[Wor21] A. Wortmann:
Model-Driven Architecture and Behavior of Cyber-Physical Systems.
In: Aachener Informatik-Berichte, Software Engineering, Band 50, ISBN 978-3-8440-8345-3. Shaker Verlag, Dec. 2021.
https://www.se-rwth.de/publications/

RWTH Aachen University

Cumulative Habilitation Treatsie

Model-Driven Architecture and Behavior of Cyber-Physical Systems

Dr. rer. nat. Andreas Wortmann

Prepared at

Lehrstuhl Informatik 3
Software Engineering

Fachgruppe Informatik

Fakultät für Mathematik, Informatik
und Naturwissenschaften

Dedicated to my parents

Acknowledgements

During the habilitation research project presented in this thesis, I was fortunate to be surrounded by many excellent, inspiring, and helpful people who have contributed to the success of this research project in various ways and to which I am very grateful for that.

Foremost, I thank Bernhard Rumpe for the opportunity to work at his Chair for Software Engineering, for entrusting me with the lead of the Model-Driven Systems Engineering working group, for his continuous interest in my research, and for many fruitful discussions contributing to shaping the research program presented in this thesis.

I also thank Uwe Aßmann of the Technische Universität Dresden and Gerti Kappel of the TU Wien for being part of the way to this thesis, many interesting discussions, and reviewing it.

Moreover, I thank Manfred Nagl for being supportive to my research program, many helpful discussions along the way, and sharing his insights into software engineering academia with me.

I also thank the colleagues and friends at the Chair for Software Engineering, including Vincent Bertram, Marita Breuer, Arvid Butting, Joel Charles, Manuela Dalibor, Imke Drave, Robert Eikermann, Christoph Engels, Arkadii Gerasimov, Sylvia Gunder, Steffen Hillemacher, Katrin Hölldobler, Nico Jansen, Oliver Kautz, Jörg Christian Kirchhof, Evgeny Kusmenko, Achim Lindt, Matthias Markthaler, Judith Michael, Joshua Mingers, Sonja Müßigbrodt, Lukas Netz, Jerome Pfeiffer, Manuel Pützer, Deni Raco, Martin Schindler, David Schmalzing, Sebastian Stüber

Simon Varga, Galina Volkova, and Louis Wachtmeister.

contributed to the success of this research project. Furthermore, I thank the colleagues and friends who also conduct research in Software Engineering and who accompanied my habilitation research project in various ways, including Olivier Barais, Christian Berger, Thorsten Berger, Wolfgang Böhm, Michel Chaudron, Federico Ciccozzi, Benoit Combemale, Timo Greifenberg, Regina Hebig, Rodi Jolak, István Koren, Markus Look, Ivano Malavolta, Daniel Méndez Fernández, Patrizio Pelliccione, Alfonso Pierantonio, Arend Rensink, Jan Oliver Ringert, Ina Schäfer, Stefan Schiffer, Hans Vangheluwe, and Mark van den Brand.

My full gratitude, however, goes to my family and friends, who supported me throughout my habilitation research project and to my wife Julia who accepted the considerable share of research in my life and always encouraged me to follow this passion.

Abstract

Systems engineering has produced striking results in many domains. Researchers and practitioners have devised concepts, methods, tools that autonomously move vehicles, enable doctors to conduct remote surgeries across continents, and sent astronauts into space. All of these cyber-physical systems are driven by software whose complexity increases tremendously. Overcompensating this growth in software and systems complexity demands novel methods that increase the abstraction in systems engineering, advance automation, and facilitate the integration of domain expert solutions. Model-based systems engineering aims to address this complexity by advancing systems engineering from its contemporary document-based processes to sophisticated model-based processes. In the latter, abstract models serve as means for systems design, communication, documentation, and basis for implementation. But to overcompensate the growth in complexity, using models as secondary artifacts is insufficient. Comprehensive research in software engineering has led to recognizing that model-driven processes, in which models are the primary engineering artifacts, can significantly improve abstraction, automation, and domain-specific modeling to address the increasing complexity in systems engineering. Yet, model-based systems engineering focuses on informal models that are hardly accessible to meaningful automation and overly generic.

This thesis summarizes 13 selected publications of a research program towards a model-driven systems engineering that operates on domain-specific modeling languages, supports sophisticated modeling methods, and enables the systematic operation of cyber-physical systems. The results of this research program cover four substantial challenges towards the model-driven engineering of cyber-physical systems: First, it contributes to understanding the use of models and modeling languages for cyber-physical systems through a comprehensive literature study on modeling for cyber-physical systems in Industry 4.0. The study surveyed over 4.000 publications and produced insights into requirements for the efficient model-driven engineering and operations of cyber-physical systems in Industry 4.0. Second, it conduces novel foundations for the efficient engineering of domain-specific modeling languages based on the requirements identified in the literature study. These foundations introduce innovative notions of language components and their composition upon which families of domain-specific modeling languages can be created systematically efficiently. Third, it leverages these foundations to produce modeling languages to describe functional architectures and geometric-physical architectures of cyber-physical systems that support unprecedented automated modeling methods, including tracing, decomposition, and semantic differencing, to facilitate modeling, maintaining, and evolving these architectures. Fourth, it exploits the novel language engineering foundations and the unprecedented automated modeling methods to alleviate the systematic operation of cyber-physical systems with digital twins that represent

and optimize the observed systems. Hence, this research program forges a bridge from observations on modeling cyber-physical systems, over software language engineering and modeling methods, to their operation that supports researchers and practitioners to advance from the contemporary document-based engineering of cyber-physical systems to their systematic model-driven engineering.

Contents

Chapter 1

Introduction

The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter - for the future. His duty is to lay the foundation for those who are to come, and point the way.

Nikola Tesla

The research results presented in this thesis plant novel foundations, concepts, and methods that aim to facilitate the systematic model-driven engineering of the cyber-physical systems of the future. This chapter motivates our research, details the challenges addressed through the publications summarized in this thesis, explains the background of our research, clarifies terminology, and describes the outline of the remainder of this thesis.

1.1 Context and Motivation

Our society thrives on cyber-physical systems. In these systems, cyber parts, *i.e.,* software, control physical sensors and actuators to make the overall system interact with its environment, often including networks or the Internet, to perform services, such as communication, creation of added-value, education, healthcare, mobility, and more. Well-known cyber-physical systems are modern cars that leverage their sensors and telematics to assist the driver, fitness trackers that compute work-out improvements in the Cloud, service robots that support medical staff and guide visitors through exhibitions, or the smart manufacturing systems of Industry 4.0 that are connected to another to optimize production. Overall, there is a variety of cyber-physical systems of different forms and functions that we rely upon in for recreational or professional activities.

Industrial engineering of cyber-physical systems always demands expertise from heterogeneous domains, including mechanical and electrical engineering, software and systems engineering, human-machine interaction, business administration, jurisprudence, and many more. The domains participating in the engineering of cyber-physical systems contribute to various concerns of the systems under development and follow different schools of thought, paradigms, methods, and techniques to address these concerns. For instance, mechanical engineers employ technical drawings in the form of 3D models to prescribe geometries and physical properties, electrical engineers use circuit diagrams to model electrical circuits, and software engineers use general-purpose programming

languages (GPLs) or domain-specific languages (DSLs) to prescribe the structure and behavior of the systems' software. For the efficient engineering and operation of cyber-physical systems, domain experts, such as mechanical engineers, electrical engineers, or roboticists need suitable solution techniques that support for expressing their contributions using concepts and terminology established in their domain of expertise. These different domain-specific techniques must support the automated analysis of integrated, heterogeneous models and the synthesis of downstream artifacts (such as GPL code or documentation) where possible. Leveraging domain-specific models raises the abstraction in the engineering of cyber-physical systems, which enables domain experts to focus on challenges in their respective problem domains.

Humanity has been using models to describe, understand, and change the world for millennia. In ancient Greece and Egypt, philosophers used mathematical models, geometric models, or physical models to describe the world, to understand what it is that holds it together, and to construct buildings and machines. Since then, scientists and engineers model to comprehend and to design (parts of) the world. Nonetheless, it is fairly recent that we have found methods to use models as blueprints that can be translated to realizations automatically. Within the last few decades, the amount and the importance of software and software-based cyber-physical systems have increased significantly and their complexity has increased as well. Therefore, software engineering, has begun to leverage models to reduce the conceptual gap [FR07] that arises from using different abstractions in the problem domains of discourse, such as mechanical engineering or jurisprudence, than in the solution domain of software engineering.

The umbrella term model-driven engineering (MDE) [AK03, FR07, VSB+13, WHR14] describes software development methodologies that leverage (domain-specific) models as primary development artifacts to reduce the conceptual gap. To support the efficient engineering with such models, these must be reified in machine-processable modeling languages. Consequently, efficiently engineering, customizing, and reusing modeling languages is a prime concern in MDE and gave rise to the field of software language engineering (SLE) [HRW18, Kle08]. Research in SLE devised concepts, methods, and techniques to design, engineer, and evolve (domain-specific) modeling languages. But as software languages are software as well, they are subject to the same challenges regarding engineering, customization, and reuse as other software [FGLP10]. Based on such modeling languages, for instance, the Unified Modeling Language (UML), the Systems Modeling Language (SysML), or various DSLs, many successful applications of MDE [BCOR15, HRW11, KR05, Rai05, Sta06, WWM+07] have been reported in the context of systems engineering for various domains and studied systematically [HPE+16, LMT+14, Pet13, Val14].

Amplified by the successes of MDE in software and by the challenges of traditional systems engineering in overcompensating the complexity of engineering cyber-physical systems, the paradigm of model-based systems engineering (MBSE) [Mic14, Ogr00, RFB11] has emerged. MBSE can be considered as the formalized application of modeling to systems engineering, ranging from requirements elicitation through development to operation. Consequently, MBSE aims at leveraging models and modeling languages to

(1) support domain experts in describing their respective system parts and concerns through suitable modeling languages that employ domain-specific concepts and terminology to reduce the conceptual gap; (2) enable automated analyses and syntheses that ease systems engineering, evolution, and operations in the presence of cyber-physical components; and (3) optimize the operation of cyber-physical systems with encoded domain expertise after deployment. This vision of MBSE demands means to efficiently engineer and customize suitable DSLs for domain experts. Such DSLs are the basis for unprecedented automation that enables novel modeling methods for engineering and evolving cyber-physical systems. Moreover, they greatly facilitate optimizing the use of cyber-physical systems through data abstraction and integration of domain expertise into their operations.

1.2 Problem Statement

MDE and model-driven systems engineering (MDSE) aim to overcompensate the growth in software and systems complexity by making models the primary development artifacts. These models can be more abstract than traditional GPL artifacts, *e.g.,* to liberate domain experts from dealing with the accidental complexities raising the conceptual gap [FR07] between problems in their domain of expertise and the technical peculiarities of the solution technologies. Moreover, these models can employ domain concepts and terminology the corresponding experts are familiar with. Yet, there is little systematic evidence on the use of (domain-specific) modeling techniques in the engineering of cyber-physical systems.

To foster the application of truly domain-specific modeling languages, a novel and effective method for the efficient development of DSLs is required that facilitates the systematic engineering DSLs based on reusing existing languages and language modules as well as customizing these. Ultimately, this fosters creating the DSLs that that domain experts need. Such a method must consider language reuse systematically and holistically by supporting reusing syntax and semantics of independently developed language modules in a black-box fashion.

When the models employed in cyber-physical system engineering conform to explicit modeling languages, *i.e.,* modeling languages reified in software, they can become machine-processable to support systems engineers, system integrators, and domain experts through modeling methods based on automated analyses that facilitate their contributions. This opens novel possibilities to address challenges regarding design and evolution of architectures and behavior of cyber-physical systems: A quintessential methodical challenge is reducing the gap between functional system architectures and their geometric-physical realizations. When both sides of this gap, functional architectures and geometric-physical realizations, are made explicit through models, they can be integrated, traced, and reused more efficiently. A method for this must combine the functional perspective typical to computer science with the geometric-physical perspective typical to mechanical engineering. A subsequent challenge is the systematic refinement and decomposition of functional architectures to support systems engineers in evolving

and these architectures and to ease the integration of domain expert solutions.

Ultimately, cyber-physical systems are deployed and operated in different contexts, where they process a plethora of data to optimize the processes they were designed to realize. Often, these systems are intended to be integrated with other systems or services, their use should be optimized, and their behavior predicted. For these purposes, research and industry have coined the notion of the digital twin [BR16, DR18, GV17], a digital representation of a cyber-physical system for a specific purpose, such as presenting selected process data of a manufacturing system to the factory's business intelligence and decision making, optimizing the use of resources, reducing the downtime or production of reject, or predicting when maintenance will be necessary.

The vision of digital twins as digital replicas of cyber-physical systems [SAOI18, ZMHK18] often aims to provide a complete digital representation of the system under investigation. Due to the technical complexity of the represented cyber-physical systems, such a complete and functionally comprehensive digital replica of the production system as a digital twin is costly and, due to the nature of digitization abstracting away from continuous properties, hardly possible. Moreover, engineering digital twins for cyber-physical systems is challenging, which is partly due to the lack of a suitable definition of what a digital twin is supposed to be and do. Based on this missing foundation, only a few systematic approaches towards engineering digital twins exist and these employ semantics-agnostic modeling languages, which prevents applying methods for systematic evolution and operation to digital twins. Moreover, where digital twins are intended to control and optimize the behavior of the represented cyber-physical systems, they demand domain expertise. Yet, no systematic methods to include this into the engineering of digital twins have been devised. This habilitation thesis presents answers to these challenges in the engineering of cyber-physical systems.

1.3 Research Objectives

The habilitation thesis at hand addresses the systematic model-driven engineering and operation of cyber-physical systems. To this end, it addresses a scope of concerns in cyber-physical systems engineering ranging from providing (1) suitable DSLs to participating domain experts; (2) systematic modeling methods to systems engineers; and (3) methods to facilitate and optimize operations of cyber-physical systems. From our vantage point, SLE, as a means to efficiently develop and customize these modeling languages, therefore, is the essential foundation for the successful MDSE of cyber-physical systems. The modeling languages precisely engineered to the domain experts' requirements enable novel modeling methods leveraging unprecedented traceability, model reuse, and automation. These, in turn, facilitate the engineering of cyber-physical systems and of the digital twins that monitor, represent, and optimize the operations of the cyber-physical systems. Figure 1.1 illustrates our view on modeling for cyber-physical systems with its foundations in SLE on which the efficient engineering of suitable DSLs, systematic modeling methods, and optimized operations of cyber-physical systems builds upon.

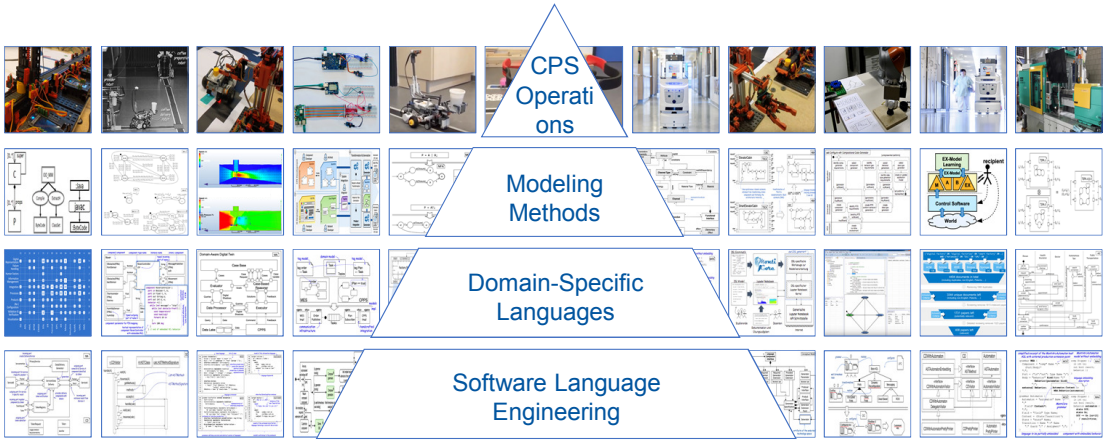


Figure 1.1: The research presented in this habilitation thesis covers foundations in software language engineering that enable domain experts in cyber-physical systems engineering to use the most appropriate modeling languages as well as systematic modeling methods for engineering cyber-physical system (CPS), and novel methods for their model-driven operation with digital twins.

Through the contributions highlighted in this thesis, we aim to support researchers and practitioners in cyber-physical systems engineering in (1) reducing the conceptual gap between the participating experts' problem domains and the solutions domains of systems engineering by means of suitable DSLs; (2) methodically guiding the engineering of functional system architectures, their evolution, as well as the mapping to geometric-physical realizations; and (3) systematically constructing and deploying digital twins to optimize the operation of cyber-physical systems. This section presents the research questions motivating the research program leading to this thesis.

1.3.1 Research Questions

RQ1: *What are challenges regarding modeling for cyber-physical systems?*

Despite many promising results, MBSE still appears to be a niche approach to the engineering of cyber-physical systems. Hence, the first objective of this thesis is to report on the state-of-the-art regarding availability, use, and limitations of modeling for cyber-physical systems. To uncover how modeling is applied to research and engineering of cyber-physical systems, which modeling techniques are used for which purposes, and where suitable modeling techniques are missing, systematic evidence is necessary. Aside from surveys regarding modeling for embedded systems [ASSS13, LMT+18], such evidence was missing.

To investigate the application of modeling to the engineering and operation of cyber-physical systems, we conducted a systematic mapping study in the advanced domains of cyber-physical systems for Industry 4.0. The first study explores the application

of modeling to the smart manufacturing of Industry 4.0 and identifies modeling trends, ranging from the physical design of cyber-physical systems to their operation with digital twins. The second survey researches the application of modeling in the engineering of mobile robotic systems and identified modeling concerns, techniques, and automation employed in robotics.

RQ2: *How can software language engineering facilitate the model-driven engineering of cyber-physical systems in collaboration with domain experts?*

To be successful, MDSE demands suitable modeling languages. Yet, most modeling for cyber-physical systems employs general-purpose modeling languages, such as UML, SysML, or Simulink, which neither directly support modeling with domain concepts, nor support the integration of contributions from experts of different domains. This often alienates domain experts, which leads to modeling inconsistencies and errors that could be prevented by employing suitable DSLs. As engineering such modeling languages is a sophisticated endeavor requiring specific expertise and skills, neither system engineers, nor domain experts can be expected to create these languages. To mitigate this, SLE needs to provide means to easily and efficiently create truly domain-specific languages from predefined modules, integrate these to obtain the desired modeling language, and customize it with little effort. Through this, domain experts can be liberated from using overly generic modeling languages, which facilitates the model-driven engineering of cyber-physical systems.

To approach this research objective, we adapted established modularity concepts from software engineering to SLE, including the notion of components with interfaces from component-based software engineering (CBSE) [Crn01, KB98], the arrangement of language components in language families from software product lines [PBL05, PM18], and a reuse process derived from reuse in object-oriented software engineering [CKM+18]. Based on these, we conceived a novel systematic method for the reuse of customizable language components, their integration in language families, resolving the families' inherent variability, and supporting open variability through customization. At the core of this method is a novel conceptual model of language components that encapsulate realizations of syntax and semantics behind explicit interfaces. Based on this model, our method is supported by a toolchain automating most of the corresponding activities.

RQ3: *How can architecture and behavior modeling systematically facilitate the engineering and evolution of cyber-physical systems?*

Successful MBSE needs to integrate the views and (mental) models of participating disciplines. Between these there exist many gaps that need to be overcome to actually engineer and deploy cyber-physical systems, including the gap between continuous and discrete solutions, the gap between problem domain abstractions and solution domain implementations, and the gap between functional and geometric-physical architectures of the systems under development. When the gap between functional and geometric-physical perspectives can be closed, models of the systems functional architectures can be decomposed to into a set of components. These components then can be implemented by

domain experts using the most suitable modeling languages. Prior to integrating these components, the systems engineers then must check their compatibility with the initial implementation. Ideally, these activities are automated.

To bridge the gap between functional architectures and geometric-physical product architectures of cyber-physical systems, we apply concepts of the Focus [BS01, RR11, RW18] theory of functional architectures to the geometric-physical perspective of mechanical solutions based on the Koller catalog [KK98] of physical effects. This enables to systematically model functional architectures and their implementation through geometric-physical realizations. For the functional architectures, we conceived methods for their automated decomposition and checking whether two versions of an architecture (or its parts) refine each other semantically. The former eases decomposing monolithic architecture specifications into components realizable by domain experts, the latter enables checking whether the implementations provided by domain experts actually conform to the specifications.

RQ4: *How can MDSE facilitate the systematic engineering of digital twins to optimize the operation of cyber-physical systems?*

MDSE aims to encompass all phases of a cyber-physical system's life cycle, from its eliciting its requirements over planning and implementation to operations and disposal. Research and industry have produced various solutions to optimize the use of cyber-physical systems through digital twins. Yet, systematic model-driven approaches to engineering digital twins for cyber-physical systems are rare. Where they exist, they either focus on digital twins being passive representations that cannot interact with, and hence, optimize the behavior of the represented cyber-physical systems, or they do not support incorporating domain expertise, which often is crucial to optimize operation of cyber-physical systems.

Based on experiences within automotive, manufacturing, and robotics, we conceived a functional and extensible reference architecture for digital twins that integrates domain-specific modeling languages to ease the systematic connection of the digital twin to the represented cyber-physical systems, data sources, and the description of actionable domain expertise. Leveraging the same functional architecture modeling technique as for modeling the CPS enables applying the systematic modeling methods mentioned above to the digital twin as well, which facilitates the integrated development of a CPS together with its digital twin.

1.3.2 Contributions

This habilitation thesis presents four substantial contributions to advance the model-driven engineering of cyber-physical systems with respect to its research objectives. First, it presents a systematic literature study to produce reliable evidence on the operation of modeling in the advanced domain of cyber-physical systems engineering for Industry 4.0, which guides researchers and practitioners to identify and select existing modeling languages for their challenges at hand as well as uncover gaps in the literature that signpost potential research directions. Second, it presents a novel conceptual

model and a systematic method for software language engineering that addresses findings from the aforementioned study. With this, language engineers can provide more appropriate domain-specific modeling languages to experts of the domains involved in engineering cyber-physical systems. Third, it summarizes modeling methods to reduce the gap between the functional view of computer science and the geometric-physical perspective of mechanical engineering and automate systems engineering activities in the presence of domain experts. Fourth, it presents novel modeling methods for the systematic operation of cyber-physical systems with digital twins that facilitate modeling platform-independent digital twins, integrating domain expertise into these, and bind these to specific platforms.

The contributions presented in this thesis enable researchers and practitioners in cyber-physical systems engineering to understand the state-of-the-art regarding modeling, to leverage truly domain-specific modeling languages that ease creation and integration of domain expert contributions, to reduce the gap between functional and geometric-physical modeling, and to systematically create digital twins that monitor and optimize the behavior of the cyber-physical systems after deployment.

1.4 Research Context

The research program summarized in this thesis presents novel concepts, methods, and tools to facilitate the engineering of cyber-physical systems. To this end, this section first illustrates the context of cyber-physical systems and their application to Industry 4.0 before it presents background relating to MDE, MDSE, and SLE.

1.4.1 Cyber-Physical Systems

Cyber-physical systems [Lee08, JCL11] are engineered systems that emerge from the networking of physical (mechanical, electrical, hydraulic, biochemical, *etc.*) and computational (control, signal processing, logical inference, planning, *etc.*) processes that often interact in highly uncertain environments and with human actors. Mundane examples of such cyber-physical systems are interconnected cell phones and fitness trackers, driver assistance systems, industrial manufacturing systems, medical devices, traffic control systems, or smart home appliances. Such systems enable many of our daily activities and they have become innovation drivers in important domains, *e.g.,* automotive, avionics, civil engineering, Industry 4.0, robotics, and more.

Industrial engineering of cyber-physical systems requires the collaboration of experts from heterogeneous domains to solve the challenges in their respective domain, but also to make all domain-specific contributions interoperate, ensure system-wide properties (*e.g.,* safety in automotive, energy-efficiency in robotics, or performance requirements in Industry 4.0). Hence, cyber-physical systems are notoriously complex because they cross domain borders in applications that are often safety-critical.

In general, cyber-physical systems are costly to fully build and maintain. Thus, modeling and simulation are crucial to their engineering as these (1) facilitate the integration

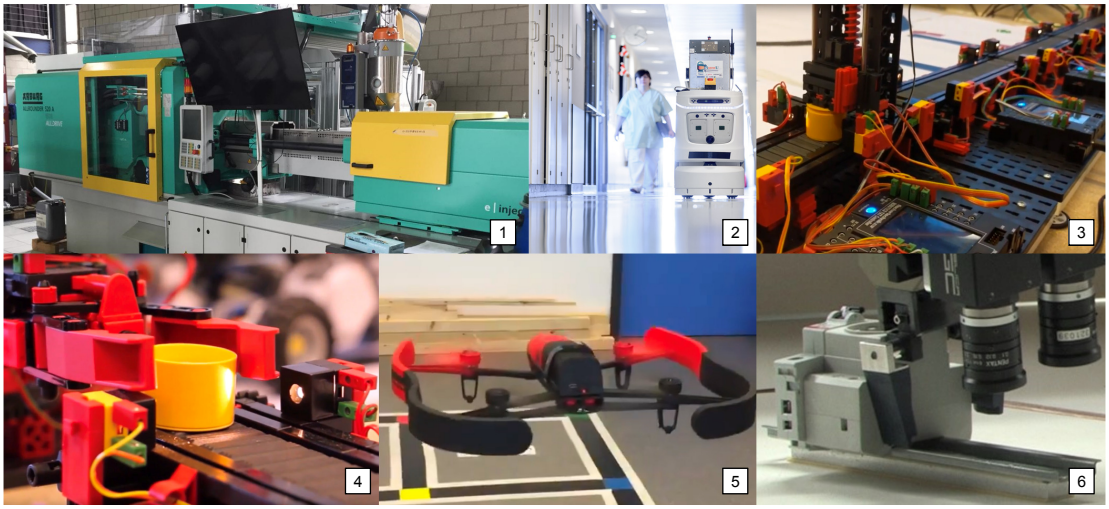


Figure 1.2: Selected cyber-physical systems employed for demonstration and validation throughout the research program presented in this thesis: (1) Arburg Allrounder 520 injection molding machine, (2) FESTO Didactics Robotino 3, (3-4) Fischertechnik Factory Simulation, (5) Parrot Bebop quadcopter; (6) KUKA LWR robotic manipulator.

of domain experts by leveraging modeling techniques using the concepts and terminology of their domain of expertise; (2) can yield the proper abstraction to reconcile and integrate the multi-domain concerns expressed in the domain experts' contributions; and (3) enable the front-loading, *i.e.,* early in the processes, integration of components and validation of system-wide properties before the physical parts of the system are available.

Ongoing research on the systematic development of cyber-physical systems has produced concepts, theories, and modeling languages for the development of their software functions and electronic functions [Alu15, Pto14], as well as for designing [SFA17], engineering [BBL+16], and operating [BSP+16] these systems in different domains. Most approaches focus on modeling through the lens of software engineering, *i.e.,* on discrete and functional systems. Where continuity and geometry are supported, the theories and languages do rarely support established processes or modeling concepts from other domains. Moreover, often the modeling techniques applied to the engineering of cyber-physical systems are overly generic, hampering the effective contribution by domain experts, complicating the integration of their solutions into the overall system, and are a common source for errors. Hence, providing suitable domain-specific modeling techniques for the different concerns of cyber-physical systems and properly integrating these is crucial to the successful industrial engineering of these systems.

Relation to this thesis:

Cyber-physical systems enable parts of our society, are important innovation drivers, and create added-value. Hence, their efficient and systematic engineering is vital to our wealth. This thesis summarizes research results contributing to advancing the model-driven, efficient, and systematic engineering of cyber-physical systems through enabling better modeling techniques in Chapter 3, systematic modeling methods in Chapter 4, and their methodical operation in Chapter 5.

1.4.2 Industry 4.0

Industrial revolutions have always been step changes in manufacturing. The first industrial revolution (18th to 19th century) introduced to machine-driven manufacturing, centralized in factories, and leveraging steam power [Dea79]. In the second industrial revolution (1870 to 1914), electric power replaced steam power and introduced the concept of interchangeable parts to enable the mass production of goods [Mok98]. The third industrial revolution (1970 to 2010) was driven by the transition from analog to (largely isolated) digital production systems. Industry 4.0 is a vision of smart manufacturing that encompasses the complete life cycle of products, from design to ordering to production, distribution, and recycling of its resources [WBCW20], which is considered the "fourth industrial revolution".

Originally, Industry 4.0 has been introduced as a part of the high-tech strategy of the German Federal Ministry for Education and Research [Bun17] in 2011. Since then, it has become an international phenomenon: The Japanese Industrial Value Chain Initiative [IVI18], the Advanced Manufacturing Initiative in the United States [Mol17], the Chinese Made in China 2025 strategy [Mer18], Manufacturing 3.0 in South Korea [Man18], and the national Catapult research center on High Value Manufacturing [CAT18] in the United Kingdom pursue the same or very similar goals.

Industry 4.0 gives rise to new challenges and opportunities for future manufacturing, which are driven by four disruptions: (1) data volumes, computational power, and connectivity; (2) the emergence of novel analytics and business-intelligence capabilities; (3) new forms of human-machine interaction; and (4) improvements in transferring digital mdoels to the physical world, such as 3D printing and advanced robotics. Based on these, stakeholders in Industry 4.0 aim to improve the participating systems towards individualized mass production ("lot size 1"), flexible (real-time) production process control, internationalization of the complete value-added chain, as well as reducing emissions, resource consumption, and cost. These aims yield a multitude of socio-technical challenges, which include addressing the increasing complexity of production processes, integrating heterogeneous and internationally distributed production chains, ensuring the security of connected production systems, the perils of workforce surveillance, standardization of technologies and norms, platformization of solutions, or data governance. To cope with the challenges of the interconnected cyber-physical systems of Industry 4.0, researchers and engineers need to overcompensate the growth in their complexity with new concepts, methods, and tools for their engineering and operation.

