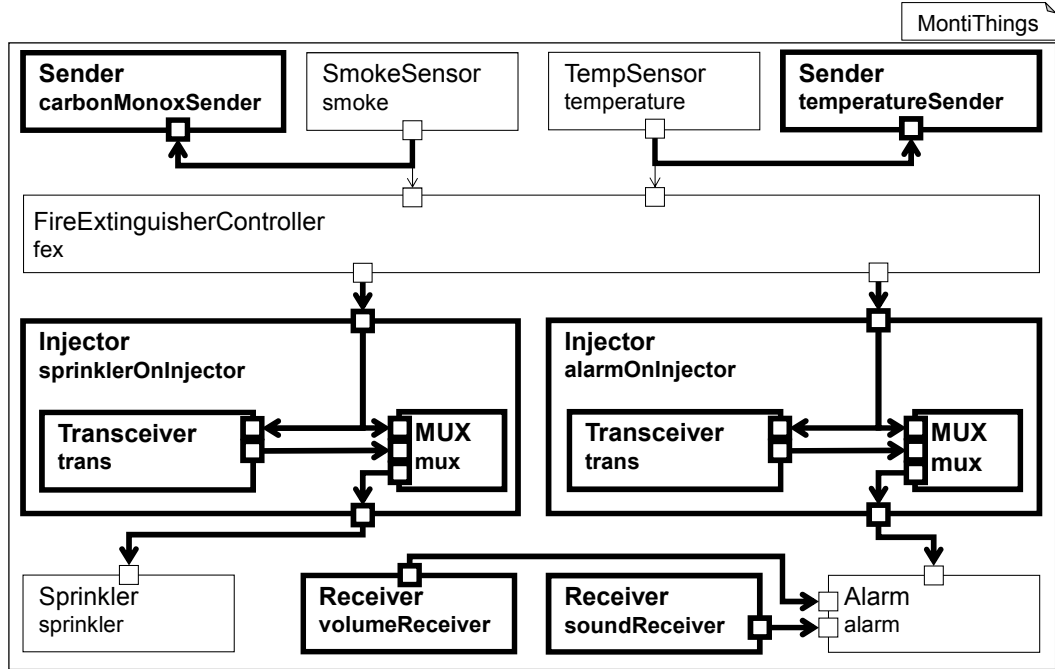
Overall, this case study shows how MontiThings' deployment system can be used to dynamically adapt an application to the available hardware. A drawback of the deployment is that by default the deployment algorithm will choose one of the devices and assign all components that have no technical requirements to it. This leads to unbalanced load distribution. To mitigate this, one should request components without technical requirements to be deployed to a virtual machine in the cloud or any similar system without practical hardware limitations. This can be achieved using e.g. whitelist rules that only allow those components to be deployed at the cloud location.

8.2 Case Study 2: Fire Alarm Digital Twin

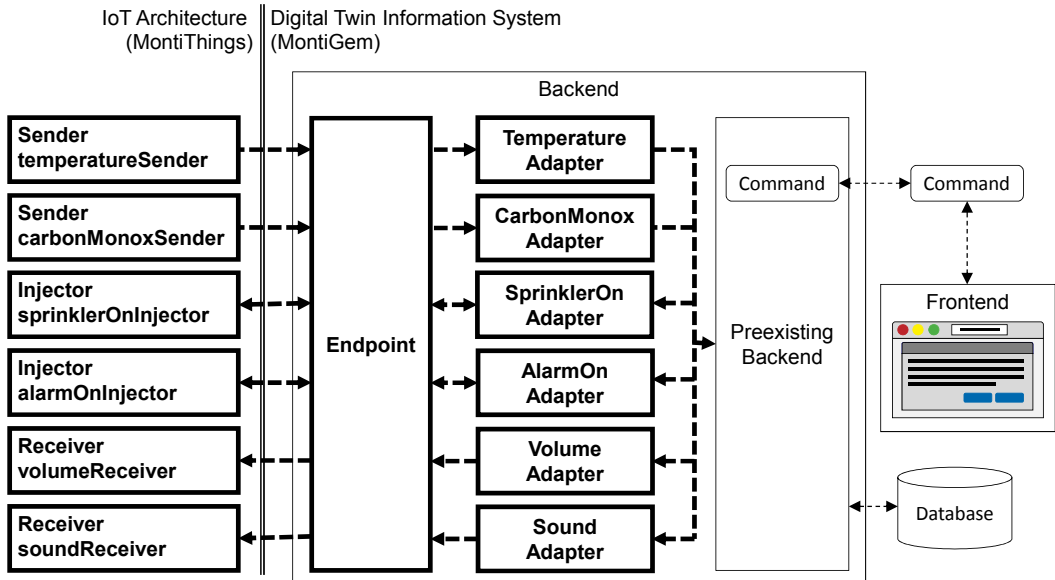
In this case study, we present a smart fire alarm and connect it to its digital twin in form of a MontiGem information system. Conceptually, the motivation for this use case is the Google Nest Protect². The Nest Protect is an IoT fire alarm that is connected to an information system that can, *e.g.*, be used to monitor and control the device. In addition to its main functionality, the Nest Protect also contains extra functionality like being able to serve as a light bulb at night. This case study, however, focuses on its main functionality as a fire alarm that is connected to a DTIS.

According to our process (*cf.* Fig. 6.2), the developers start by first developing an IoT system and a DTIS. As a result, they receive the models shown in Fig. 6.10, which the integrator connects using the tagging in Fig. 6.12. The architecture models are based on

²Google Nest Protect Product Website. [Online]. Available: https://store.google.com/us/product/nest_protect_2nd_gen Last accessed: 31.12.2021

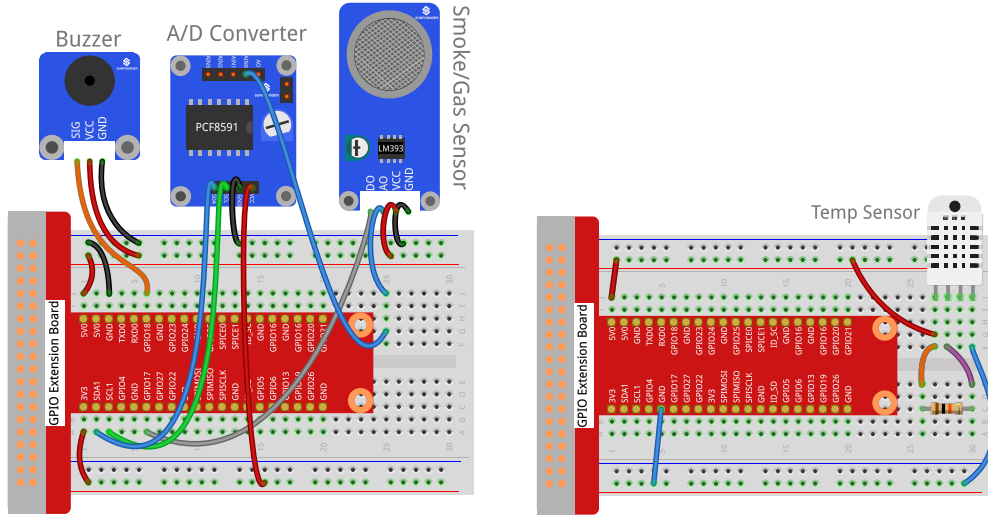


(a) Result of applying the model-to-model transformations to the MontiThings architecture.



(b) Adaptions of the MontiGem information system and its relation to the IoT system.

 Figure 8.4: Result of applying the model-to-model transformations to the IoT and information system. Elements created by model-to-model transformations are shown in bold. Figure adapted from [KMR⁺20b].



(a) A device connected to a buzzer, *i.e.*, an acoustic alarm, and a gas sensor. (b) A device connected to a temperature sensor.

Figure 8.5: Electronic setup of the Raspberry Pis. Each of the GPIO extension boards is connected to a Raspberry Pi 4 Model B. Figure taken from [Für20].

the architecture of the fire management in [MECL10]. Before executing the code generator, the developers execute the model-to-model transformations for adding the necessary components and classes for keeping the architecture and the DTIS synchronized. The resulting models are shown in Fig. 8.4.

All of the MontiThings components from the original components still exist in the transformed architecture. The Sender and Receiver components handle the communication with MontiGem. The two Injector components replaced the connectors between the FireExtinguisherController and the Sprinkler and Alarm components. Their transceiver components contain Sender and Receiver components that handle the communication with MontiGem. On the MontiGem side, adapter classes handle updating and accessing the database. The adapters communicate with the generated MontiThings components via the Endpoint that handles the network connection to the IoT devices.

Technically, we deployed the MontiThings components to three Raspberry Pi 4 Model B. Their technical setup is shown in Fig. 8.5. Two of the Raspberry Pis were each connected to a buzzer and a gas sensor (Fig. 8.5(a)). The remaining Raspberry Pi (Fig. 8.5(b)) was connected to a temperature sensor. We did not connect a Sprinkler device to the Raspberry Pis to prevent damage to our lab. Instead, the Sprinkler

components only log a message. As soon as the temperature sensor detects a high temperature or one of the gas sensors detects a high level of gas, the `fex` triggers the Alarm components.

We simulate a fire by holding a lighter close to the sensors. Accordingly, all buzzers are triggered. To trigger a test alarm, the user logs into the MontiGem web application. There they can choose to turn on the alarm of each device by sending it a message of the value `true`. Additionally, they also specify to override any values coming directly from the IoT devices. Thereby, they prevent the `fex` from directly turning off the alarm again because the sensors do not detect a fire.

Overall, the case study shows that our tagging can be used to connect and control multiple IoT devices and create digital twins for the architecture that are automatically synchronized with a web application. By specifying the connection to digital twin in its own (tagging) model, the business logic of the models is not polluted with communication- or synchronization-related code (**MC1**). Moving it to a separate model does, however, not remove the need for knowing both the MontiThings and MontiGem models at some point to create these connections. Thus, creating these connections remains a complex task. By removing the need for manually implementing such synchronizations, our method does, however, remove the chance of making programming mistakes here.

8.3 Case Study 3: HVAC Reproduction

In this section, we demonstrate MontiThings' capability of reproducing the behavior of IoT applications. For this, we implemented an HVAC application. Compared to the HVAC component of Fig. 8.1, the HVAC application in this section has a larger number of external ports and overall more complex logic. From an abstract viewpoint, this HVAC application consists of a radiator and a window that control the temperature of the room by turning the radiator on or off and opening or closing the window. The decision on how to control these actuators is influenced by a thermostat that has a display with buttons to enable the user to set a target temperature. The thermostat is modeled after a *Rock und Roll UT522* thermostat³. Additionally, the decision takes the air quality and weather forecast into account. Overall, this application is designed to make the decision on how to control the radiator and window hard-to-understand by making it depend on various different influences. For example, the window state could be changed for one of the following reasons:

- a low air quality indicates the window should be opened to let in fresh air,
- if the weather forecast predicts rainy or stormy conditions, the window is closed to prevent it from raining in,

³Rock und Roll UT522 manual. [Online, German]. Available: https://rockundroll.de/media/pdf_dateien/UT522vorabversion_Horst_Rock_GmbH.pdf Last accessed: 10.01.2022

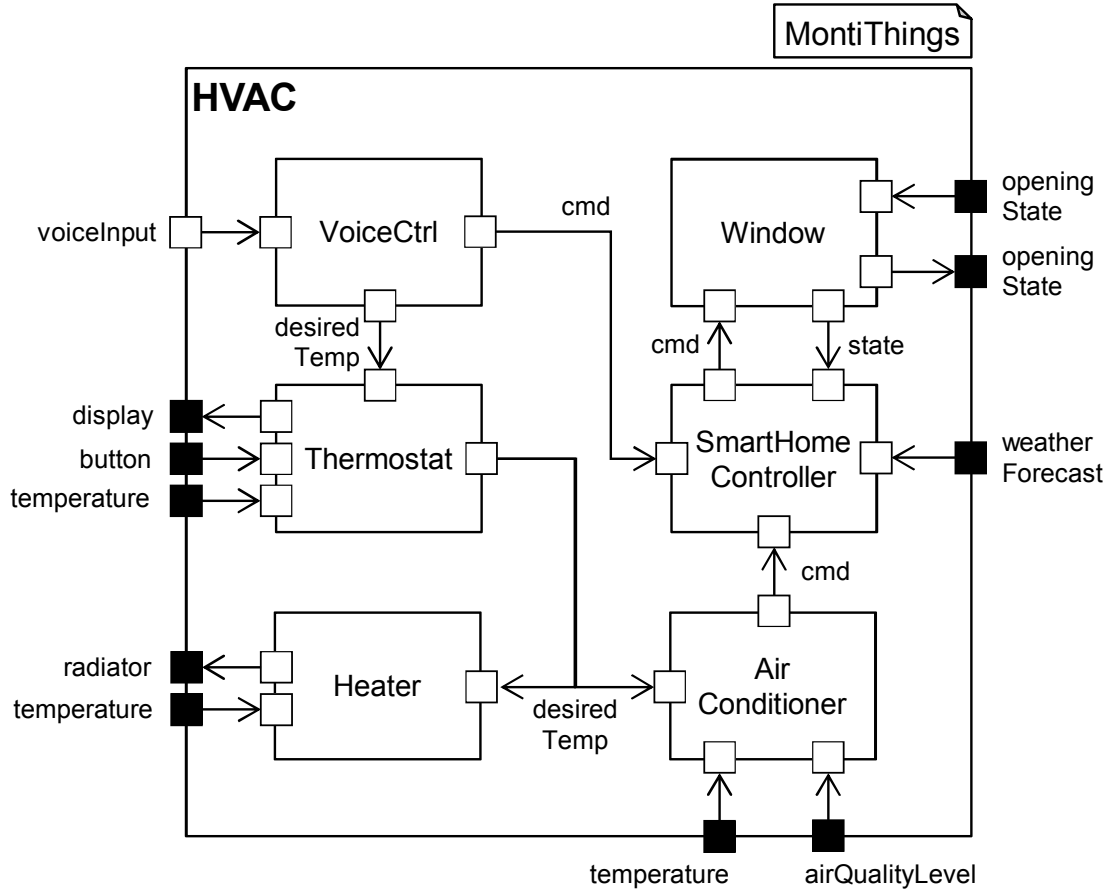


Figure 8.6: Outermost component of the HVAC application. Figure taken from [KMR21].

- if the temperature is too high, the window can be opened to lower the temperature,
- the user issues a voice command with a smart assistant to open or close the window,
- a timer automatically closes the window at certain times of day.

The outermost component of the architecture of this HVAC application is shown in Fig. 8.6. Overall, the architecture consisted of 15 component types. A more detailed overview of the models can be found in Appendix E.

The goal of this case study is to demonstrate that MontiThings can produce accurate reproductions. This requires us to have a *ground truth* of how the system is supposed to behave in the reproduction that we can compare the reproduction against. We cannot use the recorded data for this because the recorded data can already be flawed by mistakes in

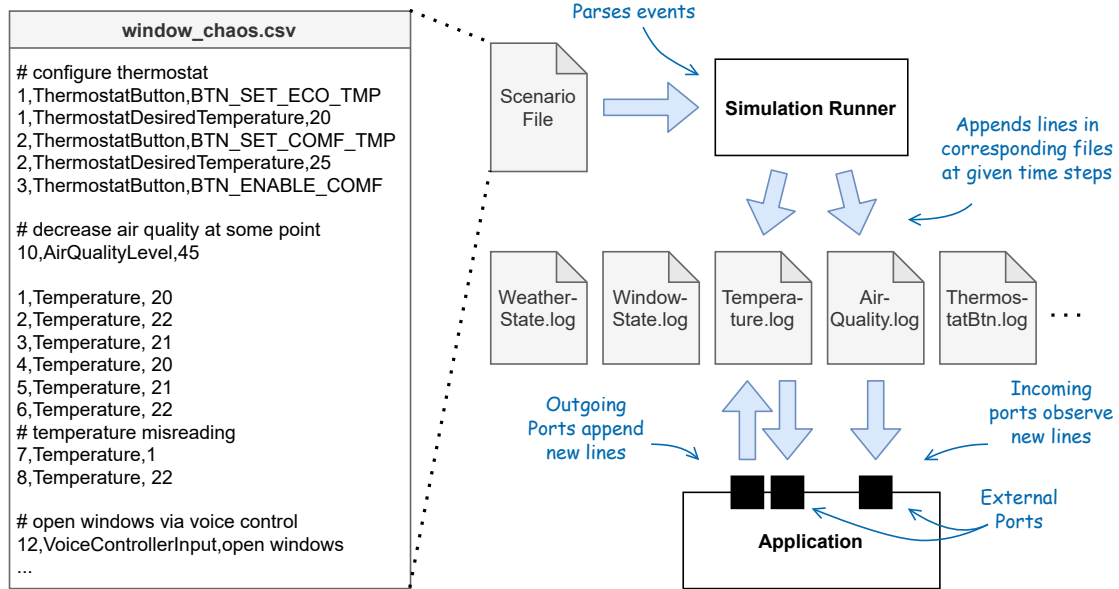
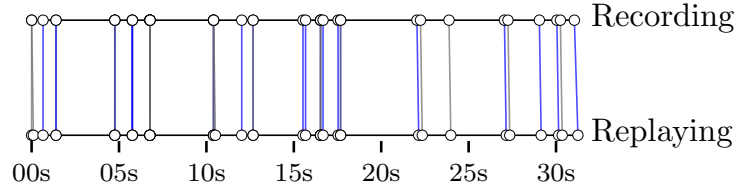


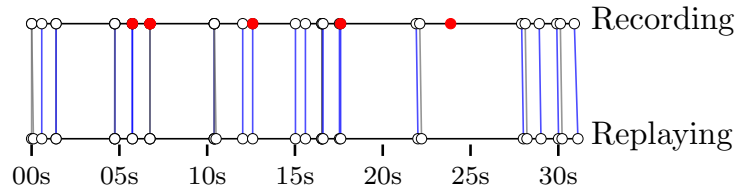
Figure 8.7: Concept of running MontiThings in a simulation. A scenario describes how the sensors are supposed to behave. A simulation runner forwards this information to the application via shared files. Figure taken from [Mal21].

the recording. Therefore, we decided to connect the generated MontiThings application to a simulator instead of real hardware. The conceptual overview of this simulation is shown in Fig. 8.7. An additional advantage of using a predefined scenario is that the scenario can be designed to make the software experience hard-to-reproduce situations by using, *e.g.*, simultaneous events or erroneous sensor values. Moreover, having a specified scenario makes the experiment reproducible.

The simulator works by creating a scenario file (`window_chaos.csv`) that specifies which values each sensor is expected to produce at which point in (simulation) time. A simulation runner, an application-independent from MontiThings, parses this scenario specification and writes the specified values to shared files at the specified time in the simulation. The ports of the MontiThings application read these shared files and forward their contents to the rest of the architecture. To control the network delay, we executed all Docker containers of the application on a single machine. During the execution of the application, we collect logs of the events. We use these logs to compare the original execution against the reproduction. Using these logs, we can validate if the events of the original execution also happened in the reproduction in the same order. Additionally, we define a criterion for when we consider a reproduction to (not) be perfect:



(a) An equal replay. The lines between the recording and replay are almost perpendicular.



(b) A non-equal replay. Multiple events of the original execution are not present in the replay.

Figure 8.8: Visualization of the accuracy of the replay. Each event is represented by a circle on the timeline. The events of the original execution are connected to their replayed version in the reproduction. The more accurate the temporal reproduction, the less skewed the lines. If there is no corresponding event in the reproduction, the circle has a red fill. Figure taken from [KMR21].

Definition 16 ((Non-)Equal Replay). *An equal replay is a reproduction in which all messages on all streams are equal (i.e., including the order of the messages) to the streams in the original execution. A non-equal replay is a reproduction in which at least one message on one stream is different or at a different position of the stream.*

Fig. 8.8 uses the event data logged during the executions of the application and its replay to visualize the difference between an equal and a non-equal replay. Each event is represented by a circle. If an event is replayed, it is connected to the original instance of the event with a line; otherwise, the circle representing the event has a red fill. A more accurate reproduction has less skew in the lines and fewer red circles. For example, at about second 24, there is a red circle in Fig. 8.8(b). This was caused by two events being incorrectly swapped in the replay causing the heater to not be turned on at second 24.

As explained in Sec. 7.5, the correctness of the reproduction depends on the configuration of the DS. The operating system’s scheduler might incorrectly change the order of two events that are very close together. DSs can mitigate this effect by putting two

Determinism Spacing (DS)	disabled	1 ms	3 ms	5 ms
non-equal replays	49 %	15 %	5 %	0 %
avg. latency error	1.9 ms	2.6 ms	2.5 ms	3.0 ms
median latency error	1.2 ms	1.5 ms	1.5 ms	2.0 ms
standard deviation σ	2.4 ms	3.5 ms	3.3 ms	3.2 ms

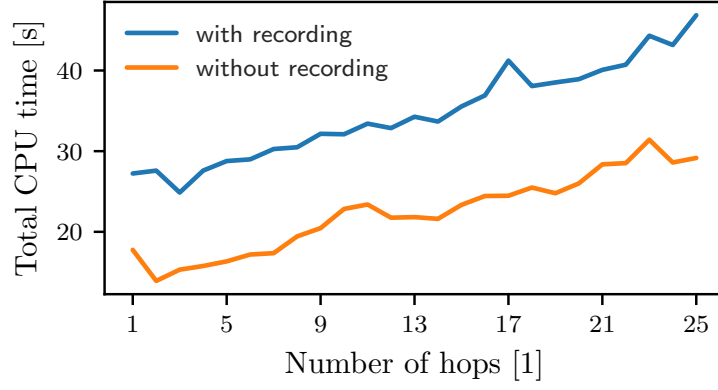
Table 8.2: Influence of DSs on the accuracy of the order of messages and timing. Table taken from [KMR21].

events that are close together further apart without affecting subsequent events. The simulation was performed by a virtual machine with 8 cores of AMD Ryzen 2600X, 10GB of RAM, and Ubuntu 20.04 as its operating system. The DS was configured to be either turned off, 1 ms, 3 ms, or 5 ms. For every configuration of the DS, we executed 100 simulations with different random seeds. Table 8.2 shows the results of this simulation. Without DSs, almost half (49 %) of the reproductions were non-equal to the original execution. By setting the DS to 5 ms all 100 runs of the simulation produced an equal replay. Note, that the DS only applies to events that are very close together. Therefore, the average and median latency error, *i.e.*, the inaccuracy in reproducing the original system’s timing, only slightly increases when increasing the DS. As already mentioned, DSs trade a slightly less accurate timing reproduction for a considerably improved accuracy in the reproduction of the streams.

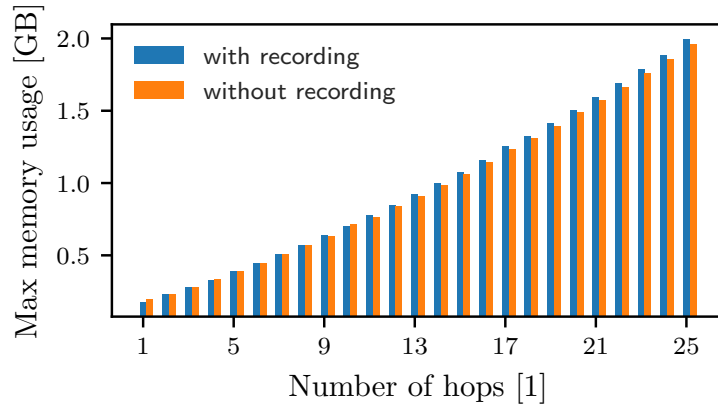
Overall, this case study showed that our model-to-model transformation-base replayer system can reproduce the behavior of IoT applications. Further, it showed that DSs can reduce the number of non-equal replays. The exact configuration of the DS depends on the developer’s computer, *i.e.*, its hardware and operating system. Because not all events need to be handled by DSs, the timing inaccuracy introduced by the DSs only has a slight effect on the overall inaccuracy.