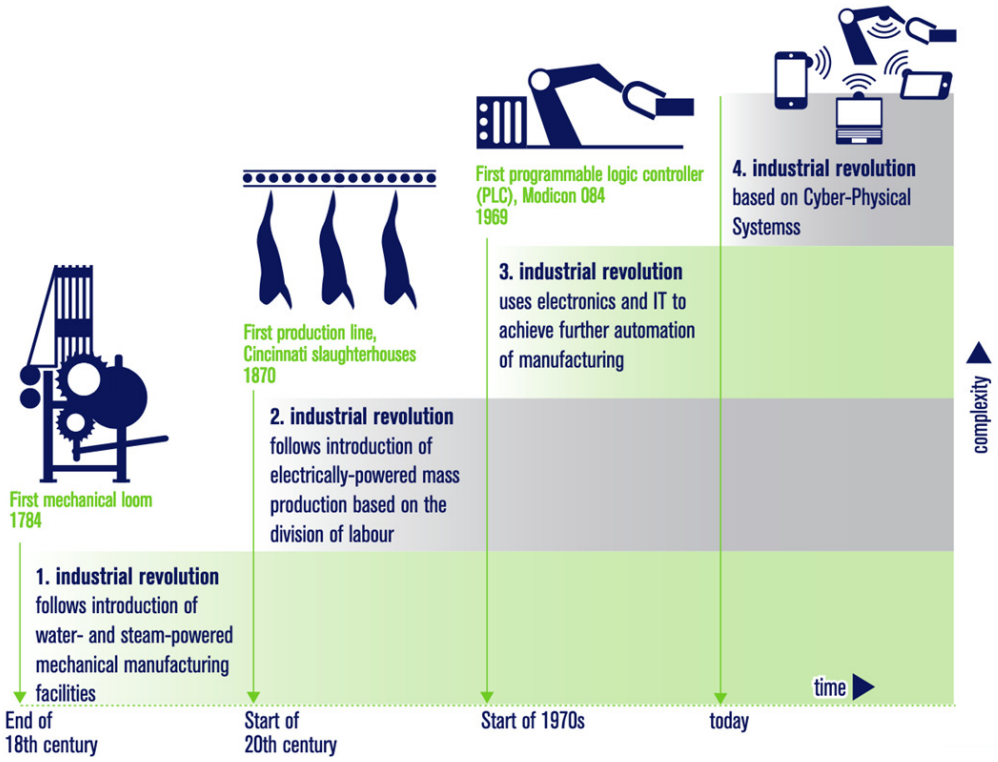


Figure 1.3: The four stages of industrial revolutions [KHHW13].

Relation to this thesis:

Industry 4.0 is an important domain of complex cyber-physical systems in which the successful deployment of systems without the collaboration of a wide variety of experts from different domains is impossible. Consequently, the challenges regarding modeling in Industry 4.0 are of particular interest and contributing to advancing the engineering of cyber-physical systems for Industry 4.0 can act as a litmus test regarding the broad applicability of modeling in other domains. Thus, this thesis reports results from investigating the challenges in modeling for the cyber-physical systems of Industry 4.0 in Chapter 3.

1.4.3 Mobile Robotics

Mobile robots are a particular complex form of cyber-physical systems that have begun supporting many aspects of our professional and personal lives. They restock supplies in automated warehouses, support caregivers in hospitals, assist in manufacturing, secure perimeters, clean floors, mow lawns, and much more. As (partially) intelligent and autonomous systems, their engineering demands expertise from a large variety of domains,

including mechanical engineering, electrical engineering, localization, motion planning, human-machine-interaction, artificial intelligence, and systems engineering.

For the efficient engineering of (mobile) robotics applications, a variety of technologies have been developed in the past that aim to integrate contributions from participating domain experts [JT09, QCG+09]. Most of these, however, focus on GPL artifacts, *i.e.,* expect the domain experts to reify their solutions in program code, which gives raise to accidental complexities [FR07]. Since then, a variety of modeling languages for robotics have been developed as well [NHWW16] By design, these generally address very specific concerns, such as kinematics, path planning, scene understanding, *etc.* and are not prepared for integration into an overall system.

Due to its intrinsic complexity, robotics is a domain of particular interests for systems engineering in which advanced modeling languages, tools, and methods for specific aspects already exist. Yet, the modeling languages and methods in robotics largely are syntactic only, which hampers automated analyses and syntheses necessary for a more efficient engineering of cyber-physical robotic systems.

Relation to this thesis:

Mobile robotic systems are a form of CPS of particular complexity for which are variety of advanced modeling technologies for different participating domains already exist. Through this, investigating the state-of-the-art in modeling for mobile robotic systems gives insights into challenges at the frontier of modeling for CPS. Thus, this thesis summarizes results from investigating these challenges in Chapter 3.

1.4.4 Model-Based and Model-Driven Engineering

The number, importance, and complexity of software functions in cyber-physical systems has increased significantly [FR07, KMS+18]. The increasing complexity of the software of such systems requires novel concepts, methods, and tools that enable overcompensating this growth in complexity and harnessing their potentials. An important reason for the complexity of cyber-physical systems' software is the conceptual gap [FR07] between the problem domain challenges and the solution domain peculiarities. Overcoming this gap with handcrafted solutions requires enormous efforts and raises accidental complexities [FR07], *i.e.,* challenges in the solution domain that the problem domains abstract away from. These accidental complexities increase software and systems engineering risks. Hence, it is paramount to reduce these.

Model-based engineering captures software and systems development methodologies that employ models to increase abstraction. To this end, researchers and practitioners in cyber-physical systems utilize models as communication and development artifacts for various engineering activities, ranging from design, to documentation, requirements modeling, implementation, or deployment. Many successful applications of model-based engineering [BCOR15, HRW11, KR05, Rai05, Sta06, WHR14, WWM+07] have been reported in the context of software engineering for certain domains, such as aviation or automotive. But despite modeling and model-based approaches becoming increasingly

popular in systems engineering, there is no common notion of the term "model" [Sei03, Kü05] as reflected in the different definitions:

- "A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system." [BG01]
- "A model is a set of statements about some system under study." [Sei03]
- "A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made." [Küh06]

Similar definitions of models focus on defining models as simplified abstractions of a system that can replace the system for certain forms of use [Bal00, BD99, HBB+94]. Some of these definitions leave only little more room for interpretation regarding what is modeled. In the following, we adopt a definition in which a model is characterized by three features [MFBC12, Sta73]:

"A model needs to possess the following three features:

- Mapping feature: A model is based on an original.
- Reduction feature: A model only reflects a (relevant) selection of an original's properties.
- Pragmatic feature: A model needs to be usable in place of an original with respect to some purpose."

Moreover, in the following, we distinguish between "model-based" and "model-driven" approaches [BCW12]. The qualification "model-based" characterizes approaches using models for communication, documentation, requirements engineering. In contrast, "model-driven" characterizes approaches where models are the primary development artifacts used for automated analysis and synthesis [Sel03, Sel06]. Hence, in this understanding, the research program summarized in this habilitation thesis contributes to a model-driven engineering and operation of cyber-physical systems.

Relation to this thesis:

Model-based and model-driven engineering are software and systems engineering paradigms that enabled to successfully reduce the conceptual gap between various problem domains and software engineering. This success highly depends on modeling with suitable abstractions, which requires appropriate modeling languages and methods. Thus, the study presented in Chapter 2 investigates the challenges in applying model-based and model-driven approaches to different the cyber-physical systems of Industry 4.0. Afterwards, Chapter 3 summarizes foundations for creating appropriate modeling language Chapter 4 reports novel modeling methods.

1.4.5 Systems Engineering

Systems engineering is an interdisciplinary approach to address the systematic design, integration, and management of complex systems during their life cycle. The term sys-

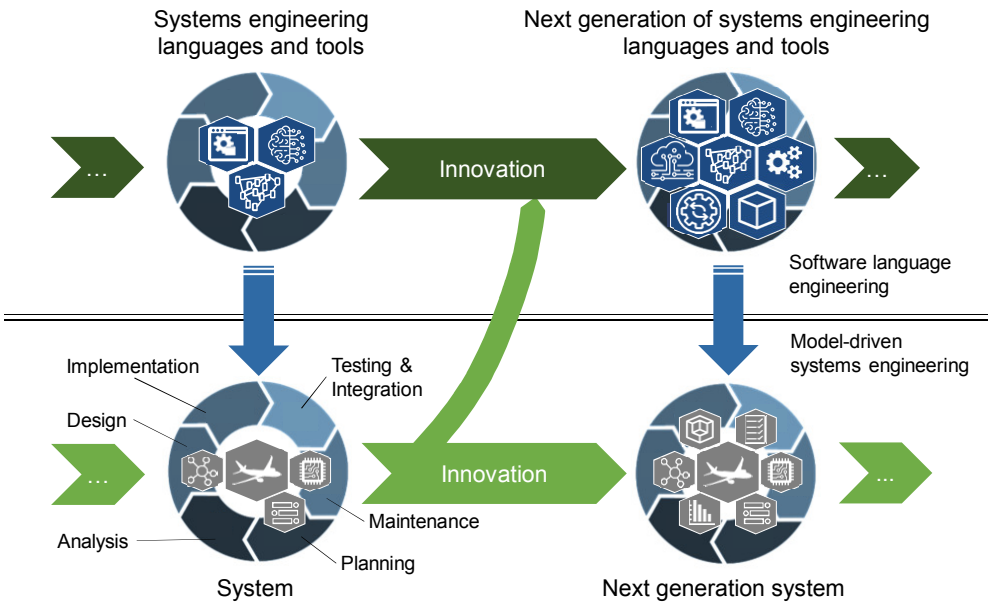


Figure 1.4: Model-driven systems engineering and software language engineering advance each other mutually.

tems engineering was defined as early as 1995 in the NASA Systems Engineering Handbook [SA95] as "a robust approach to the design, creation, and operation of systems". In contrast, modern definitions of the term focus on its interdisciplinary challenges, *e.g.,* according to International Council on Systems Engineering (INCOSE), systems engineering can be defined[1] as "a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods".

Systems engineering addresses concerns such as requirements engineering, design, reliability analysis, dimensioning, programming, logistics, coordination, verification and validation, evaluation, maintainability, and many other issues necessary for successfully realizing complex systems. It aims to ensure that all aspects of a project or system are considered and properly integrated. To this end, systems engineering encompasses processes, and methods. It overlaps with many technical and human-centered disciplines, including industrial engineering, mechanical engineering, control engineering, software engineering, organizational studies, and project management.

Currently, most systems engineering approaches rely on document-based processes and methods, in which non-formal documents, *e.g.,* natural language requirements or sketches without well-defined semantics, are the majority of artifacts. Through domain expertise and tremendous efforts, these are translated into physical components, software components, *etc.* by domain experts. Based on the experience that systems engineering with informal documents does not scale up to the challenges of modern cyber-physical

[1] `https://www.incose.org/about-systems-engineering/system-and-se-definition`

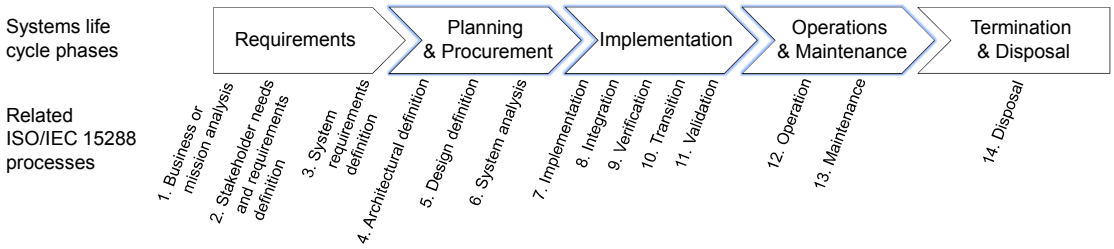


Figure 1.5: Systems life cycle phases and related ISO/IEC 15288 processes

systems, research and practice have begun to employ models in systems engineering.

According to INCOSE [INC07], MBSE "is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases". To formally support engineering across the complete life cycle of a cyber-physical system, document-based approaches have proven inappropriate. Experience in software engineering manifested in the transition from model-based to model-driven methods has shown that only lifting models to primary development artifacts can help to overcompensate the complexities in engineering modern systems. Consequently, MBSE in the terminology used by INCOSE actually must refer to model-driven methods, in which models conform to explicit modeling languages accessible to automation and are considered the primary development artifacts [Lev09]. Thus, the efficient engineering of appropriate domain-specific and semantically well-defined, modeling languages and their systematic application must be prime concerns for successful MDSE.

This also becomes obvious when considering the entangled relation between the cycles of SLE and of systems engineering as illustrated in Figure 1.4: Innovation (such as machine learning) drives systems engineering, which in turn demands suitable modeling techniques (such as differentiable languages [WZD+19]) capable of harnessing this innovation. This in turn drives innovation in software language engineering, which yields the evolution of modeling languages that then enable novel methods in systems engineering that may yield more innovation.

Systems themselves also are subject to a typical life cycle as documented in the ISO/IEC 15288 standard [ISO15] and illustrated in Figure 1.5: After analyzing business or mission requirements as well as stakeholder needs and system requirements, planning of the system begins. During planning the systems architecture and its design are defined and analyzed. Subsequently the system is implemented, which covers development, integration, verification, and validation. After successful validation, the system is deployed, operated, and maintained until it is disposed. The results presented in this thesis focus the phases of implementation and operations.

The research results presented in this thesis focus on improving planning, implementation, and operation of CPS: For all three phases, suitable domain-specific modeling languages are required to reduce the conceptual gap between the participating domain experts' problem space expertise and the required solution domain expertise of software

engineering (*cf.* Chapter 3). The planning & procurement phase then begins with modeling the functional architecture of the system under development (*cf.* Chapter 4). Its design then is systematically decomposed and components are passed to domain experts for implementation. In the implementation phase, these experts use domain-specific modeling languages and system integrators leverage refinement checking to verify the reintegration the domain experts' contributions into the architecture. In this phase, geometric/physical solutions become part of the system architecture and linked to the functional architecture to facilitate tracing of these solutions to system functions and, hence, ease operating with changing requirements and reusing of solutions. In the operations & maintenance phase, the system is deployed and operated. As systems and their behavior change over time, such as due to wear and tear, explicating the domain expertise necessary to compensate the effects of these changes, *e.g.,* the production of defect products, is crucial to optimizing system operations. By providing digital twins that carry this expertise in form of domain-specific models, system use can be improved accordingly. Obviously, such digital twins are software systems that are subject to these life cycle phases as well. Especially, they also yield a functional architecture.

Hence, a central concern in systems engineering is the architecture of the system under development. This architecture is the set of the principal design decisions [MDT07, TMD09], including its fundamental concepts embodied in its elements [ISO11]. Architecture descriptions are products expressing an architecture that can be defined in terms of architecture description languages (ADLs) [ISO11]. Prime concerns of software and system architectures and architecture descriptions are their structure and their behavior [Gro10, FMS14]. To model both, a variety of modeling languages have been developed, including UML [Gro10], SysML [FMS14], and over 120 ADLs [MLM+13]. UML, for instance, is a family of modeling languages for the object-oriented description of the structure and behavior of software architectures. As such, it does not support the representation of geometric-physical concerns in a way useful to experts from these domains. SysML is an extended subset of UML that introduces modeling languages to address requirements modeling and inter-model constraints. Yet, despite its focus on integrated systems modeling, neither SysML nor its successor SysML v2 support the systematic integration of geometric-physical models (*e.g.,* CAD, Modelica, Simulink). Moreover, neither of these modeling language families yields the semantic foundations necessary for effective automated analyses or syntheses.

In computer science, theories and well-defined mathematical frameworks for the syntactic and semantic description of system architectures, such as communicating sequential processes [Hoa78], Focus [BS01], π-calculus [Mil99], have been devised. Focus [BS01, RR11], for instance, is a mathematical framework and semantic foundation for distributed interactive systems that describes logical component & connector [MT00] architectures as stream-processing functions. The stream-processing functions of Focus describe the histories of messages exchanged over communication channels between the interfaces of the components and support underspecification. Based on this foundation, refinement and decomposition are compatible, *i.e.,* beginning with an underspecified initial architecture, this architecture can be stepwise semantically refined and decomposed

into a hierarchy of components presenting a deterministic implementation.

Notwithstanding the existence of semantically well-defined theories for systems modeling, most systems modeling languages focus on syntax, leaving giving semantics (meaning) to the models to individual tool vendors or do not support the integration of concerns from the physical domains (chemical engineering, electrical engineering, mechanical engineering, *etc.*). This prevents effective modeling processes, leveraging automation, and efficient collaboration with domain experts.

Relation to this thesis:

Systems engineering still is a predominantly document-based activity that covers the complete lifecycle of systems from eliciting their requirements to their disposal. The research results in this thesis forge a bridge from the contemporary document-based systems engineering to a model-driven systems engineering by establishing pillars for the model-driven implementation and operation of CPS.

MontiArc

MontiArc [HRR12, RRW14, BKRW17a] is an ADL for modeling the functional architecture and behavior of CPS based on the semantics of Focus [BS01]. In MontiArc, architectures consist of hierarchically composed components that reify the stream-processing functions of Focus and yield with stable interfaces of typed ports through which they communicate with other components. It distinguishes component types from their instances, supports component instance configuration, and features generic type parameters to flexibly adjust port types before reusing component types in other contexts.

Atomic components feature state-based behavior models (*e.g.,* time-synchronous port automata (TSPA) [RRW14]) to describe their behavior and composed components define configurations of subcomponents from which their behavior emerges. MontiArc can be used to design architectures, refine these semantically into implementations, and, ultimately, translate these into executable GPL artifacts. MontiArc models, architecture descriptions [ISO11], are used in a variety of domains, including automotive [HKM+13], cloud [NPR13], machine learning [KPRS19], robotics [ABH+16], and more [BHH+17].

MontiArc is realized as a MontiCore (*cf.* Section 1.4.6) modeling language. It consists of a context-free grammar (CFG) with Java well-formedness rules, and a template-based code generator that translate MontiArc models to various diverse languages, including C++, Java, and Python. It leverages MontiCore's language composition mechanisms to define the data types of ports in terms of UML/P [Rum16, Rum17] class diagrams that are aggregated with MontiArc's components. Moreover, it leverages language embedding to enable the integration of DSLs into components. Together with the natural encapsulation of concerns into components, this enables systems engineers to easily decompose the functional architecture of a cyber-physical systems under development into components. Through embedding of the most appropriate DSLs for component behavior modeling [BHH+17], these components can be implemented efficiently by domain experts using the most appropriate DSLs to this effect.

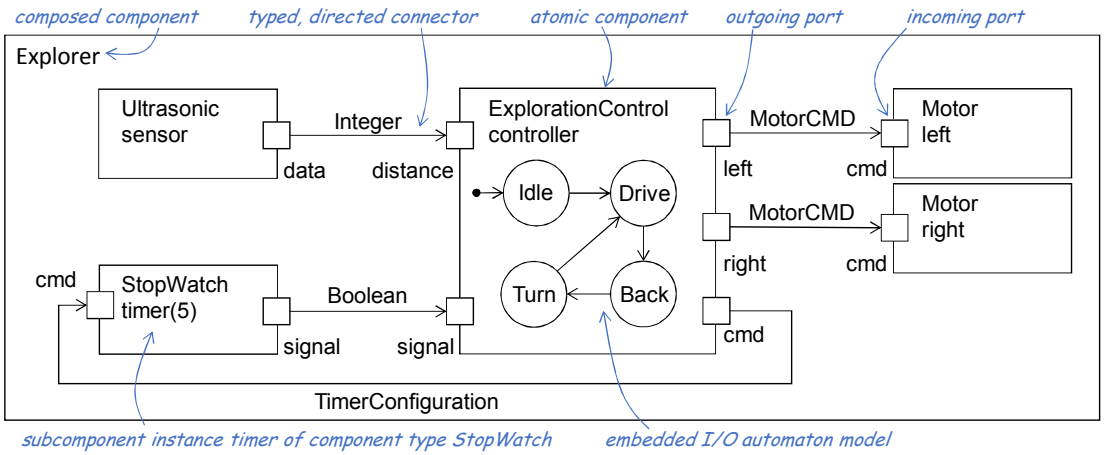


Figure 1.6: Example MontiArc architecture comprising five subcomponents of four different component types and a state-based behavior model in its central controller.

Relation to this thesis:

MontiArc is an advanced modeling technique for functional architectures and behavior of CPS. Through the language engineering techniques summarized in Chapter 3, modular variants of it have been used to enable the analyses presented in Chapter 4 and the operations recapitulated in Chapter 5.

1.4.6 Software Language Engineering

Software language engineering [Kle08, HRW18] is a field of research on investigating the engineering, maintenance, evolution, and reuse of software languages, *i.e.,* languages reified in software. The objects of SLE research include GPLs, textual or graphical modeling languages, external or internal [BDL+18] DSLs [Hud98], and more. In the following, we will use the terms "modeling language" and "domain-specific language" interchangeably as considering a language as domain-specific is in the eye of the beholder, *i.e.,* the distinction between general and domain-specific modeling languages is highly subjective. Where evident from the context, we will refer to "modeling languages" simply as "languages". Moreover, we also do not distinguish between "technical DSLs" and "application domain DSLs" [VBD+13].

Extensionally, a language can be considered as the set of sentences it comprises. Both, for more sophisticated reasoning than on the membership of sentences and for a constructive application, a more fine-grained, intensional, definition is required. Hence, research has conceived various definitions of languages definitions focusing on their constituents and meaning.

For instance, a *language definition* may consist of [CGR09] (1) a concrete syntax, *i.e.,* the form of the sentences of the language, (2) an abstract syntax, *i.e.,* the structure of the sentences of the language, (3) a semantic domain, which typically is a well-understood mathematical theory, and (4) a semantic mapping relating elements of the abstract syntax to elements of the semantic domain, which gives meaning [HR04] to sentences of the language. Another approach to characterize language definitions [CBCR15] comprises (1) a concrete syntax, (2) an abstract syntax, (3) its static semantics, *i.e.,* its well-formedness rules, and (4) the meaning of the sentences. We refer to the constituents realizing concrete syntax, abstract syntax, static semantics, and semantics in the sense of meaning [HR04] as *dimensions* of languages.

Leveraging SLE, many popular languages have been developed, such as Gradle [Ikk15], Kotlin [SB17], Matlab Simulink [Cha15], Modelica [EOH+09], SparQL [SP07], or Verilog [TM02]. Depending on the observer's perspective, these modeling languages can be considered domain-specific (*e.g.,* concerning the domain of software engineering) or not.

Similar to the term "model", research in computer science has produced various characterizations for DSLs, such as:

- "A domain-specific language is a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [vDKV00]
- "By focusing on a problem domain's idioms and jargon, DSLs avoid the notational noise required when using overly general constructs of a general-purpose language to express the same thing. Moreover, DSLs are not necessarily programming languages: they are languages tailored to express something about the solution to a problem." [Wil01]
- A "DSL is a language designed to be useful for a limited set of tasks, in contrast to general-purpose languages that are supposed to be useful for much more generic tasks, crossing multiple application domains." [JB06]

Notwithstanding the distinction between modeling languages and DSLs, employing such languages promises benefits regarding productivity and quality as DSLs may provide a platform-independent "thinking and communication tool" [VBD+13] that can leverage the concepts and terminology the experts using the DSL are familiar with. Thus, the availability of appropriate DSLs can greatly facilitate the engineering of cyber-physical systems.

Language workbenches [ESV+13] are software tools facilitating the engineering of DSLs. To this end, they often provide specific (meta-)modeling languages that enable describing the syntax or semantics of modeling languages. For the syntactic dimension, metamodels and grammars are widely applied to language engineering: For instance, Ecore of the Eclipse Modeling Framework [SBMP08] is a modeling language for the specification of metamodels. A metamodel of a language prescribes the structure of the languages' abstract syntax in terms of classes, their properties, and relations. Models (sentences) conforming to these metamodels are considered valid elements of the implied

language. The GEMOC Studio [CBW17] language workbench and others leverage Ecore for the definition of abstract syntaxes and provide features for other dimensions based on it, such as debugging, editing, or interpretation. Other language workbenches, such as MontiCore [HR17], Neverlang [VC15], or Xtext [Bet16] leverage grammars as syntax metamodeling languages instead. These grammars support prescribing the integrated concrete and abstract syntax of languages through their productions as the set of derivable sentences. Often, these syntax modeling languages are context-free [Cho56], *i.e.,* they cannot express arbitrary well-formedness restrictions. For the dimension of well-formedness, language workbenches most often employ either Object Constraint Language (OCL) constraints [HJK+10] or GPL context conditions [HR17, VC15]. Regarding semantics, language workbenches often focus on model transformations that either are model-to-model (M2M) or model-to-text (M2T) transformations. M2M transformation techniques, such as ATL [JABK08], epsilon [KPP08], or MOLA [KBC04], define translations of models between languages or within a language whereas M2T transformation techniques, such as FreeMarker [For13], Velocity [SvB02], or Xtend [Bet16], describe the transformation of models into other textual representations.

M2M transformations yield various benefits, *e.g.,* the knowledge of the target language encoded in the transformations can ensure that these always yield syntactically correct models and may optimize the resulting models en passant. However, M2M transformations require specific transformation languages [HHRW15, JAB+06], which usually are little domain-specific and, hence, demand domain-experts to learn a new modeling language. Lately, research in SLE has brought forth the notion of domain-specific transformation languages, which are generated based on an input DSL and leverage its domain-specific syntax [HRW18]. This can reduce the effort of employing M2M transformation in practice. Another challenge in M2M transformations is in the fact that the abovementioned correctness by construction only becomes possible because M2M transformation languages require that both the language of the input models and the language of the output models have been made explicit, *i.e.,* reified in software. Where the output language is not available, M2M transformations are of little use.

For M2T transformations, the output language is not required. Instead, these transformations aim to produce textual output models that resemble output language models, *e.g.,* GPL code artifacts, which enables these to be processed by tooling of the output GPL without additional effort. Hence, the transformation user must not know the abstract syntax of the output language, but only its concrete syntax. As M2T transformations are unaware of properties of the output language, they cannot ensure syntactic correctness of the results. However, not requiring an explicit representation of the target language and the relative ease of learning M2T transformations yields practical benefits. Therefore, the remainder of the thesis will leverage M2T transformations and use the terms “code generator” and “M2T transformation” accordingly. Especially, this entails that a “code generator” does not need to produce GPL code, but can produce any textual representation.

Overall, through the use of various metamodeling languages, language workbenches span multi-dimensional technological spaces [KBA02], *i.e.,* working contexts with as-

sociated concepts, bodies of knowledge, tools, and skills. Leveraging these, a variety of modeling languages have been produced for, *e.g.,* automotive software engineering [HF07a, DGH+19] avionics [EP11], business processes [GCD09], Internet of Things applications [GEVD14], manufacturing systems and processes [WBCW20] software engineering [Gro10, MLM+13], robotics [THR+13, NHW14], and many other application domains in different technological spaces. For most of these, novel implementations of the various language dimensions had to be developed. As software languages are software too [FGLP10], these are subject to the same complexities as general software engineering and impose additional complexities through the various meta levels and metamodeling technologies language engineers have to operate with.

Relation to this thesis:

Software language engineering is an essential foundation of MDSE that enables leveraging the potential of DSLs in the engineering of cyber-physical systems. DSLs enable domain experts to contribute solutions using familiar concepts and terminology while at the same time reducing the conceptual gap between the domain experts and the solution domains. Through the automated processing of models of these DSLs, *e.g.,* in form of model checking or M2T transformations, engineering of cyber-physical systems can become more efficient. To expedite the MDSE of cyber-physical systems, this thesis presents investigations on the state-of-the-art in applying modeling languages to the engineering of such systems in Chapter 2 and reports the contribution of novel methods and concepts for the systematic engineering of DSLs in Chapter 3.

MontiCore

MontiCore [GKR+08, HR17] is a language workbench for the efficient engineering of textual languages. Essentially, MontiCore employs CFGs for the integrated definition of abstract syntax and concrete syntax of modeling languages that defines which models are principally possible. From each CFG, MontiCore generates model processing infrastructure, including the abstract syntax classes representing textual models in machine-processable form, parser, lexer, a model checking framework, and a template-based code generation framework [Kra10, HR17]. The infrastructure generated from a CFG translates textual models conforming to the syntax specified in the CFG into instances of the generated abstract syntax classes and enables checking their well-formedness, as well as transforming these into arbitrary textual target representations, such as (executable) GPL code artifacts. As the language defining the syntax of MontiCore languages is a MontiCore CFG, MontiCore can bootstrap itself.

To check properties not expressible with CFGs, *e.g.,* whether a model features two elements of the same name, MontiCore supports extending its model-checking framework with context conditions. These context conditions are well-formedness rules specified in Java, that relate to the abstract syntax classes generated from the CFG of the language they belong to. To transform syntactically conforming and well-formed models into other representations, MontiCore's code generation framework can be extended

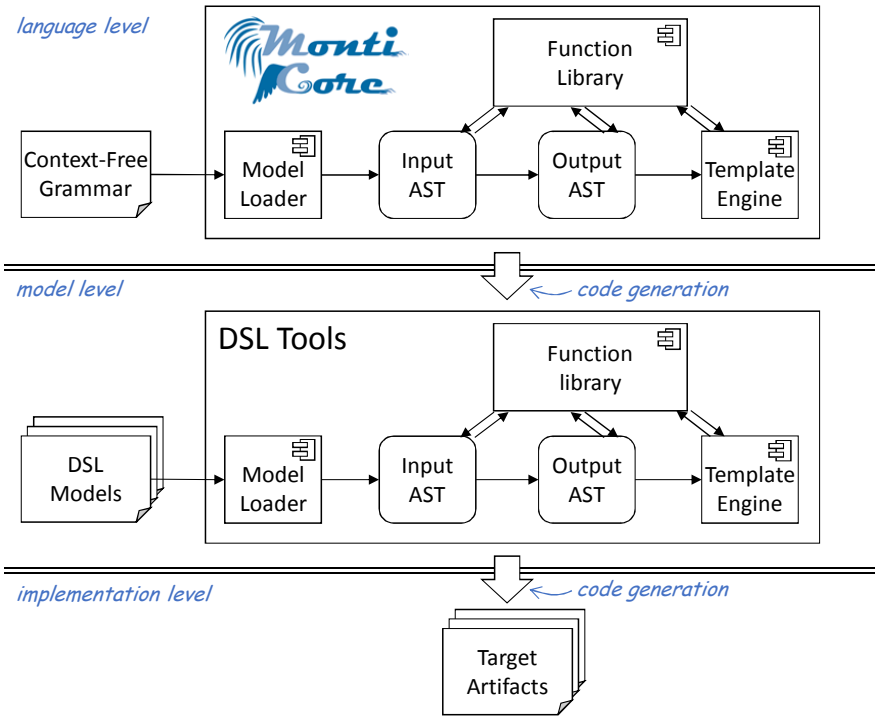


Figure 1.7: Quintessential artifacts and components of MontiCore.

with FreeMarker [For13] templates describing these transformations. In these templates, FreeMarker control structures relating to properties of the abstract syntax tree (AST) govern the use of target language text fragments to produce target language artifacts.

Figure 1.7 illustrates these parts of MontiCore's toolchain: Using MontiCore's parser for CFGs, the CFG of a DSL under development is loaded as a model of MontiCore's CFG language. Through this, MontiCore instantiates the abstract syntax classes of its CFG DSL and obtains the abstract syntax tree (AST) of the CFG. The AST is processed by MontiCore's various functions, including M2M transformations and well-formedness checking, and then passed to a template engine. The template engine translates the AST into model-processing artifacts ("DSL tools"), such as parser, AST classes, *etc.*, for the DSL specified in the CFG. The generated DSL tools then can parse models conforming to that CFG into their specific ASTs conforming to the abstract syntax classes generated from the aforementioned CFG. Afterwards, the DSL tools process these with their functions and then pass the processed AST of the model to the DSL-specific template engine. Being extended with templates describing the transformation from models of that DSL to a textual target representation, the template engine then produces the desired target artifacts.

To facilitate engineering modeling languages, MontiCore supports compositional language integration in form of language extension, language embedding, and language aggregation [HLMSN+15, HR17]. Language aggregation is the combination of multiple

independent languages into a collection ("family"). This enables describing models for different aspects in separate artifacts that can be processed together. This, for instance, is essential when experts of different domains employ different modeling techniques, such as CAD [GZ83], SysML block definition diagrams [FMS14], or Simulink [Cha15] to describe parts of the same system. In this case, language aggregation enables the SysML blocks to refer to properties of the CAD models [DJR+19] or the Simulink models and vice versa. Language embedding combines languages such that their elements can be used in a single integrated model. By embedding domain-specific language parts (*e.g.,* expressions, statements, *etc.*) into extension points of, possibly domain-independent, base language (*e.g.,* automata, architectures), this enables reusing the domain-independent modeling concepts and related tooling in a variety of domains. Thus, language embedding is a mechanism for planned language reuse that facilitates creating DSLs. Language extension enables extending, refining, or even restricting existing languages. To this effect, the CFG of a new language can extend multiple CFGs and reuse, refine, or restrict their productions independent of any extension points in the parent CFGs. Hence, language extension can be considered a generalization of language embedding that greatly facilitates creating DSLs through opportunistic reuse.

Relation to this thesis:

MontiCore is a powerful language workbench for the modular engineering of reusable DSLs. We employ MontiCore as a vehicle to experiment with modeling languages, evaluate concepts, and realize demonstrators. Hence, we realized modeling language engineering concepts and their toolchain presented in Chapter 3 as well as MontiArc [BHH+17, BKRW17a], the modeling methods employing it described in Chapter 4 and the DSLs reported in Chapter 5 with MontiCore.

1.5 Terminology

The results presented in this thesis are based on research in software language engineering and model-driven systems engineering. To ensure a consistent terminology, this section presents the most frequently used terms and principles and our current understanding of these.

Architecture Description Language (ADL) A modeling language for the description of software architectures.

Artifact An artifact is an individually storable unit with a unique name that serves a specific purpose in a software engineering process.

Cyber-physical system (CPS) Engineered systems that emerge from the networking of multi-physical (biochemical, electronic, hydraulic, mechanical, *etc.*) and computational (control, signal processing, logical inference, planning, *etc.*) processes, often operating in highly uncertain environments that include human actors.

Domain expert Person with expertise in an application domain. Often without formal software engineering education.

Domain-specific language (DSL) A language for a specific domain [CBCR15].

General-purpose programming language (GPL) A programming language that can be applied in any domain.

Language A means for communication between stakeholders (including humans and machines). It describes the set of possible sentences that can be exchanged between these stakeholders [CBCR15].

Language Component A reusable encapsulation of a, possibly incomplete, language that includes a language definition and might include explicit language interfaces [CBCR15].

Language Definition A language is defined by a concrete syntax (the form of the language's sentences), an abstract syntax (the structure of its sentences), and semantics (meaning [HR04]) [CBCR15].

Language Interface An abstraction of a provided or required part of a language component for a specific purpose [CBCR15].

Model An element of a modeling language. Each model is a purposefully abstracted representation of an original [Sta73].

Model-based A paradigm in which descriptive or prescriptive models are to support the engineering of software and systems, *e.g.,* for design-space exploration, documentation, or requirements elicitation.

Model-driven A paradigm in which descriptive or prescriptive models are the primary development artifacts for the engineering of software and systems, *e.g.,* for simulation, dimensioning, or generation of solution parts.

Model-based systems engineering (MBSE) The application of modeling to support system design, analysis, verification, validation, deployment, and operation throughout all life cycle phases.

Model-driven engineering A software engineering paradigm that considers models as primary development artifacts (*i.e.,* blueprints) and model transformations, code generators, or interpreters to transform these blueprints into solutions directly.

Model-driven systems engineering (MDSE) The vision of the pervasive, systematic application of DSLs that leverages models as the primary development artifacts to support system design, analysis, verification, validation, deployment, and operation throughout all life cycle phases.

Modeling Language A language for modeling.

Semantics Meaning of a model [HR04]. May be specified denotationally [Mos90], operationally [TP97], or translationally [LL81].

Software engineering The systematic application of scientific and technological concepts, methods, and experience to the design, implementation, validation, documentation, and deployment of software.

Software language engineering (SLE) The application of systematic, disciplined, and quantifiable approaches to developing, using, and maintaining modeling languages.

Systems engineering The interdisciplinary approach towards the successful realization, use, and retirement of systems, using systematic engineering methods.

1.6 Thesis Outline

The remainder of this thesis is structured as illustrated in Figure 1.8: Chapter 2 summarizes a study on the state-of-the-art in modeling for CPS and raises requirements on modeling language engineering, systems modeling, and systems operation. Chapter 3 to Chapter 5 summarize research results on modeling language engineering, systems modeling, and systems operation for CPS respectively. Chapter 6 concludes.

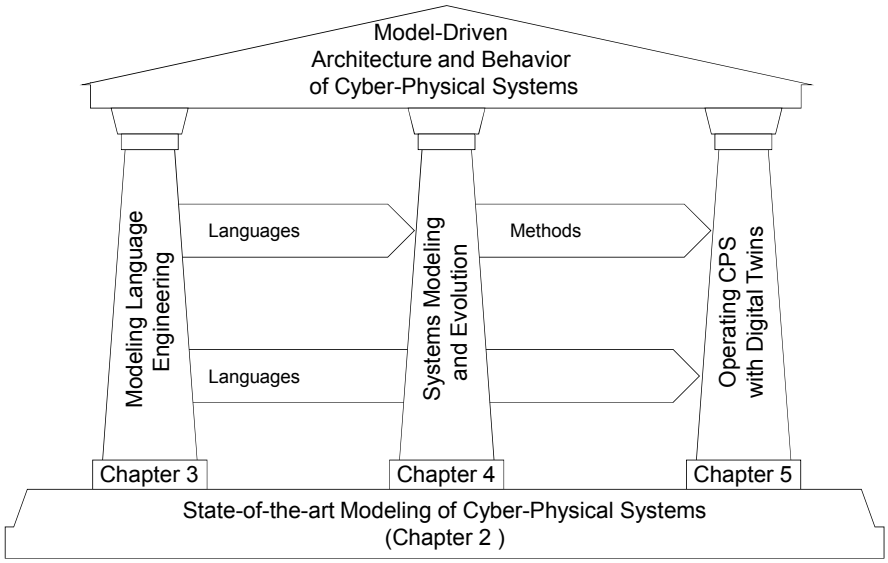


Figure 1.8: Structure of this thesis: Chapter 2 identifies requirements for modeling language engineering, systems modeling, and systems operation. Chapters 3 to 5 address these challenges. The foundations and all three pillars contribute to the model-driven engineering of CPS with domain experts.

In detail, the chapters are as follows:

Chapter 2 summarizes results from a systematic mapping study on the application of modeling in the engineering of the smart manufacturing systems of Industry 4.0. Its results describe the methods, findings, and conclusions that lead to research activities yielding the results presented in the subsequent chapters.

Chapter 3 presents methods for the systematic and efficient engineering of DSLs through the composition of independently developed DSL components in language product lines and subsequent customization. The main focus of this chapter is laid on a novel conceptual model of holistic DSL components, and their use in resolving closed variability of the language product lines, as well as for subsequent opportunistic customization. To this end, the DSL comprising realizations of syntax and semantics behind stable interfaces exposing provided and required extensions of that DSL component and we present a method for composing these according to product line specifications and customization specifications. The results outlined in this chapter are driven by insights presented in Chapter 2. They facilitate the engineering of domain-specific modeling languages that exploit concepts and terminology of the application domains contributing to the modeling of CPS to ease integration of domain experts into CPS engineering.

Chapter 4 describes modeling methods for engineering the functional architectures of CPS, for relating these functional architectures to the geometric-physical realizations of the cyber-physical systems, for automatically decomposing the functional architectures prior to distributing these to domain experts, and for automatically checking their stepwise refinement during evolution. Hence, this chapter introduces a novel conceptual model that bridges the functional view of computer scientists with the geometric view of mechanical engineers. The functional architectures are represented as functions of interconnected channels, which enables their semantic processing for automated decomposition and refinement checking. The results presented in this chapter are motivated by findings presented in Chapter 2. Moreover, they rely on the modeling language engineering achievements presented in Chapter 3 and foster the systematic modeling of CPS.

Chapter 5 presents the results of a survey and subsequent workshops on the nature of digital twins as well as the resulting definition of digital twins. Based on this, it suggests a functional and extensible architecture for digital twins that can leverage the methods presented in the previous chapter and employs models of novel DSLs facilitating integrating digital twins with data sources and manufacturing systems. Moreover, it presents integrated modeling methods to incorporate domain expertise into digital twins that is tailored to capture such expertise in a rule-based fashion and enables the digital twin to control and optimize the behavior of the represented system. The findings summarized in this chapter also are based on research driven by the outcomes presented in Chapter 2. They leverage the modeling language engineering results presented in Chapter 3 and exploit the modeling methods recapitulated in Chapter 4.

Chapter 6 concludes the thesis, summarizes its results, and briefly sketches open challenges and future research opportunities.

Appendix A describes the author's contribution to the publications summarized in this thesis.

Appendix B comprises reprints of full versions of the selected papers presented in this thesis.

All figures in the following sections are based upon similar figures that appeared first in the publications summarized in the respective section. Only their appearance has been harmonized for use in this thesis.

Trademarks appear throughout this thesis without any trademark symbol. They are the property of their respective trademark owner. There is no intention of infringement. The usage is to the benefit of the trademark owner.

Chapter 2

Modeling Languages for Cyber-Physical Systems

The limits of my language mean the limits of my world.

Ludwig Wittgenstein

Many domains have started adopting modeling to harness the increasing complexity of engineering various cyber-physical systems. Yet, many of these approaches are ad-hoc and lack the systematic rigor necessary to realize the vision of model-driven systems engineering. To uncover the modeling limits in the world of cyber-physical systems' engineering, this chapter summarizes answers to the first objective (RQ1) of this thesis regarding state-of-the-art of availability, use, and limitations of modeling for cyber-physical systems in Industry 4.0 through the lens of modeling languages. This chapter, therefore, presents the results of a systematic mapping study on modeling for cyber-physical systems in Industry 4.0. The challenges identified in this study drive the research on software language engineering, modeling methods, and operations summarized in the subsequent chapters. The study, presented in Section 2.1, investigates the use of modeling in Industry 4.0 and the domain-specific challenges the modeling languages employed in this field are applied to. It sheds light on the use of (domain-specific) modeling languages and techniques to address concerns in the engineering of cyber-physical systems and signposts research directions. Section 2.2 summarizes its results.

2.1 Modeling in Industry 4.0

Industry 4.0 is driven by technological disruptions, such as improved management and analysis of large amounts of data, increased connectivity, and enhanced autonomy of systems, that enable its four design principles [HPO16]: (1) Interoperability of production systems, processes, and people; (2) Information transparency based on large amounts of run-time data collected from shopfloor sensors; (3) Technical assistance in providing the right abstraction of the collected data to understand the complexity of Industry 4.0 systems and processes; and (4) Decentralized decision-making which enables automated manufacturing systems. As MDE has successfully been employed to address such challenges with domain-specific modeling languages and methods in the past, we investigate modeling for the cyber-physical systems of Industry 4.0 through the lens of

modeling languages and relative to diverse challenges of smart manufacturing. Through the systematic mapping study presented in this section, we aim to provide guidance and feedback for the modeling community about challenges for their research and the reception of their contributions in Industry 4.0.

Systematic mapping studies are a form of systematic literature research are an established method to investigate research landscapes. In software engineering [PFMM08], *e.g.,* they have been applied to cartograph the research landscapes on software development effort and cost estimation [JS07], on the use of experimental studies [SHH+05], object-oriented design [BBT+07], on the usage of UML diagrams [PB08], on software product lines [ER11, LC13], and on domain-specific languages [KBM16].

Moreover, this study provides an overview for the automation systems community about the contributions to modeling languages and modeling techniques in their domain. To this end, it details which challenges these modeling languages and techniques address. Its results enable identifying limitations and challenges, as well as best practices and new lines of research regarding modeling for Industry 4.0. Also, it provides a corpus for future investigation.

MDE is one of the key enablers for successfully addressing the challenges of Industry 4.0 regarding engineering, integrating, and maintaining the cyber-physical systems realizing this vision as indicated by the increasing number of related publications in key conferences and journals investigating these challenges [SUN+17, CBWM18, FT17, HGS+16, MDWA17, SZ16]. Yet the role of modeling in engineering cyber-physical systems for Industry 4.0, its potential use for integrating domain contributions, and challenges related to modeling in this domain were not systematically investigated.

To support researchers and practitioners in MDE and SLE in directing the focus of their work and to support engineers in identifying suitable modeling languages and tools for challenges in practice, we conducted a systematic mapping study on modeling in Industry 4.0. In this study, we analyzed literature on modeling in Industry 4.0, the concerns addressed with different modeling techniques and languages in this literature, and identify directions for future research. This section presents results from that study.

Section 2.1 is based on the publication:

Paper 1 A. Wortmann, O. Barais, B. Combemale, M. Wimmer. Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study. In: J. Gray and V. Kulkarni, editors, Software and Systems Modeling, 19(1), pages 67-94, Springer, 2020.
Reference: [WBCW20]

Context Engineering cyber-physical systems is an interdisciplinary endeavor [BLG17, CCQB17, MKBZ16] that brings together experts from a variety of domains, such as mechanical engineering, electrical engineering, automation engineering, factory planning, architecture, software engineering, and many more. These experts are trained in different ways to conceive, describe, and solve the challenges in their domains. All of these domains are employing modeling, sometimes even for decades, using domain-specific

terminology, concepts, and methods, including technical drawing [HQ82], circuit diagrams [Ung96], architecture drawings [BWHR08], Simulink [Bis96], UML [UML20], and more.

When collaborating, the different specific approaches, concepts, and methods employed by experts of these domains give rise to a wide conceptual gap [FR07] between their different solution domains that needs to be closed to integrate their contributions to functioning cyber-physical systems. Model-based systems engineering aims to close this gap by making a central system model the single source of truth for integrating the models contributed by domain experts that realize domain-specific solutions. This demands understanding modeling for cyber-physical systems in Industry 4.0, the modeling languages applied, and the concerns addressed with these languages. Systematic literature research can contribute to this understanding. Yet, such studies on modeling are rare and focus on specific automation techniques [MJ13] or on specific engineering concerns of cyber-physical systems [GT18, NAY17]. There are, however, empirical studies on the application of modeling in software engineering [ASSS13, LMT+14, TTR+13]. These conclude that practitioners in cyber-physical systems already leverage modeling [LMT+14], despite a "lack of skills" and "lack of tools" [ASSS13], and that the automated translation of models into programming language artifacts is widely employed [TTR+13]. While all of these studies focus on cyber-physical systems, they are restricted to modeling in software engineering and, hence, cannot contribute much to a broader understanding of modeling for cyber-physical systems in Industry 4.0.

Literature research in Industry 4.0 does not focus on modeling and its applications, but usually on broad, systemic concerns, such as deficiencies in current Industry 4.0 research [LDdP17], challenges for the domain in general [VHH16], for companies employing Industry 4.0 techniques [KT16], the state of related standards [TTG+16], the design principles of Industry 4.0 [HPO16], or key technologies enabling Industry 4.0 [Lu17]. Hence, despite modeling, as the act of prescribing the systems-to-be, being one of the key approaches in the individual domains contributing to engineering the cyber-physical systems of Industry 4.0, a study providing systematic evidence on modeling in this context was missing. Thus, until today, we still have little knowledge about how the different (domain-specific) modeling techniques employed by domain experts are used to address which concerns regarding engineering cyber-physical systems for Industry 4.0.

Contribution To improve the understanding of modeling for cyber-physical systems in Industry 4.0, what the expected benefits of modeling are, where established modeling techniques can be applied to the design principles of Industry 4.0, and where novel techniques are required, we conducted a systematic mapping study on modeling in Industry 4.0 following established guidelines [PFMM08]. Through this mapping study, we characterized the state of the art of modeling in Industry 4.0 in a broad sense, which includes modeling techniques outside of traditional software engineering and branches into modeling in systems engineering, such as 3D modeling, knowledge representation, business process modeling, and more. To this effect, we collected a total of 4404 documents from ACM Digital Library, Google Scholar, IEEE Xplore, Scopus, SpringerLink, and Web of

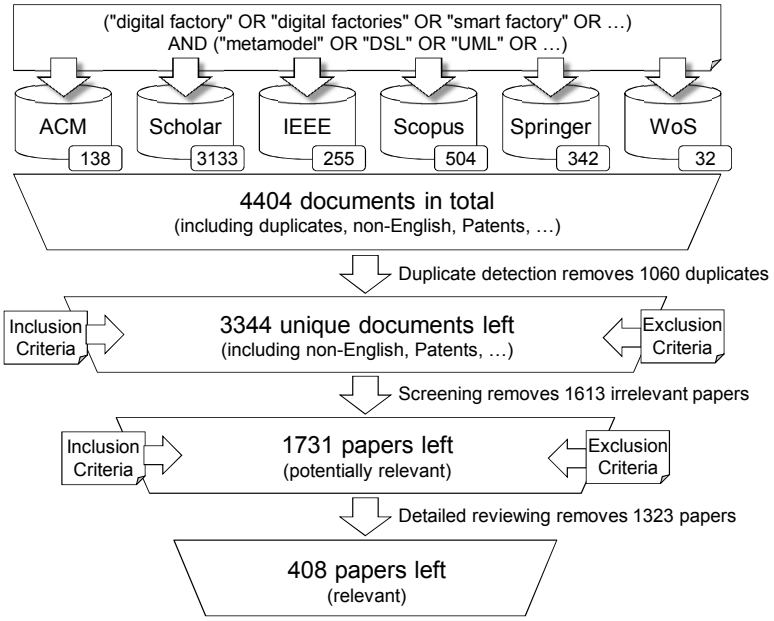


Figure 2.1: Beginning with 4404 documents returned from applying our search query to six mayor scientific literature databases, we ultimately identified 408 relevant publications for analysis.

Science that feature keywords suggesting discussing modeling in Industry 4.0. Following a systematic mapping process, as illustrated in Figure 2.1, through screening, filtering, and detailed reviewing, we identified a corpus of 408 unique publications describing contributions to this topic. Through keyword-based clustering, we obtained as classification framework including the parameters (1) Industry 4.0 concern: ten clusters ranging from the digital representation of systems to integration to their verification and validation; and (2) modeling technique: 15 clusters of modeling techniques applied in this context. The clusters are centered around

1. 3D modeling techniques, applied to product lines [SLF17], metal forming [ZLZ17], or virtual robotics [GV15];
2. Various Architecture Description Languages (ADLs), such as AADL [FG12] or EAST-ADL [DSLT04];
3. The Automation Markup Language (AutomationML) [DLPH08];
4. Business Process Modeling (BPM) [RRIG09];
5. The Core Manufacturing Simulation Data (CMSD) specification standard [RL10];
6. Various DSLs, *e.g.,* Automax [SGBvB12] for robotics motion modeling, the ETRI CPS Modeling Language (batch) Process Recipe DSL (PRD) [MBS16];
7. Entity-Relationship (ER) modeling [Che76];
8. Formal methods, including mathematical frameworks to reason about manufacturing decisions [DMMP17] and Petri nets applied to product configuration [ZR13] or

to risk analysis [FZG17];
9. Modeling with GPLs to describe the kinematic of robot manipulators [HAS+14] or information models of control systems [MW00];
10. Different knowledge representation techniques, such as the Web Ontology Language (OWL) [AVH04], *e.g.,* applied to robotic kitting [HTKR+15];
11. Metamodeling techniques used to define and arrange concepts, *e.g.,* to describe task-oriented production [BR17] or potential situations that cyber-physical systems could encounter [MHWM17];
12. Modeling of physical properties with Simulink [DH04];
13. Systems description and integration with SysML [FMS14];
14. Software modeling with UML [Gro10]; and
15. XML, *e.g.,* applied to the specification of cyber-physical system properties [LSR+19] and to the integration of Industry 4.0 components [MW17].

Based on these clusters, we analyzed which modeling techniques are used to address which Industry 4.0 concerns.

We found that modeling in Industry 4.0 leverages DSLs (21% of the contributions) as well as UML (18%), SysML (6%), and their variants, and knowledge representation techniques (14%) to describe the cyber-physical systems and their use. The use of metamodeling and DSLs, as well as UML profiles or other extension mechanisms, might suggest that specific concepts or their integration are not yet sufficiently supported by wide-adopted modeling languages or not properly integrated. UML and DSLs are most often used to address challenges in digital representation and integration of Industry 4.0 systems, which is consistent with identifying these as the most urgent challenges regarding modeling in Industry 4.0. That digital representation and integration of systems are subject to intensive research also might imply that existing modeling languages and techniques are not sufficiently expressive enough for these challenges. For process modeling, another important concern of Industry 4.0, DSLs (4%) and knowledge representation techniques (4%) are the two modeling techniques employed most often. The overall low number of process modeling solutions might suggest that existing modeling techniques are insufficient for certain process-related challenges as well. Overall, the modeling concerns of digital representation and integration, addressed by either AutomationML [LS17], various DSLs, knowledge representation techniques, SysML [FMS14], or UML [Gro10], 33% of the 700 concerns that are addressed with modeling techniques according to our corpus.

While the large body of included publications contribute modeling methods for the cyber-physical aspects of Industry 4.0 systems, we found that the overwhelming number of contributions focus on purely syntactic contributions to the ad-hoc system modeling. They contribute our employ modeling technologies without semantic foundation that prevent the systematic evolution of CPS' models. Similarly, we found few contributions to the systematic operation of CPS after deployment and where this is considered, it largely focuses on failure handling.

The study also shows that neither validation and verification, nor human factors, both

crucial to the success of Industry 4.0, are investigated as often. Whereas the former might require solving digital representation and integration first, the lack of research on the latter two is elusive. Unless the smart factory of the future is fully automated, human interaction and control are necessary and should be considered appropriately. Moreover, we identified gaps between the worlds of 3D modeling, systems modeling, and knowledge representation that only a few of the identified modeling techniques aim to reduce [Elg14, KTY+16, VLG15].

Conclusions Research on modeling for Industry 4.0 is driven in different domains, including automation engineering, knowledge engineering, and software engineering. It addresses a variety of concerns ranging from the design of cyber-physical systems to their integration in larger contexts (factories, business processes) to very specific applications. The contributions on modeling often either employ DSLs or general-purpose modeling languages, such as SysML or UML, to describe or prescribe parts of the systems under development. Where DSLs are used, the authors often aim to reduce the conceptual gap between a specific domain of expertise and software engineering. In most of the publications reporting the application of DSLs, their implementation is not discussed. Where this is the case, often JSON or similar implicit and ad-hoc language engineering technologies–in contrast to highly specialized language workbenches [ESV+15]–are employed [GLSC17, PBFS17]. Consequently, we observed little systematic reuse in modeling languages and techniques, which may pose an entry barrier to modeling the cyber-physical systems of Industry 4.0 due to the complex challenges of providing suitable (domain-specific) modeling languages and techniques. Where research addresses the gaps between 3D modeling, knowledge representation, and systems modeling, DSLs are employed rarely. Yet, DSLs are often used to reduce conceptual gaps in Industry 4.0. This also might be due to the lack of sufficiently expressive languages to represent and integrate these diverse systems concerns. Both suggest that modeling in Industry 4.0 can benefit from means to facilitate engineering explicit DSLs that can express or integrate concerns from all domains participating in engineering cyber-physical systems for Industry 4.0. Ultimately, research on modeling for Industry 4.0 CPS focuses on syntactic technologies for implementation phase of the systems life cycle (*cf.* Figure 1.5), lacks solutions for integration of participating domains, and contributes little to the systematic operation of systems.

2.2 Summary

This chapter presents the results of a systematic mapping study on the use of modeling for cyber-physical systems that aims to illuminate the state-of-the-art regarding modeling for cyber-physical systems in Industry 4.0. The study uncovered that there is a large variety of modeling languages and techniques that are applied to many different concerns in engineering cyber-physical systems, ranging from the functional and geometric-physical modeling of systems and their components, processes, knowledge, simulation, and integration. To address these concerns, a multitude of modeling languages, either created

ad-hoc, based on UML, or as explicit DSLs, have been developed. Most of these lack explicit semantic foundations, but instead are grounded through transformation into GPL code. This hampers analysis and systematic evolution of their models.

Overall, there is little systematic reuse of modeling techniques across different applications in both domains. In the domain of Industry 4.0, this might be due to being driven by different communities, contributing different perspectives on the systems under development, methods, standards, and, ultimately, modeling languages. Then again, the increasing use of DSLs over general-purpose modeling languages, such as UML or SysML, in Industry 4.0 suggests that engineering cyber-physical systems demands more domain-specific approaches to overcome the conceptual gap and overcompensate the increasing engineering complexity. This suggests that more efficient development of truly domain-specific modeling languages tailored to domain challenges, using concepts and terminology of the domain experts that are contributing solutions could facilitate the MDSE of cyber-physical systems for Industry 4.0.

Further reading The study presented in this chapter is motivated by our own research in modeling for cyber-physical systems in a variety of domains, including automotive, Industry 4.0, and robotics, which are documented in a variety of publications.

[THR+13] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. 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.

[AHRW17] K. Adam, K. Hölldobler, B. Rumpe, and A. Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations, In: International Conference on Robotic Computing (IRC'17), pages 172-179, IEEE, 2017.

[ABH+17] K. Adam, A. Butting, R. Heim, O. Kautz, J. Pfeiffer, B. Rumpe, and A. Wortmann. Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse, Aachener Informatik-Berichte, Software Engineering, Band 28, Shaker Verlag, ISBN 978-3-8440-5319-7, 2017.

[DJR+19] M. Dalibor, N. Jansen, B. Rumpe, L. Wachtmeister, and A. Wortmann. Model-Driven Systems Engineering for Virtual Product Design, In: Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, Gerti Kappel, editors, Proceedings of MODELS 2019. Workshop MPM4CPS, pages 430-435, IEEE, 2019

Chapter 3

Modeling Language Engineering

If I had nine hours to chop down a tree,
I'd spend the first six sharpening my axe.

Abraham Lincoln

Sharp, precise, and effective software tools are quintessential to successfully engineer and operate cyber-physical systems. Realizing the vision of MDSE, hence, demands for precise, *i.e.,* domain-specific, modeling languages. But engineering such DSLs from scratch is as complicated as engineering other software [FGLP10]. Consequently, facilitating the engineering of new modeling languages by reusing tried-and-tested modules of other languages [CKM+18] is crucial to the success of MDSE.

Language users (modelers) and language engineers can greatly benefit from reusing common, established, and mature language modules. For modelers, reusing language modules reifying established syntax and semantics reduces the effort of learning new languages, *e.g.,* Java reuses many concepts of C++, which lowers the barrier of using Java for C++ experts. For language engineers, reuse reduces the effort in engineering new languages from scratch and can enable reuse of language tooling (debuggers, editors, interpreter) as well. For the efficient reuse of languages and language parts, it is essential to support all relevant dimensions of languages (*cf.* Section 1.4.6), enable tailoring of existing languages to challenges of different contexts, and should be applicable to different technological spaces. In this chapter, we, therefore, present a novel systematic method for efficient creating DSLs by reusing textual, external modeling languages with translational semantics realizations that addresses the second objective of our research program (RQ2). As such, this chapter establishes the foundations for engineering the modeling languages used in the subsequent chapters and relates to the beginning of the implementation phase of the systems' life cycle, for which the suitable domains-specific modeling languages must be provided.

Our method structures the systematic reuse of modeling language modules through arranging these in the closed variability of language product lines (LPLs), deriving language variants, and systematic open customization of these variants where necessary. It rests upon a novel conceptual model separating reuse activities specific to technological spaces from independent activities and leverages language components encapsulating concrete syntax and abstract syntax, well-formedness rules, and code generators in a black-box fashion. Section 3.1 introduces a notion of syntax modules and their composi-

tion, Section 3.2 extends this with composable code generators, and Section 3.3 presents a concept for language components based on these modules and integrates these into a systematic, black-box reuse methodology. Section 3.4 summarizes our contribution to modeling language engineering for cyber-physical systems.

3.1 Reusing Modeling Language Syntaxes

For many modelers and language engineers, the syntax is the most accessible dimension of modeling languages. Hence, from these everything else, including the mapping to a semantic domain, model analyses and syntheses, editors, and other tooling, follows. As such, reusing language syntaxes in other contexts, *e.g.,* embedded within other languages or tailored to domain-specific challenges, is an essential prerequisite for efficient language engineering. Supporting forms of language reuse that are more powerful than reconfiguring language modules through premeditated parameters demands for means to compose the constituents of different languages into new languages, *e.g.,* via extension, embedding, delegation, or similar mechanisms [DCB+15, HR17, VC15].

Syntaxes of external modeling languages can be defined in a variety of ways, including metamodels [CBW17, SBMP08, VBD+13], grammars [Bet16, HR17, VOSC14], or abstract data types [WKV14]. Syntax definition in the form of metamodels and abstract data types focuses on abstract syntaxes only, whereas some grammar-based syntax definitions support describing both, concrete and abstract syntax, in an integrated fashion, increasing cohesion between both and fostering their joint reuse. Most of these syntax definition formalisms are context-free, *i.e.,* they cannot express properties over the context of the use of their elements. For instance, expressing that a model yields no two elements of the same name, such as attributes in Class Diagrams (CDs), is impossible. To mitigate this, some approaches to language engineering augment these with well-formedness rules in the form of OCL constraints [HJK+10] or GPL context conditions [HR17]. These well-formedness rules can restrict the set of valid models of the language they relate to further and in greater detail than the metamodels, grammars, and abstract data types. Consequently, efficient reuse of syntaxes must consider both, syntax definitions in the form of metamodels, grammars, or abstract data types, and related well-formedness rules.

Software variability research to SLE can facilitate this. Research in software variability has produced the notions of closed variability and open variability. Closed variability, often represented by software product lines [CN02], is a paradigm to guide the reuse of software modules by relating these as features in product line models, such as feature diagrams [SHT06]. Through the hierarchical arrangement of features in these diagrams, the options for their planned reuse are deliberately thought ahead. Thus, composition of the represented software modules can follow premeditated ways also. This liberates the users of software product lines from becoming experts in the structure and composition mechanisms of the employed software modules. Open variability expresses the support for unforeseen or unbounded tailoring choices, such as the injection of dependencies into software modules [YTM08] or their configuration with principally unbounded parameters

(*e.g.,* floating-point numbers) for which the deliberate arrangement of all alternatives is forlorn.

Hence, to support practitioners and researchers in SLE and modeling in engineering and reusing modeling languages in different contexts, we conceived a concept for the closed-variability reuse of syntaxes, including well-formedness rules. In this section, we present our notion of language modules, their integration into feature models, and their reuse through their composition based on feature selections.