8.4 Performance Evaluation: Transformation-based Replayer

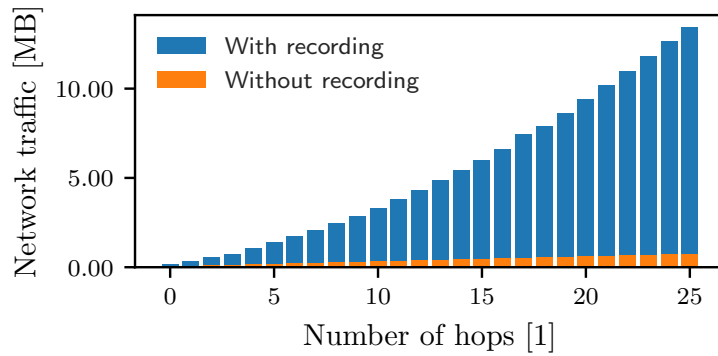
After showing the general feasibility of the transformation-based replayer, we now evaluate the overhead introduced by using the replayer. We evaluate the overhead in terms of processing (CPU), memory (RAM), and network traffic. To evaluate it we construct an architecture specifically designed to be scaled. It consists of a `Source` and a `Sink` component. The `Source` component produces numbers that it forwards on its outgoing port. The `Sink` component in return accepts values on its incoming port and logs them. Between them, there is a variable number of *hops*, *i.e.*, components with one incoming and one outgoing port that simply forwards the messages they receive. For each mea-



(a) CPU overhead of recording the execution of an IoT application.



(b) Memory overhead of recording the execution of an IoT application.



(c) Network overhead of recording the execution of an IoT application.

Figure 8.9: Performance results of of executing the recorder. Figure taken from [KMR21].

surement, the `Source` produces 100 messages that it sends to the `Sink` via the variable number of hops. We capture the resource utilization using Google’s `cAdvisor`⁴.

Fig. 8.9(a) shows that our method causes a noticeable overhead. This is mostly caused by the required parsing of JSON messages. Additionally, the communication with the recorder caused a high number of network library calls. Fig. 8.9(b) shows that our recording causes a small overhead for recording. This is expected because of the additional recording modules that are added to each port. The overhead in terms of network traffic is shown in Fig. 8.9(c). Recording causes a significant overhead that is mostly caused by our usage of vector clocks. The vector clocks require each component to keep a clock value for every other component. Hence, it scales with $\mathcal{O}(n^2)$; adding a component (here: hop) increases the size of every other vector clock.

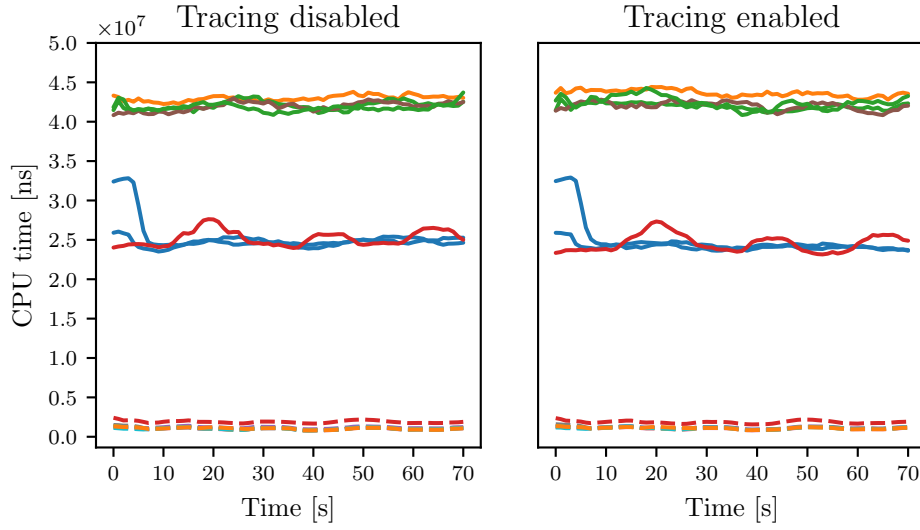
We expect that the practical effects of this can be limited by compressing the messages exchanged with the recorder. Furthermore, the overall size of the network traffic is still a low number of megabytes distributed among (up to) 27 components, *i.e.*, 25 hops and the `Source` and `Sink` component. The CPU and memory usage scales linearly. Overall, we conclude that our approach is feasible for recording the behavior of IoT applications for a limited amount of time.

8.5 Performance Evaluation: Log Tracing

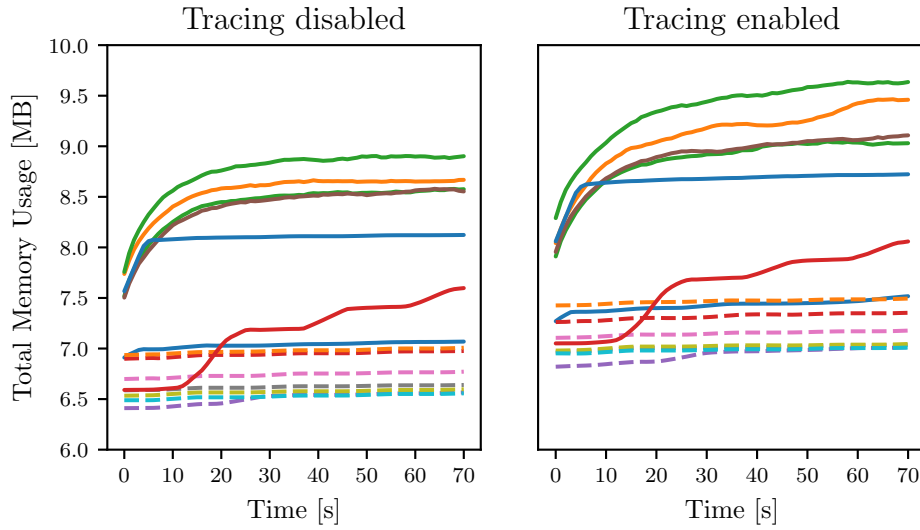
For the performance evaluation of the log tracing, we used the architecture already presented in the HVAC case study in Sec. 8.3. We recorded both the CPU and memory overhead for using tracing. In contrast to the previous performance evaluation, the network overhead was not recorded. Because the logged data is only transmitted when actually requested by the web application, the network traffic overhead depends strongly on the number of requests and the actually requested data. For example, requesting the logs of a component with twenty connections to other components that has to process messages every 100 ms is more expensive than requesting the logs of a component that barely processes any messages. Overall, the simulation was performed 100 times. 50 times, the tracing was disabled, the other 50 times, the tracing was enabled. The data about the resource usage was captured using Google’s `cAdvisor`.

Fig. 8.10 shows the results of the performance evaluation. Each combination of line color and style represents a different component instance. The consumption is shown as a running average value of the 9 last values. The zeroth second in the plots refers to the second when the simulation started. The components were started before the simulation started. This was done so that the initial setup of the components does not influence the measurement. Overall, enabling the log tracing does not noticeably increase the CPU usage. This is expected as the data structures for keeping the logs

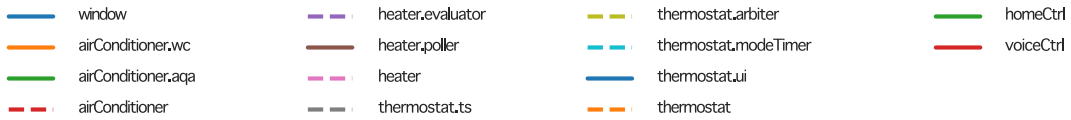
⁴`cAdvisor` project website. [Online]. Available: <https://github.com/google/cadvisor>. Last accessed: 03.01.2022



(a) CPU usage overhead.



(b) Memory footprint overhead.



(c) Legend. Each color/line type represents a different component instance.

Figure 8.10: Performance results of executing an IoT application with and without log tracing. Figure taken from. Figure taken from [KMM⁺22].

only need to be stored. This does not require a large amount of processing besides the processing that is already required for the logging that is done regardless of whether the logs are stored. The difference between the three groups of components visible in the plots comes from the frequency of their execution. Some components called their `compute` method every second, while other components called their `compute` method every 500 ms. Accordingly, the components that call their `compute` method less often have a lower CPU consumption. The components with close to zero CPU consumption are composed components that do not execute a computation themselves. Instead, they only forward messages to their subcomponents.

The memory consumption is increased by a constant factor of about 0.5 MB for the tracing module itself. As the execution goes on the traced architecture also consumes more memory compared to the execution without tracing for storing the logs. This is visible, for example in the (non-dashed) dark green and orange lines, *i.e.*, the two component instances with the highest memory consumption. While the higher memory consumption is mostly irrelevant for higher-powered IoT devices like a Raspberry Pi that is often equipped with microSD cards offering 32 GB or more, it can be a problem for low-powered devices. In these scenarios, the logs should, ideally, not be stored on the IoT devices for extended periods of time. Instead, the logs could be uploaded to a blob storage or database in the cloud. This, however, would cause a larger network traffic overhead regardless of whether the logs are ever accessed. In case neither storing the data nor uploading the (compressed) data is an option, the IoT devices will need to use a variant of Linux's `logrotate`, where only the most recent logs are kept. A disadvantage of this is that throwing away logs can make it impossible to analyze errors whose cause is only documented by these older logs.

Besides showing the overhead induced by the log tracing, the measurements with disabled log tracing also show MontiThings' general CPU and memory consumption. Even when executed as a container like in this evaluation, MontiThings usually has a low memory footprint of only a few megabytes. MontiThings also advises the operating system to schedule other threads while it is not actively processing messages (*e.g.*, using `std::this_thread::yield`) to not waste processing resources. Overall, this leads to low processing requirements that enable devices like the Raspberry Pi to run a large number of MontiThings components simultaneously. It is, however, important to note that very low-powered devices are often out of scope because their compilers sometimes may not support the full C++ standard library (**TA4**).

8.6 Student Lab: Autonomous Driving

In the winter semester 2020/21 we⁵ conducted a student lab on autonomous driving. Overall, 26 students were tasked with developing autonomous driving software. The software was supposed to be executed on three DSA VCG devices that were connected using the CAN bus. By connecting the VCGs to a driving simulator, the students were able to control the simulator's cars using the software executed on the VCGs.

One of three groups consisting of eight students (one of which had prior experience with MontiThings) was supposed to develop the autonomous driving software using MontiThings. At the beginning of the semester, all students received an introduction to MontiThings. As part of this introduction, they had to build an architecture for calculating prime numbers. This allowed us to evaluate 1. how students, *i.e.*, inexperienced developers, use MontiThings and 2. how well MontiThings fits distributed applications beyond its IoT focus. It is, however, important to note that the results of this student lab can only give a rough impression of how these two questions might be answered because of 1. the low number of students, 2. the student's lack of experience and thus their lower success chances of using MontiThings in a non-IoT domain, 3. the lack of a formal user survey. We specifically did not conduct a formal user survey because the low number of participants would not have allowed us to draw meaningful conclusions. Moreover, as we had to grade the students at the end of the semester, even a positive evaluation of MontiThings could have been caused by the students thinking that the user study may influence their grades. By inspecting the artifacts they created during the lab, we could, however, perceive how the students used MontiThings.

We made the following observations:

- The students' architecture consisted of about 43 component types across three packages. The exact number cannot be determined since the students created numerous branches for proposing modifications to each other. Each package was intended to be deployed to one VCG. Some of the components are duplicated from other packages.
- The students created C++ code unrelated to any component and placed them in the folder intended for hand-written code. When we asked the students why they did this, they answered that they wanted to use specific C++ constructs that they felt they were not able to create using class diagrams and MontiThings. Most notably, the students built a generic class for vectors and defined several functions on these vectors. As the MontiVerse's class diagrams do not offer generics, the students could not build such a class using the class diagrams. Furthermore, the students specified numerous constants and functions, *e.g.*, π or a conversion from

⁵The lab was conducted in collaboration with my colleagues Evgeny Kusmenko and Sebastian Stüber who supervised the groups that did not use MontiThings.

radians to degrees. It would, of course, have been possible to specify such a function using a MontiThings component. We assume that students are less likely to build components for data structures built using C++ because that would have required them to create a new independent file. We base this on the fact that their C++ files often not only contained a single class but many unrelated pieces of code. MontiThings, in contrast, enforces a structured way of organizing files (**MC1**), *e.g.*, by only allowing one component definition per file.

- The students generally realized communication using handwritten C++ code. For example, they created C++ files not related to any component for using the CAN bus and exchanging data with the simulator. We assume this is the case because such code is usually strongly based on existing C++ code, *e.g.*, `linux/can.h`. These files were then referenced by the handwritten implementation of components. For example, a `CANDoubleSink` component referenced their CAN bus implementation to send messages. We assume the students did this to avoid code duplication between the implementation files of components.
- Many components were either very complex, *e.g.*, had over 20 subcomponents or about 10 ports, or very simple, *i.e.*, had only one port.
- In some cases, the business logic and the technical aspects of the realization cannot be strictly separated. For example, the students assigned CAN IDs to components within the business logic, *i.e.*, in MontiThings components. As the CAN ID is not only an arbitrary message ID but also specifies the priority of the messages based on the CAN MAC protocol, business logic and technical aspects are intertwined.
- Even though it is not required by the language, students generally tried to give MontiThings components a certain structure. Mostly, the ports are defined at the beginning of the file. The rest of the file is often either a list of subcomponents followed by a list of connectors or groups of subcomponents followed by their connectors. Similarly, the handwritten C++ code written for MontiThings components was also often well-structured. We attribute this to the fact that MontiThings enforces certain good practices, *e.g.*, it is not possible to write definitions of two components in one file. In the long term, avoiding bad practices has the potential of making the applications more maintainable.

Overall, the students successfully completed their task of building an autonomous driving application using MontiThings. They realized multiple driving scenarios such as keeping the safety distance if a car in front of the controlled car brakes. Furthermore, they realized some tasks which went beyond our scope of the lab, *e.g.*, testing their car on a map of the Hockenheimring racing track. In conclusion, the lab demonstrates that MontiThings can also be used outside the IoT domain for building distributed applications. For applications that rely heavily on existing C++ code, however, the

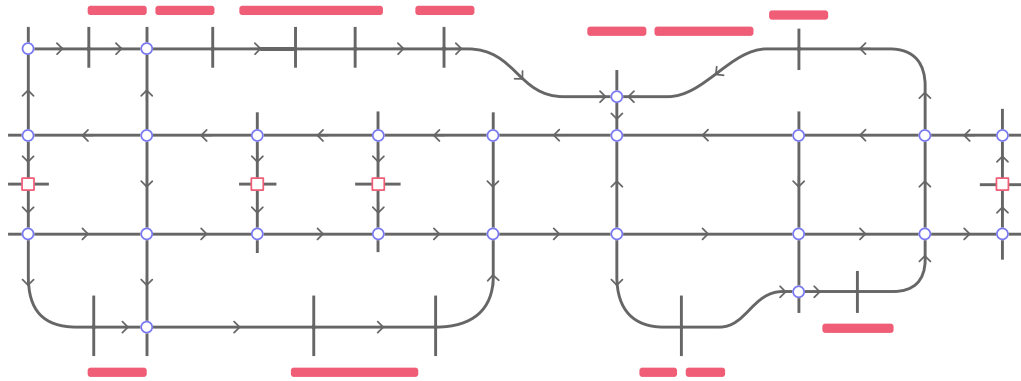


Figure 8.11: Map of the Fischertechnik setup. This figure was created by the lab’s students. Thin black lines represent the road the robots drive on. Blue circles are intersections, red squares are idle positions of the robots. Thick red lines next to the road represent Fischertechnik machines.

benefits of MontiThings over using only C++ presumably only show once the application reaches a certain level of complexity. While this student lab is certainly far from a large-scale industrial user study, the artifacts created by the students gave some interesting insights on how inexperienced developers use MontiThings.

8.7 Student Lab: Fischertechnik

In the winter semester 2021/22 we⁶ conducted another student lab with a focus on industry 4.0. For this lab, 16 students had to build an automated yogurt factory. The setup consisted of Fischertechnik machines representing the machines of a smart factory and several robot cars representing the robots in a factory. Two groups controlling the Fischertechnik machines and LEGO robots worked using MontiArc. The other two groups used MontiThings for controlling Raspberry Pi-based cars and additional sensors for the factory’s machines connected to further Raspberry Pis. Overall, seven students worked with MontiThings, one of whom had prior experience with MontiThings. All teams needed to exchanged data across language and goal boundaries to build the yogurt factory.

Fig. 8.11 shows a map of the factory created by the students. The Fischertechnik devices stood next to a network of roads. The roads were drawn on the floor using black tape. This enabled the cars to follow the black lines on the ground. The task of the cars

⁶The lab was conducted in collaboration with my colleague David Schmalzing who supervised the groups that did not use MontiThings.

was to transport containers of yogurt between the factory’s machines so that it could be further processed.

Like in the previous student lab, we inspected the artifacts created by the students and asked them about certain design choices to gain insights on how inexperienced developers use MontiThings. We gained the following insights:

- In the initial weeks, the students were tasked with calculating prime numbers to get to know MontiThings. The intended solution was to model a variant of the *sieve of Eratosthenes*. Some of the students, however, found creative solutions. For example, one student used MontiThings’s set expressions in combination with OCL constraints to filter prime numbers from a set of integers.
- One group had many problems using MontiThings (and C++ in general). For example, they needed help when the C++ compiler reported a missing semicolon. The missing semicolon was caused by a C++ Freemarker template provided by the students that did not include this semicolon. As MontiThings does not inspect the content of the user-provided templates, the error was carried over to the generated code. Apparently, the students were not able to conclude that this error might have been caused by their template. Other errors included using parentheses instead of curly braces for C++’s list initialization and returning a value in a function without a return value. We expect such errors could, however, be avoided by better tool support, *e.g.*, by showing errors in the C++ code directly in the template instead of somewhere in the generated code. The fact that MontiThings is a research prototype, where similar errors cannot be ruled out completely, probably contributed to the students giving up on trying to find the error relatively quickly and blaming MontiThings instead. Moreover, C++ is not part of the normal computer science curriculum at RWTH Aachen University probably contributed to their difficulty in debugging.
- One student said towards the end of the course (translated to English): “I did not really understand why we had to use [MontiThings].” The student was from a group that built the software to control cars using MontiThings. As all of their components were executed by the same device, they did not benefit from generating, *e.g.*, a communication infrastructure, as the same functionality could have been implemented using a monolithic Python script. This suggests that there tends to be little benefit from model-driven development in small non-distributed IoT projects, as much of the development effort is focused on hardware connectivity rather than business logic. While the students in this lab used MontiThings, we suspect that this effect would also be seen in other languages such as Calvin or ThingML, since the students mainly used the basic features of the language that are similarly implemented in the other languages.

- Components used to access sensors or actuators were mostly just empty shells. The main functionality of these components took place in the attached Python / C++ files. Students reported that they missed the wide range of libraries in MontiThings to access, for example, the front two digits of a number. The MontiThings behavior language was mainly used in controller components that did not directly access hardware or the operating system. In these components, the data usually already existed in the desired form and was then used to make decisions. For example, they used MontiThings' behavior language to implement their line-following and collision-avoidance algorithms.
- Students reported in their final presentation that they felt their project was too small to benefit from model-driven development (in general, not just related to MontiThings). Accordingly, there had been few opportunities to reuse components.
- In their final presentation, the students reported that modeling led to (translated into English): “a clearer program and process structure.” They kept this opinion after confirmation by the department head, Prof. Rumpe, that they were not forced to say this and could revise their opinion again.
- Overall, students did not pay much attention to reliability of their software. For example, no pre- or postconditions were used. When data types were used that could have been expressed as SI units, primitive data types such as `int` and `double` were used instead. When asked why there was no safeguarding against, for example, incorrect sensor values, the response was that the students had not observed any incorrect sensor values and therefore did not think it was necessary to safeguard. This is in line with the statements [TM17a], which attest developers a lack of experience in dealing with IoT projects if they are not used to developing mission-critical distributed systems.

8.8 Discussion

In this section, we evaluated MontiThings using case studies, performance measurements, and student labs. The case studies showed that MontiThings can be used for developing and deploying IoT applications including the connection to digital twins. All of the case studies are small-scale examples. MontiThings has not yet been evaluated against large-scale deployments comprising thousands of IoT devices. Our results from the deployment of the smart home/hotel case study suggest that such large-scale deployments would require the generated Prolog code to be more efficient. Currently, the Prolog backtracking mechanism checks a lot of solutions that might lead to a valid deployment but are undesired from a user experience point of view. For example, if the device owners already rejected buying a piece of hardware, they will not accept buying the same piece

of hardware and additionally relaxing one of their requirements. As Prolog’s backtracking will test each combination of a rejected proposal with other proposals, its worst-case complexity is $\mathcal{O}(2^n)$, where n is the number of possible options for proposals. While our software auto-rejects such unwanted proposals for better user interaction, they still have to be calculated. This is feasible for small-scale deployments like the one presented in our smart home. Large-scale deployments, however, would require the generated Prolog code to handle rejected proposals more efficiently.

Our student labs revealed that the concept of external connectors is not intuitive to understand for entry-level developers. Most students seem to view sensors and hardware access as components. As a result, they often develop components that have no other purpose than connecting an external connector to a normal port. For example, this could be a `TemperatureSensor` component that accesses a temperature sensor using an incoming port with an external connector and then forwards all incoming values on an outgoing port. We assume that the maintainability, testability, and flexibility benefits of using external connectors only show in larger projects that are deployed to various infrastructures. Providing only one set of devices to students tempts them to write their software only for that exact set of devices and ignore all other devices that may exist. Likewise, the students rarely used pre- and postconditions. This is consistent with the challenges stated by [TM17a] that developers of mobile apps and other more *traditional* software are not aware of the challenges of developing IoT applications and, thus, underestimate tasks like error handling.

As DSLs are usually not known to new developers, they always involve a certain learning curve. While the basic concepts of MontiThings, *i.e.*, C&C architectures, are usually easy to understand for most students, more complex topics that possibly involve multiple DSLs, *e.g.*, defining a new data type using a class diagram and using it as the type of a port, usually require more explanation. By making our behavior specifications similar to programming languages already known by students, *i.e.*, Java, we flatten the learning curve. However, currently, the lack of tool support can lead to a frustrating experience of using MontiThings. Multiple students complained that they always had to rerun the generator only to see an error message. Once they fixed the error they had to run the generator again until all errors are fixed. Modern development tools usually provide such error messages inline and directly while typing. While such tool support can make improve the experience of using a language, developing such tools was out of the scope of this thesis.

Chapter 9

Conclusion and Future Research Directions

The development of IoT applications is complex due to, *e.g.*, their distributed nature, heterogeneous target devices, and error-proneness hardware. Model-based development promises to make the complexity of IoT application development manageable [MHF17] by raising the level of abstraction. Related work proposed several C&C ADL for developing IoT applications. These, however, have various limitations such as a lack of separation of concerns. In this thesis, we presented the MontiThings ecosystem for the model-driven development and deployment of IoT applications. Compared to related work, MontiThings provides advantages such as failure-handling capabilities and a requirements-based deployment method that can propose modifications in case of unfulfillable requirements.

Fig. 9.1 gives a simplified overview of the MontiThings ecosystem. The roles of the ecosystem are based on [Zam17]. IoT developers are responsible for developing the application, device owners own the IoT devices and might have requirements for the software deployment, and users interact with the IoT devices and (optionally) information systems that enable them to achieve the IoT applications purpose. The IoT developers specify IoT applications using MontiThings architectural models for defining the behavior and class diagrams for describing data structures. Handwritten code may be used to integrate with existing libraries or to access hardware. Additionally, a model-driven information system can be developed using the MontiGem framework as described in [AMN⁺20, GHK⁺20, ANV⁺18]. The information system and IoT application can be combined using a tagging language to synthesize digital twins of the IoT application and keep the systems synchronized.