Section 3.1 is based on the publications:

Paper 2 A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. 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, 2018.
Reference: [BEK+18a]

Paper 3 A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann. Systematic Composition of Independent Language Features, In: Rafael Capilla Sevilla, Lidia Fuentes, Malte Lochau, editors, Journal of Systems and Software, 152, pages 50-69, Elsevier, 2019.
Reference: [BEK+19]

Context Software language engineering is an important driver of innovation: most new programming languages introduce concepts to facilitate software engineering. For instance, Java introduced the language requirement that everything must be defined in terms of classes to foster the reuse of software parts. Kotlin, another recent language based on Java, introduced the removal of the null value, Tony Hoare's famous "billion-dollar mistake" [Hoa09], to reduce the possibility of programming errors. Similarly, SysML [FMS14] is an extension–conceptually and technically–of a subset of UML that extends the object-oriented concepts of UML to facilitate the systematic engineering of complex systems.

Consequently, there has been fruitful research in SLE to expedite the engineering of modeling languages that produced methods to reusing language syntaxes. This research is manifested in various language workbenches that employ different language specification and reuse techniques resting on grammars [Bet16, HLMSN+15, Sto11, VC15], metamodels [DCB+15, SBPM09], or projectional editing [VV10]. All of these demand white-box insights into the language definition constituents and a thorough understanding of their structure, meaning, and options of extensibility. For instance, the language workbench Neverlang [VC15, VOSC14] and its extension AiDE [KCO15] support extension of grammars through embedding. The extension points of grammars are undefined productions spread across the complete grammar definition. Hence, language engineers reusing Neverlang grammars must understand these undefined productions, identify these in the grammars, and validate that these indeed are meant for extension.

In the Xtext [Bet16] language workbench, grammars may inherit from other grammars to extend these, which enables overwriting inherited productions to adjust the inheriting language for new challenges. Again, this demands a comprehensive understanding of the grammar to be reused. The integrated systematic reuse of syntax definitions and well-formedness rules is not supported by any of these language workbenches.

Motivated by the success of software product lines, research in SLE has produced approaches towards a more systematic, *i.e.,* guided, reuse of language modules. In a recent systematic literature review, its authors identified 14 different approaches for engineering LPLs [MAGD+16]. Most approaches found in that study supported metamodels, *i.e.,* without concrete syntax, only [JCB+15, PRB+09, WHG+09]. Hence, these enable the reuse of language structure, but not vocabulary, which is crucial for language users. Only a few support reusing abstract syntax and concrete syntax through grammars, such as FeatureHouse [LDA13], LISA [Mer13], and Neverlang [VOSC14]. All of these focus solely on metamodels or grammars without considering their conjoint systematic reuse.

Research on the systematic reuse of modeling language syntaxes, including well-formedness rules, is absent, and research on their systematic reuse through explicit variability is focusing on grammars or metamodels only. Consequently, methods for the systematic reuse of modeling language syntaxes that consider the syntax definitions (metamodels, grammars, abstract data types) and well-formedness rules were missing.

Contribution We conceived a novel method of controlled language reuse through open and closed variability that facilitates a posteriori extensibility with additional features, considers concrete syntax, and enables (re-)using languages as features without explicitly foreseeing this usage at language design time. This method follows the observation that syntax is the essential dimension of modeling languages. Consequently, extension points of syntax definitions, whether they are explicit or implicit, govern the possible combinations with other syntax definitions. As syntax definitions using grammars enable the integrated definition of concrete syntax and abstract syntax, both, the language modules, as well as their systematic reuse through closed variability, focus on reusing grammars together with their well-formedness rules. The method for their reuse leverages established concepts from software product lines, such as features, feature diagrams, and bindings to prescribe possible combinations of participating language modules. But where traditional software product lines often require that all features of the feature model are provided in a 150% artifact [GKPR08], our approach to the variability of LPLs realizes closed variability over features realized by independent language modules. This alleviates a posteriori extension and evolution of LPLs. To further support reusing language modules through LPLs, we systematically structured the process of creating and using related artifacts to the participating roles.

Based on experience in industrial language engineering from various research projects, we conceived a separation of the involved concerns between four stakeholder roles reflecting different language engineering capabilities: (1) Language engineers are experts in developing language modules in specific technological spaces [KBA02]. They create consistent language modules comprising module definitions, grammars with extension

points, and related well-formedness rules. (2) LPL managers are language engineers who create and maintain LPLs over these language modules by arranging these in the feature diagrams of the LPLs. Through modeling the bindings between the features, they decide upon possible and meaningful combinations of language features by considering the available language modules. This might entail providing adapters between grammars or well-formedness rules to ensure compatibility of features and their realizations in language modules. (3) Language owners are domain experts with profound knowledge about the concepts of a domain and, hence, responsible for selecting LPL feature to derive a suitable language for their domain. To this end, they create suitable feature configurations. (4) Modelers are domain experts that use the modeling language derived by language owners. This separation of concerns between the participating stakeholders' roles liberates the individual stakeholders from being involved in all activities of conceiving LPLs and language products and reduces the gap between language engineers that yield expertise in specific technological spaces and domain experts.

The stakeholders create models and artifacts as illustrated in Figure 3.1. Here, language engineers have produced various language modules for the description of architectures, different forms of component behavior, and object-oriented programming. The LPL manager arranges the available modules in a feature diagram, prescribing their relations based on their extension points. For example, when selecting the feature `ComponentBehavior`, a production of the grammar of `BehaviorModule`, as defined in the binding between the features, will be embedded into extension point `d` of the `CoreADLModule`. The language owner then can configure a variant of the LPL by selecting desired features and the corresponding modules are composed according to their arrangement in the feature model. Based on this configuration, a new language can be composed automatically. Afterwards, modelers can use this language fully transparently without being aware of if being a variant of an LPL.

To manage language variability, we propose arranging language modules in feature diagrams whose relations govern the composition of their grammars and the union of their well-formedness rules. Bindings between the parent features and child features in the feature diagram describe how the related grammars are embedded. To this end, bindings relate grammar extension points of the grammar contained in the parent feature's language module to productions of the grammar contained in the child feature's language module. Hence, arranging one feature as a child of another entails that (a part of) the grammar of the language module realizing the child becomes embedded into the grammar of the parent feature upon selecting both in a feature configuration. Also, it entails that their well-formedness rules are joined. To resolve variability and derive a language from the LPL, the language owner models the desired features as a feature configuration from which a new language is derived by the pairwise, bottom-up embedding of their related grammars and the union of their well-formedness rules. As not all extension points of grammars used in modules of the LPL must be bound, LPL managers and language owners can derive language modules that lack specific extensions from the LPL. This enables novel language modules that can or need to be refined in other contexts, *e.g.,* because some syntax choices should be fixed prior to reusing the

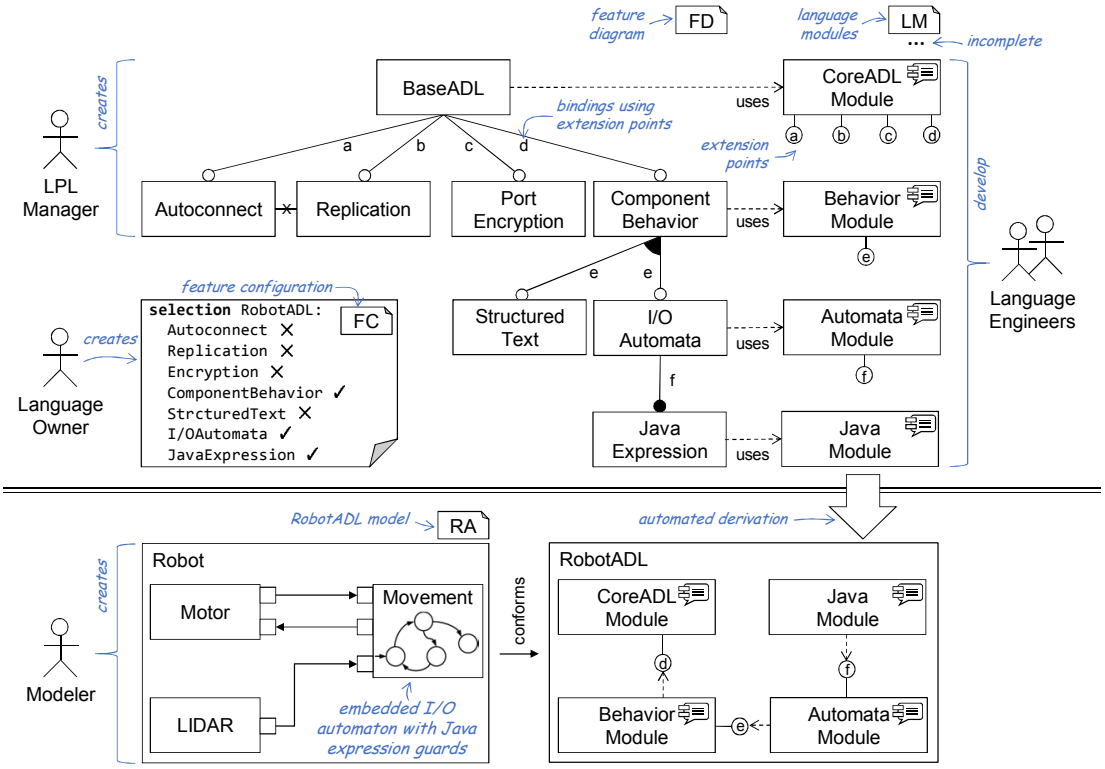


Figure 3.1: Stakeholder roles, main models, and artifacts involved in our method for systematic language reuse.

module in another LPL or because the modules require additional tailoring with syntax not provided by the LPL.

This method (1) enables compositional language reuse by decomposing languages into composable language modules, (2) supports fully automated language derivation through the composition of these modules, (3) fosters the reuse of grammars and their well-formedness rules, and (4) decouple language engineering from LPL conception, language configuration, and language use. However, as with all approaches to reuse in language engineering, it relies on strong assumptions. The three major assumptions of our approach are:

A1 Identifiable grammar extension points: A compatible grammar definition language must support the identification of grammar extension points to enable the binding of grammar productions of child features to grammar extension points of parent features. Whether this is realized through dedicated kinds of productions or through naming conventions is irrelevant. Hence, most, if not all, grammar definition formalisms can support this.

A2 Individually applicable well-formedness rules: To enable reusing the well-formedness rules of the different language modules after their related grammars have been com-

posed, they must be individually applicable to the production of these grammars that they relate to. Whether these rules are implemented in OCL [HJK+10], a GPL [HR17, Bet16], or another modeling language [VC15] is irrelevant.

A3 Conservative language composition: The composition of language modules conserves productions and well-formedness rules. Otherwise, the composition of two modules related in a parent-child relation in the LPL's feature diagram could eliminate grammar extension points required for composing the parent's module with the language module of its own parent feature. While this restricts the structure of the derivable grammars (*i.e.,* they can increase in terms of productions only), it does not restrict the languages expressiveness, as adding new well-formedness rules can restrict the accepted models of the resulting languages.

By supporting the selection of arbitrary grammar production for embedding, our approach enables reusing parts of the grammars of language modules only. For instance, in Figure 3.1, only the expression sub-language of the `JavaModule` is embedded into the `AutomataModule`. Due to our first assumption, different productions of a grammar of a language module can be declared as extension points. This significantly eases to reuse modeling languages and their parts in different contexts and liberates language engineers from being forced to create language modules for all specific embeddings. Instead, *e.g.,* the `JavaModule` might be reused with other features in the same or other LPLs to embed its statement sub-language or its type definition sub-language accordingly. Due to our second assumption, the individually applicable well-formedness rules relating to that part of the `JavaModule` can be reused without modification and applied to the expression sub-language which then is part of the automata language. Due to our third assumption, the resulting language module can be reused in another LPL as neither production of its grammar, not its well-formedness rules or extension points can be removed through composition.

Conclusions Our method for reusing language syntaxes is based on modules comprising grammars and well-formedness rules. This notion of modules facilitates the integrated engineering, maintaining, and evolution of LPLs of syntaxes. To this effect, this method is aligned with different stakeholder roles to separate the concerns of engineering language modules, guiding their composition, configuring language variants, and using these accordingly. By enabling the binding of arbitrary language module grammar productions through the LPLs' bindings and by defining them over encapsulated modules, our approach facilitates reusing modules in different language families and extending families with new modules. Overall, both can foster the engineering, and, hence, improve the availability of domain-specific modeling languages.

Leveraging LPLs for the systematic reuse of language modules presents a foundation for more efficient language engineering. Many time-honored reuse concepts from software engineering can augment this reuse paradigm further, such as (1) specifying requirements on valid extensions for extension points similar to abstract methods in object-orientation, (2) supporting the reuse of language modules via inheriting from another, or (3) lifting

language modules to language components with explicit interfaces describing their extension points. Moreover, supporting pure presentation variability [CGR09], different composition mechanisms (such as aggregation [HR17] or coordination [LDCM15]), and other dimensions of modeling languages and their tooling in language modules can further foster engineering and application of modeling languages.

3.2 Reusing Code Generators

Despite syntax being considered the quintessential language dimension, well-defined semantics [HR04] and their realization are crucial for MDE (*cf.* Section 2.1). Without unambiguous semantics of the modeling languages, neither genuine comprehension, nor automated analysis or synthesis of their models are possible. Hence, reusing semantics realizations is another important challenge in the efficient engineering of modeling languages.

While semantics often are specified denotationally [Mos90], their machine-processable realizations for the large majority of languages are specified operationally [Plo81] or translationally [LL81]. Operational semantics realization leverages interpreters, either externally [ESM08] or integrated into the languages' abstract syntax classes [JCB+15], to reify the meaning of models in software. Often, an operational semantics realization is used for the interpretation of executable models, such as statecharts (SCs). Realizing semantics operationally demands that for the target system (*i.e.,* the system using or executing the models) a suitable interpreter exists. This can restrict the application of models with operationally realized semantics severely. Translational semantics realization leverages transformations that either translate models of one language to models of another, semantically well-defined, language (M2M transformations) or to text (M2T transformations, code generators). Model-to-model transformations yield the benefit of producing correct models of the target language but require that an explicit realization of that target language exists in a technological space the transformation is compatible with. Model-to-text transformations instead can produce arbitrary text, which might represent invalid models with respect to the target language. Therefore, code generation can produce models of any language that can be represented in text without requiring an explicit representation of the target language.

Moreover, code generation enables translating models into high-performance realizations, enriched with technical details, such as serialization or network communication, and solution expertise, including leveraging established design patterns to make the resulting artifacts modular and extensible. Through this, it can greatly reduce the efforts in repetitive software engineering tasks and enforce consistency of the resulting artifacts by construction. Hence, incorporating code generators into language modules yields tremendous advantages for a more powerful language reuse that can improve availability and accessibility of modeling.

The useful composition of two code generators realizing modeling language semantics demands the compatibility of the generators as well as of the generated artifacts. The specific requirements for compatibility depend on the way the code generators and the

produced artifacts are meant to be composed. For template-based code generators, their composition often can be realized by merging their templates at well-defined extension points [GMR+16]. Such white-box approaches demand resolving name conflicts within the templates to be merged, which may require in-depth changes to the participating templates. Furthermore, they cannot ensure the compatibility of generated artifacts without additional infrastructure. Code generators that employ object-orientation to encapsulate template parts in methods, *e.g.,* Xtend [Bet16], can be composed using object-oriented mechanisms, such as inheritance, delegation, or injection. This supports making the interfaces of generators and their templates (via encapsulation in methods) explicit and facilitates their integration. As these also solely rely on templates to describe the produced artifacts, such approaches to generator composition also cannot reason about the compatibility of generated artifacts without additional infrastructure.

The research result presented in this section extends our notion of systematic language reuse based on closed variability and composition of independently developed language modules accordingly. It focuses on the embedding of code generators following the relations governed by the LPL. Thus, code generator embedding, as presented in this section, complements embedding of syntaxes by the compliant embedding of code generators at specific language extension points defined by their syntax.

This approach to generator composition gives rise to two specific challenges:

1. Generator composition challenge: Each generator responsible for translating a language with extension points must provide black-box means to invoke other generators for specific parts of embedded languages when these are processed.

2. Artifact composition challenge: The artifacts produced by the generators must be syntactically compatible and be able to interact at runtime of the product without requiring white-box insights into their production.

To address these, we conceived a novel notion of language components encapsulating realizations of syntax (grammars, well-formedness rules) and semantics (code generators), as well as a method for their composition that includes the black-box composition of participating code generators. This approach solves both, the generator composition challenge and the artifact composition challenge. In this section, we present these concepts and their integration into LPLs.

Section 3.2 is based on the publication:

Paper 4 A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann. Modeling Language Variability with Reusable Language Components, In: International Conference on Systems and Software Product Line (SPLC'18), September, ACM, 2018.
Reference: [BEK+18b]

Context Engineering code generators itself poses manifold challenges, out of which the foremost is the creation of artifacts (templates, transformations) that operate on multiple

(meta)modeling levels. Usually, such artifacts contain parts operating on the language level (such as the abstract syntax structure of the model to be processed), on the level of the specific model and its peculiarities, and on the level of the target representation (*e.g.,* the target GPL). Consequently, research on code generation focuses on developing techniques to reduce this complexity and less on the reuse of code generators.

Where code generators leverage templates to describe the translation of models, the template languages, such as FreeMarker [For13], Jinja2[1], Pug [Mar18], or Velocity [CH07], often support some forms of template reuse. For instance, the FreeMarker [For13] template language supports that templates import other templates and assign new values to the properties of the imported template. Lacking interfaces, this white-box reuse demands knowing the imported templates' properties and assigning valid (*i.e.,* type-compatible) values to their properties. Similar inclusion mechanisms are supported by Velocity [CH07]. In Pug [Mar18], a template language for HTML and JavaScript, blocks define assignments of model elements to HTML elements. It supports white-box reuse via template inclusion and inheritance. In both cases, the blocks of the included or inherited template are made available to the including or inheriting template, respectively. This modularity is possible as Pug blocks assign properties to HTML elements and adding new blocks can be translated to adding new properties to these HTML elements. Jinja2 also is based on blocks and supports their inheritance in the same fashion. Overall, the reuse mechanisms provided by template languages address reuse within code generators but do not consider reusing complete generators in different contexts and demand white-box insights. Hence, these reuse mechanisms do neither contribute to the generator composition challenge, nor to the artifact composition challenge.

With Xtend [Bet16], a modeling language extending Java with the concept of templates, the flow of control is inverted. Here, templates are embedded into method definitions and produce string objects that can be manipulated by the subsequent statements. Consequently, code generators based on Xtend can leverage the full reuse potential of object-orientation. While this provides a solution to the generator composition challenge, it does not support reasoning about generated artifacts, *i.e.,* does not contribute to the artifact composition challenge.

For the composition of complete generators, only a few approaches exist. MontiCore [HR17] features a template-based code generation framework that is based on the FreeMarker [For13] template engine. This framework supports specifying different kinds of hook points for templates, such as explicit in-template hook points for other templates or Java code, and implicit hook points available before and after each template. Using these hook points, multiple code generators can be composed through their templates. By hooking templates of one code generator into another, integrated artifacts can be produced. However, in lieu of explicit template interfaces, correctly hooking one template into another requires that the template providing the hook point also provides all data (model parts, configuration, *etc.*) required by the templates hooked into it.

GECO [Jun16] is an approach towards the composition of code generators for aspect-oriented, metamodel-based DSLs. Generators following the GECO approach must be

[1] Jinja website: `https://jinja.palletsprojects.com/`

designed as integrated "generator fragments" that can be reused at join points in the related DSL. The restriction to aspect-orientation severely limits its applicability. In the vision of a model-driven architectural framework [KR06], the authors suggest the concept of a "generic code generator generator" that leverages a repository of language building blocks to create suitable code generators. How this vision addresses the challenges of black-box composition of code generators is not explained.

There is very little research on the black-box composition of code generators, which might be due to their inherent complexity. Nonetheless, as code generators are a vital part of language realizations, their reuse must be considered. One approach to this is the investigation of arranging code generators in systematically reusable language modules.

Contribution To address both, the generator composition challenge and the artifact composition challenge, for black-box code generator composition following the embedding of their languages, we conceived a notion of generator interfaces, a method for generator composition based on these interfaces, and integrated these into the language modules presented in Section 3.1. The code generator interfaces make compatibility requirements of the generators (producers) and generated artifacts (products) explicit and support their systematic integration. Our generator composition method builds on these to ensure that (1) generator embedding can follow language embedding such that for each syntax extension point in the embedding language, there is a corresponding extension point in the code generator belonging to that language; (2) the generator (producer) interfaces of embedding and embedded generator are adapted to another to ensure that the embedding generator and invoke the embedded generator; and (3) the artifact (product) interfaces of embedding and embedded generator are also adapted to another to ensure the product of the embedding generator can interact with the product of the embedded generator. By integrating code generators into language modules and making their provided extension points and required extensions explicit, providing the product and producer interfaces, and applying our composition, their black-box reuse becomes possible.

To enable black-box composition of generators, both, the generators and the produced artifacts must be properly encapsulated. Therefore, we leverage the time-honored concept of separating interfaces and implementations to ensuring this encapsulation for object-oriented code generators that produce object-oriented artifacts. For this, we identified that, if each code generator adheres to an explicit producer interface and produces a main artifact that adheres to an explicit product interface, the syntactic extensions points of the processed language can be also be understood as extension points of the related code generator. Of course, the dedicated main artifact may relate to other artifacts. Hence, for each syntactic extension point of the processed language, the responsible code generator then yields a generator extension point. The generator extension point belongs to a syntactic language extension point, and specifies the required interfaces of compatible producers and products as illustrated in Figure 3.2.

For the grammar of the language module `BaseADL`, as illustrated in Figure 3.1, the responsible code generator yields a producer interface describing how it can be invoked

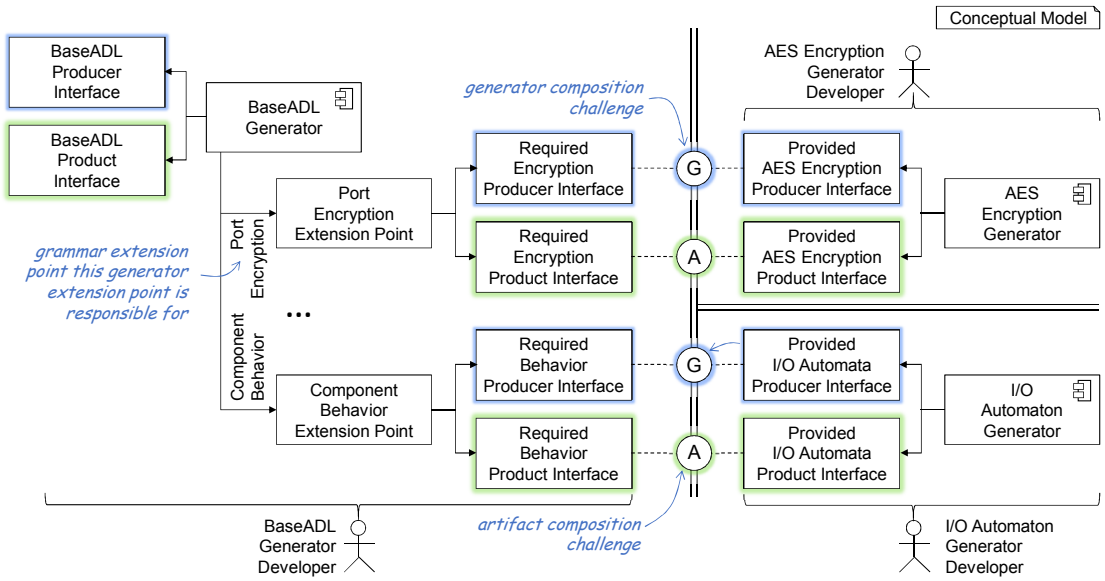


Figure 3.2: The required generator extensions follow the required grammar extensions of the containing language component and specify two kinds of interfaces to enforce compatibility of embedding and embedded generators.

and a product interface describing how the main artifact it produces can be interacted with. As the grammar of `BaseADL` yields four extension points, so does the `BaseADL Generator` (out of which only two are illustrated in Figure 3.2). For each of these generator extension points, the related grammar extension point, one required producer interface, one required product interface are made explicit, and a registration method accepting an instance of the required producer interface are specified. The required producer interface characterizes compatible generators that are responsible for translating grammar productions embedded into the specified grammar extension point and will be invoked by the `BaseADL` generator if such an embedded production is found. The required product interface describes the properties of the artifacts generated for this particular embedding that the `BaseADL` generator experts and operates with.

Through adaptation between the provided and required producer and product interfaces, the generators can be made compatible and their composition can follow the embedding of the languages they translate. This notion of generator composition does neither impose dependencies between the participating code generators, nor a common dependency to a shared library of common interfaces that the different code generators must adhere to. Instead, each code generator can be developed independently and composed by providing two adapters, for producer and product, per extension point. Given a description of the code generators participating in a composition, the structure of these adapters is be generated based on the producer and product interfaces, such that only the specific adaptation demands handcrafted efforts. Automating this as well demands a greater semantics expressiveness than supported by object-oriented GPLs.

Moreover, this method for generator composition supports transitive embedding of code generators. For instance, if the `I/O Automaton Generator` would process a language with extension points for transition guard expressions, it would yield a corresponding generator extension point of requiring a producer and a product interface for which suitable generators could be registered.

Overall, our approach to generator composition is subject to the following assumptions:

A4 Each extension point in the processed language implies an extension point in the processing code generator.

A5 Code generators and produced artifacts must support the notion of object-oriented interfaces.

A6 Each code generator must create a main GPL artifact adhering to the generator's product interface through which that artifact can be invoked during product runtime.

A7 Producer interfaces and product interfaces must be made explicit by the code generators for themselves, their produced main artifacts, and all of their generator extension points.

To integrate code generators and their black-box composition into our method for systematic language reuse, we extended the notion of language modules to language components yielding interfaces that specify grammar productions meant for embedding, context conditions meant for reuse, and code generators meant for composition. Accordingly, we extended the notions of bindings between features that are realized by language components to describe the binding of code generators as well.

Ultimately, this enables the total black-box composition of code generators for object-oriented target languages that only demands to provide suitable adapters post feature configuration.

Conclusions Our method for reusing realizations of language semantics in the form of code generators relies on four assumptions about the structure of the generators and the generated artifacts. Through these, it enables a novel, non-invasive, black-box reuse of code generators that solves the generator composition challenge and the artifact composition challenge.

These assumptions also restrict the application of our approach. The first assumption is a deliberate choice of our approach to ultimately enable the joint composition of syntax and semantics realizations. The other three assumptions impose requirements on the engineering of compatible code generators and restrict the applicability of our approach. For instance, requiring object-oriented generator interfaces prevents applying our approach to code generators developed in GPLs following other programming paradigms. Similarly, requiring object-oriented interfaces for generated artifacts prevents applying our approach to various kinds of target languages and formats (such as CSV, SQL, or

XML). Moreover, while assuming that every code generator produces a main artifact does not limit the application of our approach technically, enforcing the existence of a main artifact can make the generated code less efficient. This especially holds when embedding code generators for cross-cutting concerns.

The notion of language components presented in this section is limited to one code generator per component and only makes one grammar extension point of grammars explicit. Not supporting multiple code generator specifications for different grammar parts can hamper the reuse of code generators. Similarly, not specifying more than one grammar extension point can hampers reuse of syntaxes through components. The contribution presented in the next section addresses these challenges.

3.3 Systematic Black-Box Reuse of Language Components

The results presented in Section 3.1 and Section 3.2 enable the black-box composition of modeling language syntaxes and code generators encapsulated and structured through LPLs. The efficient black-box reuse of complete language components demands means to explicate the required extensions (*i.e.,* extension points) and provided extensions explicit on the component level. Moreover, the presented language components only explicate one provided grammar production, a set of well-formedness rules, and one provided code generator. Where these components comprise multiple reusable artifacts and artifact parts, such as additional grammar productions, well-formedness rules, or code generators that might be reusable in different contexts, their interfaces currently prevent reusing these as they are hidden from the user. Moreover, enabling the parametrization of well-formedness rules and code generators can further facilitate their reuse.

Through making their interfaces support explication of multiple provided extensions and extension points for grammars, well-formedness rules, and code generators, as well as the parametrization of well-formedness rules and code generators, reusing the individual components in different contexts becomes easier, which can improve their maturity through (re-)use [CBCR15].

A method for the black-box reuse of language components that supports this flexibility must consider language components, their interfaces, extension points and provided extension at sufficiently detailed granularity. Systematically reusing such components LPLs further requires more sophisticated bindings between the participating components and, ultimately, a feature modeling technique supporting binding of these components.

Ideally, such a framework fosters language component reuse in different technological spaces, as long as these conform to its assumptions on language specification and composition (A1-A7). This imposes a separation of concerns into:

– Generic reuse concerns: For systematic reuse through variability, these include defining LPLs, binding features to components, resolving feature configurations by orchestrating the composition of the participating components' artifacts, and adjusting the component's interfaces to reflect their post-composition properties. These concerns are independent of the specific technological space the framework is applied with and need to be taken care of for every application of the framework.

– Technology-specific reuse concerns: These include the actual composition of the language component's artifacts following the framework's assumptions and ensuring the well-formedness of resulting artifacts.

Thus, separation of concerns imposes the requirement that a framework supporting this technology-space-independent reuse must coordinate managing the generic reuse concerns and support extension with technology-space-specific software components to realize specific reuse concerns.

Section 3.3 is based on the publications:

Paper 5 A. Wortmann. Towards Component-Based Development of Textual Domain-Specific Languages, In: Luigi Lavazza, Herwig Mannaert, Krishna Kavi, editors, International Conference on Software Engineering Advances (ICSEA 2019), pages 68-73, IARIA XPS Press, 2019.
Reference: [Wor19]

Paper 6 A. Butting, J. Pfeiffer, B. Rumpe, and A. 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, 2020.
Reference: [BPRW20]

Context Various language workbenches, such as Melange [DCB+15], MontiCore [HR17], MPS [Cam14], or Spoofax [WKV14] support the composition and the customization of languages in their specific technological spaces. Similar to other approaches to systematically reuse language parts [Bet16, LDA13, KCO15, VC15], they neither consider the joint multi-dimensional reuse of language constituents nor make their provided and required interfaces explicit. This restricts language reuse to a white-box reuse of individual language dimensions (*cf.* Section 1.4.6) in specific technological spaces.

The different approaches that apply software product line techniques [PBL05] to SLE [Kle08] either focus on closed variability expressed with LPLs or do not consider open variability, *i.e.,* subsequent customization (*cf.* Section 3.1).

Overall, there is no notion of multi-dimensional language components that feature syntax, well-formedness, and code generation, which makes their integrated interfaces explicit and supports their composition based on the interfaces provided and required extensions. Consequently, there also are no top-down methods for systematic reuse that abstract from peculiarities of the technological spaces they were developed for.

Concern-oriented reuse [AKM13] is a vision of software development based on concepts from MDE, CBSE, and variability that aims to structure reuse of software parts through components arranged in software product lines. In this vision, concerns are reusable software modules supporting interfaces for expressing their variability, means for customization, and use. Concern-oriented language development [CKM+18] is a conceptual model extending concern-oriented reuse to SLE that focuses on grouping related

language dimensions in facets. These facets comprise artifacts and can be reused by language concerns that yield the three interfaces envisioned in concern-oriented reuse. As such, concern-oriented language development is a vision that neither details the structure of these interfaces, nor explains the composition of language artifacts, or provides an applicable framework.

Contribution Our method for systematic black-box reuse of language components considers the language dimensions of (1) Integrated definitions of abstract and concrete syntax in the form of grammars; (2) Restrictions to the abstract syntax through well-formedness rules; and (3) Realization of translational semantics through code generators.

The contribution presented in this section, thus, is threefold: First, we present a novel conceptual model for reuse that details the constituents of these 3D language components with their interfaces, extension points, and relations and of LPLs with feature diagrams and the binding of features to language components. Second, we describe a systematic method for reusing 3D language components through resolving closed and open variability as the composition of participating components build upon the composition mechanisms introduced in Section 3.1 and Section 3.2 that considers generic reuse concerns as well as technology-space-specific reuse concerns. Third, we present an extensible modeling framework that supports specifying language components and LPLs through dedicated modeling languages and realizes our reuse method in generic software components that can be extended with and delegate to technology-space-specific components for the composition of language constituent artifacts.

Based on the results presented in Section 3.1 and Section 3.2, we devised concepts for 3D component interfaces consisting of three kinds of provided extensions. Provided grammar extensions expose productions of comprised grammars to be reused in other contexts. Provided generator extensions reference a provided grammar extension for whose translation they are responsible and a generator specification consisting of provided and required product and producer interfaces required for that translation (*cf.* Section 3.2). Provided well-formedness rule extensions expose sets of well-formedness rules meant for their joint reuse.

Also, 3D component interfaces yield two kinds of–mandatorily or optionally–required extensions (extension points). Required grammar extensions expose extension points of the comprised grammars into which provided grammar extensions of other components can be embedded. Required generator extensions reference a required grammar extension for whose translation a responsible generator is needed and a generator specification consisting of provided and required product and producer interfaces required. There are no required well-formedness rule extensions. Exactly specifying the expected input-output behavior of well-formedness rules is generally as complicated as implementing these.

Additionally, 3D component interfaces may yield parameters for the configuration of well-formedness rules and code generators. For grammars, specification of parameters aside from via the embedding of productions of other grammars rarely is supported. Thus, lifting these to the conceptual model as well would severely restrict its appli-

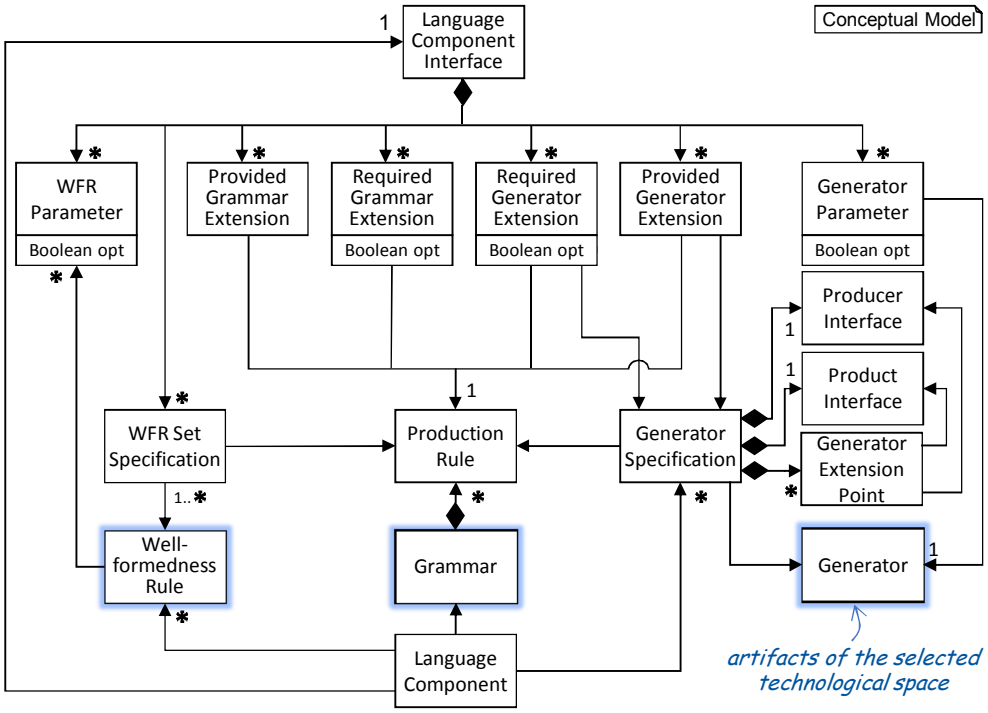


Figure 3.3: Our conceptual model of 3D components distinguishes language components for their interfaces, which make the required and provided extensions of the contained language implementation artifacts explicit.

cation of few compatible technological spaces. The resulting conceptual model for 3D language components is illustrated in Figure 3.3. The conceptual model of language families features relates to the component interfaces through bindings that match required extensions of parent features with provided extensions of child features.

Based on the separation of reuse concerns, the directed composition of two 3D language components according to the bindings from interface elements of the child component to the interface elements of the parent component then amounts to producing a new component such that:

- Its interface results from combining the interfaces of both input components, adding or removing interface elements as specified in the binding.
- The comprised grammars result from embedding the grammars of the child component into the grammars of the parent component according to the bindings defined in the feature model.
- The sets of well-formedness rules of the child component are either merged with sets of well-formedness rules of the parent component, added as individual sets to the interface, or removed.

– The code generators are composed according to the bindings and their generator specifications (*cf.* Section 3.2).

For LPLs over 3D language components, this process is applied bottom-up for all selected features and their parents until all bindings have been resolved. If the resulting language component is incomplete in the sense that there still are mandatory required extensions or unset parameters, subsequent customization dedicated customization models can complete these.

The framework realizing our reuse method consists of four main components that are illustrated in Figure 3.4 and focus on (1) processing models of language components (`ComponentProcessor`) and checking their well-formedness; (2) composing models of language components (`ComponentComposer`); (3) loading the feature models of LPLs, checking their well-formedness, and resolving feature configurations (`LPL Manager`); and (4) customizing language components through the embedding of components not provided in the LPL. These components take care of generic reuse concerns presented in this section. As such, they are independent of a specific technological space but require extension with a component taking care of reuse concerns specific to the technological space the language components are realized in. This component takes care of composing grammars, joining well-formedness rules, and embedding code generators accordingly.

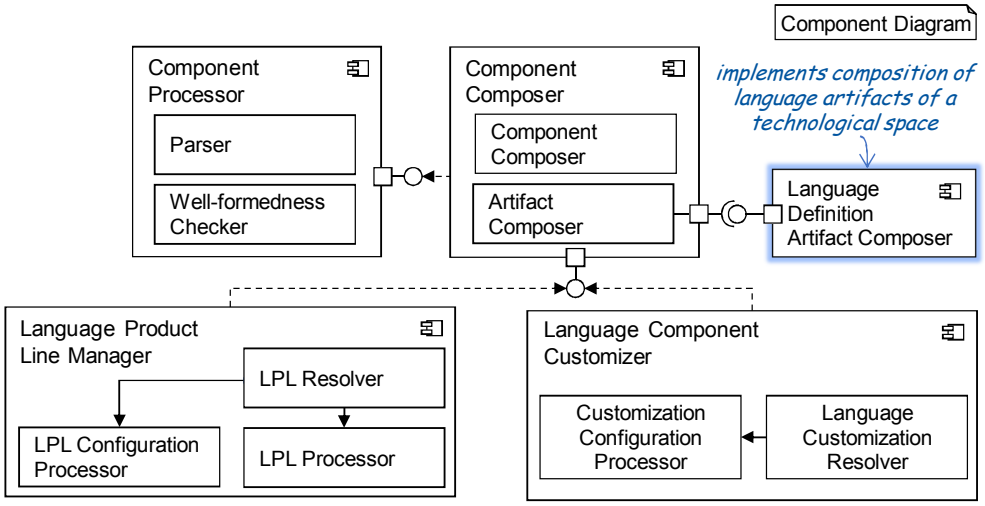


Figure 3.4: Our language reuse framework reifies the concepts presented in this section and separates the concerns of generic reuse from the reuse concerns specific to a technological space.

Conclusions By making the constituents of language components meant for reuse explicit through dedicated interfaces comprising required and provided extensions, the components can be composed in a black-box fashion. We systematically leverage this component-based composition of language components through LPLs to structure and

guide their reuse. By conceiving the concepts, process, and the framework realizing it to be independent of technological spaces (under the assumptions A1-A7), our novel reuse method can be applied to a wide variety of language engineering challenges. As such, our approach can be considered a novel foundation for technology-space-independent systematic language reuse that can be extended to support other dimensions of language definition (transformations, editors, *etc.*), specification mechanisms for required well-formedness rules, or different means of language composition.

3.4 Summary

This chapter presents a novel method that enables software language engineers to provide domain experts with more precise, and thus more effective, modeling languages. To this end, they can define LPLs over independently developed language components encapsulating realizations of syntax and semantics behind their explicit interfaces and relate these in the LPLs feature models as seen fit for the intended domain. Based on a selection of features from the LPLs, a novel DSL can be derived mostly automatically; only implementations of the adapters between participating generators must be provided manually. In the derivation process, grammars, well-formedness rules, and code generators of the language components of selected features are combined pairwise according to their features' relations in the LPL's feature model. Our reuse method generally is independent of specific technological language engineering spaces as long as these conform to the requirements laid out in this chapter. This novel form of systematic, planned language engineering facilitates providing more truly domain-specific language to experts contributing solutions to the engineering of CPS.

Further reading The publications summarized in this chapter relate to results of a research program on SLE that investigates architecture-centric language engineering, systematic modeling language reuse, and bridges between different technological spaces of language engineering. Its results are documented in a multitude of publications.

[BHH+17] A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, and A. 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, 2017.

[ABK+18] K. Adam, A. Butting, O. Kautz, J. Pfeiffer, B. Rumpe, and A. Wortmann. Retrofitting Type-safe Interfaces into Template-based Code Generators, In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pages 179 - 190, SciTePress, 2018.

[HRW18] K. Hölldobler, B. Rumpe, and A. Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages, Computer Languages, Systems & Structures, 54, pages 386-405, Elsevier, 2018.

[DJK+19a] M. Dalibor, N. Jansen, J. Kästle, B. Rumpe, D. Schmalzing, L. Wachtmeister, and A. Wortmann. Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext, In: Jeff Gray, Matti Rossi, Jonathan Sprinkle, Juha-Pekka Tolvanen, editors, International Workshop on Domain-Specific Modeling (DSM'19), pages 40-49, ACM, 2019.

Chapter 4

Systems Modeling and Evolution

Architecture starts when you carefully put two bricks together.
There it begins.

Ludwig Mies van der Rohe

A quintessential challenge of modeling cyber-physical systems is defining their architectures, *i.e.,* fixing their principal design decisions (*cf.* Section 1.4.5). The "bricks" of such architectures can assume many shapes, from the geometric-physical appearance of product architectures to the abstract function networks of functional architectures. Putting these together carefully is one of the prime challenges of product development processes [UE09], including planning, design, validation, and production, for cyber-physical systems. The research summarized in this chapter, thus, contributes to the third objective of our research program (RQ3) and establishes novel methods for the systematic modeling and evolution of architectures and behavior of CPS and their digital twins. As such, it largely focuses on the implementation phase of the system life cycle but can be expected to extend into their operations through the increasing application of DevOps (the combination of software development and IT operations) to CPS.

The dichotomy between the functional perspective on cyber-physical systems popular in computer science and the geometric-physical perspective popular in mechanical engineering gives rise to a conceptual gap between functionally defined, reusable, traceable, and, ideally, automatically processable functional architectures and their geometric-physical realizations. Successful engineering of cyber-physical systems needs to overcome this gap. Due to incompatible abstractions on both sides of the gap, this demands significant efforts by experts of the implementation domains, who usually prefer to start with geometric-physical implementations instead of functional architectures. This hampers reusing solutions between different product development processes and product architectures.

Embracing semantically-grounded, functional architectures for the systematic engineering of cyber-physical systems yields many benefits, including advanced possibilities for automated analysis and synthesis of these architectures. Among these are the possibilities to systematically evolve these architectures employing the methodology of stepwise refinement [BS01]. In this, engineering of functional architectures starts with coarse, largely underspecified, specifications of the architecture in question and properties that must hold for this architecture. If each subsequent version of this architecture

is a semantic refinement of its predecessor, *i.e.,* its observable behavior is a refinement of the behavior of its predecessor, then the properties that held for the previous version still hold for the successor version. This liberates engineers from validating and verifying these properties again.

In stepwise refinement, decomposing the initial, monolithic, architectures into connected components that can be implemented by experts of the participating domains imposes significant effort to the systems engineers. Hence, in this activity, many errors may occur. Through automated analysis and decomposition of the initial monoliths, this can be mitigated.

Section 4.1 presents metamodels for supporting a functionally-driven product development process for cyber-physical systems that enable reusing mechanical engineering expertise reified in time-honored design catalogs [KK98]. Section 4.2 presents a novel method for the automated, semantics-aware decomposition of functional architectures based on the Focus [BS01] theory and its implementation in MontiArc (*cf.* Section 1.4.5). This fosters the independent development of its components and facilitates integrating domain experts into the overall systems engineering by assigning such components to the respective experts. Section 4.3 introduces a method for the fully automated semantic differencing of functional architectures. This method supports system integrators and domain experts in ensuring the semantic compatibility of their independently developed components with their predecessor versions assigned to them by the systems engineers. Section 4.4 summarizes our contributions to systems modeling and evolution.

4.1 Functional Modeling of Cyber-Physical Systems

Cyber-physical systems are characterized by the interaction of mechanical, electronic, and software systems. This raises various socio-technical challenges [Bro06, FR07] that render engineering of such systems inherently complex. As customers increasingly desire innovative functionalities and a shorter time-to-market, engineers developing cyber-physical systems need to manage these complexities fast and cost-efficiently [DGH+19, KMS+18, PBFG07]. Innovation in cyber-physical systems is driven by functionalities and features that arise from the interaction of solutions from mechanical engineering, electrical engineering, software engineering, and other contributing disciplines.

In mechanical engineering, engineers integrate the physical components of a product into assemblies such that various constraints, *e.g.,* regarding design space, mounting, and maintenance are satisfied [PBFG07]. The resulting geometric-physical product architecture fulfills the desired functionalities through physical effects acting between its components and assemblies. In the product architecture, engineers control the impact of physical effects by manipulating the geometric shape of components or their material, as the effects themselves are set by laws of nature. Consequently, mechanical engineers tend to directly design the geometry of components based on given requirements, *i.e.,* without explicating the functionality to implement, which is why the product architecture has become the quintessential structuring element in mechanical engineering [PBFG07].

Where mechanical engineering is mainly concerned with the physical product architec-

ture, software engineering has been very successful in adopting a functional perspective on systems, which fosters the holistic design, engineering, and validation of functions.

In mechanical design methodology, functional architectures describe the functionality of the system under development but are typically not considered in practice, let alone modeled. Existing approaches utilizing mechanical functional architectures do not formalize the relation between the functional architecture and the product architecture. Hence, the components of the physical product architectures are not directly linked to the functions they implement. This raises a conceptual gap [FR07] between the functional requirements of cyber-physical systems [KMS+18, DGH+19] (problem domain) and their resulting physical product architecture (solution domain). Therefore, reusing component implementations in other systems is hardly possible, and validation occurs late in the product development process, *i.e.,* when changes are cost-intensive.

This conceptual gap arises when the solution to a problem is described at a lower level of abstraction than the problem itself [FR07]. A holistic MDSE for cyber-physical systems, therefore, needs to bridge the gap between system functions and their implementation in a physical product architecture. In the following, we present two metamodels to describe functional architectures that can be linked to their geometric-physical realizations. This enables describing functional architectures of cyber-physical systems and explaining their geometric-physical realizations based on mechanical design catalogs in integrated models, which facilitates tracing of changes, joint evolution and maintenance of the cyber parts and the physical parts, as well as systematic reuse of functional architectures and their parts with different geometric-physical realizations.

Section 4.1 is based on the publication:

Paper 7 I. Drave, B. Rumpe, A. Wortmann, J. Berroth, G. Höpfner, G. Jacobs, K. Spütz, T, Zerwas, C. Guist, J. 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, 2020.
Reference: [DRW+20]

Context Research has contributed modeling theories and languages for engineering software and electronic functions of cyber-physical systems, *e.g.,* [Alu15, Pto14], as well as for designing [SFA17], engineering [BBL+16], and operating [BSP+16] these systems in various domains. Most of these approaches solely consider modeling through the lens of software engineering, *i.e.,* they focus on discrete and functional systems. Where continuity and geometry are supported, the theories and languages do not support established processes or modeling concepts from other domains (such as mechanical engineering).

In the Focus theory [BS01, RR11, RW18] for distributed interactive systems, systems are hierarchically composed of components that realize stream processing functions. These functions communicate via channels only. Hence, they can be (de)composed (*cf.* Section 4.2) and refined (*cf.* Section 4.3) systematically. Applying the Focus notion of

refinement to a product development process based on mechanical function architectures is subject to ongoing research.

Research in systems engineering produced various modeling languages specifically tailored to engineering cyber-physical systems, including MechatronicUML [BGT04], UML4IoT [TC16] SysML4Mechatronics [FHK+15], or SysML4Modelica [BFK15]. Neither of these are tailored to support the product development process.

Research in mechanical engineering produced methods supporting the modeling of functional architectures in general [EGZ12, GDP+10, MKG+15, WS09, ZAMM12]. These focus on informal modeling with sketches that do not distinguish between functional architecture, physical geometry, or physical effect, which is hardly accessible to automation. Hence, using these for systematic MDSE still demands to overcome the conceptual gap.

In mechanical engineering, design catalogs are means to support the design processes [FLD04] by providing methods, processes, or components for recurring design tasks. Consequently, various design catalogs have been conceived [KK98, GA09, Kol14, PBFG07, Rot94, Rot96, Rot11]. The guidelines given in these catalogs are often given in prose, which is ambiguous and imprecise, and, hence, cannot be leveraged directly for MDSE. Approaches digitizing such catalogs solely focus on making their content digitally accessible [FLD04, MEE+11]; again in form of drawings and prose. Hence, engineers still need to overcome the conceptual gap when trying to make the digitized design catalogs directly usable in MDSE. Currently, precise modeling languages tailored to support the product development process based on the established foundations of functional architectures in the sense of [KK98, Kol14] do not exist yet.

Contribution In mechanical design theory, the concept of functions is based on the observation that a system or a part of it can be delimited by a boundary. Through this boundary, physical quantities can enter and leave the system as functional flows [PBFG07]. The function of the delimited system is responsible for transforming the incoming functional flows to the outgoing flows, which can be flows of signals, energies, and materials [UE03, PBFG07]. The function of a mechanical system then is composed of sub-functions linked by functional flows [PBFG07]. Atomic functions, *i.e.,* the transformation of flows they represent is not further decomposed, are referred to as elementary functions [KK98]. Physical effects and geometries realize elementary functions. The "Koller Catalog", for instance, comprises 350 physical effects that are mapped to the elementary functions they can realize [KK98]. Principle solutions then characterize how a physical effect, given a qualitative geometry with certain material properties [PBFG07], fulfills a physical function, *i.e.,* the leaf of a functional structure [KK98].

To support a functional product development process of cyber-physical systems based on established design catalogs, we applied concepts of CBSE, decomposition along channels of Focus [BS01], and separation of concerns to the engineering of functional architectures of cyber-physical systems. Based on these concepts, we conceived two metamodels that make functional architectures and their geometric-physical realizations explicit as well as accessible to automated analysis, and traceable.

The first metamodel, illustrated in Figure 4.1, comprises concepts to describe the

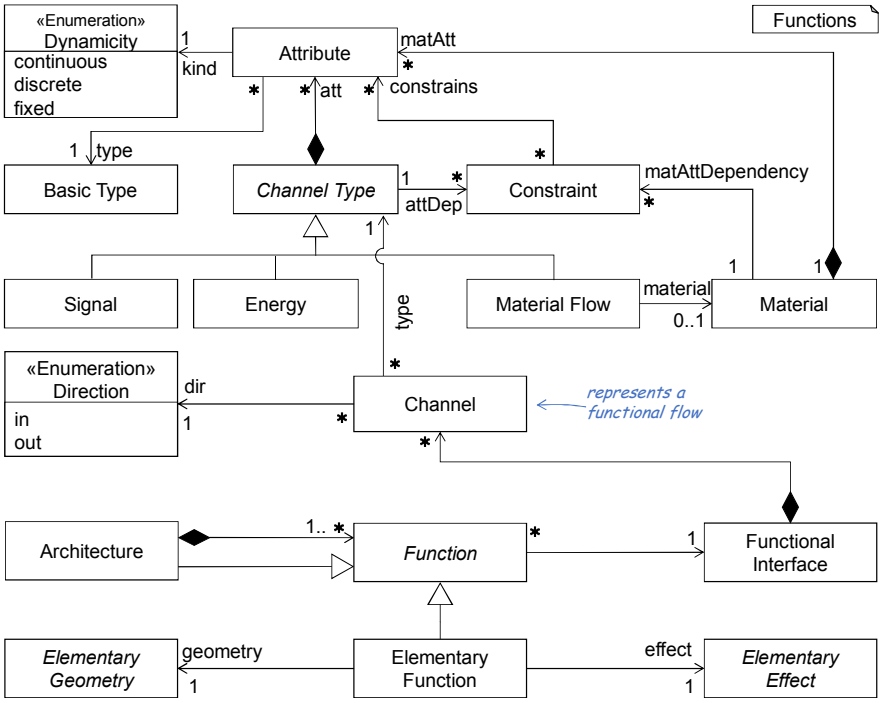


Figure 4.1: Our metamodel of functional architectures is inspired by the Focus theory and combines selected concepts with mechanical design theory.

functional architecture of a CPS as functions that are connected via directed and typed channels between their interfaces. Essentially channel type can be either signal, energy flow, or material flow with their corresponding refinements, *e.g.,* data structures, thermal energy, or coolant fluid, respectively. Channel types feature attributes that are either continuous, discrete, or fixed and represent properties of the flow modeled by the channel type. Functions either are decomposed or elementary functions that require a geometry and a effect.

The second metamodel, *cf.* Figure 4.2, features concepts to describe the geometric-physical principle solutions in terms of attributes describing their characteristics, a principle geometry, a single principle effect, and constraints over attributes. A principle geometry consists of a number of geometric elements, each of which are of a material, may feature attributes, and might be subject to constraints between their attributes. A principle effect describes the interaction of multiple physical laws given in the form of constraints over attributes. Solutions are special architectures that redefine the inherited (elementary) functions of architectures to (principle) solutions, which are interconnected by functional flows as inherited from the architecture.

With these metamodels in place, engineers of CPS can leverage the potential of functional structures trace their components back to requirements, bridge the gap between function and geometry in mechanics, apply a systematic separation of concerns between

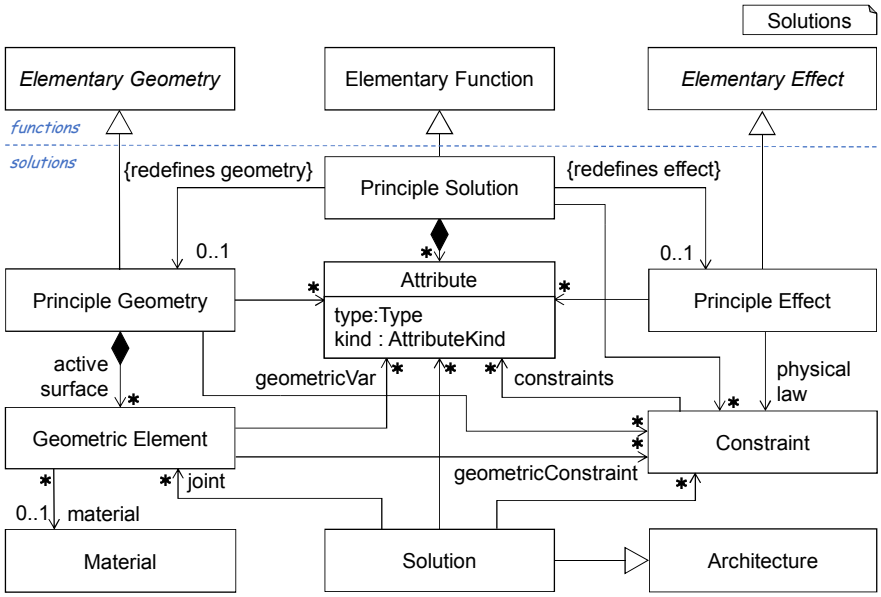


Figure 4.2: Out metamodel of principle solutions explicates concepts of the "Koller catalog" [KK98] and relates these to functional architectures.

functional design and physical implementation, and facilitate integrating the heterogeneous domains participating in engineering CPSs. To further support the industrial application of our metamodels for functional architectures and product architectures, we also have encoded it as a SysML profile.

Conclusions In this contribution, we formalized the notion of functional CPS architectures and combined modeling these with applying the knowledge of design catalogs [KK98, PBFG07]. To this end, we applied concepts from software engineering in metamodels that enable the explicit modeling of functional architectures, physical solutions, and their connection to facilitate the functional design of CPS. Our results are a first step towards the rigorous pervasive modeling of CPS that is necessary to overcompensate their increasing complexity. Moreover, the metamodels enable separating the concerns of involved domain experts into reusable functions and solutions, and facilitate their contribution to the model-driven engineering of CPS.

4.2 Automated Semantics-Preserving Decomposition of Architectures

Systems architectures combine and integrate concerns from a variety of stakeholders from different domains. The better the architecture can be decomposed into domain-specific components, the more cohesive components can be provided by the domain experts us-

ing the concepts, methods, and paradigms of their choice. To support the systematic and semantics-aware evolution of initial, monolithic components into architectures of cohesive components, some theories for architecture modeling, such as communicating sequential processes [Hoa78], Focus [BS01], and π-calculus [Mil99], support the notion of stepwise refinement, a methodology for continuous architecture modeling based on controlled evolution and progressive decomposition of components. During stepwise refinement, monolithic components that realize multiple independent concerns are decomposed and component behaviors are concretized, *i.e.,* made increasingly deterministic, until, ultimately, an implementation, a hierarchical architecture of deterministic components, is achieved. Thereby, each subsequent (possibly decomposed) version of a component must adhere to properties already proven for its predecessors. A component refactoring then is a special refinement step where the resulting component's semantics is equal to the semantics of the predecessor. Hence, from an external observer's viewpoint, the behaviors of the original and the resulting components are indistinguishable. Manual stepwise refinement and refactoring without tool support, however, are tedious and error-prone.component & connector

For the systematic engineering of software architectures with experts from multiple domains contributing various components, automating the parallel decomposition of software components into architectures of smaller, more cohesive software components is important to achieve an efficient, modular, and parallel systems engineering. This raises two challenges.

1. *Analysis Challenge:* Proving architectural properties already for initial, monolithic architectures enables fixating relevant system properties early. However, model checking the complete initial architecture might be challenging or impossible due to it intertwining different concerns unrelated to the properties under consideration.

2. *Implementation Challenge:* Evolving functionalities implemented by monolithic architectures is usually overly complicated: in the worst case, parts of the domain-specific implementations are scattered over different components and hardly documented, which makes evolution error-prone and costly.

To support systems architects in semantically refining initial monoliths into architectures of cohesive components that can be implemented independently by domain experts, we conceived a novel method for the automated semantics-preserving decomposition of component & connector architectures based on the *influence* exerted by messages received via incoming ports on the messages sent via outgoing ports of components.

Section 4.2 is based on the publication:

Paper 8 O. Kautz, B. Rumpe, and A. Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures, In: Automated Software Engineering, 27, pages 119–151, Springer, 2020.
Reference: [KRW20]

Context Only few ADLs support semantics-preserving stepwise refinement through parallel decomposition and these are grounded in Focus [BS01] or the π-calculus [Mil99]: AutoFocus 3 [HF07b] is a tool featuring an ADL for architectures of reactive software systems grounded in Focus. While AutoFocus 3 supports model checking of architecture behavior against temporal logics properties [CHN11], it lacks methods for the fully automated, semantics-preserving, decomposition of architectures. Hence, the challenge of manually decomposing monolithic, initial architectures into components realizable by domain experts retains. The π-ADL [CQT$^+$16, Oqu04] is an ADL based on the π-calculus that supports model checking of software architectures against DynBLTL [QCT$^+$16] properties. To this end, a statistical model of system executions is created based on which the probability of satisfying properties within confidential bounds is calculated. We are unaware of any methods for automated, semantics-preserving decomposition based on the π-calculus.

As our approach decomposes initial, monolithic components with large behavior automata, into architectures of smaller components, it also relates to the (automated) parallel decomposition of automata [GG67]. More specifically, we aim for the practical decomposition [Noz78, US13], *i.e.,* the resulting components feature fewer states than the component they were decomposed from, of TSPA. And while TSPA generally can be decomposed into compositions comprising FIFOs and XORs only [KC09], the resulting granularity produces automata too detailed for constructive systems engineering. For probabilistic automata [CY15] and linear automata [PP15], related approaches exist. None of these consider automated, semantics-preserving, decomposition in the presence of influencing ports. Our method closes this gap to ease the modeling of fine-grained software architectures that consist of independent component models, which can be implemented by domain experts.

Contribution To support the automated semantics-preserving parallel decomposition of architectures, we conceived a concept of influence between the ports of its components. Based on the identification of non-influencing specifications, monolithic components can be decomposed into independent, more cohesive and comprehensible components, which are better accessible for analyses and development (*e.g.,* for stepwise refinement or refactoring), automatically. The independence of resulting components facilitates their realization by experts of different domains.