After uploading the artifacts to a development platform such as GitLab or GitHub, a CI pipeline processes the artifacts to create deployable packages. Then, model-to-model transformations are applied and model-to-text generation creates C++ code from the models. This code is integrated with a runtime environment providing core functionality such as communication. The generated code is (cross-)compiled and packaged as container images for further distribution. Using an information system, the device owner specifies requirements for the deployment. Based on these requirements, the deployment manager decides which software to deploy to which devices by generating and evaluating Prolog code corresponding to the requirements. The devices pull the container images

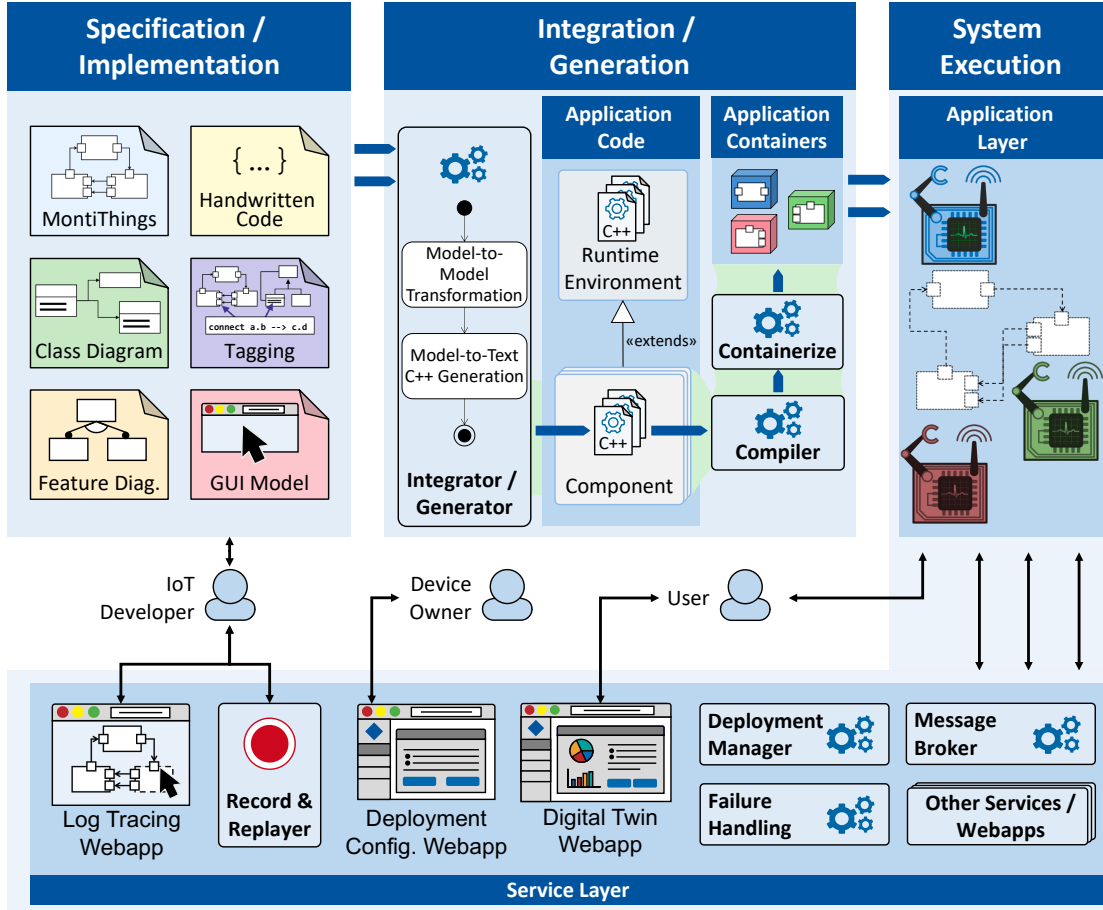


Figure 9.1: High-level overview of the MontiThings ecosystem. Figure adapted from [KRSW22].

from a container registry and start their execution. During this execution, they can be combined with various services, *e.g.*, digital twins, failure recovery, or a recorder that observes their behavior for analysis purposes. Users can influence the application during runtime by either interacting with the IoT devices, *e.g.*, by pressing a button, or via information systems that may control (parts of) the IoT application.

While MontiThings already provides a feature-rich ecosystem, we envision the following directions for future research:

Integration with cloud systems MontiThings is focused on the software executed by IoT devices. While cloud resources can be integrated, *e.g.*, by using virtual machines to execute components or accessing cloud services via handwritten code, this integration requires manual work to set up these cloud resources and integrate

them with the IoT system. Using Infrastructure-as-Code languages such as Terraform, the code generator could also automatically deploy the needed resources and integrate them with the IoT system. In future IoT app stores, this could also give the device owner more control over the cloud resources compared to using cloud services that the IoT developers provide for all instances of an application.

User-specified behavior Currently, the behavior of the system is mostly defined by the IoT developer. Since IoT systems can be highly user-specific, users could also be enabled to specify parts of the behavior. In commercial systems, such user-specified behavior is often based on a rules engine that users can utilize to specify behavior using a lightweight scripting language (*cf.* Apple Siri Shortcuts). To lower the entry barrier, vendors often offer a graphical editor. An overview of existing work can be found in [DRF22] as *end-user development*.

Plug-and-play interoperability Much research has been conducted on engineering single IoT applications. Future use cases like smart cities will require a high degree of interoperability between different systems. Some authors thus envision systems that can be combined similar to putting LEGO bricks together [DeF21, PA15]. This degree of interoperability, however, requires a higher degree of standardization and analyses checking the compatibility of software components (not only syntactically but also semantically).

Standardization IoT development is hampered by a lack of standardization [DRF22]. This lack of standardization includes many aspects of the development including 1. communication (including data exchange formats), 2. accessing sensors, actuators, and other hardware, 3. interacting with low-powered microcontrollers, and 4. accessing cloud resources. While the standards themselves have to be provided by large industrial alliances of relevant vendors, academia can contribute by, *e.g.*, providing requirements, analyses, or proposals of individual aspects for such standards.

Energy efficiency Application energy efficiency is often ignored by model-driven IoT frameworks. While energy efficiency is not an issue in a lab where all devices are connected to a power source, energy efficiency is an important consideration in real-world setups as many devices are battery-powered. MontiThings provides mechanisms for bridging temporary outages that can be caused by, *e.g.*, entering battery-saving modes. Future IoT DSLs may, however, need to give developers more precise control over power management. However, the lack of such power management features in code generators is also caused by the lack of standards.

Explainability IoT applications often depend on a large number of influences such as sensor readings. This can make their behavior incomprehensible to developers and

users. MontiThings aims to make IoT applications easier for developers to understand by providing a reproduction service and a tracing service that can filter irrelevant log messages. Future research may seek to also provide natural language explanations for application behavior. The explainability problem is further exacerbated by the ever-expanding use of machine learning.

Overall, this thesis aims to contribute to the goal of distributing IoT applications via future smartphone-like app stores [BSS⁺17]. To this end, we have shown how model-driven IoT applications can be deployed and integrated with IoT devices that are not necessarily fully known at the time of development, but are only specified via interfaces to sensors or actuators. We have outlined how non-technically trained domain experts can be involved in the deployment process to adapt the deployment to their needs. For this purpose, we presented a requirement-based deployment method that is able to propose modifications to the domain experts. This allows IoT applications distributed through app stores to be adapted to the different infrastructures of their customers. Using a variety of analytics and resilience methods, MontiThings also addresses handling the error-proneness of IoT devices deployed under harsh environmental conditions.

Bibliography

- [ABH⁺17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.
- [AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.
- [AKKR21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21)*, pages 55–68. ACM SIGPLAN, October 2021.
- [AMMK19] Shabir Ahmad, Faisal Mehmood, Asif Mehmood, and DoHyeun Kim. Design and Implementation of Decoupled IoT Application Store: A Novel Prototype for Virtual Objects Sharing and Discovery. *Electronics*, 8(3), 2019.
- [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)*, volume P-304 of *LNI*, pages 59–66. Gesellschaft für Informatik e.V., May 2020.
- [ANV⁺18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA '18)*, volume 2097 of *CEUR Workshop Proceedings*, pages 75–79. CEUR-WS.org, May 2018.

- [AP17] Ola Angelsmark and Per Persson. Requirement-Based Deployment of Applications in Calvin. In Ivana Podnar Žarko, Arne Broering, Sergios Sourso, and Martin Serrano, editors, *Interoperability and Open-Source Solutions for the Internet of Things*, pages 72–87, Cham, 2017. Springer International Publishing.
- [ASD19] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Characterizing Internet of Things Systems through Taxonomies: A Systematic Mapping Study. *Internet of Things*, 7:100084, 2019.
- [BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., USA, 1998.
- [BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.
- [BCPP20] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, 2020.
- [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 Shahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE’20)*, volume 12127 of *Lecture Notes in Computer Science*, pages 85–100. Springer International Publishing, June 2020.
- [BDJ⁺22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Zieffle. A Computer Science Perspective on Digital Transformation in Production. *ACM Trans. Internet Things*, 3(2), feb 2022.
- [BEK⁺16] Uwe Breitenbücher, Christian Endres, Kálmán Képes, Oliver Kopp, Frank Leymann, Sebastian Wagner, Johannes Wettinger, and Michael Zimmermann. The OpenTOSCA Ecosystem – Concepts & Tools. *European Space project on Smart Systems, Big Data, Future, Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016*, 2016.
- [BEK⁺18] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings*

- of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.
- [BJK⁺18] Christopher Brooks, Chadlia Jerad, Hokeun Kim, Edward A. Lee, Marten Lohstroh, Victor Nouvellet, Beth Osyk, and Matt Weber. A Component Architecture for the Internet of Things. *Proceedings of the IEEE*, 106(9):1527–1542, September 2018.
- [BKK⁺22] Arvid Butting, Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Radoslav Orlov, and Bernhard Rumpe. Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices. In *Proceedings of the 21th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22)*, pages 108–121. ACM, December 2022.
- [BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.
- [BKRW17b] 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.
- [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.
- [Bre07] Ulrich Breymann. *C++ Einführung und Professionelle Programmierung*. Carl Hanser Verlag München Wien, 9 edition, 2007.
- [Bro01] Manfred Broy. Refinement of time. *Theoretical Computer Science*, 253(1):3–26, 2001. ARTS 97.
- [Bro10] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 53(10):1758–1782, 2010.
- [BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Heidelberg, 2001.

- [BSS⁺17] Arne Bröring, Stefan Schmid, Corina-Kim Schindhelm, Abdelmajid Khelil, Sebastian Käbisch, Denis Kramer, Danh Le Phuoc, Jelena Mitic, Darko Anicic, and Ernest Teniente. Enabling IoT Ecosystems through Platform Interoperability. *IEEE Software*, 34(1):54–61, 2017.
- [CAF21] Angel Cañete, Mercedes Amor, and Lidia Fuentes. Supporting IoT applications deployment on edge-based infrastructures using multi-layer feature models. *Journal of Systems and Software*, page 111086, 2021.
- [CCS15] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödín. On the Generation of Full-Fledged Code from UML Profiles and ALF for Complex Systems. In *12th International Conference on Information Technology - New Generations*, pages 81–88, 2015.
- [CDO⁺15] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert Use in GitHub Projects. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 755–766, 2015.
- [CDRS20] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. How is Open Source Software Development Different in Popular IoT Projects? *IEEE Access*, 8:28337–28348, 2020.
- [CHS⁺17] Steve Counsell, Tracy Hall, Thomas Shippey, David Bowes, Amjed Tahir, and Stephen MacDonell. Assert Use and Defectiveness in Industrial Code. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 20–23, 2017.
- [Cle96] Paul C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, 1996.
- [CPD16] Bruno Costa, Paulo F. Pires, and Flávia C. Delicato. Modeling IoT Applications with SysML4IoT. In *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 157–164, 2016.
- [CS16] Federico Ciccozzi and Romina Spalazzese. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In *10th International Symposium on Intelligent and Distributed Computing*, October 2016.
- [DeF21] Joanna DeFranco. 12 Flavors of IoT. *Computer*, 54(10):133–137, 2021.
- [DHH⁺20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese

- von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.
- [DJK⁺19] Manuela Dalibor, Nico Jansen, Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Tagging Model Properties for Flexible Communication. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 39–46. CEUR Workshop Proceedings, September 2019.
- [DRF22] João Pedro Dias, André Restivo, and Hugo Sereno Ferreira. Designing and constructing internet-of-Things systems: An overview of the ecosystem. *Internet of Things*, 19:100529, 2022.
- [DSF21] Rustem Dautov, Hui Song, and Nicolas Ferry. Towards a Sustainable IoT with Last-Mile Software Deployment. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2021.
- [EBC⁺22] Romina Eramo, Francis Bordeleau, Benoit Combemale, Mark van den Brand, Manuel Wimmer, and Andreas Wortmann. Conceptualizing Digital Twins. *IEEE Software*, 39(2):39–46, 2022.
- [Ecl20] Eclipse Foundation. IoT Developer Survey 2020. [Online]. Available: <https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020>. Last accessed: 20.06.2021, October 2020.
- [Ecl22] Eclipse Foundation. IoT & Edge Developer Survey Report. [Online]. Available: <https://outreach.eclipse.foundation/iot-edge-developer-survey-2022>. Last accessed: 16.10.2022, September 2022.
- [FG08] Conny Franke and Michael Gertz. Detection and Exploration of Outlier Regions in Sensor Data Streams. In *IEEE International Conference on Data Mining Workshops*, pages 375–384, 2008.
- [FGH06] Peter Feiler, David Gluch, and John Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.
- [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.

- [FN19] Nicolas Ferry and Phu H. Nguyen. Towards Model-Based Continuous Deployment of Secure IoT Systems. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 613–618, 2019.
- [FNS⁺19] Nicolas Ferry, Phu Nguyen, Hui Song, Pierre-Emmanuel Novac, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems. In *IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 870–875, 2019.
- [FNS⁺20] Nicolas Ferry, Phu H. Nguyen, Hui Song, Erkuden Rios, Eider Iturbe, Satur Martinez, and Angel Rego. Continuous Deployment of Trustworthy Smart IoT Systems. *Journal of Object Technology*, 19:16:1–23, 2020.
- [Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.
- [Fow19] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. A Martin Fowler signature book. Addison-Wesley, 2019.
- [FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, pages 37–54, May 2007.
- [Für20] Joshua Christian Fürste. Model-Driven Development and Deployment of Distributed Internet of Things Applications. Master Thesis. RWTH Aachen University. Software Engineering Group., January 2020.
- [Gai86] Jason Gait. A Probe Effect in Concurrent Programs. *Software: Practice and Experience*, 16(3):225–233, March 1986.
- [GASS06] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference, ATEC '06*, page 27, USA, 2006. USENIX Association.
- [GBLL15] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. *Proceedings - 2015 5th International Conference on the Internet of Things, IoT 2015*, pages 155–162, 2015.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

- [GHK⁺08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GHK⁺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.
- [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.
- [GKR19] Nicola Gatto, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansoor Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MDE Intelligence*, pages 196–202, September 2019.
- [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.
- [GMW10] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *CASCON First Decade High Impact Papers*, CASCON ’10, pages 159–173, USA, 2010. IBM Corp.
- [GVM⁺17] Gordana Gardašević, Mladen Veletić, Nebojša Maletić, Dragan Vasiljević, Igor Radusinović, Slavica Tomović, and Milutin Radonjić. The IoT Architectural Framework, Design Issues and Application Domains. *Wireless Personal Communications*, 92(1):127–148, 2017.
- [Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

- [Häu20] Jan Häusler. Fehlertolerante, modellgetriebene IoT-Architekturen mittels zuverlässigem Message Broking. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., September 2020.
- [HBJD20] Nicolas Hili, Mojtaba Bagherzadeh, Karim Jahed, and Juergen Dingel. A model-based architecture for interactive run-time monitoring. *Software and Systems Modeling*, 19(4):959–981, 2020.
- [HFMH16] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS '16*, pages 125–135, New York, NY, USA, 2016. ACM.
- [HKK⁺22] Mattis Hoppe, Jörg Christian Kirchhof, Evgeny Kusmenko, Chan Yong Lee, and Bernhard Rumpe. Agent-Based Autonomous Vehicle Simulation with Hardware Emulation in the Loop. In *2022 IEEE Intelligent Vehicles Symposium (IV)*, pages 16–21, 2022.
- [HKKR19] Alexander Hellwig, Stefan Kriebel, Evgeny Kusmenko, and Bernhard Rumpe. Component-based Integration of Interconnected Vehicle Architectures. In *30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles*, pages 146–151. IEEE, June 2019.
- [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.
- [HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.
- [HLR17] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. Model-driven Development of Adaptive IoT Systems. In *Proceedings of MODELS 2017. Workshop ModComp*, volume 2019, pages 17–23, Austin, United States, September 2017. CEUR.
- [HNPR13] Lars Hermerschmidt, Antonio Navarro Perez, and Bernhard Rumpe. A Model-based Software Development Kit for the SensorCloud Platform. In *Workshop Wissenschaftliche Ergebnisse der Trusted Cloud Initiative*, pages 125–140. Springer, Schweiz, 2013.

- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What’s the Semantics of ”Semantics”? *IEEE Computer*, 37(10):64–72, October 2004.
- [HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.
- [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.
- [HSG18] Rashina Hoda, Norsaremah Salleh, and John Grundy. The Rise and Evolution of Agile Software Development. *IEEE Software*, 35(5):58–63, 2018.
- [HT06] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.
- [JBD21] Karim Jahed, Mojtaba Bagherzadeh, and Juergen Dingel. On the benefits of file-level modularity for EMF models. *Software and Systems Modeling*, 20(1):267–286, 2021.
- [JCO17] Gonalo Jesus, Ant3nio Casimiro, and Anabela Oliveira. A Survey on Data Quality for Dependable Monitoring in Wireless Sensor Networks. *Sensors*, 17(9), 2017.
- [KADAS19] Ala’ Khalifeh, Khaled Aldahdouh Aldahdouh, Khalid A. Darabkh, and Waleed Al-Sit. A Survey of 5G Emerging Wireless Technologies Featuring LoRaWAN, Sigfox, NB-IoT and LTE-M. In *International Conference on Wireless Communications Signal Processing and Networking (WiSPNET)*, pages 561–566, 2019.
- [Kay20] Paridhika Kayal. Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope : Invited Paper. In *IEEE 6th World Forum on Internet of Things (WF-IoT)*, pages 1–6, 2020.
- [KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
- [KGR20] Heiko Kozi3lek, Sten Gr3uner, and Julius R3uckert. A Comparison of MQTT Brokers for Distributed IoT Edge Computing. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, pages 352–368, Cham, 2020. Springer International Publishing.

- [KKM⁺22] Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. Efficiently Engineering IoT Architecture Languages—An Experience Report (Poster). STAF 2022 Workshop Proceedings: 10th International Workshop on Bidirectional Transformations (BX 2022), 2nd International Workshop on Foundations and Practice of Visual Modeling (FPVM 2022) and 2nd International Workshop on MDE for Smart IoT Systems (MeSS 2022) (co-located with Software Technologies: Applications and Foundations federation of conferences (STAF 2022)), July 2022.
- [KKMR19] Jörg Christian Kirchhof, Evgeny Kusmenko, Jean Meurice, and Bernhard Rumpe. Simulation of Model Execution for Embedded Systems. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MLE*, pages 331–338. IEEE, September 2019.
- [KKR⁺22a] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *ACM Transactions on Internet of Things*, 3(4), 2022.
- [KKR⁺22b] Jörg Christian Kirchhof, Evgeny Kusmenko, Jonas Ritz, Bernhard Rumpe, Armin Moin, Atta Badii, Stephan Günnemann, and Moharram Challenger. MDE for Machine Learning-Enabled Software Systems: A Case Study and Comparison of MontiAnna & ML-Quadrat. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS ’22, pages 380–387, New York, NY, USA, October 2022. ACM.
- [KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.
- [Kle21] Anno Kleiss. Using OCL/P to Improve the Reliability of Model-Driven Internet of Things Applications. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., March 2021.

- [KMM⁺22] Jörg Christian Kirchhof, Lukas Malcher, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. Web-Based Tracing for Model-Driven Applications. In *Proceedings of the 48th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA'22)*. In Press, 2022.
- [KMMN16] Aimad Karkouch, Hajar Mousannif, Hassan Al Moatassime, and Thomas Noel. A model-driven architecture-based data quality management framework for the internet of Things. In *2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, pages 252–259, 2016.
- [KMR20a] Jörg Christian Kirchhof, Judith Michael, and Bernhard Rumpe. *Softwarequalität in Energieprojekten*, pages 273–279. Fraunhofer IRB Verlag, Stuttgart, July 2020.
- [KMR⁺20b] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.
- [KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21)*, pages 197–209. ACM SIGPLAN, October 2021.
- [KNB06] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation. In *17th International Symposium on Software Reliability Engineering*, pages 204–212, 2006.
- [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.
- [KNS⁺21] Jörg Christian Kirchhof, Michael Nieke, Ina Schaefer, David Schmalzing, and Michael Schulze. *Variant and Product Line Co-Evolution*, pages 333–351. Springer, January 2021.

- [Kou16] Anis Koubaa. *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Springer Publishing Company, Incorporated, 1st edition, 2016.
- [KPRR20] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety. In *AIAA Scitech 2020 Forum*. American Institute of Aeronautics and Astronautics, January 2020.
- [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.
- [Kre20] Julian Jérôme Krebber. Generierung von Schnittstellen zwischen Informationssystemen und Cyber-Physischen Systemen zur Entwicklung digitaler Zwillinge. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., April 2020.
- [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)*, volume 2308 of *CEUR Workshop Proceedings*, pages 87–94. CEUR Workshop Proceedings, February 2019.
- [KRRvW18] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc: Textual Modeling Alternative to Simulink. In *Proceedings of MODELS 2018. Workshop EXE*, October 2018.
- [KRSvW18] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. 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.
- [KRSW20] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Structurally Evolving Component-Port-Connector Architectures of Centrally Controlled Systems. In Maxime Cordy, Mathieu

- Acher, Danilo Beuche, and Gunter Saake, editors, *International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, February 2020.
- [KRSW22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, 183:111087, January 2022.
- [KSGW20] Jörg Christian Kirchhof, Martin Serror, René Glebke, and Klaus Wehrle. Improving MAC Protocols for Wireless Industrial Networks via Packet Prioritization and Cooperation. In *Proceedings of the 21st International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoW-MoM). Workshop CCNCPS.*, pages 367–372. IEEE, August 2020.
- [KSJ00] R. Konuru, H. Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed java applications. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 219–227, 2000.
- [Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.
- [Lan] Lantronix, Inc. Product Life Cycle In The Age Of IoT. [Online]. Available: https://cdn.lantronix.com/wp-content/uploads/pdf/Product_Life_Cycle_in_the_Age_of_IoT_Final_04-1.pdf.
- [LCFT17] Xabier Larrucea, Annie Combelles, John Favaro, and Kunal Taneja. Software Engineering for the Internet of Things. *IEEE Software*, 34(1):24–28, Jan 2017.
- [Lis87] Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. Association for Computing Machinery.
- [LLPZ07] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation, NSDI'07*, page 19, USA, 2007. USENIX Association.
- [LSCPE18] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. Microservices. *IEEE Software*, 35(3):96–100, 2018.

BIBLIOGRAPHY

- [Lt04] Marc Lankhorst and the ArchiMate team. ArchiMate Language Primer, Version 1.0. Technical Report TI/RS/2004/024, Telematica Instituut, 2004.
- [Mal21] Lukas Malcher. Reconstructing the Behavior of Cyber-Physical Systems through Digital Shadows and Deterministic Replay in Component & Connector Architectures. Master Thesis. RWTH Aachen University. Software Engineering Group., August 2021.
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008.
- [Mat21] Simulink® User’s Guide, R2021b. Technical report, The MathWorks, Inc., 2021.
- [MECL10] Julien Mercadal, Quentin Enard, Charles Consel, and Nicolas Lorient. A domain-specific approach to architecturing error handling in pervasive computing. *SIGPLAN Not.*, 45(10):47–61, oct 2010.
- [MF19] Brice Morin and Nicolas Ferry. Model-Based, Platform-Independent Logging for Heterogeneous Targets. In *ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 172–182, 2019.
- [MHF17] Brice Morin, Nicolas Harrand, and Franck Fleurey. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software*, 34(1):30–36, January 2017.
- [MLM⁺13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013.
- [MM12] Dejan Munjin and Jean-Henry Morin. Toward Internet of Things Application Markets. In *IEEE International Conference on Green Computing and Communications*, pages 156–162, 2012.
- [MNZC20] Samuel J. Moore, Chris D. Nugent, Shuai Zhang, and Ian Cleland. IoT reliability: a review leading to 5 key research directions. *CCF Transactions on Pervasive Computing and Interaction*, 2(3):147–163, 2020.
- [MRG18] Armin Moin, Stephan Rössler, and Stephan Günnemann. ThingML+ Augmenting Model-Driven Software Engineering for the Internet of Things with Machine Learning. In *Proceedings of MODELS 2018. Workshop MDE4IoT*, pages 521–523. CEUR Workshop Proceedings, October 2018.

- [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.
- [MSPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.
- [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.
- [MVH17] Frédéric Mallet, Eugenio Villar, and Fernando Herrera. MARTE for CPS and CPSoS. *Cyber-Physical System Design from an Architecture Analysis Viewpoint*, pages 81–108, 2017.
- [NCM⁺20] Thiago Nepomuceno, Tiago Carneiro, Paulo Henrique Maia, Muhammad Adnan, Thalysen Nepomuceno, and Alexander Martin. *AutoIoT: A Framework Based on User-Driven MDE for Generating IoT Applications*, pages 719–728. Association for Computing Machinery, New York, NY, USA, 2020.
- [NFE⁺19] Phu Nguyen, Nicolas Ferry, Gencer Erdogan, Hui Song, Stéphane Lavitrotte, Jean-Yves Tigli, and Arnor Solberg. Advances in Deployment and Orchestration Approaches for IoT - A Systematic Review. In *IEEE International Congress on Internet of Things (ICIOT)*, pages 53–60, 2019.
- [NTBG15] Xuan Thang Nguyen, Huu Tam Tran, Harun Baraki, and Kurt Geihs. FRASAD: A framework for model-driven IoT Application Development. In *IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 387–392, December 2015.
- [Obj14] Object Management Group. Object Constraint Language (Version 2.4), February 2014.
- [Obj15] Object Management Group (OMG). Data Distribution Service (DDS), Version 1.4. [Online]. Available: <https://www.omg.org/spec/DDS/1.4/PDF> Last accessed: 21.03.2021, 2015.
- [Obj17] Object Management Group. Unified Modeling Language Specification (Version 2.5.1), December 2017.