Our method uses TSPA to represent the abstract syntax (*cf.* Chapter 3) of common component & connector ADLs, such as AADL [FG12], AutoFocus [HF11], EAST-ADL [DSLT04], SysML's block diagrams [FMS14], MontiArc (*cf.* Section 1.4.5), and similar languages. Given an initial component implementation, our method automatically decomposes it into hierarchically composed subcomponents according to their influence relation that yield the same behavior as the initial implementation. To achieve this, we extend the Focus theory of time-synchronous components with the concept of influence, present a decomposition procedure leveraging this, and prove that the resulting system is semantically equivalent. Hence, our contributions are (1) A notion of influence between ports of a software architecture that is grounded in the Focus theory; and (2) A method

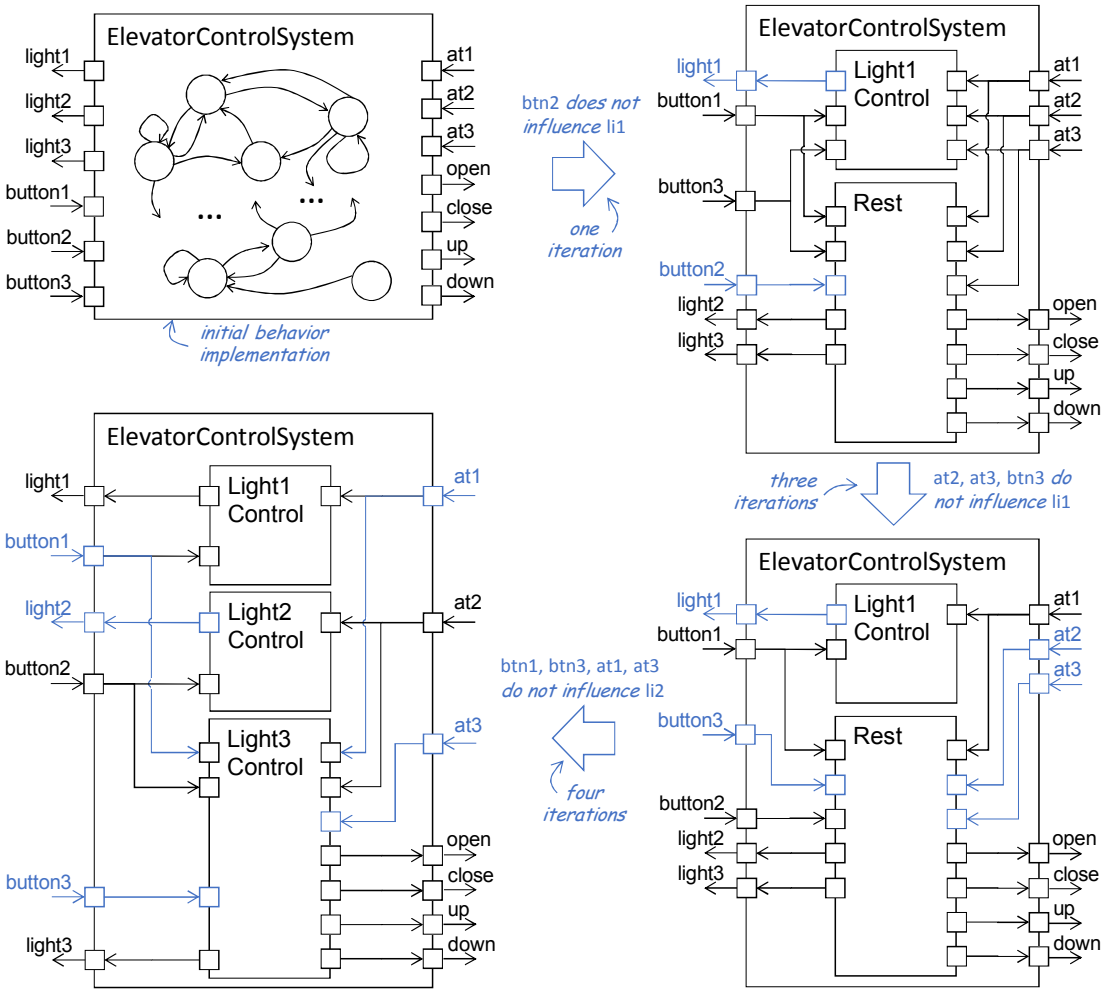


Figure 4.3: Initial architecture and intermediate, semantics-preserving, refinements of an elevator control system.

to automatically refactor components with finite state spaces via parallel decomposition according to the influence relation. The resulting architecture is semantically equivalent to the previous architecture but can be evolved more efficiently by different stakeholders. Thus, all original properties still hold, despite being less complicated and better to evolve and maintain.

We identify an incoming port i and an outgoing port o of a component c as influencing each other, if there are histories of messages with the same messages on all ports of c except i that produce different behavior on o. By iteratively partitioning a component's implementation into two, with respect to two non-influencing ports, independent (sub-)implementations, encapsulating these into new subcomponents, and composing these based on their interfaces, we achieve a new architecture that is semantically equivalent

but structurally refined.

Figure 4.3 illustrates the process of automated semantics-preserving decomposition on the example architecture of an elevator control system [SW99]. The architecture is responsible for controlling an elevator that serves three floors, including the movement of the elevator cabin, opening and closing its doors, and operating the buttons and lights of the elevator's controls.

Beginning with an initial monolithic implementation (top left) that specifies a state-based behavior, our method identifies that the incoming port `btn2` and the outgoing port `light1` do not influence each other. Hence, the monolithic behavior implementation can be partitioned into an architecture of two subcomponents (top right), one, `Light1Control` producing the behavior of the outgoing port `light1`, the other, `Rest` depending on the incoming port `button2`. After three more iterations (bottom right), our method identifies that the ports `at2`, `at3`, and `button3` also to not influence the port `light1`, which is why they are removed from the subcomponent `Light1Control`. After four more iterations, (bottom left), the elevator control system is refined further into three subcomponents responsible for controlling the lights on the three floors this elevator serves. Further refinement will identify that the behavior of subcomponent `Light3Control` is independent of the ports responsible for the behavior of `Light1Control` and `Light2Control`. Moreover, a new subcomponent will be identified that depends on `button1 - button3`, as well as on`at1 - at3`, and is responsible for the behavior of the elevator cabin and its outgoing ports `open`, `close`, `up`, and `down`.

Domain experts now can easily implement the different subcomponents with modeling languages of their choice as long as the behavior of the components adheres to their specification.

Conclusions Automatically decomposing monolithic architectures into hierarchically composed subcomponents that retain the semantics of the initial architecture supports providing independent components to domain experts who then can use the most appropriate domain-specific modeling languages to implement these independent components. To this end, we have extended the focus theory with the concept of influence between ports and proven that this refinement actually is a refactoring, *i.e.,* semantics-preserving.

4.3 Continuously Analyzing Architecture Models

For engineering the software architectures of cyber-physical systems, knowing the precise semantics of the participating models is essential with respect to automation, analysis, communication, synthesis, and many domain-specific concerns (such as safety or reliability). Current architecture modeling rarely considers the semantics of ADL, which also is due to only very few ADLs being semantically well-defined. Stepwise refinement [BF92, Bro10] is a software engineering methodology for the continuous modeling of software architectures based on the controlled evolution and the progressive improvement of the architecture's components. Stepwise refinement requires that each subsequent ver-

sion of a component model adheres to properties already proven for its predecessors. This greatly reduces the effort for verifying these properties again for subsequent component model versions. Hence, checking whether successor component model versions semantically refine their predecessors in terms of observable input-output behavior is essential. Understanding the semantic differences of a continuously evolving architecture through semantic analyses supports systems engineers during evolution analysis in understanding the impact of the changes between two versions of the architecture under development.

However, where fully detailed denotational or operational architecture semantics are available, *e.g.,* as with Focus [BS01], these usually are too complex for fully automated refinement checking. Hence, they typically require to (at least partially) manually prove refinement between two versions of the same component model. This exacerbates stepwise refinement so drastically that it becomes a "highly idealistic" [Bro10] idea.

To support architecture modelers in leveraging effective semantic differencing in practice, this requires means to fully automatically check whether one version of a system admits behaviors that are not possible in another version. With automated refinement checking and disproving, manual proofs become redundant, which enables domain experts without training in formal methods to validate the refinement of components they evolved. This facilitates the efficient integration of domain experts into the engineering of cyber-physical systems.

Section 4.3 is based on the publications:

Paper 9 A. Butting, O. Kautz, B. Rumpe, A. Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In: International Conference on Software Architecture (ICSA'17), Gothenburg, pages 145-154. IEEE, 2017.
Reference: [BKRW17b]

Paper 10 A. Butting, O. Kautz, B. Rumpe, and A. Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution, Patrizio Pelliccione, Jan Bosch, Mikic Marija, editors, In: Journal of Systems and Software, 149, pages 437-461, Elsevier, 2019.
Reference: [BKRW19]

Context Studies on the verification techniques of ADLs emphasize the need for automation in architecture verification [TX00, ZML10]. Yet, only a few architecture modeling techniques are sufficiently well-defined to support automated refinement and they are grounded in Focus [BS01] or the π-calculus [Mil99]. AutoFOCUS 3 [HF07b] is based on the semantics on Focus and supports model checking of architectures against LTL and CTL formulas that specify properties of component behavior [CHN11]. It does not support fully automated refinement checking. The π-ADL, based on the π-calculus, could support refinement, but considers finite traces of dynamic architectures only, where our approach considers infinite traces of static architectures.

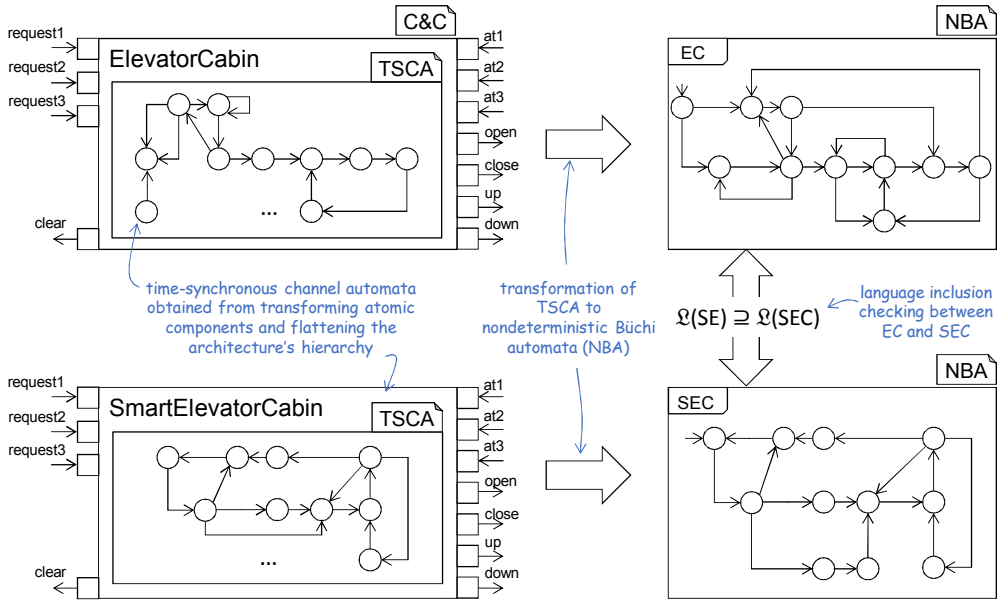


Figure 4.4: Reduction of refinement checking to inclusion of ω-regular languages.

There is a concept for the refinement of architectures specified with timed I/O automata [KLSV03]. In that concept, the semantics of a timed I/O automaton is given by a set of traces and the refinement of automata is defined in terms of trace inclusion. However, timed I/O automata are limited to one message per transition, which complicates architecture modeling and refinement with these[GR95]. Another approach to refinement checking of software architectures also relies on Focus as its semantic domain [Rin14, RRW16]. To this end, it translates component semantics into WS1S and uses the Mona [EKM98] model checker to (dis)prove refinement. The approach suffers from the high computational complexity of solving W1S1 problems.

An efficient, fully automated refinement checking for complete architectures and individual components, hence, still is missing. We conceived a representation of functional architectures as message-driven time-synchronous systems and that leverages TSPA to efficiently check their refinement.

Contribution To enable automated verification of component refinement by domain experts without training in formal methods, we conceived a novel model of message-driven time-synchronous system architectures. In this model, architectures are be represented as time-synchronous channel automata (TSCA), which are inspired by port automata [GR95], I/O* automata [RR11, Rum96], and MAA_{ts} automata [Rin14]. All of these models operate in time slices, in which they consume inputs and produce outputs, but in contrast to these models, TSCA neither require initial outputs, nor control states, consume and produce only a single message per time slice, and are guaranteed to have finitely many transitions. In contrast to TSPA, TSCA support a more powerful compo-

sition operator that is associative and commutative. Leveraging this operator enbables defining an intuitive notion of system architecture, which is not possible with the TSPA composition operator. Overall, checking refinement based on TSCA eliminates unnecessary complexity, yet TSCA can represent common component & connector architecture models. Based on TSCA, we present a sound and complete method for the semantic differencing of such architectures. Given an architecture model as a TSCA, our approach reduces the semantic differencing problem for such automata to the language inclusion problem for Büchi automata, which is PSCPACE-complete for nondeterministic Büchi automata and decidable in general [KV96]. Through this reduction, we can calculate semantic differences between two TSCA on a push-button basis, automatically produce witnesses (counterexamples), and ultimately facilitate the semantic evolution of such system architectures.

To this end, our contributions are the notion of TSCA, their novel composition operator, a method for their semantics-preserving trimming to reduce the complexity of refinement, and an efficient method for refinement checking that mitigates the state explosion problem by leveraging the aforementioned contributions. If an architecture is not a refinement of another, our method fully automatically calculates a behavior that is possible in the one architecture but not in the other. This behavior serves as a witness and is a concrete disproof for refinement. Domain experts can use the witness as evidence for efficiently identifying the model parts that cause the non-refinement.

Figure 4.4 illustrates our approach on the example of an architecture model of an elevator cabin `ElevatorCabin`, which is part of the elevator control system [SW99] illustrated in Figure 4.3, and its successor version `SmartElevatorCabin`. In contrast to the predecessor version, the `SmartElevatorCabin` only reacts to requests at floors where there has not been made a request yet. The observable behavior of it shall be a refinement of the observable behavior of `ElevatorCabin`. To verify that, first both architectures are translated into TSCA (depicted left). In this process, their hierarchies are eliminated. Then both are translated into the nondeterministic Büchi automata `EC` and `SEC` (depicted right). Ultimately, it is checked whether the language of `SEC` is included in the language of `EC`. If that holds, the behaviors of `SmartElevatorCabin` are a subset of the behaviors of `ElevatorCabin` and the architecture `SmartElevatorCabin` indeed is a refinement of the architecture `ElevatorCabin`.

Applying this transformation reduces semantic component refinement to language inclusion on Büchi automata, which can be solved automatically. Hence, for architectures representable as TSCA, domain experts can easily check whether their contribution in form of an atomic or composed components still is a refinement of its previous version prior to contributing it to systems integration. If refinement is refuted, counterexamples, *i.e.,* difference-witnessing pairs of input/output message sequences are produced. Hence, domain experts can easily comprehend why their evolved successor component is no refinement of its predecessor.

Conclusions Refinement checking for components representable as TSCA can be fully automated in an efficient fashion by translating the component and its predecessor into

weak Büchi automata. With these, component refinement can be reformulated as the language inclusion problem between these Büchi automata for which efficient algorithms exist. Fully automating refinement checking can greatly increase the pace of each refinement step, improve overall systems engineering efficiency, and facilitate the integration of domain experts.

4.4 Summary

This chapter summarizes novel modeling techniques and methods to reduce the conceptual gap between functional and geometric-physical architectures of CPS, automatically decompose functional CPS architectures and evolve these systematically. To this end, we conceived a novel metamodel that combines results from the Focus theory and design catalogs and enables modeling, as well as tracing the implementation of desired functionality within product architectures of geometric-physical realizations. The functional architectures can be decomposed according to the influence relation between two ports automatically and iteratively, yielding a collection of interconnected (less complex) components that can be realized independently by experts of the participating domains. Whether these realizations and other evolution of the architecture are indeed refinements of their predecessors can then be checked automatically, significantly reducing the effort for integration. And while we contributed three methods that can facilitate solving central challenges in engineering CPS, the vision of a pervasive MDSE of CPS demands additional contributions on modeling concepts, methods, and tools from all participating domains.

Further reading The results presented in this chapter address three essential challenges in the MBSE of CPS. To achieve the vision of a truly pervasive MDSE for CPS, many other challenges, such as improving the reuse of architecture parts and components, efficient verification of CPS properties, or the systematic structuring and analysis of related development processes, need to be addressed as well. In the research program leading to this thesis, solutions to some these challenges were conceived as well.

[ARW17] K. Adam, B. Rumpe, and A. Wortmann. Improving Reuse in Architecture Modeling with Higher-Order Components, In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIII (MBEES'17), Universität Hamburg, 2017.

[BGRW18] A. Butting, T. Greifenberg, B. Rumpe, and A. Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects, In: Martina Seidl, Steffen Zschaler, editors, Software Technologies: Applications and Foundations, LNCS 10748, pages 146-153, Springer, 2018.

[KMS+18] S. Kriebel, M. Markthaler, K. S. Salman, T. Greifenberg, S. Hillemacher, B. Rumpe, C. Schulze, A. Wortmann, P. Orth, and J. 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, 2018.

[DGH+18] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, M. Markthaler, B. Rumpe, and A. Wortmann. Model-Based Testing of Software-Based System Functions, In: Conference on Software Engineering and Advanced Applications (SEAA'18), pages 146-153, IEEE, 2018.

Chapter 5

Operating Cyber-Physical Systems with Digital Twins

There is a better way for everything. Find it.

Thomas Edison

Industrially operated cyber-physical systems can produce tremendous amounts of data. The information hidden in this wealth of data can be used to optimize the behavior of these systems and their use of resources. Yet, globally collecting all the data and analyzing it in realtime, *i.e.,* fast enough to act on analysis results with changes to the process emitting that data, often is impossible due to the sheer amount of data. A locally optimized data aggregation, abstraction, and processing that selects and combines the relevant data for various purposes could be a better way for optimizing the operation of cyber-physical systems. Model-driven concepts, methods, and tools can facilitate and improve all of these operations. Thus, this chapter summarizes model-driven contributions to the operation of cyber-physical systems (RQ4).

A pillar of Industry 4.0 and other domains of cyber-physical systems is in the digitization of participating systems, processes, and stakeholders to facilitate design space exploration, integration verification & validation, monitoring, and their optimization. Under the umbrella term "digital twinL" [BR16, GV17, SCD+12], research and industry from many domains have produced various approaches to modeling the digital representations of systems for specific purposes. These approaches define digital twins as

- a "digital equivalent to a physical product" [ASP17],
- an "always current digital image of the production system" [BMKW19],
- "a mimic of a real-world asset displaying up to date information of what is currently happening" [EDF+18],
- "an integrated virtual model of a real-world system containing all of its physical information and functional units." [PNL+19], or
- "a virtual representation of an asset used from early design through building and operation." [SHB+17].

And while there are many more approaches to define digital twins, most of these assume the same main functionality inherent in the definitions above: the main function of a digital twin is to digitally represent a system, process, or person during their (potentially

virtual) operation. Yet, most contributions on digital twins present ad-hoc solutions to very particular challenges, *e.g.,* representation of aircraft tires [ZM17], CNC machines [KLMX18], drilling rigs [MSO18], industrial robots [KOTB19], organs [LBK+19], or complete manufacturing plants [LWG+19]. Hence, regarding the components of digital twins, their systematic engineering, the integration of domain expertise into digital twins, and the use of data aggregation and abstraction in digital twins, there is a severe need for research.

This chapter aims to shed light on these challenges and to guide researchers and practitioners in leveraging MDSE concepts, methods, and tools in data aggregation and abstraction as well as modeling and operating the digital twins of cyber-physical systems. To this end, it leverages the software language engineering results and modeling methods presented in previous chapters to combine modeling language for digital twins and engineer these systematically.

Section 5.1 investigates the conception of "Digital Shadows", data structures not aiming to be a "digital equivalent" or and "integrated model [...] containing all of its [...] information", but specific, purposefully created, collected, aggregated, and abstracted data used as the basis for optimizing operation if cyber-physical systems. Section 5.2 presents a novel method for the MDSE of digital twins of cyber-physical systems that leverages the combination of MontiArc (*cf.* Section 1.4.5) and domain-specific languages to ease the engineering, deployment, and reuse of digital twins for different contexts. Section 5.3 summarizes results on the model-driven integration of digital twins with information systems to facilitate their representation. Section 5.4 summarizes our contributions to systems modeling and evolution.

5.1 On Digital Twins and Digital Shadows

Digital shadows [RSDT19, SDT18, WWB17] are the vision of collecting, aggregating, and abstracting manufacturing data for specific purposes fast enough, such that decisions made on this data can impact processes and cyber-physical systems having produced the data. In contrast to digital twins, digital shadows therefore comprise only a reduced or abstracted subset of the available data (and possibly models) to represent a system with respect to a specific purpose. For this purpose, digital shadows need to comprise various kinds of data (*e.g.,* measurement data, simulation data, models) from different sources that is abstracted and aggregated in domain-specific and application-specific ways and augmented with necessary metadata to fulfill their purpose. Through their specific abstraction and aggregation, digital shadows can be the optimal data structures to ensure timely decision-making in cyber-physical systems.

Figure 5.1 illustrates the potential benefits of digital shadows using road systems and traffic data as an example. To represent the physical road system, its digital topological model (top) can serve for general orientation or for estimating distances. Yet, the complete model is rarely necessary, but instead purposefully abstracted shadows of it, enriched with additional data are required. For instance, to optimize route planning, flexibly react to changes in the system (*e.g.,* due to congestion), and reduce the cost

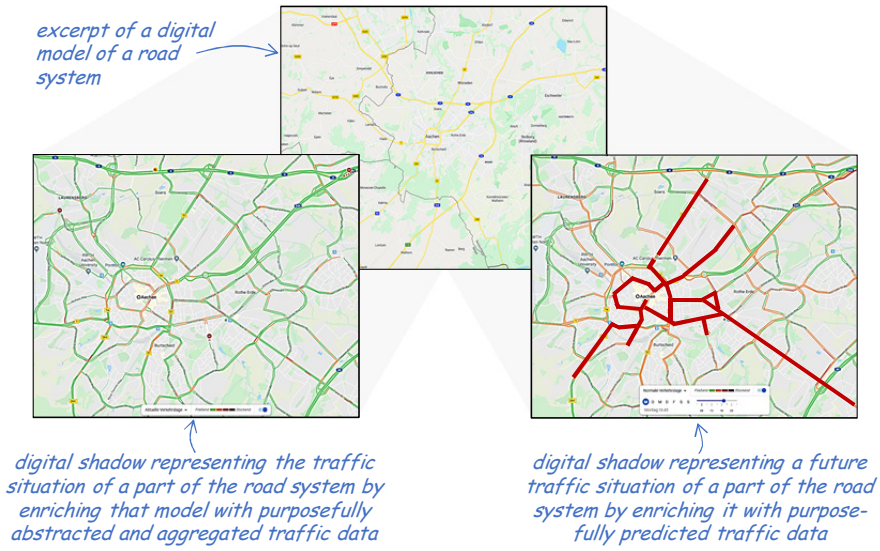


Figure 5.1: Digital shadows are data structures produced by illuminating a represented system for a specific purpose.

of traveling (in terms of time and fuel consumption), the aggregation of dynamic data on traffic, weather, and other circumstances of potential impact is required. However, this data is not required to precisely cover the complete road system model but needs precision only for excerpts of it, whereas for areas related to the route planning in question, details might be abstracted–and for unrelated areas the data is not needed at all. Moreover, additional available data related to the model (*e.g.,* restaurants in the related area and their latest reviews) can be omitted completely. By abstracting from the complete topological road system model to a specific part and aggregating related data on the current (bottom left) or predicted (bottom right) traffic situation, digital shadows for different purposes can be created more efficiently than by enriching the complete model. These purposefully created digital shadows comprise parts of the road system model as well as specific data and can be processed more efficiently (*e.g.,* to plan routes in the depicted region) than the totality of models and data they were derived from. A complete digital twin (in the sense of complete, high-precision digital replica) of the earth with all possible data, not limited to traffic, is not necessary for this. On the contrary, overwhelming decision-making systems with all available data might delay decisions dramatically.

Analogous to this, *e.g.,* manufacturing orders (models) can also be dispatched to the manufacturing process on the basis of their rigid characteristics (necessary work steps, average throughput times), but their completion dates can only be reliably planned by taking into account the current capacity utilization, machine availability, *etc.* (data). We, therefore, conceived a first explicit definition and model of digital shadows to facilitate data-driven process optimization.

The concept of digital shadows is novel and must integrate data, concerns, and methods from a variety of domains participating in Industry 4.0. Hence, realizing it needs multi-disciplinary collaboration across the different stakeholders involved in designing, engineering, deploying, operating, and maintaining manufacturing systems. Establishing such a collaboration is challenging as the involved stakeholders follow different paradigms, employ different methods, and use different terminology based on their domain of expertise. A first necessary foundation towards this collaboration, hence, is an agreement on the concept of digital shadows in general.

The Internet of Production[1] excellence cluster of RWTH Aachen University pursues the vision of interdisciplinary collaboration based on continuous horizontal and vertical exchange of manufacturing data, integrated development of specifications, efficient processing of domain-specific tasks, and cross-domain validation. For a first conceptualization of digital shadows, their use, and capabilities, we conducted a survey in the Internet of Production through which we reached out to its 200 researchers of 25 departments conducting research in a variety of domains, including artificial intelligence, computer science, innovation research, labor science, mechanical engineering, and manufacturing technology.

Section 5.1 is based on the publication:

Paper 11 G. Schuh, C. Häfner, C. Hopmann, B. Rumpe, M. Brockmann, A. Wortmann, J. Maibaum, M. Dalibor, P. Bibow, P. Sapel, and M. Kröger. Effizientere Produktion mit Digitalen Schatten, In: Wilhelm Bauer, Wolfram Volk, Michael Zäh, editors, In: ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb, 115, pages 105-107, Hanser, 2020.
Reference: [SHH+20]

Context The term "digital shadow" has been proposed by members of the Internet of Production earlier [SBRB16, SWL+16] and is still considered essential for enabling the data exchange necessary for the vision of the Internet of Production. The early publications on digital shadows have in common that the concept itself is neither sufficiently distinguished from the concept of the digital twin, nor sufficiently precise.

This also explains why many publications use terms digital twin and digital shadow interchangeably, whereas others consider qualitative differences between both concepts. For instance, there are various conceptions of digital twins that consider these as models of datasets aggregated and abstracted for a specific purpose that represent a cyber-physical system, person, or process but cannot interact with it [DR18, HSFH18, Kue18, SF18, SSP+16, WB17]. Others consider digital twins as systems that are able to interact with the represented systems or processes. Such digital twins often are considered active models or systems that, maybe automatically, perform operations with the represented systems or processes to optimize its use [PJIZ18, ZZL+19]. Hence, there is a qualitative difference between both concepts that warrants investigation. Yet,

[1] Internet of Production: `https://www.iop.rwth-aachen.de/`

surveys on digital twins focus on success stories [MSO18], state-of-the-art application of digital twins [DHA+18, TZLN18], or specific challenges digital twins are applied to [BWK18, DHA+18, Kra16]. There are no studies investigating or even considering the distinction between digital twins and digital shadows.

Contribution To shed light onto the distinction between digital shadows and digital twins, we conceived a novel interdisciplinary questionnaire of 21 questions arranged in three sections on (1) the purpose and understanding of digital shadows in contrast to digital twins; (2) the expected functions and capabilities of digital shadows and digital twins; and (3) the use of digital shadows and digital twins through the engineering lifecycle of a system. After a pilot survey with selected participants from mechanical engineering, systems engineering, and factory planning, we revised our survey to use less specific terminology and presumptions from computer science and software engineering and rephrased the questions to be better comprehensible by experts of other domains. Based on the results of the survey, we conducted multiple workshops with participants of different research domains of the Internet of Production to better understand the answers given in response to the questionnaire and to refine the findings further. Based on these findings, we conceived the following definitions:

> A digital shadow is a set of contextual data traces and/or their aggregation and abstraction collected concerning a system for a specific purpose with respect to the original system.

Digital shadows are digital artifacts without physical representation that consist of data traces that are generated, preprocessed, and possibly augmented for a specific purpose with respect to a specific system. In addition to the data traces, they need to contain contextual data (*e.g.,* source of data, time of creation, precision, reliability) that is required for the semantic processing of the shadows by other systems, *e.g.,* digital twins, manufacturing systems, product lifecycle management systems, or others.

This entails that digital shadows can originate from different sources (*e.g.,* sensor signals, simulation data, strategic enterprise-level data) and that this data can be created by aggregating or filtering other data traces (*e.g.,* to include the temperature gradient from several temperature measurements). This also entails that digital shadows can contain complex models produced by the represent system or related systems. Especially, the digital shadows might be models of modeling languages themselves.

While there are different opinions on whether a represented system can have multiple digital twins, it is crucial that it can cast a multitude of different digital shadows depending on the illuminated aspects of observation. Thus, purpose-specific digital shadows are created by selecting the required parameters, gathering the raw data from the source of interest, cleaning and abstracting it. In contrast to digital twins, digital shadows always are passive in the sense that they do not interact with the represented system but serve as the basis for making decisions about that system and for influencing it through other systems and services. Based on these findings, we propose the conceptual model of digital shadows as presented in Figure 5.2.

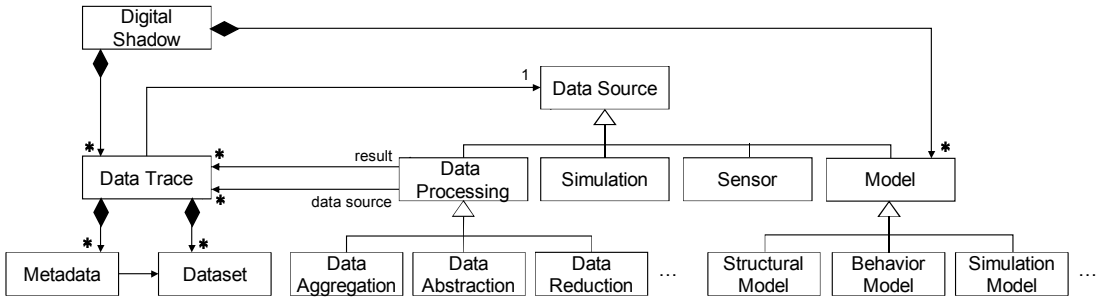


Figure 5.2: Conceptual model of digital shadows.

In this model, digital shadows comprise data traces and models. Data traces consist of datasets and related metadata. Each data trace yields a single data source describing its origin, which can be a sensor, results from a simulation, a model, or a data trace processing activity that itself is based on data traces. The models can be of different kinds, such as structural models, *e.g.,* SysML block diagrams or CAD models, behavior models, such as physical equations or state-based behavior descriptions, knowledge bases, and many more. Metadata comprises domain-specific data relative to the purpose of the digital shadow, which may explain the semantics of data. Typical data processing activities related to digital shadows are abstraction, aggregation, or data transformation. Based on the survey and the subsequent workshops, we also conceived the following definition of digital twins:

> 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.

Again, we use the term system in a wide sense that includes, *e.g.,* systems, processes, and persons. Where the digital shadows are passive data structures focusing on data that changes during system operation, the digital twins focus on representing a cyber-physical system using structural and behavioral models as well as digital shadows. To this end, we conceive digital twins as systems featuring models, data, and related services that may manipulate the represented system themselves (*e.g.,* to control or optimize its behavior or prevent failures). Consequently, a digital twin might include CAD models of manufacturing systems, models of the stakeholders of related business processes, or name and birthday of a patient.