- [Obj19] Object Management Group. OMG System Modeling Language Specification (Version 1.6), December 2019.
- [Orl22] Radoslav Orlov. Feature-based Deployment of IoT Applications in Monti-Things. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., May 2022.
- [PA15] Per Persson and Ola Angelsmark. Calvin – Merging Cloud and IoT. *Procedia Computer Science*, 52:210 – 217, 2015. 6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015).
- [PA17] Per Persson and Ola Angelsmark. Kappa: Serverless iot deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing*, WoSC '17, pages 16–21, New York, NY, USA, 2017. Association for Computing Machinery.
- [PBS⁺22] Julian Eduardo Plazas, Sandro Bimonte, Michel Schneider, Christophe de Vault, Pietro Battistoni, Monica Sebillio, and Juan Carlos Corrales. Sense, transform & send for the Internet of Things (STS4IoT): UML profile for data-centric IoT applications. *Data & Knowledge Engineering*, page 101971, 2022.
- [PHPH19] Yusuf Perwej, Kashiful Haq, Firoj Parwej, and Mumdouh M. Mohamed Hassan. The Internet of Things (IoT) and its Application Domains. *International Journal of Computer Applications*, 182(49):36–49, Apr 2019.
- [Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
- [QCG⁺09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [RBF⁺16] Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. Internet of Things Patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2016.
- [RLC⁺20] Gianna Reggio, Maurizio Leotta, Maura Cerioli, Romina Spalazzese, and Fahed Alkhabbas. What are IoT systems for real? An experts' survey on software engineering aspects. *Internet of Things*, 12:100313, 2020.

- [ROL18] Leila Fatmasari Rahman, Tanir Ozcelebi, and Johan Lukkien. Understanding IoT Systems: A Life Cycle Approach. *Procedia Computer Science*, 130:1057–1062, 2018. The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops.
- [RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.
- [RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArc-Automaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA’13)*, pages 10–12. IEEE, 2013.
- [Rui22] Julian Jérôme Ruiz. Model-Driven Digital Twin Plugins: Interfacing Information Systems, IoT Devices, and External Services. Master Thesis. RWTH Aachen University. Software Engineering Group., November 2022.
- [Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.
- [Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.
- [RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.
- [Sas22] Maya Sastges. Integrating Model-Driven IoT Applications with their Environment. Master Thesis. RWTH Aachen University. Software Engineering Group., January 2022.
- [Sch21] Philipp Schneider. Orchestrierung modellgetriebener IoT-Anwendungen durch generierte Informationssysteme. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., September 2021.
- [SDF⁺20] Hui Song, Rustem Dautov, Nicolas Ferry, Arnor Solberg, and Franck Fleurey. Model-Based Fleet Deployment of Edge Computing Applications. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS ’20, pages 132–142, New York, NY, USA, 2020. Association for Computing Machinery.

BIBLIOGRAPHY

- [Sel96] Bran Selic. Tutorial: Real-Time Object-Oriented Modeling (ROOM). In *Proceedings Real-Time Technology and Applications*, pages 214–217, 1996.
- [Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
- [Sel08] Bran Selic. Accounting for platform effects in the design of real-time software using model-based methods. *IBM Systems Journal*, 47(2):309–320, 2008.
- [SGME92] Bran Selic, Garth Gullekson, Jim McGee, and Ian Engelberg. ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, pages 230–240, 1992.
- [Sho04] Jim Shore. Fail Fast. *IEEE Software*, 21(5):21–25, 2004.
- [Sie04] Johannes Siedersleben. *Moderne Softwarearchitektur - Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, 1st edition, 2004.
- [SKS⁺17] Martin Serror, Jörg Christian Kirchhof, Mirko Stoffers, Klaus Wehrle, and James Gross. Code-Transparent Discrete Event Simulation for Time-Accurate Wireless Prototyping. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, pages 161–172, New York, NY, USA, 2017. Association for Computing Machinery.
- [SKS18] Joshua E. Siegel, Sumeet Kumar, and Sanjay E. Sarma. The future internet of things: Secure, efficient, and model-based. *IEEE Internet of Things Journal*, 5(4):2386–2398, 2018.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, Wien, New York, 1973.
- [Sta14] John A. Stankovic. Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, 1(1):3–9, 2014.
- [Sti21] Lukas Stief. Steuerung digitaler Komponenten für Internet-of-Things-Anwendungen mit Hilfe grafischer Benutzeroberflächen. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., September 2021.
- [SYDZ16] Wentao Shang, Yingdi Yu, Ralph Droms, and Lixia Zhang. Challenges in IoT Networking via TCP/IP Architecture. *NDN Technical Report NDN-0038*, 2016.

- [Tan11] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Vrije University, Amsterdam, The Netherlands, 5th edition, 2011.
- [TGPH20] Sergio Trilles, Alberto González-Pérez, and Joaquín Huerta. An IoT Platform Based on Microservices and Serverless Paradigms for Smart Farming Purposes. *Sensors*, 20(8), 2020.
- [TJSW18] Behrang Ashtari Talkhestani, Nasser Jazdi, Wolfgang Schlögl, and Michael Weyrich. A concept in synchronization of virtual production system with real factory based on anchor-point method. *Procedia CIRP*, 67:13–17, 2018.
- [TM17a] Antero Taivalsaari and Tommi Mikkonen. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software*, 34(1):72–80, Jan 2017.
- [TM17b] Antero Taivalsaari and Tommi Mikkonen. Beyond the Next 700 IoT Platforms. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3529–3534, Oct 2017.
- [TM18] Antero Taivalsaari and Tommi Mikkonen. On the development of IoT systems. In *Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 13–19, April 2018.
- [VSID15] Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar. DIANE - Dynamic IoT Application Deployment. In *2015 IEEE International Conference on Mobile Services*, pages 298–305, 2015.
- [vW20] Michael von Wenckstern. *Verification of Structural and Extra Functional Properties in Component and Connector Models for Embedded and Cyber Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 44. Shaker Verlag, March 2020.
- [WMW18] Sabine Wolny, Alexandra Mazak, and Bernhard Wally. An Initial Mapping Study on MDE4IoT. In *Proceedings of MODELS 2018. Workshop MDE4IoT*, pages 524–529. CEUR Workshop Proceedings, October 2018.
- [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.
- [wwwa] docker.com - What is a Container? . [Online]. Available: <https://www.docker.com/resources/what-container>.

BIBLIOGRAPHY

- [wwwb] Eclipse Mita Project Website. [Online]. Available: <https://www.eclipse.org/mita/>. Last accessed: 16.06.2021.
- [www20] Ericsson Mobility Report November 2020. [Online]. Available: <https://www.ericsson.com/4adc87/assets/local/mobility-report/documents/2020/november-2020-ericsson-mobility-report.pdf>, November 2020.
- [XSXH18] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend Cloud to Edge with KubeEdge. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.
- [YMLL17] Emre Yigitoglu, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. Foggy: A framework for continuous automated iot application deployment in fog computing. In *2017 IEEE International Conference on AI Mobile Services (AIMS)*, pages 38–45, 2017.
- [Zam17] Franco Zambonelli. Key Abstractions for IoT-Oriented Software Engineering. *IEEE Software*, 34(1):38–45, 2017.
- [Zha20] Han Zhang. Integration of Deep Learning into Internet of Things Applications. Bachelor Thesis. RWTH Aachen University. Software Engineering Group., September 2020.

Appendix A

Acronyms

AADL	architecture analysis & design language	35
ADL	architecture description language	3
ALF	action language for foundational UML	139
API	application programming interface	138
AST	abstract syntax tree	20
AWS	Amazon Web Services	3
CAN	controller area network	44
C&C	component and connector	3
CD	continuous deployment	114
C2A2	Capture, Communicate, Analyse and Act	32
CD4A	class diagrams for analysis	25
CI	continuous integration	6
CPU	central processing unit	81
CoCo	context condition	22
CLI	command line interface	100
DDC	diagnostic data cache	87
DDS	data distribution service	91
DNS	domain name system	101
DT	digital twin	18
DS	determinism spacing	158
DSL	domain-specific language	4
DTIS	digital twin information system	111
EBNF	extended Backus-Naur form	21
EMA	EmbeddedMontiArc	28

APPENDIX A ACRONYMS

GCP	Google Cloud Platform	3
GPIO	general-purpose input/output	44
GPL	general purpose programming language	46
GUI	graphical user interface	112
HVAC	heating, ventilation, and air conditioning	167
IoT	Internet of Things	3
IP	Internet Protocol	136
JSON	JavaScript object notation	113
NAT	network address translation	106
MDE	model-driven engineering	5
MARTE	modeling and analysis of real-time and embedded systems	67
MAC	media access control	10
MDE	model-driven engineering	5
MDSE	model-driven software engineering	13
MQTT	message queue telemetry transport	42
MUX	multiplexer	
OCL	object constraint language	8
RFID	radio-frequency identification	167
ROS	robot operating system	162
RTE	run-time environment	6
SC	Statechart	59
SI	international system of units	6
TA	technical assumption	31
UML	unified modeling language	60
UPS	uninterruptible power supply	81
VCG	vehicle connectivity gateway	81
WARP	wireless open access research platform	10

Appendix B

Selected Grammars from the MontiVerse

As MontiThings includes many languages from the MontiVerse (sometimes called *Monti-Zoo*), this section gives an overview of selected grammars that are used by MontiThings. The author of this thesis wants to point out that while he was involved in the development of grammars in the MontiVerse, he does *not* claim the grammars in this appendix are his own work. This appendix only gives a small overview over the grammars most important for understanding this thesis. Due to space reasons, a complete reprint of all used MontiVerse grammars is not given here. All MontiVerse projects and their grammars can be found at <https://github.com/monticore>.

B.1 ArcBasis (MontiArc)

```
1 /* (c) https://github.com/MontiCore/monticore */
2
3 /* This is a MontiCore stable grammar.
4  * Adaptations -- if any -- are conservative. */
5
6 /**
7  * This grammar defines the basic structural elements of component & connector
8  * architecture descriptions in form of component models. The grammar contains
9  * definitions for components, ports, and connectors.
10 *
11 * This grammar is part of the MontiArc language definition,
12 * which are organized according to this extension hierarchy:
13 * * ArcBasis.mc4
14 * * -- ComfortableArc.mc4
15 * * -- GenericArc.mc4
16 *
17 * * ArcCore.mc4 composes
18 *   + ArcBasis.mc4 + ComfortableArc.mc4 + GenericArc.mc4
19 *   and builds the core of architectural modelling
20 *   (still without concrete expressions, literals, etc.)
21 *
22 * Furthermore MontiArc.mc4 extends Arc.mc4 to a complete
23 * language, with Expressions, Literals, etc. defined
24 * * ArcCore.mc4
25 * * -- MontiArc.mc4
26 *
```

MCG

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

27  * The grammar relies on basic expressions, literals and types only.
28  * All these are meant to be extended.
29  */
30
31  component grammar ArcBasis extends
32      de.monticore.MCBasics,
33      de.monticore.types.MCBasicTypes,
34      de.monticore.expressions.ExpressionsBasis,
35      de.monticore.symbols.OOSymbols {
36
37      /**
38       * ASTArcElement is the top-level interface for all elements of the component.
39       * A component may contain arbitrary many elements. This interface may be
40       * used as an extension point to enrich components with further elements.
41       */
42      interface ArcElement;
43
44      /**
45       * ASTComponent represents the definition of a component type. A component is
46       * a unit of computation or a data store. The size of a component may scale
47       * from a single procedure to a whole application. A component ist either
48       * atomic or decomposed into subcomponents.
49       *
50       * @attribute name The type name of this component.
51       * @attribute head Defines configuration options and extensions of this component
52       *
53       * @attribute componentInstances List of identifiers used to create instances
54       * of this component type. Only available for inner components.
55       * @attribute body Contains the architectural elements of this component.
56       */
57      symbol scope ComponentType implements ArcElement =
58          key("component") Name
59          head:ComponentHead
60          (ComponentInstance || ",")*
61          body:ComponentBody
62      ;
63
64      /**
65       * ASTComponentHead holds the definitions of generic type parameters that may
66       * be used as prt types in the component, definitions of configuration parameters
67       * that may be used to configure the component, and this component's parent.
68       *
69       * @attribute arcParamers A list of parameters that define the configuration
70       * options of the component.
71       * @attribute parent The type of the parent component.
72       */
73      ComponentHead =
74          ( "(" (ArcParameter || ",")* ")" )?
75          ( "extends" parent:MCType )?
76      ;
77
78      /**
79       * ASTParameter defines the configuration usage interface of the
80       * component type.
81       *
82       * @attribute type The type of the parameter.
83       * @attribute name The identifier of the parameter.
84       * @attribute value Default value used for the parameter if no argument is

```



```

84  * given during instantiation.
85  */
86  ArcParameter implements Variable =
87    MCType Name ("=" default:Expression)?
88  ;
89
90  /**
91   * ASTComponentBody holds the architectural elements of the component.
92   *
93   * @attribute arcElements A list of architectural elements.
94   */
95  ComponentBody = "{" ArcElement* "}" ;
96
97  /**
98   * ASTComponentInterface defines the interface of the component in terms of
99   * in- and outgoing ports.
100  *
101  * @attribute portDeclarations A list of port declarations.
102  */
103  ComponentInterface implements ArcElement =
104    key("port") (PortDeclaration || ",")+ ";"
105  ;
106
107  /**
108   * ASTPortDeclaration declares one or more ports by specifying their
109   * direction and type.
110   *
111   * @attribute portDirection The direction of the port. Can be in- or outgoing.
112   * @attribute type The type of the port.
113   * @attribute ports A list of declared port identifiers.
114   */
115  PortDeclaration =
116    PortDirection MCType (Port || ",")+
117  ;
118
119  /**
120   * ASTPortDirection defines the direction of the port.
121   */
122  interface PortDirection;
123
124  PortDirectionIn implements PortDirection = key("in");
125
126  PortDirectionOut implements PortDirection = key("out");
127
128  /**
129   * ASTPort defines the port identifier and functions as an extension point
130   * for other port identifier kinds.
131   * ASTPort also creates PortSymbols.
132   *
133   * @attribute name The name of the port.
134   */
135  symbol Port = Name;
136
137  /**
138   * ASTArcFieldDeclaration declares one or more component fields by
139   * specifying their type and identifier.
140   *
141   * @attribute type The type of the component field.

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

142  * @attribute fields A list of field identifier.
143  */
144  ArcFieldDeclaration implements ArcElement =
145    MType (ArcField || ",")+ ";";
146
147  /**
148   * ASTArcField defines the field identifier.
149   *
150   * @attribute name The name of the field.
151   * @attribute value The initialization value of the field.
152   */
153  ArcField implements Variable <100> = Name "=" initial:Expression;
154
155  /**
156   * ASTComponentInstantiation holds one or more component instances that are
157   * used in topology spanned by the component. This way the hierarchical
158   * structure of decomposed components is defined.
159   *
160   * @attribute type The type of the component instance.
161   * @attribute componentInstance A list of instantiated components.
162   */
163  ComponentInstantiation implements ArcElement <100> =
164    MType (ComponentInstance || ",")+ ";";
165  ;
166
167  /**
168   * ASTComponentInstance defines the component instance identifier and functions
169   * as an extension point for other identifier kinds of component instances.
170   *
171   * @attribute name The name of the component instance.
172   * @attribute arguments A list of configuration arguments.
173   */
174  symbol ComponentInstance = Name Arguments?;
175
176  /**
177   * ASTConnector connects one source port with one or more target ports.
178   *
179   * @attribute source The qualified identifier of the source port.
180   * @attribute targets A list of the qualified identifiers of the target ports.
181   */
182  Connector implements ArcElement =
183    source:PortAccess "->" target:(PortAccess || ",")+ ";";
184  ;
185
186  /**
187   * ASTPortAccess refers to a port via its qualified identifier.
188   *
189   * @attribute component The name of the component the port belongs to.
190   * @attribute port the name of the port.
191   */
192  PortAccess = (component:Name@ComponentInstance ".")? port:Name@Port ;
193
194  }

```

Listing B.1: ArcBasis grammar. ArcBasis is the basis grammar of MontiArc.

B.2 Class Diagrams

MCG

```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore;
3
4  /* This is a MontiCore stable grammar.
5   * Adaptations -- if any -- are conservative. */
6
7  /**
8   * This is the basis language component for CD4Analysis and CD4Code.
9
10   * It contains class structures with attributes, but omits
11   * interfaces, enums, modifiers, methods.
12
13   * It uses
14   *   * ExpressionsBasis, MCLiteralsBasis (for the Expressions)
15   *   * MCBasicTypes (for the types in CS and AST) and
16   *   * TypesSymbols (for imported/exported TypeSymbols)
17   * as holes that shall be filled in concrete, complete languages
18   * by a variety of available grammars.
19  */
20
21  component grammar CDBasis extends
22    de.monticore.literals.MCLiteralsBasis,
23    de.monticore.expressions.ExpressionsBasis,
24    de.monticore.types.MCBasicTypes,
25    de.monticore.symbols.OOSymbols,
26    de.monticore.UMLStereotype,
27    de.monticore.UMLModifier {
28
29    /* ==== General ==== */
30    /** The artifact header:
31     * import statements, package definition
32     */
33    CDCompilationUnit =
34      MCPackageDeclaration?
35      MCImportStatement*
36      CDTargetImportStatement*
37      CDDefinition;
38
39    /** This import statement allows to explain in the model, which
40     * additional imports are relevant for the target, this includes
41     * e.g. includes generated code.
42     * (beyond the imports that the generator identifies itself).
43     */
44    CDTargetImportStatement = "targetimport" MCQualifiedName ( "." Star:["*"] )? ";";
45
46    /** The class diagram: defines a set of elements
47     */
48    CDDefinition implements Diagram =
49      Modifier "classdiagram" Name "{" CDElement* "}";
50
51    /* ==== CDElement ==== */
52    /** CDElement denotes the basic elements of a class diagram.
53     * This includes Classes, Interfaces, Enums, Associations
54     * and is meant for extension if needed.

```


```
55  */
56  interface CDElement;
57
58  /** CDPackage span a scope which can contain any CDElement.
59      The name of the package is a flat name and can not create
60      a hierarchical package structure.
61  */
62  scope symbol CDPackage implements CDElement =
63    "package" MCQualifiedName "{"
64    CDElement*
65    "}";
66
67  /** CDType are all of the CDElements, which can be used to describe a
68      type for e.g. variables, method arguments, ...
69  */
70  interface symbol CDType extends CDElement, OOType;
71
72  /* Remark:
73      Because people that know Java also know interface implementation,
74      we decide to add "implements" already here
75      and restrict the grammar to classes only through a CoCo.
76      Advantage: better explanation of the error if someone
77      actually uses "implement" with the base grammar .
78      This CoCo restriction will be switched off in the
79      CDInterfaceAndEnum extension.
80  */
81
82  /** This adds the possibility for classes to implement
83      interfaces (by filling the external NT defined there)
84  */
85  CDInterfaceUsage =
86    "implements" interface: (MCObjectType || ",")+;
87
88  /** This adds the possibility for classes to extend
89      other classes or interfaces extend other interfaces
90      (by filling the external NT defined there)
91  */
92  CDExtendUsage =
93    "extends" superclass: (MCObjectType || ",")+;
94
95  /** CDClass defines a class including extensions and its body.
96  */
97  scope CDClass implements CDType =
98    Modifier "class" Name
99    CDExtendUsage?
100   CDInterfaceUsage?
101   ( "{"
102     CDMember*
103     "}"
104     | ";" );
105
106  /* ==== Attributes ==== */
107  /** The CDMember interface is for all possible members of class.
108      * This grammar only uses it for CDAttribute.
109  */
110  interface CDMember;
111
112  /** An attribute has a type, a name and an optional initializing expression.
```

```

113  */
114  CDAttribute implements CDMember, Field =
115      Modifier MCType Name ("=" initial:Expression)? ";" ;
116  }

```

Listing B.2: CDBasis grammar. CDBasis is the basis grammar of class diagram grammars.



```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore;
3
4  /* This is a MontiCore stable grammar.
5   * Adaptations -- if any -- are conservative. */
6
7  /**
8   * This component grammar defines interfaces and enum classes
9   * for CD4Analysis and CD4Code.
10
11   * This includes directions, qualifiers, multiplicities and compositions.
12  */
13
14  component grammar CDInterfaceAndEnum extends de.monticore.CDBasis {
15
16      /** An interface is quite similar to a class:
17       */
18      scope CDInterface implements CDType =
19          Modifier "interface" Name
20          CDExtendUsage?
21          ( "{"
22              CDMember*
23              "}"
24              | ";" );
25
26      /** Enumerations allow to define a list of names that act as constants.
27       * Please note that Java allows attributes for enums:
28       * this possibility will be added in CD4Code, but not yet here.
29       */
30      scope CDEnum implements CDType =
31          Modifier "enum" Name
32          CDInterfaceUsage?
33          ( "{"
34              (CDEnumConstant || ",")* ";"
35              CDMember*
36              "}"
37              | ";" );
38
39      /** Have the enum constants as separate NT. To allow extensions
40       * an enum constant is a static final attribute of the enum with the type
41       * of the enum
42       */
43      CDEnumConstant implements Field = Name;
44  }

```

Listing B.3: CDInterfaceAndEnum grammar. CDInterfaceAndEnum extends CDBasis with interfaces and enums.

```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore;
3
4  /* This is a MontiCore stable grammar.
5   * Adaptations -- if any -- are conservative. */
6
7  /**
8   * This component grammar defines associations for CD4Analysis and CD4Code.
9
10  * This includes directions, qualifiers, multiplicities and compositions.
11  */
12
13  component grammar CDAssociation extends de.monticore.CDBasis {
14
15    /** The AssociationType is a separate NT to allow for extension later
16     */
17    interface CDAssocType;
18
19    /** The list of association types available as variants of CDAssocType:
20     */
21    CDAssocTypeAssoc implements CDAssocType = "association";
22    CDAssocTypeComp implements CDAssocType = "composition";
23
24    /** An association has a name, a left, right part and a direction
25     */
26    scope symbol CDAssociation implements CDElement =
27      Modifier
28      CDAssocType Name?
29      left:CDAssocLeftSide
30      CDAssocDir
31      right:CDAssocRightSide
32      ";";
33
34    /** ==== Association ==== */
35    interface CDAssocDir;
36
37    /** The list of association directions available as variants of
38     * CDAssociationDirection:
39     */
40    CDLeftToRightDir implements CDAssocDir = "->";
41    CDRightToLeftDir implements CDAssocDir = "<-";
42    CDBiDir implements CDAssocDir = "<->";
43    CDUnspecifiedDir implements CDAssocDir = "--";
44
45    splittoken "->", "<-", "<->", "--";
46
47    CDOordered = {noSpace(2,3)}? "{" "ordered" "}";
48
49    /** CDAssociationSide defines the properties of one side of
50     * an association. This construction allows developers to access
51     * both sides of the associations using the same interface,
52     * even though the concrete syntax differs in their order.
53     */
54    interface CDAssocSide =
55      CDOordered? Modifier CDCardinality?
56      MCQualifiedType CDQualifier? CDRole?;

```

MCG

```

57 CDAssocLeftSide implements CDAssocSide =
58   CDOordered? Modifier CDCardinality?
59   MCQualifiedType CDQualifier? CDRole?;
60
61 CDAssocRightSide implements CDAssocSide =
62   CDRole? CDQualifier? MCQualifiedType
63   CDCardinality? Modifier CDOordered?;
64
65 /** A role symbol is a simple name. It is embedded in the association.
66 */
67 symbol CDRole implements CDMember = "(" Name ";";
68
69 /** CDCardinality captures the cardinality of an association side.
70 Associations currently allow for standard cardinalities, but
71 this is extensible. Visitors help to identify the correct cardinality.
72 Cardinality from de.monticore.Cardinality is not used, because we only
73 want the simple variants here
74 */
75 interface CDCardinality;
76 CCardMult implements CDCardinality = "[*]";
77 CCardOne implements CDCardinality = {noSpace(2,3) && _input.LT(2).getText
78   ().equals("1")}? "[" Digits "; // matches "[1]"
79 CCardAtLeastOne implements CDCardinality = {noSpace(2,3,4,5) && _input.LT(2).
80   getText().equals("1") }? "[" Digits "." "." "*" "; // matches "[1..*]"
81 CCardOpt implements CDCardinality = {noSpace(2,3,4,5) && _input.LT(2).
82   getText().equals("0") && _input.LT(5).getText().equals("1") }? "[" Digits "."
83   "." Digits "; // matches "[0..1]"
84
85 // TODO SVA: MC#2548, use token when available: _input.LT(2).getText().equals
86 ("1") = token(2).equals("1")
87
88 splittoken "[*]";
89
90 /** CDQualifier describes the two forms of explicit qualifications:
91 either through an attribute value contained in the value-object
92 or through a type (such as String) , which the value-object doesn't
93 know about.
94 */
95 CDQualifier =
96   "[" byAttributeName:Name@Variable "]"
97   | "[" byType:MCType ";";
98
99 splittoken "[[, ";";
100
101 /** Write a composition in short form inside the composite class:
102 class A { -> (r) B [*]; }
103 transforms to:
104 composition [1] A -> (r) B [*];
105 */
106 CDDirectComposition implements CDMember =
107   "->" CDAssocRightSide ";";
108
109 /* ==== Symbols ==== */
110
111 /** The symbol for CDRole contains all the information of one side of an
112 association.
113 It contains the link to the type and the SymAssociation, which contains the

```

```

109     basic information of an association.
110     */
111     symbolrule CDRole =
112         isDefinitiveNavigable: boolean
113         cardinality: Optional<de.monticore.cdassociation._ast.ASTCDCardinality>
114         field: Optional<de.monticore.symbols.oosymbols._symboltable.FieldSymbol>
115
116         // Defined exactly if a qualifier is given:
117         attributeQualifier: Optional<de.monticore.symbols.basicsymbols._symboltable.
            VariableSymbol>
118         typeQualifier: Optional<de.monticore.types.check.SymTypeExpression>
119
120         // the CDRole symbol only links to a SymAssociation if the other role also has
            a Symbol
121         assoc: Optional<de.monticore.cdassociation._symboltable.SymAssociation>
122         isOrdered: boolean
123         isLeft: boolean
124
125         type: de.monticore.types.check.SymTypeExpression
126         isReadOnly: boolean
127         isPrivate: boolean
128         isProtected: boolean
129         isPublic: boolean
130         isStatic: boolean
131         isFinal: boolean
132     ;
133
134     /** The symbol for CDAssociation is defined for named associations.
        The SymAssociation object provides all relevant
135     information about the association and the roles.
136     */
137
138     symbolrule CDAssociation =
139         assoc: Optional<de.monticore.cdassociation._symboltable.SymAssociation>;
140 }

```

Listing B.4: CDAssociation grammar. CDAssociation defines associations in class diagrams.

```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore;
3
4  /* This is a MontiCore stable grammar.
5  * Adaptations -- if any -- are conservative. */
6
7  /**
8      CD4A is the textual representation to describe UML class diagrams
9      (it uses the UML/P variant).
10     CD4A covers classes, interfaces, inheritance, attributes with types,
11     visibilities, and all kinds of associations and composition,
12     including qualified and ordered associations.
13
14     CD4A focusses on the analysis phase in typical data-driven development
15     projects and is therefore mainly for data modelling.
16     Consequently, it omits method signatures and complex generics.
17
18     This grammar brings together all CD components

```

MCG


```

19  * CDBasis.mc4
20      for the core class structures
21  * CDInterfaceAndEnum.mc4
22      for interfaces and enumerations
23  * CDAssociation.mc4
24      for associations
25
26  It builds on MCCollectionTypes, which allow. e.g. List<int>
27
28  And it includes MCCCommonLiterals, CommonExpressions, BitExpressions
29  to allow a rich set of expressions
30  (e.g. to initialize attributes and enum values.)
31  */
32
33  grammar CD4Analysis extends
34      de.monticore.CDInterfaceAndEnum,
35      de.monticore.CDAssociation,
36      de.monticore.types.MCCollectionTypes,
37      de.monticore.types.MCArrayTypes,
38      de.monticore.literals.MCCCommonLiterals,
39      de.monticore.expressions.BitExpressions,
40      de.monticore.expressions.CommonExpressions {
41
42      start CDCompilationUnit;
43
44      // Certain keywords of the used grammars shall not become
45      // restricted words in other context:
46      nokeyword "targetpackage", "targetimport", "classdiagram";
47      nokeyword "association", "composition", "ordered";
48
49  }

```

Listing B.5: CD4Analysis grammar. CD4Analysis combines CDBasis, CDInterfaceAndEnum, and CDAssociation to define full class diagrams.

B.3 MCCOMMONSTATEMENTS

```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore.statements;
3
4  /* This is a MontiCore stable grammar.
5   * Adaptations -- if any -- are conservative. */
6
7  import de.monticore.statements.*;
8
9  /**
10   * This grammar defines typical statements, such as
11   * method calls (which are actually expressions),
12   * assignment of variables, if, for, while, switch statements, and blocks.
13   *
14   * This embodies a complete structured statement language, however does not
15   * provide return, assert, exceptions, and low-level constructs like break.
16   */

```



APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```
17 * This grammar is part of a hierarchy of statements, namely
18 * * statements/MCStatementsBasis.mc4
19 * * -- statements/MCAssertStatements.mc4
20 * * -- statements/MCVarDeclarationStatements.mc4
21 * * -- -- statements/MCArrayStatements.mc4
22 * * -- -- statements/MCCommonStatements.mc4
23 * * -- -- -- statements/MCExceptionStatements.mc4
24 * * -- -- -- statements/MCSynchronizedStatements.mc4
25 * * -- statements/MCLowLevelStatements.mc4
26 * * -- statements/MCReturnStatements.mc4
27 *
28 * and the composition of all statement grammars to full Java:
29 * * -- -- statements/MCFullJavaStatements.mc4
30 *
31 */
32
33 component grammar MCCommonStatements
34     extends MCVarDeclarationStatements {
35
36 /**
37  * Standard Form of a block { ... }
38  * it allows to define local variables that are not exported
39  * and can only be used after defined (typical within code bodies).
40  */
41 scope (non_exporting ordered) MCJavaBlock implements MCStatement
42     = "{" MCBlockStatement* "}" ;
43
44 /**
45  * All the Java Modifier
46  */
47 JavaModifier implements MCM Modifier =
48     Modifier:["private" | "public" | "protected" | "static"
49             | "transient" | "final" | "abstract" | "native"
50             | "threadsafe" | "synchronized" | "const" | "volatile"
51             | "strictfp"] ;
52
53 IfStatement implements MCStatement
54     = "if" "(" condition:Expression ")"
55       thenStatement:MCStatement
56       ("else" elseStatement:MCStatement)? ;
57 // we use "elseStatement", because the
58 // generated Java code doesn't allow "else" as Name
59
60
61 scope (non_exporting ordered) ForStatement implements MCStatement
62     = "for" "(" ForControl ")" MCStatement ;
63
64 interface ForControl ;
65
66 CommonForControl implements ForControl
67     = ForInit? ";" condition:Expression? ";" (Expression || ",")* ;
68
69 ForInit
70     = ForInitByExpressions | LocalVariableDeclaration ;
71
72 ForInitByExpressions
73     = (Expression || ",")+ ;
74
```

```

75 EnhancedForControl implements ForControl
76     = FormalParameter ":" Expression;
77
78 FormalParameter
79     = JavaModifier* MCType DeclaratorId;
80
81 WhileStatement implements MCStatement
82     = "while" "(" condition:Expression ")" MCStatement ;
83
84 DoWhileStatement implements MCStatement
85     = "do" MCStatement "while" "(" condition:Expression ")" ";" ;
86
87 SwitchStatement implements MCStatement
88     = "switch" "(" Expression ")"
89       "{" SwitchBlockStatementGroup* SwitchLabel* "}" ;
90
91 EmptyStatement implements MCStatement
92     = ";" ;
93
94 ExpressionStatement implements MCStatement
95     = Expression ";" ;
96
97 // Matches cases then statements, both of which are mandatory.
98 // To handle empty cases at the end, SwitchLabel* is explicitly added
99 // in the statement body
100 SwitchBlockStatementGroup
101     = SwitchLabel+ MCBLOCKStatement+ ;
102
103 interface SwitchLabel ;
104
105 ConstantExpressionSwitchLabel implements SwitchLabel
106     = "case" constant:Expression ":" ;
107
108 EnumConstantSwitchLabel implements SwitchLabel
109     = "case" enumConstant:Name ":" ;
110
111 DefaultSwitchLabel implements SwitchLabel
112     = "default" ":" ;
113
114 BreakStatement implements MCStatement
115     = "break" ";" ;
116
117 }

```

Listing B.6: MCCCommonStatements grammar. MCCCommonStatements is the basis MontiThings' Java-like behavior descriptions.

B.4 MCCCommonLiterals

```

1 /* (c) https://github.com/MontiCore/monticore */
2 package de.monticore.literals;
3
4 /* This is a MontiCore stable grammar.

```

MCG

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```
5  * Adaptations -- if any -- are conservative. */
6
7  import de.monticore.literals.*;
8
9  /**
10 * This grammar defines Java compliant literals.
11 * The scope of this grammar is to
12 * ease the reuse of literals structures in Java-like sublanguages, e.g., by
13 * grammar inheritance or grammar embedment.
14 * The grammar contains literals from Java, e.g., Boolean, Char, String, ....
15 */
16
17 component grammar MCommonLiterals
18     extends de.monticore.MCBasics,
19             MCLiteralsBasis {
20
21
22     /*=====*/
23     /*===== INTERFACE DEFINITIONS =====*/
24     /*=====*/
25
26
27     /** ASTSignedLiteral is the interface for all literals (NullLiteral,
28         BooleanLiteral, CharLiteral, StringLiteral and all NumericLiterals).
29         Compared to Literal it also includes negative NumericLiterals
30     */
31     interface SignedLiteral;
32
33
34     /** The interface ASTNumericLiteral combines the numeric literal types for
35         Integer, Long, Float and Double without '-' at the beginning)
36     */
37     interface NumericLiteral extends Literal <100>;
38
39
40     /** The interface ASTNumericLiteral combines the numeric literal types for
41         Integer, Long, Float and Double.
42         Compared to NumericLiteral it also includes negative numbers.
43     */
44     interface SignedNumericLiteral extends SignedLiteral <100>;
45
46
47     /*=====*/
48     /*===== PARSER RULES =====*/
49     /*=====*/
50
51     /** ASTNullLiteral represents 'null'
52     */
53     NullLiteral implements Literal, SignedLiteral =
54         "null";
55
56
57     /** ASTBooleanLiteral represents "true" or "false"
58         @attribute source String-representation (including '"').
59     */
60     BooleanLiteral implements Literal, SignedLiteral =
61         source:["true" | "false"];
62
```

```

63
64  /** ASTCharLiteral represents any valid character parenthesized with "'".
65      @attribute source String-representation (including "'").
66  */
67  CharLiteral implements Literal, SignedLiteral =
68    source:Char;
69
70
71  /** ASTStringLiteral represents any valid character sequence parenthesized
72      with "'".
73      @attribute source String-representation (including "'").
74  */
75  StringLiteral implements Literal, SignedLiteral =
76    source:String;
77
78
79  /** ASTNatLiteral represents a positive Decimal number.
80      @attribute source String-representation (including "'").
81  */
82  NatLiteral implements NumericLiteral<1> =
83    Digits;
84
85  /** ASTSignedNatLiteral represents a positive or negative Decimal number.
86      @attribute source String-representation (including "'").
87  */
88  SignedNatLiteral implements SignedNumericLiteral<1> =
89    {noSpace(2)}? (negative:["-"]) Digits |
90    Digits;
91
92  /** ASTLongLiteral represents a positive Decimal number.
93      @attribute source String-representation (including "'").
94  */
95  BasicLongLiteral implements NumericLiteral<1> =
96    { cmpToken(2,"1","L") && noSpace(2) }? Digits key("1" | "L");
97
98  /** ASTSignedLongLiteral represents a positive or negative Decimal number.
99      @attribute source String-representation (including "'").
100 */
101 SignedBasicLongLiteral implements SignedNumericLiteral<1> =
102 { cmpToken(3,"1","L") && noSpace(2,3) }?
103   negative:["-"] Digits key("1" | "L")
104   |
105   { cmpToken(2,"1","L") && noSpace(2) }?
106     Digits key("1" | "L");
107
108 /** ASTFloatLiteral represents a positive float.
109     @attribute source String-representation (including "'").
110 */
111 BasicFloatLiteral implements NumericLiteral<1> =
112 { cmpToken(4,"f","F") && noSpace(2,3,4) }?
113   pre:Digits "." post:Digits key("f" | "F");
114
115 /** ASTSignedFloatLiteral represents a positive or negative float.
116     @attribute source String-representation (including "'").
117 */
118 SignedBasicFloatLiteral implements SignedNumericLiteral<1> =
119 { cmpToken(5,"f","F") && noSpace(2,3,4,5) }?
120   negative:["-"] pre:Digits "." post:Digits key("f" | "F") |

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```
121     { cmpToken(4,"f","F") && noSpace(2,3,4) }?
122     pre:Digits "." post:Digits key("f" | "F");
123
124     /** ASTDoubleLiteral represents a positive double.
125     @attribute source String-representation (including '').
126     */
127     BasicDoubleLiteral implements NumericLiteral<1> =
128     { noSpace(2,3) }? pre:Digits "." post:Digits;
129
130     /** ASTSignedDoubleLiteral represents a positive or negative double.
131     @attribute source String-representation (including '').
132     */
133     SignedBasicDoubleLiteral implements SignedNumericLiteral<1> =
134     { noSpace(2,3,4) }? negative:["-"] pre:Digits "." post:Digits |
135     { noSpace(2,3) }? pre:Digits "." post:Digits;
136
137     /*=====*/
138     /*===== LEXER RULES =====*/
139     /*=====*/
140
141
142     /*=====*/
143     /* The following section is adapted from */
144     /* https://github.com/antlr/grammars-v4/blob/master/java/Java.g4 */
145     /*=====*/
146
147     // §3.10.1 Integer Literals
148
149     token Digits
150     = Digit+;
151
152     fragment token Digit
153     = '0'..'9';
154
155     // §3.10.4 Character Literals
156     token Char
157     = '\'' (SingleCharacter|EscapeSequence) '\''
158     : {setText(getText().substring(1, getText().length() - 1));};
159
160     fragment token SingleCharacter
161     = ~ ('\'' );
162
163
164     // §3.10.5 String Literals
165     token String
166     = '"' (StringCharacters)? '"'
167     : {setText(getText().substring(1, getText().length() - 1));};
168
169     fragment token StringCharacters
170     = (StringCharacter)+;
171
172     fragment token StringCharacter
173     = ~ ('"' | '\'' | EscapeSequence);
174
175
176     // §3.10.6 Escape Sequences for Character and String Literals
177     fragment token EscapeSequence
178     = '\\\' ('b' | 't' | 'n' | 'f' | 'r' | '"' | '\'' | '\\\' )
```

```

179         | OctalEscape | UnicodeEscape;
180
181     fragment token OctalEscape
182         = '\\\ OctalDigit | '\\\ OctalDigit OctalDigit
183         | '\\\ ZeroToThree OctalDigit OctalDigit;
184
185     fragment token UnicodeEscape
186         = '\\\ 'u' HexDigit HexDigit HexDigit HexDigit;
187
188     fragment token ZeroToThree
189         = '0'..'3' ;
190
191     fragment token HexDigit
192         = '0'..'9' | 'a'..'f' | 'A'..'F' ;
193
194     fragment token OctalDigit
195         = '0'..'7' ;
196
197
198     /*=====*/
199     /*===== AST DEFINITIONS =====*/
200     /*=====*/
201
202     astrule BooleanLiteral =
203         method public boolean getValue() {
204             return this.source == ASTConstantsMCCCommonLiterals.TRUE;
205         }
206     ;
207
208     astrule CharLiteral =
209         method public char getValue() {
210             return de.monticore.literals.MCLiteralsDecoder.decodeChar(
211                 getSource());
212         }
213     ;
214
215     astrule StringLiteral =
216         method public String getValue() {
217             return de.monticore.literals.MCLiteralsDecoder.decodeString(
218                 getSource());
219         }
220     ;
221
222     astrule NatLiteral =
223         method public String getSource() {
224             return getDigits();
225         }
226         method public int getValue() {
227             return de.monticore.literals.MCLiteralsDecoder.decodeNat(
228                 getSource());
229         }
230     ;
231
232     astrule SignedNatLiteral =
233         method public String getSource() {
234             return (negative?"-":"" ) + getDigits();
235         }
236         method public int getValue() {

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```
237     return de.monticore.literals.MCLiteralsDecoder.decodeNat(  
238         getSource());  
239     }  
240 ;  
241  
242 astrule BasicLongLiteral =  
243     method public String getSource() {  
244         return getDigits() + "L";  
245     }  
246     method public long getValue() {  
247         return de.monticore.literals.MCLiteralsDecoder.decodeLong(getSource());  
248     }  
249 ;  
250  
251 astrule SignedBasicLongLiteral =  
252     method public String getSource() {  
253         return (negative?"-":"" ) + getDigits() + "L";  
254     }  
255     method public long getValue() {  
256         return de.monticore.literals.MCLiteralsDecoder.decodeLong(getSource());  
257     }  
258 ;  
259  
260 astrule BasicFloatLiteral =  
261     method public String getSource() {  
262         return getPre() + "." + getPost() + "F";  
263     }  
264     method public float getValue() {  
265         return de.monticore.literals.MCLiteralsDecoder.decodeFloat(getSource());  
266     }  
267 ;  
268  
269 astrule SignedBasicFloatLiteral =  
270     method public String getSource() {  
271         return (isNegative?"-":"" ) + getPre() + "." + getPost() + "F";  
272     }  
273     method public float getValue() {  
274         return de.monticore.literals.MCLiteralsDecoder.decodeFloat(getSource());  
275     }  
276 ;  
277  
278 astrule BasicDoubleLiteral =  
279     method public String getSource() {  
280         return getPre() + "." + getPost();  
281     }  
282     method public double getValue() {  
283         return de.monticore.literals.MCLiteralsDecoder.decodeDouble(getSource());  
284     }  
285 ;  
286  
287 astrule SignedBasicDoubleLiteral =  
288     method public String getSource() {  
289         return (isNegative?"-":"" ) + getPre() + "." + getPost();  
290     }  
291     method public double getValue() {  
292         return de.monticore.literals.MCLiteralsDecoder.decodeDouble(getSource());  
293     }  
294 ;
```


295
296 }

Listing B.7: MCCCommonLiterals grammar. MCCCommonLiterals provides the basis of MontiThings' literals.

B.5 OCL Expressions

```

1 // (c) https://github.com/MontiCore/monticore
2 package de.monticore.ocl;
3
4 /* This is a MontiCore stable grammar.
5  * Adaptations -- if any -- are conservative. */
6
7 import de.monticore.expressions.*;
8 import de.monticore.types.*;
9 import de.monticore.symbols.*;
10
11 /**
12  * This grammar defines expressions typical to UMLs OCL
13  *
14  * This includes among others the
15  *   * typeif, forall and exists quantifiers,
16  *   * set selection with any, iteration,
17  *   * @pre and transitive closure **
18  *
19  * OCL expressions can safely (i.e. as conservative extension)
20  * be composed if with other forms of expressions
21  * given in the MontiCore core project.
22  * Especially common expressions should be added.
23  *
24  * This grammar is part of a hierarchy of expressions, namely
25  * * expressions/ExpressionsBasis.mc4
26  * * -- expressions/CommonExpressions.mc4
27  * * -- -- expressions/JavaClassExpressions.mc4
28  * * -- expressions/AssignmentExpressions.mc4
29  * * -- expressions/BitExpressions.mc4
30  * * -- ocl/OCLExpressions.mc4
31  * * -- ocl/SetExpressions.mc4
32  * * -- ocl/OptionalOperators.mc4
33  *
34  * Care: other grammars may include a syntactically similar casting
35  * function. The parser then only takes one alternative (dependend
36  * on the priority (which is here <200>)).
37  *
38  * Note that it may be useful to use the "nokeyword"-keyword statement
39  * for a variety of keywords used here, such as "implies", "forall", "in"
40  */
41
42 component grammar OCLExpressions
43     extends ExpressionsBasis,
44             MCBasicTypes,
45             BasicSymbols

```

MCG

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

46 {
47   /**
48    * ASTTypeCastExpression casts an expression to a given type
49    *   @attribute MCType
50    *       type to cast the expression to
51    *   @attribute Expression
52    *       the expression that should be casted
53    */
54   TypeCastExpression implements Expression <200> =
55     "(" MCType ")" Expression;
56
57   /**
58    * ASTOCLVariableDeclaration defines a variable
59    *   @attribute MCType
60    *       type of the variable
61    *   @attribute Name
62    *       name of the variable
63    *   @attribute Expression
64    *       initial value of the variable
65    */
66   OCLVariableDeclaration implements Variable =
67     MCType? Name (dim:[" "]*) ("=" Expression)?;
68
69   /*=====*/
70
71   /**
72    * ASTTypeIfExpression
73    *   Type-safe version of type-cast for variables.
74    *   typeof m instanceof Subtype
75    *   then (m known here as Subtype)
76    *   else (m here as only Supertype)
77    *
78    *   @attribute Name
79    *       Name of a variable of which the type should be checked
80    *   @attribute MCType
81    *       The type to which the variable should be compared
82    *   @attribute thenExpression
83    *       resulting expression in which the variable can be used with
84    *       the type defined by MCType (if the type check returns true)
85    *   @attribute elseExpression
86    *       resulting expression which will be evaluated if the variable
87    *       is not of the type defined by MCType
88    *
89    *   Example:
90    *   typeof bm instanceof BidMesssage
91    *   then    bm.auction==copper912
92    *   else    false
93    */
94   TypeIfExpression implements Expression <100> =
95     "typeof" Name@Variable "instanceof" MCType
96     "then"   thenExpression:Expression
97     "else"   elseExpression:Expression
98     ;
99
100  /**
101   * IfThenElseExpression defines a case distinction operator.
102   *   If the condition is true, thenExpression will be returned,
103   *   otherwise the elseExpression will be returned.