Conclusions Considering the digital twin as an exact replica of a represented system is a highly idealistic vision hardly feasible in reality. To achieve such a vision, every concern of the represented system, from its subatomic behavior to the impact that celestial bodies effect on it, which, *e.g.,* would be relevant to fluid processes such as injection molding, must be considered. Our concept of digital shadows aims to capture, process, and analyze only the data and models necessary for a specific purpose. Collecting, constructing, and

combining digital shadows for specific purposes instead of aiming for all-encompassing digital twins can greatly facilitate making the vision of Industry 4.0 (*cf.* Section 2.1) a reality.

5.2 Pervasive Model-Driven Digital Twins

The investigation of digital twins detailed in the previous section assumes a birds-eye view on the concept and its understanding in manufacturing. Through this, we found that the large majority of publications on digital twins either reports on generic concepts for digital twin engineering that abstract from technical realizations or presents successful applications of nondescript digital twin realizations. The abstract presentations of digital twins and their implementations make applying the presented solutions in other application contexts or domains impossible.

There are only a few publications presenting reproducible and reusable architectures for digital twins. Where such architectures are proposed, their integration with domain-specific expertise is not considered. To mitigate this, we propose investigating digital twins through the lens of MDE (1) top-down, from a systems engineering vantage point, and (2) bottom-up, from the challenges of a specific domain, at the same time. Focusing on the models of discourse can facilitate making domain expertise explicit. Investigating their engineering top-down and bottom-up can shed light on the cross-domain systems engineering concerns, challenges, and solutions for reproducible digital twin architectures as well as on the domain-specific concepts, methods, and techniques relating to these.

This approach to digital twin research demands the close collaboration of experts from participating domains, which the Internet of Production excellence cluster enables. Hence, to support researchers and practitioners in engineering digital twins for cyber-physical systems in manufacturing, we conceived a method for the pervasive model-driven engineering of digital twins together with experts from plastics processing. This method focuses on (1) a generic and extensional component & connector architecture; (2) structure and behavior models reifying expertise, data sources, and machinery of plastics processing; (3) the systematic customization of the architecture with the domain-specific models; and (4) producing executable digital twins from their combination.

Section 5.2 is based on the publication:

Paper 12 P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, A. Wortmann. Model-Driven Development of a Digital Twin for Injection Molding, In: Advanced Information Systems Engineering, 32nd International Conference, CAiSE 2020, Grenoble, France, June 8–12, 2020, Proceedings, pages 85-100, Springer, 2020.
Reference: [BDH+20]

Context Digitization poses major challenges for companies worldwide. In the vision of Industry 4.0, comprehensive data acquisition and analysis enables highly flexible, versa-

tile value-added systems to increase productivity at reduced costs. One pillar of these versatile value-added systems in Industry 4.0 is the idea of digital twins, replicas of cyber-physical systems that represent these for various purposes, ranging from monitoring, behavior optimization and prediction in a multitude of application domains including automotive [DB18, KMC+18], construction [KX18, SDLJ19], drilling and mining [NMR+18, PNBL+19], manufacturing [WYG+19, ZLX18], medicine [CCSN19, LBK+19], robotics [HE17, VCGP19], and sports [BC19], in which they represent the relevant systems, persons, or processes.

Most of this research focuses on very specific applications with little generalizability. Where more general architectures are presented, these most often focus on abstract conceptual guidelines [BVM19, CAM+17, DDP+17] for engineering digital twins, such as arranging the digital twins together with their environments in specific layers [ASLC17, BS17, RBK18], or their implementation with specific GPLs [PJIZ18]. Overall, there is a lack of research regarding the systematic engineering of generalizable architectures for digital twins that support integrating domain expertise.

Contribution The systematic engineering of digital twins for cyber-physical systems needs to consider the overall digital twin behavior, its connection to various kinds of data sources, its connection to its cyber-physical system counterpart, and the integration of (domain-specific) expertise. Making these aspects of a digital twin explicit in domain-specific models that can be understood and developed by experts of the respective domains, integrated, and translated into a digital twin implementation is a necessary prerequisite for that.

To facilitate the engineering of digital twins with experts of participating domains, we conceived a model-driven method centered around an extensible MontiArc architecture that provides components to process operation data from the cyber-physical systems into digital shadows, evaluate the necessity to act on the observed systems based on domain-specific rules, and act accordingly. We conceived this architecture in close collaboration with experts from plastics processing and evaluated it in the context of injection molding.

The digital twin architecture, illustrated in Figure 5.3, describes a single digital twin, operates in the context of a UML/P class diagram domain model, and relies on two DSLs tailored to domain experts: An event DSL describes the conditions under which the digital twins need to act and a design of experiments (DoE) DSL that describes experiments to find optimal values for manufacturing parameters. To further ease reusing this architecture, it makes its connections to sources and the cyber-physical systems explicit. Therefore, our method also includes a compact DSL for specifying connections to the Apache Kafka [Gar13], a popular platform to stream manufacturing data [PC16, PRCO16], to read live data from the observed CPS. To control the observed system, it features a second compact DSL for specifying OPC-UA [LM06] connections to system. With OPC-UA being a popular middleware for industrial automation [CJOC10, HS14, IJ13], this enables applying our method and architecture to a variety of cyber-physical systems.

Overall, the digital twin architecture comprises five subcomponents that itself are

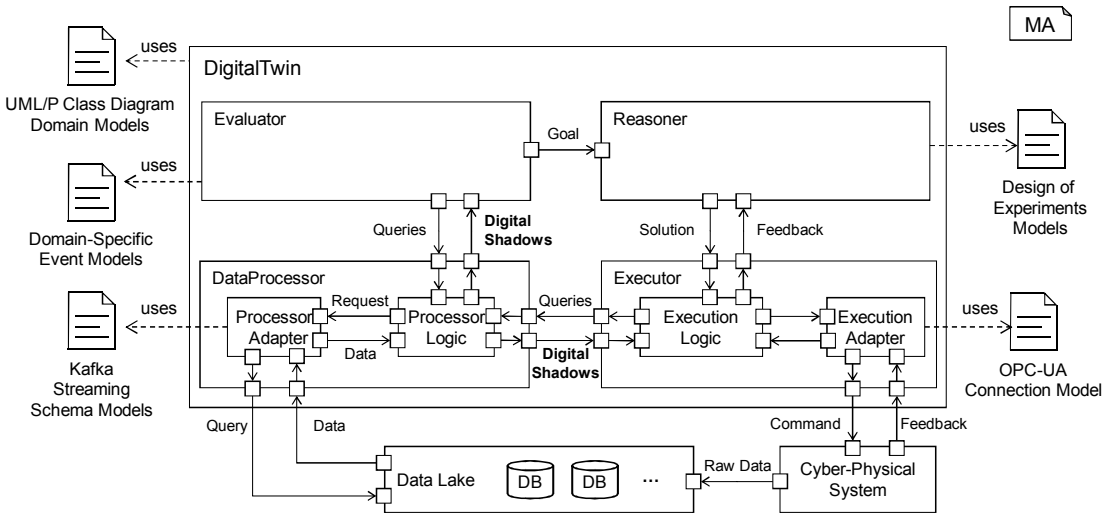


Figure 5.3: At the core of our method to the model-driven engineering of digital twins is a MontiArc architecture connected to models of various DSLs.

composed again: To decouple the technical details of data retrieval from the logical translation of data into digital shadows, the `DataProcessor` comprises the subcomponents `ProcessorAdapter` and `ProcessorLogic`. The `ProcessorAdapter` subcomponent leverages novel Kafka streaming schema models to connect to the various data sources of a data lake. The subcomponent `ProcessorLogic` which translates data received from the to digital shadows based on queries from the `Evaluator`. It creates the digital shadows and emits these through the corresponding outgoing port of `DataProcessor`. The `Evaluator` interprets domain-specific event models to decide whether the digital twin needs to react to changes in the observed system or its environment. To clarify whether this is necessary, additional information might be required, which can be requested by sending queries for additional digital shadows to the `DataProcessor`. During this, the knowledge base of the digital twin might be updated to reflect these changes. If an event's condition is fulfilled, a goal is emitted to the `Reasoner`. The `Reasoner` receives goals from the `Evaluator` and selects suitable DoE models to conduct experiments to find valid parameters for the cyber-physical systems. During this, it might update its knowledge base. Once a suitable DoE is identified, this is emitted as a `Solution` to the `Executor`. Similar to the `DataProcessor`, the `Executor` comprises two subcomponents to decouple execution logic from the technical connection to the cyber-physical systems. The subcomponent `ExecutionLogic` translates the DoE into a sequence of platform-specific actions that should be applied to the cyber-physical systems. The subcomponent `ExecutionAdapter` leverages an OPC-UA connection model to translate these actions into OPC-UA commands that it sends to the systems.

From the UML/P class diagram domain model, the architecture model, the event-

action models, the DoE models, and the connection models, an integrated implementation of the digital twin is generated. This implementation connects to a cyber-physical system and a data source, produces digital shadows according to event models and based on data of the observed system, evaluates these with respect to the conditions of the event models. If an event triggers, *i.e.,* observed properties of the observed system deviate from expectations, a DoE is conducted to identify an operational configuration of the system again. Unless continuing the operation is identified as hazardous, the cyber-physical system continues to operate during the DoE, *e.g.,* it continues producing goods, to automatically improve operation parameters and evaluate results.

Through exchanging the event models and DoE models, our digital twin architecture can easily be tailored towards different applications and by adjusting the Kafka models and OPC-UA model, the architecture also can be used with different cyber-physical systems with little effort. To adjust the architecture to different applications and cyber-physical systems, we conceived the systematic customization process illustrated in Figure 5.4.

First, domain experts create the domain model and the event models relating to the data types defined in the domain model. After all relevant events have been identified and modeled, domain experts create the DoE models to react to these events. To bind the digital twin architecture to data sources and the cyber-physical systems, the afterwards create Kafka schema models and a single OPC-UA connection model. Based on the architecture model and the models create during the customization process, our toolchain generates an implementation of the architecture comprising Java artifacts reifying the domain model's data types, the architecture's components, and the bindings for data sources and the cyber-physical systems. Moreover, these artifacts generated to be aware of the event models and DoE are capable of interpreting these. Ultimately, the `Reasoner` and the `Executioner` can be adjusted further if necessary.

Conclusions Overall, leveraging MDE with DSLs for the engineering of digital twins eases the inclusion of domain expertise in the digital twins, which is crucial to their effective application. Using an extensible architecture as the basis for digital twins facilitate their future extension with novel components, such as user interfaces, sophisticated planners, or additional failure handling. By selecting MontiArc (*cf.* Section 1.4.5) as the basis of our method, engineers developing digital twins can leverage automated decomposition (Section 4.3) and refinement checking (Section 4.3) without additional effort. The separation of digital twin business logic and technical realization through Kafka schema models and OPC-UA connection models encourages reusing the architecture with different data sources and cyber-physical systems.

The presented architecture leverages an event DSL and a DoE DSL tailored to domain experts. These languages might need adjustment for application in different domains. Through the modular language engineering (*cf.* Chapter 3), this becomes feasible. Further challenges are (1) that different digital twins will require different forms of reasoning to control, optimize, and predict the behavior of the represented cyber-physical systems; (2) the domain-specific visualization of digital shadows by the digital twin; (3) the diver-

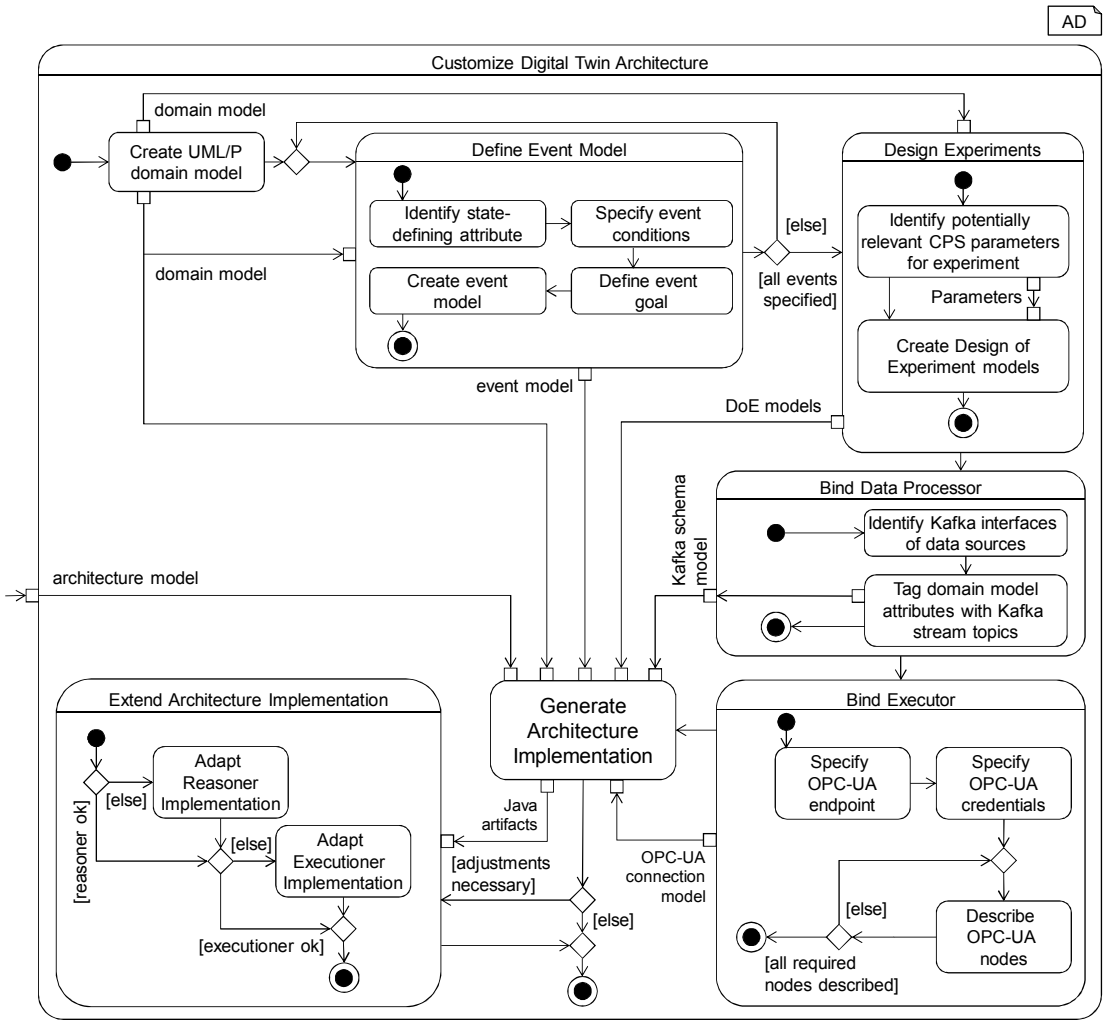


Figure 5.4: Our method for the systematic MDSE of domain-specific and reusable digital twins leverages an extensible MontiArc architecture and four DSLs.

gence of expected behavior (*e.g.,* as reified in event models) and actual behavior of the observed systems, which might degrade the effectiveness of the digital twin; (4) encoding mode complex domain expertise; and (5) the composition of digital twins of multiple cyber-physical systems to a system of digital twins.

5.3 Representing Digital Twins with Information Systems

The efficient and agile creation of digital twins [TCQ+18] is a concern of interest in many disciplines of cyber-physical systems, such as marine [JN19], smart manufacturing [TCQ+18, TQWN19], avionics [MPPU19], infrastructure and energy manage-

ment [LXH+20] or automotive [BW19]. In many of these domains, the digital twins are considered to comprise models of the systems, data traces (digital shadows), and often services that enable using the digital twins purposefully with respect to the original system (*cf.* Section 5.1). As the quintessential purpose of digital twins is representing (the activities of) a cyber-physical system to human operators or other systems (*e.g.,* production dashboards), means for their efficient representation and integration into existing representation mechanisms are vital to the success of digital twins.

Information systems [AMN+20, BD20], such as manufacturing dashboards [GHH+13, GS14] or finance cockpits [ANV+18], provide established and effective means to manage cyber-physical systems [JYE19]. Hence, they also are an obvious interface for the representation of digital twins and their data (*i.e.,* shadows) as well as for structuring the interaction with operators and the cyber-physical systems. Accordingly, the participating cyber-physical systems and information systems have to share many interfaces to communicate about data, models, and user inputs.

Traditionally, the connections between the interfaces of information systems and cyber-physical systems are crafted manually. The corresponding implementation activities are highly repetitive and do not demand a high cognitive performance of the developers. Yet they are, due to the multitude of interfaces, time-consuming and error-prone. To support researchers and practitioners in the engineering and operation of digital twins, applying MDSE to creating the connections between digital twins and information systems can mitigate these challenges, facilitate realizing the representation of digital twins, and hence, foster their application.

To efficiently support engineers in representing digital twins via information systems, a model-driven approach to their connection, thus, should ensure that (1) both systems need to be able to exchange data of types known to both systems; (2) the communication infrastructure connecting both should be completely generated; (3) the handcrafted parts of cyber-physical systems and information system should be transparent to the systems' communication; and (4) the information system should be able to influence the behavior of the digital twin.

Section 5.1 is based on the publication:

Paper 13 J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, A. 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, 2020.
Reference: [KMR+20]

Context There are already various solutions to the MDSE of Internet of Things systems. For instance, ThingML [HFMH16, MHF17] is a modeling framework for describing IoT systems and their behavior in a DSL. From these models, the framework generates C, Java, or JavaScript artifacts realizing the different components of an IoT system. Calvin [PA15] is a modeling language for defining the distributed architecture of IoT

systems in a syntax similar to MontiArc (*cf.* Section 1.4.5) using its CalvinScript DSL. MDE4IoT [CS16] uses a variant of UML to describe IoT systems and their deployment to physical devices. SysML4IoT [HLR17] is a modeling language for the engineering of adaptive IoT systems using a variant of SysML. CapeCode [BJK+18], based on Ptolemy [Pto14], provides a graphical user interface for combining IoT components as reusable building blocks.

These solutions support specifying message exchange and serialization between IoT systems but lack means for automatically synchronizing the messages from the IoT systems with information systems. And while it might be possible to leverage the DSLs of these solutions to manually craft digital twins and information systems, there is no systematic support for that in any of them. Hence, principally, the synchronization of digital twins with information systems is generally supported but requires considerable manual effort and in-depth expertise about information systems, cyber-physical systems, and digital twins. Hence, even with popular IoT solutions, connecting devices as simple as a temperature sensor can be unnecessarily complicated [LG18].

Contribution To address the abovementioned challenges and facilitate the representation of digital twins with information systems, we have conceived a model-driven method for their integration. This method exploits that both the data structures the information system is based upon, the architecture of the digital twins (*cf.* Section 5.2), and their connection are modeled using appropriate modeling languages (*cf.* Chapter 2). Therefore, it comprises the model-driven integration of the digital twins and the and information systems and a fully automated transformative extension of digital twin architecture models and information system data models to keep their data synchronized.

MontiGem [AMN+19, GHK+20] is a code generation framework for enterprise information systems that translates UML/P [Rum16, Rum17] class diagram models that are augmented with user interfaces models into web-based information systems that operate on databases [GMN+20] employing schemata generated from the class diagrams. Hence, the essential information represented in the information systems are data conforming to the class diagrams and stored in the generated databases. Our method for the integration of information systems generated with MontiGem with digital twins modeled as MontiArc architectures therefore aims at connecting attributes of MontiGem's data model with ports of the digital twin's architecture. Hence, we leverage the notion of tagging languages [GLRR15].

Tagging is a modeling mechanism in which a base model is enriched with additional information, such as platform-specific details, to prevent pollution of the base model and to facilitate reusing it in different contexts [MRRW16]. To this effect, from a tagging schema relative for the base DSL is specified from which a tagging language specific to the base DSL, *i.e.,* by reusing parts of its syntax, is generated. Using models of this domain-specific tagging language, models of the base DSL can be annotated in a type-safe fashion, which reduces the possibilities for error. Usually, models of the base language then are processed together with their corresponding tag models, *e.g.,* to consider platform-specific information in the transformation of the base model [DJK+19b].

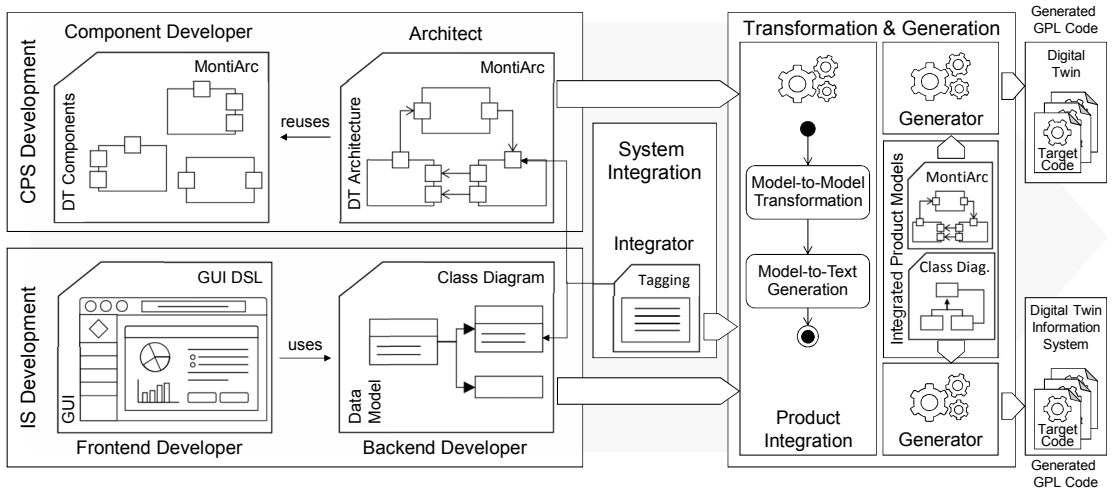


Figure 5.5: Tagging facilitates the connection of digital twins and information systems while clearly separating the related concerns.

To connect the information system with the digital twin, we developed a tagging schema that supports tagging the attributes of a base UML/P class diagram model with information from a MontiArc model, *i.e.,* its ports. Through this, engineers can easily establish a directed relation between a port reading or writing the value of an attribute of a class presented by the information system (or vice versa). Based on this tagging, the models of both systems, the information system class diagram and the digital twin architecture are augmented with classes or components responsible for sending or receiving information accordingly before both models are translated into executable systems by their corresponding generators.

The overall process is illustrated in Figure 5.5: The activities of creating the digital twin architecture and creating the information system models can be performed in parallel and without any exchange of information between the responsible developers. A system integrator or digital twin operator then creates tag models that link ports of the MontiArc digital twin architecture to information system class diagram attributes. Our toolchain takes the digital twin models, information system models, and tag models as input, transforms the architecture and the class diagram data models accordingly, and adds GPL implementations for the behavior of digital twin components and information system classes responsible for communication. Afterwards, these models are passed to the respective generators. As the digital twin models and the data models are unaware of any communication (which is realized through their GPL implementations), these generators do not need to consider their communication at all.

Conclusions Representing digital twins properly is a quintessential concern in operating cyber-physical systems with digital twins. By connecting digital twins to information systems, their data can be represented using their powerful graphical interfaces. To ease

the integration of cyber-physical systems with information systems, we exploited the generative processes of the MontiGem information system and linked the data structures that are generated from the class diagrams to the ports of the digital twin architecture. Generalizing this demands that information systems are developed in a model-driven fashion as well. Where this is possible, Leveraging the mechanism of tagging for the model-driven integration of digital twins with information systems separates the concerns of digital twin engineering and information system development, enables the automated transformation of their models, and can be realized transparently with respect to the individual toolchains of digital twins and information systems. Hence, our approach yields significant benefits over their handcrafted integration and contributes to the overall MDSE of cyber-physical systems and their operation.

5.4 Summary

The contributions summarized in this chapter address selected challenges in the digitization of CPS. To this end, it presented results of an interdisciplinary study aiming to identify, shape, and separate the concepts of digital shadows and digital twins. Based on the understanding gained from this study and constructive research conducted throughout the research program presented in this thesis, we conceived a reusable, model-driven architecture for digital twins that operates on digital shadows and incorporates domain expertise through novel DSLs. These DSLs support the description of digital shadows, the modular and platform-independent development of domain-aware digital twin architectures, and the binding of these architectures to specific platforms. Through linking the digital twin model to the data model of an information system, we also contributed a novel method for the model-driven integration of digital twins with information systems that greatly facilitates their representation.

Our contributions, hence, foster an interdisciplinary understanding of the concepts of digital shadows and digital twins and greatly facilitate the systematic engineering of digital twins. The results of the presented contributions, thus, not only guide and support researchers and engineers focusing on CPS to systematically conceive, develop, and operate CPS with digital twins but also serve as a foundation for creating novel model-driven methods to advance the digitization through digital twins.

Further reading Digitization is a tremendous challenge with manifold concerns to consider. Throughout the research program presented in this thesis, we made further contributions to the digitization of activities involving CPSs that address connecting the digitized components of a smart factory, connecting CPS architectures to the Internet, and making these self-explainable.

[BKL+18] C. Brecher, E. Kusmenko, A. Lindt, B. Rumpe, S. Storms, S. Wein, M. von Wenckstern, and A. 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), pages 1-9, ACM, 2018.

[DJK+19c] M. Dalibor, N. Jansen, J. C. Kirchhof, B. Rumpe, D. Schmalzing, and A. Wortmann. Tagging Model Properties for Flexible Communication, In: Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, Andreas Wortmann, editors, Proceedings of MODELS 2019. Workshop MDE4IoT, pages 39-46, CEUR Workshop Proceedings, 2019.

[BGG+19] M. Blumreiter, J. Greenyer, F. J. C. Garcia, V. Klös, M. Schwammberger, C. Sommer, A. Vogelsang, and A. Wortmann. Towards Self-Explainable Cyber-Physical Systems, In: Jeff Gray, Matti Rossi, Jonathan Sprinkle, Juha-Pekka Tolvanen, editors, Proceedings of MODELS 2019. Workshop Models@run.time, pages 542-547, IEEE, 2019,

Chapter 6

Conclusion

The habilitation thesis at hand summarizes substantial results for the pervasive model-driven systems engineering of CPS with domain experts. This vision demands suitable domain-specific modeling languages, semantics-aware automated modeling methods, and systematic concepts for the efficient operation of CPS. Consequently, we pursued four research questions.

First, we investigated the use of modeling and modeling languages in the advanced and important domain of CPS for Industry 4.0. To produce systematic evidence on modeling for the CPS of Industry 4.0, we conducted a systematic mapping study, including more than 4.000 publications. In this study, we investigated which modeling languages are applied in Industry 4.0, which concerns these languages are applied to, and which automation support modeling contributes to Industry 4.0. We found that modeling for CPS needs to address a variety of concerns, including the representation of object-oriented concerns as well as geometric-physical concerns, and knowledge about the systems under development. To this end, often general-purpose modeling languages, such as UML or SysML, variants of these, or ad-hoc DSLs are employed. Our literature study shows that there is a strong interest in leveraging the benefits of DSLs in systems engineering, especially to address concerns and concern combinations hardly expressible with general purpose modeling languages. Moreover, code generation appears to be an important motivation for modeling. We infer that easing the integration of truly domain-specific modeling languages that cover the (combinations of) concerns identified essential in the study and employ concepts and terminology of the corresponding domains as first-order elements, can greatly facilitate the model-driven engineering of CPS. This, however, needs novel language engineering techniques that systematically advance the engineering and combination of such languages.

Second, we conceived concepts and methods to expedite engineering of such DSLs based on applying time-honored concepts from software engineering, including component-based software engineering, software product lines, and separation of concerns, to software language engineering. We presented a novel systematic method for the efficient engineering of DSLs based on the efficient composition of grammar-based syntaxes and semantics realizations based on code generators. For such languages, we conceived a concept of language components with explicit interfaces that make their provided and required extensions explicit. Through these, independently developed language components can be composed into new components, which fosters the engineering of reusable DSL building blocks and their efficient combination to truly domain-specific languages

as required by different experts. Leveraging a two-phase reuse process based on planned reuse expressed through the closed variability of language product lines over these components and open variability through post-hoc customization of these components, the creation of complete DSLs actually tailored to domain experts can be achieved with unprecedented efficiency. Of course, modeling languages need to be supported by useful processes.

Third, we conceived novel modeling methods to address challenge typical to engineering CPS, *i.e.,* the integrated development of functional architectures and geometric-physical product architectures, the decomposition of monolithic functional architectures into components to be realized by domain experts, and the automated validation that the domain experts' solutions indeed are refinements of these components. To reduce the conceptual gap between functional architectures and geometric-physical product architectures, we conceived a modeling method integrating both, that facilitates tracing functional requirements to geometric-physical realizations and hence fosters reuse in product development processes for CPS. For separating monolithic functional architectures into components to be realized by domain experts, we devised a notion of influence relation between ports of the architecture's components and a process to fully automatically decompose such architectures according to that relation. This reduces the efforts of systems engineering is systematically integrating components reifying domain expertise. To validate that the component implementations provided by domain experts indeed are refinements for the components received for implementation, we contrived a method that enables checking such refinements automatically and produces witnesses if refinement is violated. This facilitates systems engineering by liberating systems engineers from checking refinement manually. But CPS, especially in Industry 4.0, are meant to be operated for many years, hence supporting their operation is crucial to their success.

Fourth, to support the operation of CPS, we conceived a notion of digital shadows and digital twins that support distinguishing passive data traces (digital shadows) from active systems operating a CPS (digital twins). Based on the latter, we conceived an extensible architecture for digital twins as a functional architecture that itself can be subjected to the methods summarized in this thesis. This platform-independent architecture operates on digital shadows and leverages a variety of DSLs to bind itself to data sources and CPSs, as well as to reify the domain expertise necessary for the optimization of operation. As the proper representation of a CPS is the quintessential purpose of digital twins, we leverage this MontiArc architecture, the MontiGem information system generator, and domain-specific tagging languages to connect digital twins to information systems with minimal effort and while clearly separating the concerns of both. Through both, model-driven methods greatly contribute to facilitating the operation of digital twins with CPSs.

To realize the vision of pervasive MDSE, future work is necessary in, at least, three areas, including SLE, digital twins, and support of model-driven development processes. The notion of systematic language engineering is an unprecedented foundation for the holistic composition and black-box reuse of DSLs. Yet, it focuses on textual DSLs with code generators and relies on conservative embedding as the composition opera-

tor of choice. As there are other technological options for language engineering, such as metamodels or M2M transformations, as well as other operations for combining languages [EGR12], such as inheritance or aggregation, research in unifying these into a holistic concept of language modules is necessary. To add further benefit, such modules should comprise language-related technology crucial to the success of DSLs, such as editor fragments or debuggers. Moreover, functional architectures often exist in the context of other modeling paradigms, such as geometric-physical modeling or knowledge representation. Currently, the automated analyses and syntheses capable of processing purely functional architectures cannot process such information. Here, novel concepts for the translation of other modeling paradigms into functional architectures or language engineering techniques enabling extending these analyses and syntheses to other modeling paradigms are required. Research in the foundations of multi-paradigm modeling [ABH+19] might yield contributions to this. Also, the integration of modeling languages and tools that reify the concerns of the different domains participating in the development of CPS is crucial to the success of MDSE for their systematic engineering. And while there are various research contributions as well as commercial tools for this, the integration of modeling tools still is one of the major challenges in industrial practice.