```

```

104 *
105 * @attribute condition
106 * the condition to be evaluated
107 * @attribute thenExpression
108 * the expression to return if the condition is true
109 * @attribute elseExpression
110 * the expression to return if the condition is false
111 */
112 IfThenElseExpression implements Expression <100> =
113     "if" condition:Expression
114     "then" thenExpression:Expression
115     "else" elseExpression:Expression
116 ;
117
118
119 /*=====*/
120
121 /**
122  * ASTImpliesExpression defines a logical implies operator.
123  * Example: a.startTime >= Time.now() implies a.numberOfBids == 0
124  */
125 ImpliesExpression implements Expression <116> =
126     left:Expression
127     "implies"
128     right:Expression
129 ;
130
131 /**
132  * ASTEquivalentExpression defines a logical equals operator.
133  * Example: sa.equals(sb) <=> sa==sb
134  */
135 EquivalentExpression implements Expression <115> =
136     left:Expression operator:"<=>" right:Expression;
137
138
139 /*=====*/
140
141 /**
142  * ASTForAllExpression defines a quantified expression for collections e.g.
143  * "forall x in Y : ...".
144  * @attribute InDeclaration
145  * List of collection variable declarations, e.g:
146  * "forall a in A: ..."
147  * "forall a in List <..> : ..."
148  * "forall a: ..."
149  * @attribute OCLExpression
150  * The body of forall iteration as an expression.
151  */
152 scope (non_exporting) ForallExpression implements Expression <90> =
153     "forall"
154     (InDeclaration || ",")+
155     ":"
156     Expression
157 ;
158
159 /**
160  * ASTExistsExpression defines a quantified expression for collections e.g.
161  * "exists x in Y : ...".

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

162 * @attribute InDeclaration
163 * List of collection variable declarations, e.g:
164 * "exists a in A: ..."
165 * "exists a in List <..> : ..."
166 * "exists a: ..."
167 * @attribute OCLExpression
168 * The body of exists iteration as an expression.
169 */
170 scope (non_exporting) ExistsExpression implements Expression <90> =
171     "exists"
172     (InDeclaration || ",")+
173     ":"
174     Expression
175 ;
176
177 /**
178 * ASTOCLAnyExpression selects an element from a non-empty collection e.g.
179 * any x in set or any Auction. The result is underspecified.
180 * @attribute OCLExpression
181 * A collection defined by an expression.
182 */
183 AnyExpression implements Expression <100> =
184     "any" Expression;
185
186
187 /*=====*/
188
189 /**
190 * ASTLetinExpression are used to define local vars or methods. The defined
191 * vars and methods are visible in the in-expression body.
192 * @attribute letDeclaration
193 * A list of variable declarations.
194 * @attribute expression
195 * An expression where previous declarations are used.
196 */
197 scope (non_exporting) LetinExpression implements Expression <100> =
198     "let" (OCLVariableDeclaration || ";")+
199     "in" Expression ;
200
201 /**
202 * ASTIterateExpression is used to iterate collections. It differs from
203 * Java5-Iterator.
204 * Example:
205 * iterate{ elem in Auction; int acc=0 : acc = acc+elem.numberOfBids }.
206 * @attribute iterationDeclarator
207 * The elements of a collection that will be iterated as an
208 * OCLCollectionVarDeclaration.
209 * @attribute init
210 * Definiton of a accumulation variable as an
211 * OCLVariableDeclaration.
212 * @attribute Name
213 * Name of the accumulation assignment variable. This has to be
214 * the variable introduced by init:OCLVariableDeclaration
215 * @attribute value
216 * Right hand of the accumulation as an expression.
217 */
218 scope (non_exporting) IterateExpression implements Expression <100> =
219     "iterate" "{"

```

```

220     iteration:InDeclaration ","
221     init:OCLVariableDeclaration ":"
222     Name@Variable "=" value:Expression
223     "};
224
225     /*=====*/
226
227     /**
228     * ASTInDeclaration defines a collection like "int x in y" or "Auction a" as
229     * shortform of "Auction a in Auction.allInstances".
230     */
231     InDeclaration =
232         MCType (InDeclarationVariable || ",")+
233         | MCType? (InDeclarationVariable || ",")+ ("in" Expression)
234         ;
235
236     /**
237     * ASTInDeclarationVariable defines the name of the variable
238     * used in the InDeclaration nonterminal
239     * ASTInDeclarationVariable also creates VariableSymbols.
240     *
241     * @attribute Name
242     *         name of the variable.
243     *
244     * Variable defines an according symbol.
245     */
246     InDeclarationVariable implements Variable = Name;
247
248     /*=====*/
249
250     /**
251     * ASTInstanceOfExpression checks if an expression has a certain type
252     * evaluates to true if the expression has the type given by MCType
253     * otherwise evaluates to false
254     *
255     * @attribute Expression
256     *         expression whose type is to be checked
257     * @attribute MCType
258     *         type against which the expression should be checked
259     */
260     InstanceOfExpression implements Expression <150> =
261         Expression "instanceof" MCType;
262
263     /*=====*/
264
265     /**
266     * ASTOCLArrayQualification is used to access elements of
267     * an array of elements.
268     *
269     * @attribute Expression
270     *         The expression whose elements will be accessed
271     * @attribute arguments
272     *         The expression defining which element to access
273     */
274     OCLArrayQualification implements Expression <250> =
275         Expression ("[" arguments:Expression "]" );
276
277     /**

```

```

278 * ASTOCLAtPreQualification
279 *   Value of the expression in the precondition
280 *   Example: post: messageList == messageList@pre.add(m)
281 */
282 OCLAtPreQualification implements Expression <400> =
283   Expression atpre:["@pre"];
284
285 /**
286 * ASTOCLTransitiveQualification
287 *   Transitive closure of an association. The operator ** is
288 *   only directly applied to a reflexive association. It
289 *   cannot be applied on chains of associations of the form (a.b.c)**
290 *   The transitive closure of an association is also calculated if the
291 *   association's source and target are not identical or even if they
292 *   are not subclasses of each other. In that case, the transitive
293 *   closure is identical to the initial association.
294 *   Example: this.clique = this.friend**
295 */
296 OCLTransitiveQualification implements Expression <400> =
297   Expression transitive:["**"];
298
299 }

```

Listing B.8: OCLExpressions grammar. OCLExpressions are imported by MontiThings.

B.6 Set Expressions

```

1 // (c) https://github.com/MontiCore/monticore
2 package de.monticore.ocl;
3
4 /* This is a MontiCore stable grammar.
5 * Adaptations -- if any -- are conservative. */
6
7 import de.monticore.expressions.*;
8 import de.monticore.types.*;
9 import de.monticore.symbols.*;
10
11 /**
12 * This grammar defines set expressions, such as
13 * union, intersect, setand, setor and set comprehension.
14 *
15 * Set expressions can safely (i.e. as conservative extension)
16 * be composed if with other forms of expressions
17 * given in the MontiCore core project.
18 * Especially common expressions should be added.
19 *
20 * This grammar is part of a hierarchy of expressions, namely
21 * * expressions/ExpressionsBasis.mc4
22 * * -- expressions/CommonExpressions.mc4
23 * * -- -- expressions/JavaClassExpressions.mc4
24 * * -- expressions/AssignmentExpressions.mc4
25 * * -- expressions/BitExpressions.mc4
26 * * -- ocl/OCLExpressions.mc4

```

MCG

```

27 * * -- ocl/SetExpressions.mc4
28 * * -- ocl/OptionalOperators.mc4
29 *
30 */
31
32 component grammar SetExpressions
33     extends ExpressionsBasis,
34             MCBasicTypes,
35             BasicSymbols
36 {
37     /*=====*/
38
39     SetInExpression implements Expression <150> =
40         elem:Expression
41         operator:"isin"
42         set:Expression;
43
44     SetNotInExpression implements Expression <150> =
45         elem:Expression
46         operator:"notin"
47         set:Expression;
48
49     /*=====*/
50
51     UnionExpression implements Expression <180> =
52         left:Expression
53         operator:"union"
54         right:Expression;
55
56     IntersectionExpression implements Expression <180> =
57         left:Expression
58         operator:"intersect"
59         right:Expression;
60
61     SetMinusExpression implements Expression <180> =
62         left:Expression
63         operator:"\\"
64         right:Expression;
65
66     /*=====*/
67
68     // sets of sets united (i.e. flattened) and intersected
69     SetUnionExpression implements Expression <170> =
70         "union" set:Expression;
71
72     SetIntersectionExpression implements Expression <170> =
73         "intersect" set:Expression;
74
75     /*=====*/
76
77     // Logical expressions extended to sets of arguments
78     SetAndExpression implements Expression <130> =
79         "setand" set:Expression;
80
81     SetOrExpression implements Expression <130> =
82         "setor" set:Expression;
83
84     /*=====*/

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

85
86 SetVariableDeclaration implements Variable =
87   MCType? Name (dim:"[" "]"")* ("=" Expression)?;
88
89 /*=====*/
90
91 /**
92  * ASTSetComprehension defines a comprehension with given
93  * characteristic.
94  * @attribute MCType
95  * Optional type of comprehension, e.g. Set, List or Collection.
96  * @attribute left
97  * A comprehension-item (e.g. "x*x" or "x in Y") that describes
98  * or introduces the elements stored in the set.
99  * @attribute setComprehensionItems
100  * Characterization of comprehension as a list of
101  * comprehension-items. This can be generators, vardefinitions
102  * or filters.
103  * Example:
104  * {x * x | x in y, x < 10}
105  * Note that we assume at least one generator (e.g. x in Y) in this AST.
106  */
107 scope (non_exporting) SetComprehension implements Expression <40> =
108   MCType?
109   "{" left:SetComprehensionItem "|"
110   (SetComprehensionItem || ",")+ "}"
111   ;
112
113 /**
114  * ASTSetComprehensionItem defines the items that can occur
115  * on the right hand side of a comprehension.
116  * This can be
117  * Boolean expressions that act as filter, e.g. x < 6
118  * introductions on new local variables that act as
119  * intermediate result, e.g. int y = 2*x
120  * and generators that introduce a new variable and let them
121  * range over a set of values, e.g. x in S,
122  * y in {3..10}, z in Set{3,5,10..20}
123  */
124 SetComprehensionItem =
125   Expression |
126   SetVariableDeclaration |
127   GeneratorDeclaration
128   ;
129
130 /**
131  * ASTGeneratorDeclaration defines a generator that introduces a new
132  * variable and lets it range over a set
133  * @attribute MCType
134  * Optional type of variable
135  * @attribute Name
136  * Name of the variable
137  * @attribute Expression
138  * Expression that describes or references a set
139  */
140 GeneratorDeclaration implements Variable =
141   MCType? Name "in" Expression;
142

```



```

143  /**
144   * ASTSetEnumeration is used for an enumeration of
145   * comprehension elements. Note that collection items are optional.
146   * @attribute MCType
147   * Optional type of comprehension, e.g. Set, List or Collection.
148   * @attribute setCollectionItems
149   * Enumerated elements as a list separated by , (comma).
150   * (e.g.: "1..3, x, y..z")
151   * Example:
152   * {1 .. 3, x+1 .. 10, 2*y, 21}
153   */
154  SetEnumeration implements Expression <40> =
155      MCType?
156      "{" (SetCollectionItem || ",")* "}"
157      ;
158
159  interface SetCollectionItem;
160
161  // list of allowed values
162  SetValueItem implements SetCollectionItem =
163      (Expression || ",")+
164      ;
165
166  // range of allowed values
167  SetValueRange implements SetCollectionItem =
168      lowerBound:Expression ".." upperBound:Expression
169      ;
170 }

```

Listing B.9: SetExpressions grammar. SetExpressions are imported by MontiThings.

B.7 SI Units

```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore;
3
4  /* Beta-version: This is intended to become a MontiCore stable grammar. */
5
6  /**
7   * This grammar defines SI units and other derived units such as
8   * 'm', 'km', 'km^2' or 'mm^2/kVA^2h'. Spaces in a unit are
9   * prevented by semantic predicates.
10  *
11  * The definitions are fully compliant to the definitions given in
12  * International Bureau of Weights and Measures (20 May 2019),
13  * SI Brochure: The International System of Units (SI) (9th ed.).
14  *
15  * SI units are declared as independent Nonterminal and can then
16  * be used as part of a
17  * * value definition, such as "5kg", or
18  * * type definition, such as "km/h"
19  *
20  * The grammar extends the MontiCore common literals, because it uses

```



```

21  * natural numbers e.g. as exponent.
22  */
23
24  grammar SIUnits extends de.monticore.literals.MCCommonLiterals {
25
26
27    /**
28     * The SIUnit is either a SIUnitPrimitive, e.g. "km"
29     * the division of two SIUnitPrimitives, e.g. "km/h"
30     * or the division of "1" and a SIUnitPrimitive, e.g. "1/h"
31     *
32     */
33    SIUnit =
34      // The lookahead is needed for the parser to
35      // decide which alternative to take.
36      { isSIOneDiv() }?
37      one:NatLiteral "/" denominator:SIUnitPrimitive |
38      { isSIDiv() }?
39      numerator:SIUnitPrimitive "/" denominator:SIUnitPrimitive |
40      { !isSIDiv() }?
41      SIUnitPrimitive;
42
43
44    /**
45     * The SIUnitPrimitives are the primitives of the SIUnit
46     *
47     * SIUnitPrimitives contains the basic SI units without prefixes
48     * such as 'm', 's' or 'kg'
49     * as well as the basic SI units with prefixes
50     * such as 'km', 'mm' or 'ms'.
51     *
52     * Other derived or officially accepted units are also contained
53     * ('h', 'day', 'Ohm', ...)
54     *
55     * Do not take a SIUnitWith(out)Prefix if it is in fact a
56     * SIUnitKindGroupWithExponent.
57     */
58    SIUnitPrimitive =
59      { !isSIUnitKindGroupWithExponent(1) }? SIUnitWithPrefix |
60      { !isSIUnitKindGroupWithExponent(1) }? SIUnitWithoutPrefix |
61      SIUnitDimensionless |
62      CelsiusFahrenheit |
63      { isSIUnitKindGroupWithExponent(1) }? SIUnitKindGroupWithExponent;
64
65
66    /**
67     * SIUnitWithPrefix
68     *
69     * The regular expression is defined according to:
70     * * https://en.wikipedia.org/wiki/Metric\_prefix
71     * * https://en.wikipedia.org/wiki/SI\_base\_unit
72     * * https://en.wikipedia.org/wiki/SI\_derived\_unit
73     * * https://en.wikipedia.org/wiki/Non-SI\_units\_mentioned\_in\_the\_SI
74     *
75     * The expression matches an SI unit starting
76     * with a prefix. An SI unit that can have a
77     * prefix is one of the following:
78     * 'm, g, s, A, K, mol, cd, Hz, N, Pa, J, W, C, V, F, Ohm,

```

```

79  *   Ω, S, Wb, T, H, lm, lx, Bq, Gy, Sv, kat, l, L'
80  *
81  * Alternatively the SIUnitWithPrefix is
82  * followed by any other SIUnitWithPrefix or
83  * SIUnitWithoutPrefix (see below) for a
84  * SI unit group, e.g. 'kVAh'.
85  *
86  * The regular expression is needed, because SI
87  * units shall not be defined as keywords
88  * because they would not be usable e.g. as
89  * variable names in other places anymore.
90  * See also functions available to handle the
91  * stored unit.
92  */
93  SIUnitWithPrefix =
94      { isSIUnitWithPrefix(1) }? (Name | NonNameUnit);
95
96
97  /**
98   * SIUnitWithoutPrefix
99   *
100  * The regular expression is defined according to:
101  * * https://en.wikipedia.org/wiki/Metric\_prefix
102  * * https://en.wikipedia.org/wiki/SI\_base\_unit
103  * * https://en.wikipedia.org/wiki/SI\_derived\_unit
104  * * https://en.wikipedia.org/wiki/Non-SI\_units\_mentioned\_in\_the\_SI
105  *
106  * The expression matches an SI unit not starting
107  * with a prefix. An SI unit that does not need a
108  * prefix is one of the following:
109  * 'm, g, s, A, K, mol, cd, Hz, N, Pa, J, W, C, V, F, Ohm,
110  *   Ω, S, Wb, T, H, lm, lx, Bq, Gy, Sv, kat, l, L'
111  * and
112  * 'min, h, d, ha, t, au, Np, B, dB, eV, Da, u'
113  *
114  * Alternatively the SIUnitWithPrefix is
115  * followed by any other SIUnitWithPrefix (see
116  * above) or SIUnitWithoutPrefix for a
117  * SI unit group, e.g. 'VAh'.
118  *
119  * The regular expression is needed, because SI
120  * units shall not be defined as keywords
121  * because they would not be usable e.g. as
122  * variable names in other places anymore.
123  * See also functions available to handle the
124  * stored unit.
125  */
126  SIUnitWithoutPrefix =
127      { isSIUnitWithoutPrefix(1) }? (Name | NonNameUnit);
128
129
130  /**
131   * CelsiusFahrenheit matches °"C" and °"F"
132   *
133   * Lookahead needed at the beginning to
134   * distinguish with other alternatives
135   */
136  CelsiusFahrenheit =

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

137     { isCelsiusFahrenheit(1) }? "°" unit:Name;
138
139
140 /**
141  * SIUnitDimensionless matches °"" and "deg/rad/sr"
142  * according to https://en.wikipedia.org/wiki/SI_derived_unit
143  *
144  * Lookahead needed at the beginning to
145  * distinguish with other alternatives
146  */
147 SIUnitDimensionless =
148     "°" |
149     { isDimensionless(1) }? unit:Name;
150
151
152 /**
153  * The SIUnitKindGroupWithExponent combines
154  * several SIUnitWithPrefix and SIUnitWithoutPrefix
155  * with exponents as one SI unit group, such as
156  * 'kV^2Ah' and 's^2m'
157  */
158 SIUnitKindGroupWithExponent =
159     { isSIUnitKindGroupWithExponent(1) }?
160     (SIUnitGroupPrimitive "^" exponent:SignedNatLiteral)+
161     SIUnitGroupPrimitive?;
162
163 /**
164  * The SIUnitGroupPrimitive is either a
165  * SIUnitWithPrefix or SIUnitWithoutPrefix
166  */
167 SIUnitGroupPrimitive =
168     SIUnitWithPrefix | SIUnitWithoutPrefix;
169
170
171
172 /** This Token is needed because the Name-Token
173  * does not cover greek Ohm 'Ω' and greek mu 'μ'.
174  * The Token contains at least one of those
175  * symbols.
176  */
177 token NonNameUnit =μ
178     ' ' UnitChar+ |
179     UnitChar* 'Ω' |μ
180     ' ' 'Ω' ;
181
182 fragment token UnitChar =
183     'a'..'z' | 'A'..'Z' ;
184
185
186 // Defining semantic predicates
187 concept antlr {
188     parserjava {
189         public static final String prefix =
190             "(Y|Z|E|P|T|G|M|k|h|da|d|c|m|μ|n|p|f|a|z|y)";
191         public static final String unitWithPrefix =
192             "(m|g|s|A|K|mol|cd|Hz|N|Pa|J|W|C|V|F|Ohm|Ω|S|Wb|T|H|lm|lx|Bq|Gy|Sv|kat|l|L)";
193         public static final String unitWithoutPrefix =

```

```

194     "(min|h|d|ha|t|au|Np|B|dB|eV|Da|u)";
195     public static final String units =
196         "(m|g|s|A|K|mol|cd|Hz|N|Pa|J|W|C|V|F|Ohm|Ω|S|Wb|T|H|lm|lx|Bq|Gy|Sv|kat|l|L|
           min|h|d|ha|t|au|Np|B|dB|eV|Da|u)";
197
198     /* returns true iff the next token matches a
199      * SI unit (group) starting with a prefix
200     */
201     public boolean isSIUnitWithPrefix(int i) {
202         String regex = "(" + prefix + unitWithPrefix + units + "*" + ")" + units +
           "+";
203         return cmpTokenRegEx(i, regex);
204     }
205
206     /* returns true iff the next token matches a
207      * SI unit (group) starting without a prefix
208     */
209     public boolean isSIUnitWithoutPrefix(int i) {
210         return cmpTokenRegEx(i, units + "+");
211     }
212
213     /* returns true iff the next token matches a
214      * dimensionless SI unit
215     */
216     public boolean isDimensionless(int i) {
217         return cmpToken(i, "°", "deg", "rad", "sr");
218     }
219
220     /* returns true iff the next token matches a
221      * ° C or °F
222     */
223     public boolean isCelsiusFahrenheit(int i) {
224         return cmpToken(1, "°") && cmpToken(2, "C", "F") && noSpace(2);
225     }
226
227     /* returns true iff the next token matches a
228      * SI unit (group) starting with or without
229      * a prefix
230     */
231     public boolean isSIUnitGroupPrimitive(int i) {
232         return isSIUnitWithPrefix(i) || isSIUnitWithoutPrefix(i);
233     }
234
235     /* counts the tokens used for a UnitGroup with
236      * exponents, e.g. kV^2Ah
237      * returns -1, iff ht next tokens do not match
238     */
239     public int countSIUnitKindGroupWithExponent(int i) {
240         if (!isSIUnitGroupPrimitive(i))
241             return -1;
242
243         /* A SI unit group with exponents cannot
244          * be a simple primitive and must be
245          * followed by an exponent
246         */
247         if (!cmpToken(i + 1, "^") || !(
248             cmpTokenRegEx(i + 2, "\\d+") && noSpace(i + 1, i + 2) ||
249             cmpToken(i + 2, "-") && cmpTokenRegEx(i + 3, "\\d+") &&

```

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```

250         noSpace(i + 1, i + 2, i + 3))
251     return -1;
252
253     boolean loop = true;
254     int counter = 0;
255     while (loop) {
256         // have seen a primitive
257         counter++;
258
259         // exponent ^2
260         if (cmpToken(i + counter, "^")
261             && cmpTokenRegEx(i + counter + 1, "\\d+")
262             && noSpace(i + counter, i + counter + 1))
263             counter += 2;
264
265         // exponent ^-2
266         else if (cmpToken(i + counter, "^")
267             && cmpToken(i + counter + 1, "-")
268             && cmpTokenRegEx(i + counter + 2, "\\d+")
269             && noSpace(i + counter, i + counter + 1, i + counter + 2))
270             counter += 3;
271
272         /* break if there are spaces between the tokens
273          * or the next tokens do not match an exponent
274          */
275         else
276             loop = false;
277         /* break if the next token is not a SIUnitGroup
278          * or there are spaces between the tokens
279          */
280         if (!isSIUnitGroupPrimitive(i + counter) || !noSpace(i + counter))
281             loop = false; // break
282     }
283     return counter;
284 }
285
286 /* returns true iff the next tokens match a
287  * SI unit (group) with exponents
288  */
289 public boolean isSIUnitKindGroupWithExponent(int i) {
290     return countSIUnitKindGroupWithExponent(i) > 0;
291 }
292
293 /* counts the tokens used for a SI unit primitive
294  * returns -1, iff ht next tokens do not match
295  */
296 public int countPrimitive(int i) {
297     int j = countSIUnitKindGroupWithExponent(i);
298     if (j > 0) return j;
299     if (isSIUnitWithPrefix(i)) return 1;
300     if (isSIUnitWithoutPrefix(i)) return 1;
301     if (isDimensionless(i)) return 1;
302     if (isCelsiusFahrenheit(i)) return 2;
303     return -1;
304 }
305
306 /* returns true iff the next token(s) matches a
307  * SI unit primitive

```

```

308     */
309     public boolean isPrimitive(int i) {
310         return countPrimitive(i) > 0;
311     }
312
313     /* returns true iff the next tokens match a
314      * SI unit Div
315     */
316     public boolean isSIDiv() {
317         int j = countPrimitive(1);
318         if (j > 0
319             && cmpToken(1 + j, "/")
320             && isPrimitive(2 + j)
321             && noSpace(1 + j, 2 + j))
322             return true;
323         return false;
324     }
325
326     /* returns true iff the next tokens match a
327      * 1 divided by a SI unit primitive
328     */
329     public boolean isSIOneDiv() {
330         return cmpToken(1, "1") && isPrimitive(3) && noSpace(2, 3);
331     }
332 }
333 }
334
335 }

```

Listing B.10: SIUnits grammar. The SIUnits grammar provides the basis for using SI units in literals and types.

```

1  /* (c) https://github.com/MontiCore/monticore */
2  package de.monticore;
3
4  /* This is a MontiCore stable grammar.
5   * Adaptations -- if any -- are conservative. */
6
7  /**
8   * This grammar defines SI unit literals
9   * based on all the available SI units such as
10  * '3 m', '2.5 km', '1 km^2' or '3.541 m*deg/(h^2*mg)'
11  *
12  * The definitions are fully compliant to the definitions given in
13  * International Bureau of Weights and Measures (20 May 2019),
14  * SI Brochure: The International System of Units (SI) (9th ed.)
15  *
16  * Caution:
17  * Java long and float unfortunately conflict with SI Units "F" and "L".
18  * We therefore decided:
19  * If the number is followed by L (or F respectively), the literal will
20  * be parsed as BasicLongLiteral "30L" or BasicFloatLiteral "30.2F"
21  * Only if a space is inbetween, it becomes a Liter Literal "30 L"
22  * or Farad Literal "30.2 F".
23  * Accordingly "30Lkg" is not parsable, but "30L kg" and "30 L*kg" are.
24  *

```

MCG

APPENDIX B SELECTED GRAMMARS FROM THE MONTIVERSE

```
25 * The grammar extends the Monticore common literals, because it uses
26 * natural numbers e.g. as exponent.
27 */
28
29 component grammar SIUnitLiterals extends
30     de.monticore.SIUnits,
31     de.monticore.literals.MCCommonLiterals {
32
33     // The unsigned SI unit literals
34     SIUnitLiteral implements Literal <10> =
35         NumericLiteral SIUnit ;
36
37     // The signed SI unit literals
38     SignedSIUnitLiteral implements SignedLiteral <10> =
39         SignedNumericLiteral SIUnit ;
40
41 }
```

Listing B.11: SIUnitLiterals grammar. The SIUnitLiterals grammar defines how to use SI units in literals.

```
1 /* (c) https://github.com/MontiCore/monticore */
2 package de.monticore;
3
4 /* This is a Monticore stable grammar.
5  * Adaptations -- if any -- are conservative. */
6
7 /**
8  * This grammar declares SI unit also as generic types
9  * based on all the available SI unit definitions.
10  * The type parameter can be a numeric type, describing the
11  * range of possible values (such as int, long, float, double).
12  *
13  * The definitions of the SI Unit type themselves (without type parameter)
14  * are fully compliant to the definitions given in
15  * International Bureau of Weights and Measures (20 May 2019),
16  * SI Brochure: The International System of Units (SI)
17  * (9th ed.)
18  *
19  * With this definition an SI Unit such as "kg/m<float>"
20  * can also be used as type definition.
21  *
22  * An extension of the typecheck algorithms is available.
23  * The typecheck ensures correct typing of mathematical expressions.
24  *
25  * As a shortcut the type parameter may be omitted assuming double as
26  * default, i.e.
27  * "km" is identical to "km<double>"
28  * (quite like "List" is identical to "List<Object> in Java")
29  */
30
31 component grammar SIUnitTypes4Computing extends
32     de.monticore.types.MCBasicTypes,
33     de.monticore.SIUnitTypes4Math {
34
35     interface SIUnitType4ComputingInt extends MCType =
```

MCG


```
36      MPrimitiveType SIUnit;  
37  
38      SIUnitType4Computing implements SIUnitType4ComputingInt =  
39          SIUnit "<" MPrimitiveType ">" ;  
40 }
```

Listing B.12: SIUnitTypes4Computing grammar. The SIUnitTypes4Computing grammar defines how to use SI units as types.

Appendix C

MontiThings Grammars

This section shows grammars of the MontiThings project. The grammars have been developed over multiple years including many student theses (*cf.* Sec. 1.4). Due to the student theses, they are not the sole work of the author of this thesis, even though the author supervised the theses that influenced the grammars. The grammars are also largely based on grammars of the MontiVerse (see Appendix B). Deprecated non-terminals not discussed in this thesis are removed to save space.

C.1 Behavior

```
1 // (c) https://github.com/MontiCore/monticore
2
3 /**
4  * This grammar is supposed to extend the functionality for writing
5  * behavior for components directly into the MontiThings models
6  */
7 component grammar Behavior extends de.monticore.literals.MCCommonLiterals,
8                                     de.monticore.statements.MCCommonStatements,
9                                     de.monticore.SIUnitLiterals,
10                                    ArcBasis
11 {
12     /**
13      * ASTAfterStatement can be used to defer the execution
14      * of a MCJavaBlock by the specified amount of time.
15      * This is done asynchronously, i.e. statements after the
16      * statement will be executed without delay.
17      *
18      * @attribute SIUnitLiteral
19      * Amount of time by which the execution of the statements shall be
20      * deferred
21      * @attribute MCJavaBlock
22      * Statements to be executed later
23      */
24     AfterStatement implements MCStatement =
25         "after" SIUnitLiteral MCJavaBlock;
26 }
```

MCG

```

27  * ASTEveryBlock contains statements that shall be executed
28  * periodically. The interval between two executions refers
29  * to the time since the last execution was started. For example,
30  * if the code should be executed "every 5s" and the computation
31  * takes 2 seconds, then the next computation will start 3 seconds
32  * after the last execution finished. If the execution takes longer
33  * than the interval between two execution a warning will be logged.
34  *
35  * @attribute Name
36  *           Name of the block (can be used to start / stop execution)
37  * @attribute SIUnitLiteral
38  *           Distance between two executions
39  * @attribute MCJavaBlock
40  *           Statements to be executed periodically
41  */
42  symbol EveryBlock =
43      (Name ":")? "every" SIUnitLiteral MCJavaBlock;
44
45  /**
46   * ASTLogStatement can be used to log to the console.
47   * Similar to Bash, variables referenced in the StringLiteral
48   * prefixed with a dollar symbol (e.g. "$variable") will be
49   * replaced by their value.
50   *
51   * @attribute StringLiteral
52   *           Text to print to the console
53   */
54  LogStatement implements MCStatement =
55      "log" StringLiteral ";" ;
56  nokeyword "log";
57
58  /**
59   * ASTAgoQualification can be used to access the values of
60   * variables and ports at an earlier point in time.
61   * Using "variable@ago(2s)" accesses the value of "variable"
62   * 2 seconds before the execution of the AgoQualification.
63   *
64   * @attribute Expression
65   *           Variable or port to be accessed
66   * @attribute SIUnitLiteral
67   *           Time at which value is accessed
68   */
69  AgoQualification implements Expression <400> =
70      Expression "@ago" "(" SIUnitLiteral ")";
71
72  /**
73   * ASTConnectStatement can be used to describe behavior which dynamically
74   * connects ports at runtime.
75   *
76   * @attribute Connector
77   *           Connector which specifies which ports should be connected
78   */
79  ConnectStatement implements MCStatement =
80      Connector;
81
82  /**
83   * ASTDisconnectStatement can be used to describe behavior which dynamically
84   * removes connections of ports at runtime.

```

```

85  *
86  * @attribute Source
87  *      Name of the source port which another port should be
88  *      disconnected from
89  * @attribute Target
90  *      Name of the port(s) which should be disconnected from the
91  *      source port
92  */
93  DisconnectStatement implements MCStatement =
94      source:PortAccess "-/>" target:(PortAccess || ",")+ "; "
95  ;
96  }

```

Listing C.1: Behavior grammar. Behavior defines MontiThings' extensions to MCCCommonStatements (Appendix B.3).

C.2 Error Handling

```

1  // (c) https://github.com/MontiCore/monticore
2
3  /**
4   * Pre- and Postconditions for components. Preconditions are evaluated before
5   * executing the behavior
6   * of a component. Postconditions are evaluated after executing the behavior of a
7   * component.
8   */
9  component grammar PrePostCondition extends ConditionBasis,
10                                     de.monticore.expressions.
11                                     ExpressionsBasis
12  {
13      /**
14       * Preconditions are evaluated before executing the behavior
15       * of a component. If a precondition is not fulfilled, the
16       * component throws an error. Preconditions may not access
17       * the output ports of a component (as they only hold a valid
18       * value after the behavior is executed).
19       */
20      Precondition implements Condition =
21          "pre" guard:Expression ";"
22      ;
23      /**
24       * Preconditions are evaluated before executing the behavior
25       * of a component. If a precondition is not fulfilled, the
26       * component throws an error.
27       */
28      Postcondition implements Condition =
29          "post" guard:Expression ";"
30      ;
31  }

```

MCG

Listing C.2: PrePostCondition grammar. PrePostCondition defines pre- and postconditions.

```

1 // (c) https://github.com/MontiCore/monticore
2
3 /**
4  * Catches violated conditions
5  */
6 component grammar ConditionCatch extends ConditionBasis,
7                                     de.monticore.statements.MCCommonStatements
8 {
9
10  /**
11   * Defines a catch statement describing how to handle a violated assumption.
12   * If the condition is violated, the handler is executed.
13   */
14  ConditionCatch =
15      Condition // the catch statement must directly follow a condition
16      "catch" handler:MCJavaBlock
17  ;
18
19 }
```

MCG

Listing C.3: ConditionCatch grammar. ConditionCatch defines how to handle violated pre- and postconditions.

C.3 Set Definitions

```

1 // (c) https://github.com/MontiCore/monticore
2
3 /**
4  * This grammar can be used to define sets of values using, e.g., single values,
5  * ranges of values, or regular expressions.
6  */
7 component grammar SetDefinitions extends de.monticore.literals.MCCommonLiterals,
8                                     de.monticore.ocl.SetExpressions
9 {
10  // range of allowed values (optionally with stepsize)
11  @Override
12  SetValueRange implements SetCollectionItem =
13      lowerBound:Expression
14      (".." stepsize:Expression)?
15      ".." upperBound:Expression
16  ;
17
18  // RegEx to which value has to conform
19  SetValueRegEx implements SetCollectionItem =
20      "format" ":" format:StringLiteral
21  ;

```

MCG

```
22 }
```

Listing C.4: SetDefinitions grammar. SetDefinitions extends the set definitions from the OCL project (Appendix B.6) with sets containing string that conform to a regular expression.

C.4 MontiThings Main Grammar

```

1 // (c) https://github.com/MontiCore/monticore
2
3 import de.monticore.ocl.*;
4 import de.monticore.*;
5
6 grammar MontiThings extends MontiArc,
7     ClockControl,
8     PortExtensions,
9     ConditionBasis,
10    PrePostCondition,
11    ConditionCatch,
12    SetDefinitions,
13    Behavior,
14    OCLExpressions,
15    OptionalOperators,
16    SIUnitTypes4Computing,
17    SIUnitLiterals
18 {
19     start MACompilationUnit;
20
21     /**
22      * Component Types. Matches MontiArc's component types but additionally
23      * allows a "component modifier", i.e. an additional keyword.
24      *
25      * @attribute name The type name of this component.
26      * @attribute head Defines configuration options and extensions of this
27      *                   component.
28      * @attribute componentInstances List of identifiers used to create instances
29      *                               of this component type.
30      *                               Only available for inner components.
31      * @attribute body Contains the architectural elements of this component.
32      */
33     MTComponentType extends ComponentType =
34         MTComponentModifier Name
35         head:ComponentHead
36         MTImplements?
37         (ComponentInstance || ",")*
38         body:ComponentBody
39     ;
40
41     /**
42      * An additional keyword for components.
43      * Interface keyword marks interface components, i.e. components without

```

MCG

```

44     * behavior.
45     * See [Wor16 Sec. 4.1.1]
46     */
47     MTComponentModifier = ["interface"]? "component";
48
49     /**
50     * An additional keyword for components, which marks a component as
51     * implementing an interface component.
52     */
53     MTImplements = "implements" (Name@ComponentType || ",")+;
54
55     /* ===== */
56     /* ===== ArcElement Insertions ===== */
57     /* ===== */
58
59     MTCondition implements ArcElement = Condition;
60     MTCatch implements ArcElement = ConditionCatch;
61
62     /**
63     * Store and restore component states in case of failure
64     * Priority higher than 100 to prevent MontiCore from trying to
65     * interpret "retain" as a component type's name.
66     */
67     MTRetainState implements ArcElement <110> = key("retain") key("state") ";";
68
69
70     /* ===== */
71     /* ===== Behavior ===== */
72     /* ===== */
73
74     interface MTBehavior extends ArcBehaviorElement =
75         (Name@Port || ",")* MCJavaBlock;
76
77     /**
78     * Behavior of components get executed whenever all ports listed in front
79     * of the MCJavaBlock have a new message available. If no ports are listed
80     * the behavior is executed whenever any port has a new message available.
81     */
82     Behavior implements MTBehavior =
83         "behavior" (Name@Port || ",")* MCJavaBlock;
84
85     /**
86     * InitBehavior of components get executed once all ports listed in front
87     * of the MCJavaBlock have a new message available for the first time.
88     * During this first cycle, normal behavior gets skipped and resumes
89     * normally after.
90     */
91     InitBehavior implements MTBehavior =
92         key("init") (Name@Port || ",")* MCJavaBlock;
93
94     /**
95     * Behavior that gets executed in regular time intervals independent of the
96     * availability of new messages.
97     */
98     MTEveryBlock implements ArcBehaviorElement = EveryBlock;
99
100     // In contrast to MontiArc, we use the "statechart" keyword from the
101     // statechart language instead of "automaton"

```



```

102     @Override
103     ArcStatechart implements ArcBehaviorElement =
104         "statechart" "{"
105             SCStatechartElement*
106         "}";
107 }

```

Listing C.5: MontiThings grammar. This is the main grammar of MontiThings that defines component definitions.

C.5 Configuration

MCG

```

1 // (c) https://github.com/MontiCore/monticore
2
3 /**
4  * MTConfig provides additional properties for MontiThings language elements.
5  */
6 grammar MTConfig extends MontiThings {
7
8     start MTConfigUnit;
9
10    /**
11     * MTConfigUnit represents the complete properties for MontiThings
12     * configuration.
13     *
14     * @attribute package The package declaration of the elements.
15     * @attribute Element List of elements.
16     */
17    scope MTConfigUnit = ("package" package:MCQualifiedName ";")? Element+;
18
19    /**
20     * Element is an extension point that is used for the top-level elements in
21     * the MTConfig.
22     */
23    interface Element;
24
25    /**
26     * MTCFGTag is an extension point that is used for tags that refer to a
27     * specific component and platform combination
28     */
29    interface MTCFGTag;
30
31    /**
32     * CompConfig represents the configuration of a
33     * MontiThings Component for a specific platform.
34     *
35     * @attribute Name Name of the MontiThings component.
36     * @attribute platform Deployment platform. E.g. GENERIC, Windows, Arduino etc.
37     */
38    scope symbol CompConfig implements Element =
39        "config" componentType:Name "for" platform:Name
40        "{"
41        MTCFGTag*

```

```
42     "};";
43
44     interface SinglePortTag;
45
46     /**
47      * PortTemplateTag specifies the templates that are used for processing of the
48      * specified port.
49      *
50      * @attribute Name Name of the MontiThings port.
51      */
52     scope symbol PortTemplateTag implements MTCFGTag =
53         port:Name "{"
54         SinglePortTag+
55         "}";
56
57     /**
58      * Hookpoint specifies the template and the arguments that should be supplied
59      * to the template.
60      *
61      * @attribute Name Identification of the Hookpoint.
62      * @attribute template Unqualified name of the template.
63      * @attribute Arguments Attributes required by the template.
64      */
65     symbol Hookpoint implements SinglePortTag =
66         Name "=" template:String Arguments? ";";
67
68     /**
69      * Specifies how often a port should be read out. This is important for
70      * analog inputs that do not have explicit messages.
71      */
72     EveryTag implements SinglePortTag = "every" SIUnitLiteral ";";
73
74     /**
75      * RequirementStatement contain information about what physical requirements
76      * a component has. E.g Sensors/Actuators etc.
77      */
78     scope RequirementStatement implements MTCFGTag =
79         "requires" (property:Property | "{" property:Property+ "}");
80
81     /**
82      * A specific property a device needs to fulfill to execute a component.
83      * Properties have a name and a content. The content can either be a
84      * String or a number.
85      *
86      * @attribute Name Identifier of the property (e.g. "sensor")
87      * @attribute stringValue Content of the property (e.g. "DHT22")
88      * @attribute numericValue Content of the property
89      */
90     symbol Property = Name& ":"
91     (
92         stringValue:StringLiteral | numericValue:SignedNumericLiteral
93     )
94     ";";
95
96     /**
97      * A SeparationHint gives instructions about how the component should
98      * be split up if splitting is enabled.
99      * "none" means that the component and its subcomponents should not be
```

```
100  * splitted. This can be useful to prevent creating container images with
101  * almost no functionality (e.g. a component that adds two numbers).
102  */
103  SeparationHint implements MTCFGTag = "separate" "none" ";";
104 }
```

Listing C.6: MTConfig grammar. This grammar defines MontiThings' configuration language (Sec. 4.3.2).

Appendix D

Open Source Software Used In RTE

MontiThings also utilizes various (sometimes slightly adapted) open-source projects in its RTE:

SPSCQueue ¹ provides a single producer, single consumer queue. MontiThings' Ports use them to buffer messages.

Sole ² is used to generate unique identifiers. MontiThings uses UUID version 4.

TCLAP ³ parses command line arguments of the binaries created from the generated code.

nlohmann/json ⁴ offers a fast JSON parser. MontiThings uses it for storing, *e.g.*, for parsing configuration files.

Easylogging++ ⁵ provides a logging library that, unlike `std::cout`, can be filtered, includes timestamps and has different levels of logging.

Optional ⁶ offers optionals called `tl::optional`, because C++'s `std::optional` is only available since C++17 and MontiThings is C++11 compatible (**TA4**).

Cereal ⁷ (de)serializes C++ objects into JSON format. Adapted to also support `tl::optional` values. Classes generated from MontiThings' class diagram code generator also include the necessary code to be compatible with Cereal.

¹SPSCQueue GitHub Project. [Online]. Available: <https://github.com/rigtorp/SPSCQueue>. Last accessed: 11.11.2021

²Sole GitHub Project. [Online]. Available: <https://github.com/r-lyeh-archived/sole>. Last accessed: 11.11.2021

³TCLAP SourceForge Project. [Online]. Available: <https://sourceforge.net/projects/tclap/>. Last accessed: 11.11.2021

⁴nlohmann JSON GitHub Project. [Online]. Available: <https://github.com/nlohmann/json>. Last accessed: 11.11.2021

⁵Easylogging++ GitHub Project. [Online]. Available: <https://github.com/amrayn/easyloggingpp>. Last accessed: 11.11.2021

⁶Optional GitHub Project. [Online]. Available: <https://github.com/TartanLlama/optional>. Last accessed: 11.11.2021

⁷Cereal GitHub Project. [Online]. Available: <https://github.com/USCiLab/cereal>. Last accessed: 11.11.2021

NNG ⁸ provides websocket and inter-process communication. For example, the `WSPort` is implemented using NNG.

Mosquitto ⁹ provides a client library for communicating with MQTT brokers.

OpenDDS ¹⁰ provides communication using the OMG's DDS standard [Obj15].

⁸NNG GitHub Project. [Online]. Available: <https://github.com/nanomsg/nng>. Last accessed: 11.11.2021

⁹Mosquitto Project. [Online]. Available: <https://mosquitto.org/>. Last accessed: 14.11.2021

¹⁰OpenDDS Project. [Online]. Available: <https://opendds.org/>. Last accessed: 14.11.2021

Appendix E

Models of the HVAC Case Study

Disclosure of prior publication: The models in this section were developed and taken from [KMR21, Mal21, KRSW22].

This appendix shows the composed components and data types of the HVAC case study. Components not shown in this appendix are atomic components. The behavior of the atomic components only consists of logging and if-statements for choosing the next message to send.

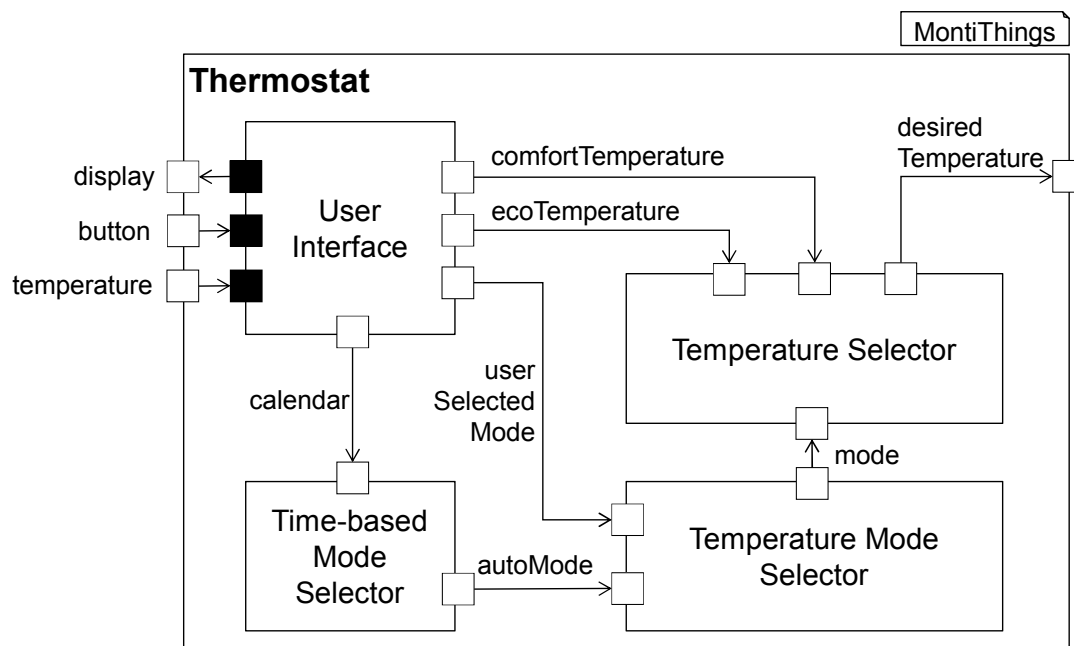


Figure E.1: Thermostat component of the HVAC application. Figure adapted from [KRSW22].

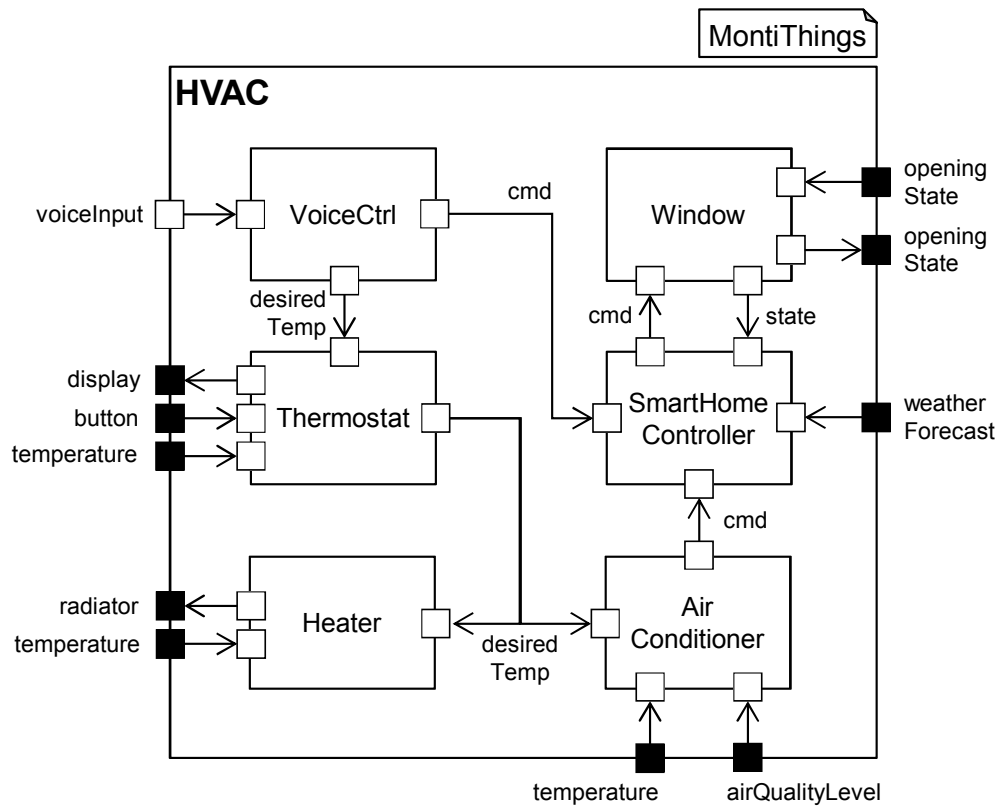


Figure E.2: Outermost component of the HVAC application. Figure taken from [KMR21].

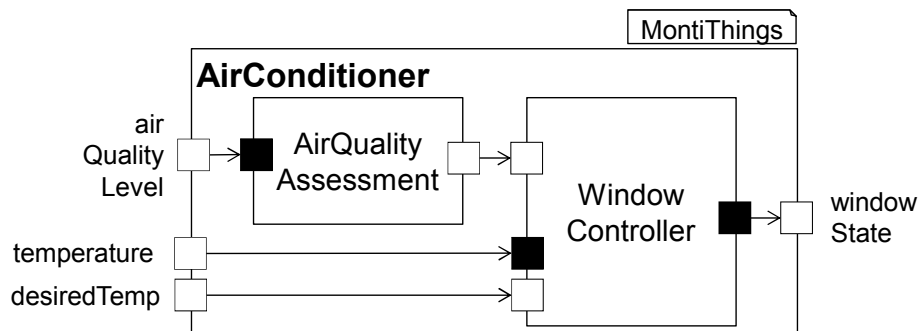


Figure E.3: AirConditioner component of the HVAC application. Figure adapted from [KRSW22].

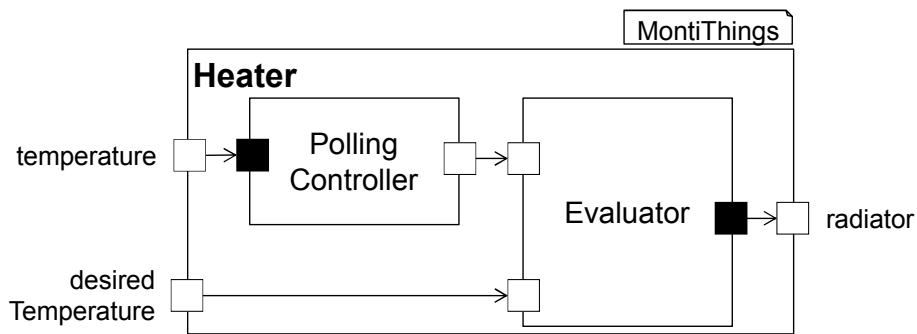


Figure E.4: Heater component of the HVAC application. Figure adapted from [Mal21].

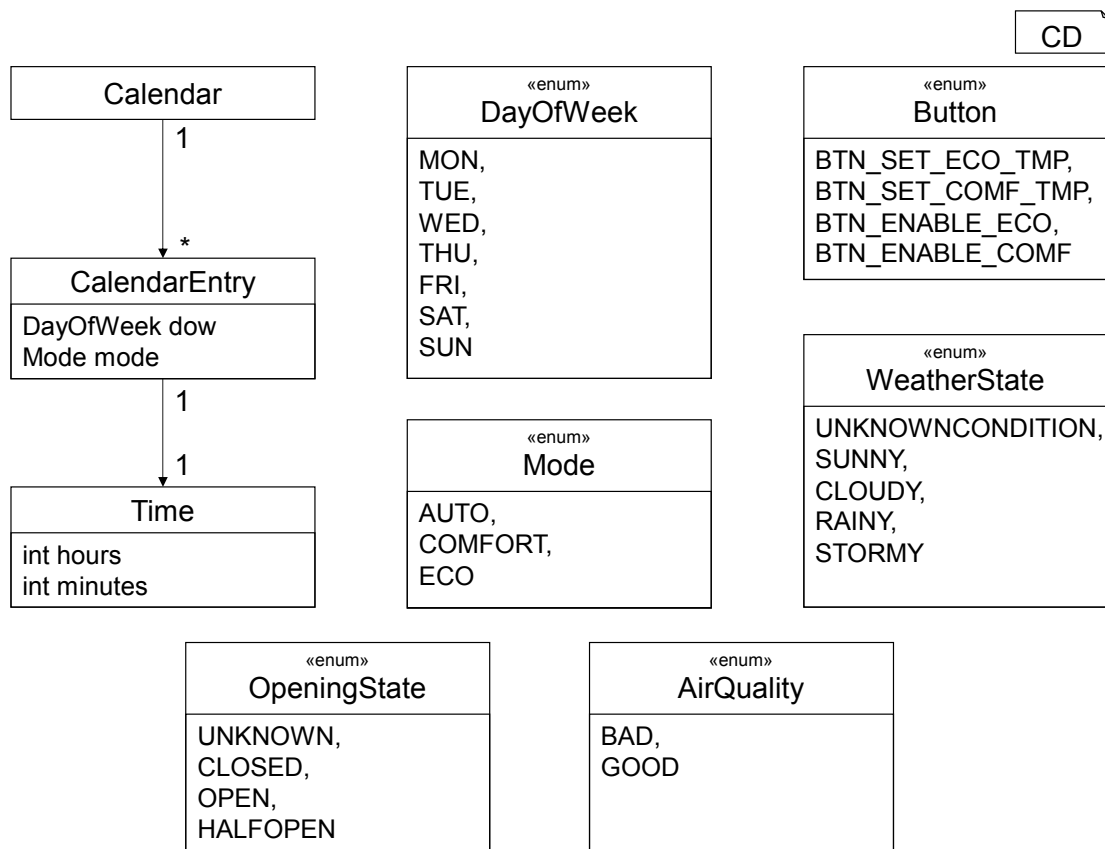


Figure E.5: Additional types used by the HVAC case study defined in class diagrams. As many class diagrams only define single enums, the class diagrams were merged for this figure.

Appendix F

Diagram and Listing Tags






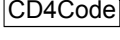








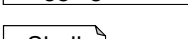
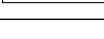
Tag	Description
	Activity Diagram
	CD4A Adapter
	C++ Source Code
	Class Diagram
	Class Diagram for Analysis Diagram
	Class Diagram for Code
	Component Diagram
	Feature Diagram
	Java Source Code
	MontiCore Grammar
	MontiCore Languages
	Object Diagram
	Statechart Diagram
	Sequence Diagram
	Tagging Model (for Digital Twins)
	Shell command

Table F.1: Explanation of the used tags in listings and figures.

Stereotype	Description
«concept»	Conceptual model. Elements do not directly show implementation.
«enum»	Enumeration.
«EXT»	External elements
«extends»	Inheritance relationship.
«GEN»	Generated elements
«HWC»	Handcoded elements
«interface»	Interface component or class.
«pseudo»	Pseudocode.
«real system»	Model refers to what is executed by IoT devices at runtime.
«reproduction»	Model refers to what is executed by developers to reproduce the real system's behavior.
«RTE»	Run-time Environment elements

Table F.2: Explanation of the used stereotypes in listings and tags.

List of Definitions

1	Definition ((IoT) Developer)	16
2	Definition (Device Owner)	16
3	Definition (User)	16
4	Definition (Digital Twin)	18
5	Definition (Software Architecture)	23
6	Definition (Component)	23
7	Definition (Microservice)	24
8	Definition (Container)	25
9	Definition (Container Image)	25
10	Definition (Stream)	27
11	Definition (Interface Component)	55
12	Definition (Subcomponent)	56
13	Definition (Enclosing Component)	56
14	Definition (Technical Requirement)	114
15	Definition (Local Requirement)	115
16	Definition ((Non-)Equal Replay)	179

List of Figures

2.1	Common architecture of IoT applications as identified by [TM17a] (simplified and redrawn from [TM17a]).	18
2.2	Architecture of (code) generators developed using MontiCore (adapted from [HR17, HKR21]).	21
2.3	An example for MontiArc’s graphical syntax (taken from [KMR ⁺ 20b]). . .	26
3.1	Lifecycle of model-driven IoT systems.	33
3.2	High-level overview of MontiThings’ code generation. Figure conceptually based on [HR17, HKR21].	43
3.3	High-level overview of iterative development cycle using MontiThings. . .	44
3.4	A smart home at different points of its lifetime. Some devices fail and new devices get added to the system. Figure taken from [KKR ⁺ 22a]. . . .	47
4.1	A teaser of the graphical and textual syntax of MontiThings. The constructs in this figure are explained throughout this chapter. Figure adapted from [KKR ⁺ 22a].	52
4.2	Overview of MontiThings’ relation to languages of the MontiVerse. Figure taken from [KKM ⁺ 22].	53
4.3	MontiThings’ system boundary is at the hardware access. External services can be used to access sensors, actuators, and other hardware via ports.	66
4.4	Conceptual overview of MontiThings’ dynamics. The environment informs the deployment manager about hardware changes. As a result, the dynamics manager may choose to instantiate components. Their interface is given to the outermost component of the application modeled by the developers. Figure adapted from [KKR ⁺ 22a].	68
4.5	Example for using dynamic reconfiguration: A smart speaker gets connected to a microphone at runtime. Figure conceptually based on [KKR ⁺ 22a].	70
4.6	Adapters can specify that class diagram types shall be converted to a type of a target language to facilitate the integration with hand-written code. Figure taken from [KRSW22].	72
4.7	Sequence diagram specifying a white box test of a fire detector. The graphical syntax of placing ports below components is taken from [HNPR13].	77

LIST OF FIGURES

5.1	Methodology of developing IoT applications using MontiThings. Figure taken from [KRSW22].	86
5.2	Overview of MontiThings' RTE. Figure is an extended version of the overview from [KRSW22].	88
5.3	InOutPorts forward messages and can translate between different communication technologies. In this case a message is received via Bluetooth and forwarded via MQTT. MultiPorts defer the decision which communication technology to use to runtime by providing multiple alternative implementations for the same port. The BluetoothPort is only shown for better understanding and not part of the RTE.	91
5.4	Example for the object structure created when connecting ports.	92
5.5	Communication with independently executed external ports uses local message broker (<i>cf.</i> Fig. 4.3). The hardware access manager connects the architecture to external ports. Figure taken from [BKK ⁺ 22].	94
5.6	Overview of the C++ code MontiThings generates from architecture models. Figure is an extended version from [KRSW22].	95
5.7	Partitioning an architecture into different container images and setting up the component instances started from these container images.	99
6.1	High-level overview of MontiThings' deployment process. Figure taken from [KRSW22].	110
6.2	Workflow for the development of integrated IoT systems and information systems. Both the IoT system and the DTIS are developed using model-driven development. Figure adapted from [KMR ⁺ 20b].	111
6.3	Workflow of developing and deploying software for IoT applications. Figure taken from [KKR ⁺ 22a].	114
6.4	Overview of the components of the deployment system. Figure taken from [KKR ⁺ 22a].	117
6.5	Interaction between the target providers and deployment clients. Figure taken from [KKR ⁺ 22a].	118
6.6	Workflow how the deployment manager utilizes Prolog and Docker Compose for deploying software to an IoT device. Figure adapted from [KRSW22].	121
6.7	Activity diagram of how the generated Prolog code calculates which to which devices a component shall be deployed. Figure taken from [KKR ⁺ 22a].	122

6.8	Example of IoT app store-based specification of technical requirements. The app store provides a hardware ontology as class diagram. Device developers associate each IoT with an object diagram that specifies its hardware. IoT application developers define the components' requirements using OCL. Prolog code generated from the object diagrams and OCL expressions checks whether a device can execute a component. Figure adapted from [BKK ⁺ 22].	123
6.9	A fire extinguisher application for a smart home. IoT developers tag features with the component instances that implement them. Tagging is shown using the dashed arrows. Device owners can choose a feature configuration without knowing which components are required to implement the feature. Figure taken from [BKK ⁺ 22].	126
6.10	Models of a fire extinguishing application. The information system and the IoT system can be connected by connecting attributes of the class diagram to ports within the architecture. Figure taken from [KMR ⁺ 20b].	127
6.11	Overview of the app store concept. Application development and device development are decoupled from each other. By specifying a hardware ontology, the app store ensures compatibility between software and hardware. Figure taken from [BKK ⁺ 22].	128
6.12	Example of the tagging language that connects the class diagram and the MontiThings model from Fig. 6.10. Figure taken from [KMR ⁺ 20b].	131
6.13	Model-to-model transformations for keeping adding synchronization elements to MontiThings models. Elements created by model-to-model transformations are shown in bold. Figure adapted from [KMR ⁺ 20b]. . .	133
6.14	Generic components for exchanging data with the information system. Generated elements are shown in bold. Figure adapted from [KMR ⁺ 20b].	134
6.15	Extension of the information system's class diagrams to synchronize with the IoT system. Elements created by model-to-model transformations are shown in bold. Figure taken from [KMR ⁺ 20b].	135
6.16	Process of retrieving an object from the information system's database in response to receiving a message from the IoT system. Figure taken from [KMR ⁺ 20b].	136
7.1	Motivating example for MontiThings' error analysis. Figure taken from [KMR21].	146
7.2	Overview of MontiThings' transformation-based replay. Figure taken from [KMR21].	147
7.3	MontiThings' failure recovery process. If a component fails, the system replays messages received by the failed component to restore the state of the failed component. Figure taken from [KRSW22].	150

LIST OF FIGURES

7.4	Concept of bundling logs for tracing. Figure taken from [KMM ⁺ 22] and based on [Mal21].	152
7.5	GUI of the web-based log tracing tool. Figure taken from [Mal21].	153
7.6	Handling non-determinism using the <code>nd</code> function. The results of non-deterministic function calls are stored while recording the system's executions. If the system is replayed the actual results of the function call get replaced by the recorded results. Figure taken from [KMR21].	155
7.7	Model-to-model transformations for transforming the original models into the reproduction model. Figure taken from [KMR21].	156
7.8	Relation between the IoT system and the digital twin used for the replay. Figure taken from [KMR21].	159
8.1	MontiThings architecture of the Smart Home / Hotel application. Figure taken from [KKR ⁺ 22a].	168
8.2	Screenshot of the deployment web application for entering location requirements. Labels in the screenshot were translated from the original German web application to English. Unnecessary website elements were removed to save space. Figure taken from [KKR ⁺ 22a].	170
8.3	Screenshot of the deployment web application suggesting to buy a new device with a <code>sensorRFID</code> and <code>actuatorLock</code> capability and placing it in the living room. Labels in the screenshot were translated from the original German web application to English. Figure taken from [KKR ⁺ 22a].	171
8.4	Result of applying the model-to-model transformations to the IoT and information system. Elements created by model-to-model transformations are shown in bold. Figure adapted from [KMR ⁺ 20b].	174
8.5	Electronic setup of the Raspberry Pis. Each of the GPIO extension boards is connected to a Raspberry Pi 4 Model B. Figure taken from [Für20]. . .	175
8.6	Outermost component of the HVAC application. Figure taken from [KMR21].	177
8.7	Concept of running MontiThings in a simulation. A scenario describes how the sensors are supposed to behave. A simulation runner forwards this information to the application via shared files. Figure taken from [Mal21].	178
8.8	Visualization of the accuracy of the replay. Each event is represented by a circle on the timeline. The events of the original execution are connected to their replayed version in the reproduction. The more accurate the temporal reproduction, the less skewed the lines. If there is no corresponding event in the reproduction, the circle has a red fill. Figure taken from [KMR21].	179
8.9	Performance results of of executing the recorder. Figure taken from [KMR21].	181

8.10	Performance results of executing an IoT application with and without log tracing. Figure taken from. Figure taken from [KMM ⁺ 22].	183
8.11	Map of the Fischertechnik setup. This figure was created by the lab's students. Thin black lines represent the road the robots drive on. Blue circles are intersections, red squares are idle positions of the robots. Thick red lines next to the road represent Fischertechnik machines.	187
9.1	High-level overview of the MontiThings ecosystem. Figure adapted from [KRSW22].	192
E.1	Thermostat component of the HVAC application. Figure adapted from [KRSW22].	265
E.2	Outermost component of the HVAC application. Figure taken from [KMR21].	266
E.3	AirConditioner component of the HVAC application. Figure adapted from [KRSW22].	266
E.4	Heater component of the HVAC application. Figure adapted from [Mal21].	267
E.5	Additional types used by the HVAC case study defined in class diagrams. As many class diagrams only define single enums, the class diagrams were merged for this figure.	267

Listings

2.1	A guitar tab specifying how to play music on a guitar.	20
2.2	A non-terminal defining an If-Then-Else statement (taken from Monti-Core's <code>MCCCommonStatements</code>).	21
2.3	Excerpt of the textual representation of the MontiArc architecture in Fig. 2.3.	27
4.1	Definition of a component type <code>Sink</code> with a single incoming port of type <code>int</code> named <code>value</code>	54
4.2	Definition of a component type <code>Source</code> with a parameter of type <code>int</code> named <code>startValue</code> , a single outgoing port of type <code>int</code> named <code>value</code> , and a state variable of type <code>int</code> named <code>lastValue</code>	54
4.3	Definition of an interface component type <code>MathOperation</code> . Interface components have no behavior.	55
4.4	Definition of a generic component type <code>Calc</code> . The argument passed to the type parameter <code>T</code> must conform to the interface of <code>MathOperation</code>	55
4.5	Definition of a composed component type <code>Example</code>	56
4.6	Definition of a component type <code>Source</code> that uses SI units.	57
4.7	Definition of a component type <code>LowPassFilter</code> that replaces messages higher than a threshold with a default value.	59
4.8	Definition of a component <code>Source</code> with a periodic behavior.	60
4.9	A component that sends messages to itself in an endless loop.	61
4.10	A component for a smart light bulb that turns on or off based on voice commands from the user.	62
4.11	A component that calculates a running sum and for monotonically increasing values.	63
4.12	A component that takes an integer and only forwards it if it is a prime number.	64
4.13	Configuration file for a scale component. Two weight sensors provide values via Freemarker templates on the <code>weightLeft</code> and <code>weightRight</code> ports. An OLED display receives messages via an MQTT topic.	75
4.14	Textual representation of the test specification from Fig. 4.7.	78
B.1	ArcBasis grammar. ArcBasis is the basis grammar of MontiArc.	217

B.2	CDBasis grammar. CDBasis is the basis grammar of class diagram grammars.	221
B.3	CDInterfaceAndEnum grammar. CDInterfaceAndEnum extends CDBasis with interfaces and enums.	223
B.4	CDAssociation grammar. CDAssociation defines associations in class diagrams.	224
B.5	CD4Analysis grammar. CD4Analysis combines CDBasis, CDInterfaceAndEnum, and CDAssociation to define full class diagrams.	226
B.6	MCCCommonStatements grammar. MCCCommonStatements is the basis MontiThings' Java-like behavior descriptions.	227
B.7	MCCCommonLiterals grammar. MCCCommonLiterals provides the basis of MontiThings' literals.	229
B.8	OCLEExpressions grammar. OCLEExpressions are imported by MontiThings.	235
B.9	SetExpressions grammar. SetExpressions are imported by MontiThings.	240
B.10	SIUnits grammar. The SIUnits grammar provides the basis for using SI units in literals and types.	243
B.11	SIUnitLiterals grammar. The SIUnitLiterals grammar defines how to use SI units in literals.	249
B.12	SIUnitTypes4Computing grammar. The SIUnitTypes4Computing grammar defines how to use SI units as types.	250
C.1	Behavior grammar. Behavior defines MontiThings' extensions to MCCCommonStatements (Appendix B.3).	253
C.2	PrePostCondition grammar. PrePostCondition defines pre- and postconditions.	255
C.3	ConditionCatch grammar. ConditionCatch defines how to handle violated pre- and postconditions.	256
C.4	SetDefinitions grammar. SetDefinitions extends the set definitions from the OCL project (Appendix B.6) with sets containing string that conform to a regular expression.	256
C.5	MontiThings grammar. This is the main grammar of MontiThings that defines component definitions.	257
C.6	MTConfig grammar. This grammar defines MontiThings' configuration language (Sec. 4.3.2).	259

List of Tables

4.1	Primitive data types of MontiThings.	57
4.2	Overview of related IoT modeling methods. ● = fulfilled, ◐ = partly fulfilled, ○ = not fulfilled.	83
6.1	Overview of related IoT modeling and deployment approaches. ● = fulfilled, ◐ = partly fulfilled, ○ = not fulfilled. Table and annotations largely taken from [KKR ⁺ 22a].	139
8.1	Overview of the devices used in the case study. All devices are Raspberry Pi 4 Model B.	169
8.2	Influence of DSs on the accuracy of the order of messages and timing. Table taken from [KMR21].	180
F.1	Explanation of the used tags in listings and figures.	269
F.2	Explanation of the used stereotypes in listings and tags.	270

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website www.se-rwth.de/topics/ or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques, concepts, and methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: “Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.”, [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g. with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.

The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able, e.g., to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was

introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded 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, [Hab16] for MontiArc is used in domains such as cars or robotics [HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and, e.g., in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.

[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].

Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the “System Model” [BCGR09], [BCGR09a], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].

The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99a], [FEL+98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed

computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a, HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10]. [BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented, e.g., in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.

We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.

We use streams, statemachines, 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 [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to

the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15a]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the “globalized” use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. And [BKRW17] compares component and connector architectures similar to SysML’ block definition diagrams.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between a system, a view, and a complete model in the context of the UML.

[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10].

[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].

[Rum04c] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].

In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CreEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include

DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement

management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMarDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].

[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO₂ emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research

projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.

We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [PR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.

We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

- [ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.
- [ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.
- [ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.
- [AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.
- [AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSEr)*, 8(1):3–16, 2017.
- [AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM SIGPLAN, October 2021.
- [AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.
- [ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.
- [BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Pádraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Boris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensche, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics*, 12, 2018.

- [BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.
- [BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.
- [BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.
- [BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.
- [BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.
- [BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.
- [BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.
- [BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.
- [BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Shahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.
- [BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas

- Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *ACM Trans. Internet Things*, 3:1–32, February 2022.
- [BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE’18)*, pages 187–199. ACM, 2018.
- [BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.
- [BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology*, 19(3):3:1–16, October 2020. Special Issue dedicated to Martin Gogolla on his 65th Birthday.
- [BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC’18)*. ACM, September 2018.
- [BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.
- [Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.
- [BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA’97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.
- [BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Computer Standards & Interfaces*, 19(7):335–345, November 1998.
- [BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.
- [BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND’17) Workshop*, July 2017.

- [BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.
- [BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA’17)*, LNCS 10376, pages 53–70. Springer, July 2017.
- [BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.
- [BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.
- [BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE’97)*, LNCS 1526, pages 43–68. Springer, 1998.
- [BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.
- [BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.
- [BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE’18)*, pages 174–186. ACM, 2018.
- [BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC’18)*. ACM, September 2018.

- [BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case*. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.
- [BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.
- [BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.
- [BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software*, 149:437–461, March 2019.
- [BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.
- [BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.
- [BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.
- [BR12b] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.
- [BR12] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.
- [BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.
- [CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.
- [CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

- [CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.
- [CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.
- [CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.
- [CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS’09)*, LNCS 5795, pages 670–684. Springer, 2009.
- [DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODEL-SWARD’19)*, pages 274–282. SciTePress, February 2019.
- [DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA’18)*, pages 146–153, August 2018.
- [DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019.
- [DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology*, 20:1–24, November 2021.
- [DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.
- [DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Cus-

- tomized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.
- [DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnav, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.
- [DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.
- [DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.
- [EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluwer Academic Publisher, 1999.
- [EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.
- [EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnav, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.
- [ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.
- [FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

- [FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.
- [FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.
- [FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.
- [FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.
- [FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.
- [FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.
- [GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.
- [GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.
- [GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.
- [GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.
- [GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wort-

- mann. 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.
- [GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.
- [GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.
- [GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.
- [GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.
- [GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.
- [GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.
- [GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.
- [GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.
- [GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Hei-

- delberg, April 2015.
- [GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.
 - [GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.
 - [GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.
 - [GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.
 - [GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.
 - [Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.
 - [GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.
 - [GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009*, LNCS 5522, Lisbon, Portugal, 2009.
 - [GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.
 - [Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.
 - [Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy*. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.
 - [HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In

- Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.
- [HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.
- [HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.
- [HHK+15a] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.
- [HHR+15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.
- [HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.
- [HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.
- [HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.
- [HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.
- [HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineering in Research and Practice (SERP'09)*, pages 172–176, July 2009.
- [HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

- [HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.
- [HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016.
- [HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.
- [HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.
- [HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.
- [HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019.
- [HNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.
- [Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.
- [HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What’s the Semantics of ”Semantics”? *IEEE Computer*, 37(10):64–72, October 2004.
- [HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.
- [HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.
- [HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop*

- MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010.
- [HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.
- [HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.
- [HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 1 – 10. fortiss GmbH, 2011.
- [HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.
- [HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.
- [HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.
- [HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018.
- [JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology*, 21, July 2022.
- [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.
- [KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.
- [KKP+09] Gabor Karsai, Holger Krah, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

- [KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications.
- [KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *ACM Transactions on Internet of Things*, November 2022.
- [KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansoor Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.
- [KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW’12)*, pages 2:1–2:6. ACM, October 2012.
- [KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Beloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR’16)*, LNCS 9679, pages 122–137. Springer, June 2016.
- [KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.
- [KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.
- [KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM SIGPLAN, October 2021.
- [KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE’18)*, pages 172–180. ACM,

June 2018.

- [KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.
- [KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.
- [KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.
- [KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O’Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.
- [KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.
- [Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.
- [KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapman & Hall, 1996.
- [KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.
- [KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.
- [KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy’s Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of*

- the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.
- [KRRW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.
- [KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.
- [KRS+18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.
- [KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, 183:1–21, January 2022.
- [KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.
- [KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.
- [KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.
- [KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.
- [KRV10] Holger Krahn, Bernhard Rumpe, and Stefan Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.
- [KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020.

- [Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.
- [LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Zieffle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.
- [Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.
- [LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.
- [MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.
- [MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.
- [MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.
- [MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.
- [MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.
- [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.
- [MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

- [MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 9–12. IEEE, May 2022.
- [MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME’10)*, LNCS 6627, pages 194–203. Springer, 2010.
- [MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE ’11)*, pages 179–189. ACM, 2011.
- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.
- [MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS’11)*, LNCS 6981, pages 592–607. Springer, 2011.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.
- [MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP’11)*, LNCS 6813, pages 281–305. Springer, 2011.
- [MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS’11)*, LNCS 6981, pages 153–167. Springer, 2011.
- [MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.
- [MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’13)*, pages 444–454. ACM New York, 2013.
- [MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

- [MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.
- [MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.
- [MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.
- [MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.
- [MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.
- [Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.
- [NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.
- [NRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.
- [PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.
- [PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.
- [PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS

- 2379, pages 188–197. Springer, 2002.
- [Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen*. Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.
- [Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project*. Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.
- [PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME’94)*, LNCS 873, pages 154–174. Springer, 1994.
- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM’99)*, LNCS 1708, pages 96–115. Springer, 1999.
- [PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.
- [PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.
- [Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.
- [Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.
- [RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.
- [RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.
- [Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.
- [RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.
- [RRRW15b] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering*

- for Robotics (JOSER)*, 6(1):33–57, 2015.
- [RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.
- [RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.
- [RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Kozirolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.
- [RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.
- [RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.
- [RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.
- [RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.
- [RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.
- [Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.
- [Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.
- [Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852,

- pages 380–402. Springer, November 2003.
- [Rum04c] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.
 - [Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.
 - [Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.
 - [Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.
 - [Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.
 - [RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.
 - [Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.
 - [SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.
 - [SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.
 - [SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.
 - [SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.
 - [TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70.

- Springer, July 2021.
- [THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.
 - [Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.
 - [WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.
 - [Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.
 - [Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.
 - [Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, Oktober 2021.
 - [ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.