Overall, the research program presented in this thesis addresses selected interconnected concerns in the engineering and operation of CPS with model-driven concepts, methods, and tools that contribute to the efficient and systematic engineering of complex CPS with domain experts. It produced systematic evidence on the use of modeling for CPS, foundations for the efficient integration of domain experts through suitable DSLs, novel methods for the systematic and automated modeling of architecture and behavior of CPS architectures and unprecedented model-driven concepts for their operation. Yet, there is much research to be done.

Bibliography

[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.

[ABH+19] Moussa Amrani, Dominique Blouin, Robert Heinrich, Arend Rensink, Hans Vangheluwe, and Andreas Wortmann. Towards a Formal Specification of Multi-paradigm Modelling. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MPM4CPS*, pages 418–423. IEEE, September 2019.

[ABK+18] Kai Adam, Arvid Butting, Oliver Kautz, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Type-safe Interfaces into Template-based Code Generators. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 179 – 190. SciTePress, January 2018.

[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.

[AK03] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[AKM13] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-Oriented Software Design. In *International Conference on Model Driven Engineering Languages and Systems*, pages 604–621. Springer, 2013.

[Alu15] Rajeev Alur. *Principles of Cyber-Physical Systems.* MIT Press, 2015.

[AMN+19] 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 *Digital Ecosystems of the Future: Methods, Techniques and Applications (EMISA'19)*, LNI, pages 1–8, 2019. (in press).

[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.

[ARW17] Kai Adam, Bernhard Rumpe, and Andreas Wortmann. Improving Reuse in Architecture Modeling with Higher-Order Components. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIII (MBEES'17)*, Univ. Hamburg, March 2017.

[ASLC17] Atin Angrish, Binil Starly, Yuan-Shin Lee, and Paul H Cohen. A flexible data schema and system architecture for the virtualization of manufacturing machines (VMM). *Journal of Manufacturing Systems*, 45:236–247, 2017.

[ASP17] Giuseppe Avventuroso, Marco Silvestri, and Paolo Pedrazzoli. A networked production system to implement virtual enterprise and product lifecycle information loops. *IFAC-PapersOnLine*, 50(1):7964–7969, 2017.

[ASSS13] Luciane Telinski Wiedermann Agner, Inali Wisniewski Soares, Paulo CéZar Stadzisz, and Jean Marcelo SimãO. A Brazilian survey on UML and model-driven practices for embedded software development. *Journal of Systems and Software*, 86(4):997–1005, 2013.

[AVH04] Grigoris Antoniou and Frank Van Harmelen. Web Ontology Language: OWL. In *Handbook on ontologies*, pages 67–92. Springer, 2004.

[Bal00] Helmut Balzert. Software-Entwicklung (Lehrbuch der Software-Technik, Band 1), 2000.

[BBL+16] Luca Berardinelli, Stefan Biffl, Arndt Lüder, Emanuel Mätzler, Tanja Mayerhofer, Manuel Wimmer, and Sabine Wolny. Cross-disciplinary engineering with AutomationML and SysML. *at-Automatisierungstechnik*, 64(4):253–269, 2016.

[BBT+07] John Bailey, David Budgen, Mark Turner, Barbara Kitchenham, Pearl Brereton, and Stephen Linkman. Evidence Relating to Object-Oriented Software Design: A Survey. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 482–484, Washington, DC, USA, 2007. IEEE Computer Society.

[BC19] S Balachandar and R Chinnaiyan. Reliable digital twin for connected footballer. In *International conference on computer networks and communication technologies*, pages 185–191. Springer, 2019.

[BCOR15] Jean-Michel Bruel, Benoit Combemale, Ileana Ober, and Héléne Raynal. MDE in Practice for Computational Science. In *International Conference on Computational Science (ICCS 2015)*, Reykjavík, Iceland, June 2015.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 2012.

[BD99] Bernd Bruegge and Allen A Dutoit. *Object Oriented Software Engineering, Conquering Complex and Changing Systems*. Prentice Hall, 1999.

[BD20] Paul Beynon-Davies. *Business information systems*. Red Globe Press, 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 Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *Advanced Information Systems Engineering*, pages 85–100, Cham, 2020. Springer International Publishing.

[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.

[BEK+18a] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*, pages 75–82. ACM, January 2018.

[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.

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[BF92] Manfred Broy and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TU Munich, 1992.

[BFK15] Olaf Berndt, Uwe Freiherr von Lukas, and Arjan Kuijper. Functional modelling and simulation of overall system ship-virtual methods for engineering and commissioning in shipbuilding. In *ECMS*, pages 347–353, 2015.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[BGG+19] Mathias Blumreiter, Joel Greenyer, Francisco Javier Chiyah Garcia, Verena Klös, Maike Schwammberger, Christoph Sommer, Andreas Vogelsang, and Andreas Wortmann. Towards Self-Explainable Cyber-Physical Systems. In Jeff Gray, Matti Rossi, Jonathan Sprinkle, and Juha-Pekka Tolvanen, editors, *Proceedings of MODELS 2019. Workshop Models@run.time*, pages 542–547. ACM, September 2019.

[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.

[BGT04] Sven Burmester, Holger Giese, and Matthias Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In *Model Driven Architecture*, pages 47–61. Springer, 2004.

[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.

[Bis96] Robert H Bishop. *Modern control systems analysis and design using MATLAB and SIMULINK*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[BJK+18] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvelletz, B. Osyk, and M. Weber. A Component Architecture for the Internet of Things. *Proceedings of the IEEE*, 106(9):1527–1542, September 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.

[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.

[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.

[BLG17] Stefan Biffl, Arndt Lüder, and Detlef Gerhard. *Multi-Disciplinary Engineering for Cyber-Physical Production Systems: Data Models and Software Solutions for Handling Complex Engineering Projects*. Springer, 2017.

[BMKW19] Florian Biesinger, Davis Meike, Benedikt Kraß, and Michael Weyrich. A digital twin for production planning based on cyber-physical systems: A Case Study for a Cyber-Physical System-Based Creation of a Digital Twin. *Procedia CIRP*, 79:355–360, 2019.

[BPRW20] 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*, MODELS '20, page 35–46, New York, NY, USA, 2020. Association for Computing Machinery.

[BR16] Stefan Boschert and Roland Rosen. Digital twin—the simulation aspect. In *Mechatronic futures*, pages 59–74. Springer, 2016.

[BR17] Julian Backhaus and Gunther Reinhart. Digital description of products, processes and resources for task-oriented programming of assembly systems. *Journal of Intelligent Manufacturing*, 28(8):1787–1800, 2017.

[Bro06] Manfred Broy. Challenges in Automotive Software Engineering. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, 2006.

[Bro10] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 2010.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.* Springer Verlag Heidelberg, 2001.

[BS17] Matthias Blum and Guenther Schuh. Towards a data-oriented optimization of manufacturing processes. In *Proceedings of the 19th International Conference on Enterprise Information Systems, Porto, Portugal*, pages 26–29, 2017.

[BSP+16] Patrick Bareiss, Daniel Schütz, Rafael Priego, Marga Marcos, and Birgit Vogel-Heuser. A model-based failure recovery approach for automated production systems combining sysml and industrial standards. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–7. IEEE, 2016.

[Bun17] Bundesministerium für Bildung und Forschung. Zukunftsprojekt Industrie 4.0. `https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html`, 2017. Accessed: 2017-04-20.

[BVM19] Michael Borth, Jacques Verriet, and Gerrit Muller. Digital twin strategies for sos: 4 challenges and 4 architecture setups for digital twins of sos. In *2019 14th Annual Conference System of Systems Engineering, SoSE 2019, 14th Annual Conference System of Systems Engineering, SoSE 2019, 19 May 2019 through 22 May 2019, 164-169*. Institute of Electrical and Electronics Engineers Inc., 2019.

[BW19] F. Biesinger and M. Weyrich. The facets of digital twins in production and the automotive industry. In *23rd Int. Conference on Mechatronics Technology (ICMT)*, pages 1–6. IEEE, 2019.

[BWHR08] Gary R Bertoline, Eric N. Wiebe, Nathaniel Hartmann, and William Ross. *Technical Graphics Communication.. 2003.* McGraw-Hill Education, 2008.

[BWK18] Christoph Brosinsky, Dirk Westermann, and Rainer Krebs. Recent and prospective developments in power system control centers: Adapting the digital twin technology for application in power system control centers. In *2018 IEEE International Energy Conference (ENERGYCON)*, pages 1–6. IEEE, 2018.

[Cam14] Fabien Campagne. *The MPS language workbench: volume I*, volume 1. Fabien Campagne, 2014.

[CAM+17] Michele Ciavotta, Marino Alge, Silvia Menato, Diego Rovere, and Paolo Pedrazzoli. A microservice-based middleware for the digital factory. *Procedia Manufacturing*, 11:931–938, 2017.

[CAT18] High Value Manufacturing Catapult. `https://hvm.catapult.org.uk/`, 2018. Accessed: 2018-06-05.

[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.

[CBW17] Benoit Combemale, Olivier Barais, and Andreas Wortmann. Language engineering with the GEMOC studio. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 189–191. IEEE, 2017.

[CBWM18] Dante Chavarría-Barrientos, Rafael Batres, Paul K. Wright, and Arturo Molina. A methodology to create a sensing, smart and sustainable manufacturing enterprise. *International Journal of Production Research*, 56(1-2):584–603, 2018.

[CCQB17] Isidro Calvo, Itziar Cabanes, Jerónimo Quesada, and Oscar Barambones. A multidisciplinary PBL approach for teaching industrial informatics and robotics in engineering. *IEEE Transactions on Education*, 61(1):21–28, 2017.

[CCSN19] Neeraj Kavan Chakshu, Jason Carson, Igor Sazonov, and Perumal Nithiarasu. A semi-active human digital twin model for detecting severity of carotid stenoses from head vibration—a coupled computational mechanics and computer vision method. *International journal for numerical methods in biomedical engineering*, 35(5):e3180, 2019.

[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.

[CH07] John Carnell and Rob Harrop. Velocity Template Engine. *Pro Apache Struts with Ajax*, pages 359–389, 2007.

[Cha15] Stephen J Chapman. *MATLAB programming for engineers*. Nelson Education, 2015.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model—toward a Unified View of Data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, March 1976.

[CHN11] Alarico Campetelli, Florian Hölzl, and Philipp Neubeck. User-friendly Model Checking Integration in Model-based Development. In *International Conference on Computer Applications in Industry and Engineering*, 2011.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[CJOC10] Gonçalo Cândido, François Jammes, José Barata de Oliveira, and Armando W Colombo. SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 598–603. IEEE, 2010.

[CKM+18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, et al. Concern-oriented language development (COLD): Fostering reuse in language engineering. *Computer Languages, Systems & Structures*, 54:139–155, 2018.

[CN02] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[CQT+16] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flávio Oquendo, Thaís Batista, and Axel Legay. Statistical Model Checking of Dynamic Software Architectures. In *European Conference on Software Architecture*, 2016.

[Crn01] Ivica Crnkovic. Component-based software engineering—new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[CS16] Federico Ciccozzi and Romina Spalazzese. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In *10th International Symposium on Intelligent Distributed Computing*, October 2016.

[CY15] Gunnar Carlsson and Jun Yu. A Prime Decomposition of Probabilistic Automata. *arXiv preprint arXiv:1503.01502*, 2015.

[DB18] Violeta Damjanovic-Behrendt. A digital twin-based privacy enhancement mechanism for the automotive industry. In *2018 International Conference on Intelligent Systems (IS)*, pages 272–279. IEEE, 2018.

[DCB+15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015.

[DDP+17] Gicu Calin Deac, Crina Narcisa Deac, Cicerone Laurentiu Popa, Mihalache Ghinea, and Costel Emil Cotet. Machine vision in manufacturing processes and the digital twin of manufacturing architectures. In *DAAAM Proceedings of the 28th International DAAAM Symposium*, volume 2017, pages 0733–0736, 2017.

[Dea79] Phyllis M Deane. *The first industrial revolution*. Cambridge University Press, 1979.

[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.

[DH04] James B Dabney and Thomas L Harman. *Mastering Simulink*. Pearson, 2004.

[DHA+18] Luiz Fernando C. S. Durão, Sebastian Haag, Reiner Anderl, Klaus Schützer, and Eduardo Zancul. Digital twin requirements in the context of industry 4.0. In *IFIP International Conference on Product Lifecycle Management*, pages 204–214. Springer, 2018.

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

[DJK+19b] Manuela Dalibor, Nico Jansen, Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Tagging Model Properties for Flexible Communication. In *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 39–46. IEEE, September 2019.

[DJK+19c] 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.

[DJR+19] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, Louis Wachtmeister, and Andreas Wortmann. Model-Driven Systems Engineering for Virtual Product Design. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MPM4CPS*, pages 430–435. IEEE, September 2019.

[DLPH08] Rainer Drath, Arndt Lüder, Jorn Peschke, and Lorenz Hundt. AutomationML-the glue for seamless automation engineering. In *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, pages 616–623. IEEE, 2008.

[DMMP17] Manuel Díaz-Madroñero, Josefa Mula, and David Peidro. A mathematical programming model for integrating production and procurement transport decisions. *Applied Mathematical Modelling*, 52:527–543, 2017.

[DR18] Ulrich Dahmen and Jürgen Roßmann. Simulation-based verification with experimentable digital twins in virtual testbeds. In *Tagungsband des 3. Kongresses Montage Handhabung Industrieroboter*, pages 139–147. Springer, 2018.

[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*, MODELS '20, page 79–89, New York, NY, USA, 2020. Association for Computing Machinery.

[DSLT04] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST-ADL—An Architecture Description Language. In *IFIP World Computer Congress, TC 2*, pages 181–195. Springer, 2004.

[EDF+18] Jonathan M Eyre, Tony J Dodd, Chris Freeman, Richard Lanyon-Hogg, Aiden J Lockwood, and Rab W Scott. Demonstration of an industrial

framework for an implementation of a process digital twin. In *ASME 2018 International Mechanical Engineering Congress and Exposition*. American Society of Mechanical Engineers Digital Collection, 2018.

[EGR12] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, New York, NY, USA, 2012. ACM.

[EGZ12] Martin Eigner, Torsten Gilz, and Radoslav Zafirov. Proposal for functional product description as part of a PLM solution in interdisciplinary product development. In *DS 70: Proceedings of DESIGN 2012, the 12th International Design Conference, Dubrovnik, Croatia*, 2012.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In *Computer-Aided Verification*, 1998.

[Elg14] Fredrik Elgh. Automated engineer-to-order systems a task oriented approach to enable traceability of design rationale. *International Journal Agile Systems and Management*, 7(3/4):324–347, 2014.

[EOH+09] Hilding Elmqvist, Martin Otter, Dan Henriksson, Bernhard Thiele, and Sven Erik Mattsson. Modelica for embedded systems. In *Proceedings of the 7th International Modelica Conference*, pages 354–363. Linköping University Electronic Press, 2009.

[EP11] Christof Efkemann and Jan Peleska. Model-based testing for the second generation of integrated modular avionics. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 55–62. IEEE, 2011.

[ER11] Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology*, 53(1):2–13, January 2011.

[ESM08] George Edwards, Chiyoung Seo, and Nenad Medvidovic. Model interpreter frameworks: A foundation for the analysis of domain-specific software architectures. *Journal of Universal Computer Science*, 14(8):1182–1210, 2008.

[ESV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk,

editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.

[ESV+15] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[FG12] Peter H Feiler and David P Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[FGLP10] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In *SLE*, pages 316–326. Springer, 2010.

[FHK+15] Stefan Feldmann, Sebastian JI Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan JJ Paredis, and Birgit Vogel-Heuser. Towards effective management of inconsistencies in model-based engineering of automated production systems. *IFAC-PapersOnLine*, 48(3):916–923, 2015.

[FLD04] Hans-Joachim Franke, Stefan Löffler, and Markus Deimel. Increasing the Efficiency of Design Catalogues By Using Modern Data Processing Techniques. In *DS 32: Proceedings of DESIGN 2004, the 8th International Design Conference, Dubrovnik, Croatia*, 2004.

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, 2014.

[For13] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing Ltd, 2013.

[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.

[FT17] Theodoros Foradis and Kleanthis Thramboulidis. From Mechatronic Components to Industrial Automation Things-An IoT model for cyber-physical manufacturing systems. *Journal of Software Engineering and Applications*, 10(08):734, 2017.

[FZG17] Yonggui Fu, Jianming Zhu, and Sheng Gao. CPS information security risk evaluation system based on Petri Net. In *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pages 541–548. IEEE, 2017.

[GA09] Karl-Heinrich Grote and Erik K. Antonsson. *Springer Handbook of Mechanical Engineering*. Springer, Berlin, 2009.

[Gar13] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[GCD09] Oscar González, Rubby Casallas, and Dirk Deridder. MCM-BPM: A domain-specific language for business processes analysis. In *International Conference on Business Information Systems*, pages 157–168. Springer, 2009.

[GDP+10] Jürgen Gausemeier, Rafal Dorociak, Sebastian Pook, Alexander Nyßen, and Axel Terfloth. Computer-aided cross-domain modeling of mechatronic systems. In *DS 60: Proceedings of DESIGN 2010, the 11th International Design Conference, Dubrovnik, Croatia*, volume 2, 05 2010.

[GEVD14] Cristian González García, Jordán Pascual Espada, Edward Rolando Núñez Valdez, and Vicente García Díaz. Midgar: Domain-specific language to generate smart objects for an internet of things platform. In *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 352–357. IEEE, 2014.

[GG67] Giovanni Battista Gerace and Giuseppe Gestri. Decomposition of synchronous sequential machines into synchronous and asynchronous submachines. *Information and Control*, 11(5):568 – 591, 1967.

[GHH+13] Christoph Gröger, Mark Hillmann, Friedemann Hahn, Bernhard Mitschang, and Engelbert Westkämper. The operational process dashboard for manufacturing. *Procedia CIRP*, 7(Supplement C):205–210, 2013.

[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.

[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.

[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.

[GLSC17] Omid Givehchi, Klaus Landsdorf, Pieter Simoens, and Armando Walter Colombo. Interoperability for industrial cyber-physical systems: An approach for legacy systems. *IEEE Transactions on Industrial Informatics*, 13(6):3370–3378, 2017.

[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 *25th Americas Conference on Information Systems (AMCIS 2020)*. Association for Information Systems (AIS), 2020. in press.

[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*, volume 254 of *LNI*, 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.

[Gro10] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03), 2010.

[GS14] Christoph Gröger and Christoph Stach. The mobile manufacturing dashboard. In *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*, pages 138–140. IEEE, 2014.

[GT18] Havva Gulay Gurbuz and Bedir Tekinerdogan. Model-based testing for software safety: a systematic mapping study. *Software Quality Journal*, 26(4):1327–1372, 2018.

[GV15] Xenofon V. Gogouvitis and George-Christopher Vosniakos. Construction of a virtual reality environment for robotic manufacturing cells. *International Journal of Computer Applications in Technology*, 51(3):173–184, 2015.

[GV17] Michael Grieves and John Vickers. Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In *Transdisciplinary perspectives on complex systems*, pages 85–113. Springer, 2017.

[GZ83] Mikell Groover and EWJR Zimmers. *CAD/CAM: computer-aided design and manufacturing*. Pearson Education, 1983.

[HAS+14] Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein, and Wolfgang Reif. Service-oriented robotics manufacturing by reasoning about the scene graph of a robotics cell. In *ISR/Robotik 2014; 41st International Symposium on Robotics*, pages 1–8. VDE, 2014.

[HBB+94] Wolfgang Hesse, Georg Barkow, Hubert von Braun, Hans-Bernd Kittlaus, and Gert Scheschonk. Terminologie der Softwaretechnik. Ein Begriffssystem fur die Analyse und Modellierung von Anwendungssystemen. Teil 2: Tätigkeits-und ergebnisbezogene Elemente. *Informatik Spektrum*, 17(2):96–105, 1994.

[HE17] Gergely Horváth and Ferenc Gábor Erdős. Gesture control of cyber physical systems. *Procedia Cirp*, 63:184–188, 2017.

[HF07a] Florian Hölzl and Martin Feilkas. 13 autofocus 3-a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pages 317–322. Springer, 2007.

[HF07b] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, 2007.

[HF11] Florian Hölzl and Martin Feilkas. AutoFocus 3-A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 317–322. Springer, 2011.

[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.

[HGS+16] Tiago Heineck, Enyo Gonçalves, Aêda Sousa, Marcos Oliveira, and Jaelson Castro. Model-Driven Development in Robotics Domain: A Systematic Literature Review. In *Software Components, Architectures and Reuse (SBCARS), 2016 X Brazilian Symposium on*, pages 151–160. IEEE, 2016.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on*

Model-Driven Engineering for Component-Based Software Systems (ModComp'15), volume 1463 of *CEUR Workshop Proceedings*, pages 18–23, 2015.

[HJK+10] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating OCL and textual modelling languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2010.

[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.

[HLMSN+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.

[HLR17] M. Hussein, S. Li, and A. Radermacher. Model-driven Development of Adaptive IoT Systems. In *Proceedings of MODELS 2017. Workshop ModComp*, volume 2019, pages 17–23, Austin, United States, 2017. CEUR.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa09] Tony Hoare. Null References: The Billion Dollar Mistake (Presentation). `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`, 2009. Accessed: 2020-05-07.

[HPE+16] Rogardt Heldal, Patrizio Pelliccione, Ulf Eliasson, Jonn Lantz, Jesper Derehag, and Jon Whittle. Descriptive vs Prescriptive Models in Industry. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 216–226, New York, NY, USA, 2016. ACM.

[HPO16] Mario Hermann, Tobias Pentek, and Boris Otto. Design Principles for Industrie 4.0 Scenarios. In *Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS)*, HICSS '16, pages 3928–3937, Washington, DC, USA, 2016. IEEE Computer Society.

[HQ82] Robert M. Haralick and David Queeney. Understanding engineering drawings. *Computer Graphics and Image Processing*, 20(3):244–258, 1982.

[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.

[HRW11] John Hutchinson, Mark Rouncefield, and John Whittle. Model-Driven Engineering Practices in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 633–642, May 2011.

[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.

[HS14] Robert Henssen and Miriam Schleipen. Interoperability between OPC UA and AutomationML. *Procedia Cirp*, 25:297–304, 2014.

[HSFH18] Jason Hatakeyama, Daniel Seal, Don Farr, and Scott Haase. Systems Engineering "V" in a Model-Based Engineering Environment: Is it still relevant? In *2018 AIAA SPACE and Astronautics Forum and Exposition*, 2018.

[HTKR+15] Dirk Holz, Angeliki Topalidou-Kyniazopoulou, Francesco Rovida, Mikkel Rath Pedersen, Volker Krüger, and Sven Behnke. A skill-based system for object perception and manipulation for automating kitting tasks. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–9. IEEE, 2015.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No. 98TB100203)*, pages 134–142. IEEE, 1998.

[IJ13] Jahanzaib Imtiaz and Jürgen Jasperneite. Scalability of OPC-UA down to the chip level enables "Internet of Things". In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 500–505. IEEE, 2013.

[Ikk15] Hubert Klein Ikkink. *Gradle Dependency Management*. Packt Publishing Ltd, 2015.

[INC07] INCOSE Technical Operations. Systems Engineering Vision 2020, version 2.03, 2007.

[ISO11] ISO/IEC/IEEE. Systems and software engineering – architecture description. Technical report, International Organization for Standardization, 1 2011.

[ISO15] ISO/IEC. Systems and software engineering —- system life cycle processes. Technical report, International Organization for Standardization, 5 2015.

[IVI18] The Industrial Value Chain Initiative. `https://iv-i.org/wp/en/about-us/whatsivi/`, 2018. Accessed: 2018-06-04.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[JB06] Frederic Jouault and Jean Bezivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (LNCS 4037)*, pages 171–185, 2006.

[JCB+15] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, 14(2):905–920, 2015.

[JCL11] Jeff C Jensen, Danica H Chang, and Edward A Lee. A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1666–1671. IEEE, 2011.

[JN19] Sigrid S. Johansen and Amir R. Nejad. On Digital Twin Condition Monitoring Approach for Drivetrains in Marine Applications. In *International Conference on Offshore Mechanics and Arctic Engineering*, volume Volume 10: Ocean Renewable Energy, 06 2019.

[JS07] Magne Jorgensen and Martin Shepperd. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, January 2007.

[JT09] Kyle Johns and Trevor Taylor. *Professional Microsoft Robotics Developer Studio*. John Wiley & Sons, 2009.

[Jun16] Reiner Jung. *Generator-Composition for Aspect-Oriented Domain-Specific Languages*. PhD thesis, Christian-Albrechts Universität Kiel, 2016.

[JYE19] Klementina Josifovska, Enes Yigitbas, and Gregor Engels. A Digital Twin-Based Multi-modal UI Adaptation Framework for Assistance Systems in Industry 4.0. In Masaaki Kurosu, editor, *Human-Computer Interaction. Design Practice in Contemporary Societies*, pages 398–409. Springer International Publishing, 2019.

[KB98] Wojtek Kozaczynski and Grady Booch. Component-based software engineering. *IEEE Software*, 15(5):34, 1998.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces: an Initial Appraisal. *CoopIS, DOA*, 2002, 2002.

[KBC04] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. In *Model Driven Architecture*, pages 62–76. Springer, 2004.

[KBM16] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology*, 71:77 – 91, 2016.

[KC09] Christian Koehler and Dave Clarke. Decomposing port automata. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1369–1373. ACM, 2009.

[KCO15] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line*, pages 71–80, 2015.

[KHHW13] Henning Kagermann, Johannes Helbig, Ariane Hellinger, and Wolfgang Wahlster. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group*. Forschungsunion, 2013.

[KK98] Rudolf Koller and Norbert Kastrup. *Prinziplösungen zur Konstruktion technischer Produkte*. Springer Berlin, 1998.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008.

[KLMX18] Tsubasa Kubota, Chao Liu, Khamdi Mubarok, and Xun Xu. A cyber-physical machine tool framework based on STEP-NC. In *Proceedings of the 48th International Conference on Computers and Industrial Engineering (CIE 48)*, volume 2018, 2018.

[KLSV03] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS 2003)*, 2003.

[KMC+18] Sathish A. P. Kumar, R. Madhumathi, Pethuru Raj Chelliah, Lei Tao, and Shangguang Wang. A novel digital twin-centric approach for driver intention prediction and traffic congestion avoidance. *Journal of Reliable Intelligent Environments*, 4(4):199–209, 2018.

[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*, MODELS '20, page 90–101, New York, NY, USA, 2020. Association for Computing Machinery.

[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.

[Kol14] Rudolf Koller. *Konstruktionslehre für den Maschinenbau - Grundlagen zur Neu- und Weiterentwicklung technischer Produkte mit Beispielen.* Springer, Berlin, 4 edition, 2014.

[KOTB19] Vladimir Kuts, Tauno Otto, Toivo Tähemaa, and Yevhen Bondarenko. Digital twin based synchronised control and simulation of the industrial robotic cell using virtual reality. *Journal of Machine Engineering*, 19, 2019.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[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.

[KR05] Vinay Kulkarni and Sreedhar Reddy. Model-Driven Development of Enterprise Applications. In *UML Modeling Languages and Applications*, pages 118–128. Springer, 2005.

[KR06] Vinay Kulkarni and Sreedhar Reddy. A model-driven architectural framework for integration-capable enterprise application product lines. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–12. Springer, 2006.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[Kra16] Edward M. Kraft. The air force digital thread/digital twin-life cycle integration and use of computational and experimental knowledge. In *54th AIAA Aerospace Sciences Meeting*, 2016.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, April 2020.

[KT16] Ateeq Khan and Klaus Turowski. A Survey of Current Challenges in Manufacturing Industry and Preparation for Industry 4.0. In *Proceedings of the First International Scientific Conference on Intelligent Information Technologies for Industry (IITI'16)*, pages 15–26, January 2016.

[KTY+16] Alexey Kashevnik, Nikolay Teslya, Eugeny Yablochnikov, Valery Arckhipov, and Kirill Kipriianov. Development of a prototype Cyber Physical Production System with help of Smart-M3. In *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 4890–4895. IEEE, 2016.

[Kue18] Wolfgang Kuehn. Digital twins for decision making in complex production and logistic enterprises. *International Journal of Design & Nature and Ecodynamics*, 13(3):260–271, 2018.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[KV96] Orna Kupferman and Moshe Y. Vardi. Verification of Fair Transition Systems. In *International Conference on Computer Aided Verification*, 1996.

[KX18] Sakdirat Kaewunruen and Ningfang Xu. Digital twin for sustainability evaluation of railway station buildings. *Frontiers in Built Environment*, 4:77, 2018.

[Kü05] Thomas Kühne. What is a Model? In *Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl*, pages 200–0, 2005.

[LBK+19] Nathan Lauzeral, Domenico Borzacchiello, Michael Kugler, Daniel George, Yves Rémond, Alexandre Hostettler, and Francisco Chinesta. A model order reduction approach to create patient-specific mechanical models of human liver in computational medicine applications. *Computer methods and programs in biomedicine*, 170:95–106, 2019.

[LC13] Miguel A. Laguna and Yania Crespo. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *Sci. Comput. Program.*, 78(8):1010–1034, August 2013.

[LDA13] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[LDCM15] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A behavioral coordination operator language (bcool). In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 186–195. IEEE, 2015.

[LDdP17] Yongxin Liao, Fernando Deschamps, Eduardo de Freitas Rocha Loures, and Luiz F. Pierin Ramos. Past, present and future of Industry 4.0 - a systematic literature review and research agenda proposal. *International Journal of Production Research*, 55(12):3609–3629, 2017.

[Lee08] Edward A Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[Lev09] Anatoly Levenchuk. SysML is the Point of Departure for MBSE, Not the Destination. *INSIGHT*, 12(4):54–55, 2009.

[LG18] Milica Lekić and Gordana Gardašević. IoT sensor integration to Node-RED platform. In *17th International Symposium INFOTEH-JAHORINA*, pages 1–5, March 2018.

[LL81] Ernest Lepore and Barry Loewer. Translational semantics. *Synthese*, pages 121–133, 1981.

[LM06] Stefan-Helmut Leitner and Wolfgang Mahnke. OPC UA–service-oriented architecture for industrial applications. *ABB Corporate Research Center*, 48:61–66, 2006.

[LMT+14] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *International Conference on Model Driven Engineering Languages and Systems*, pages 166–182. Springer, 2014.

[LMT+18] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[LS17] Arndt Lüder and Nicole Schmidt. Automationml in a nutshell. In *Handbuch Industrie 4.0 Bd. 2*, pages 213–258. Springer, 2017.

[LSR+19] David Lechevalier, Seung-Jun Shin, Sudarsan Rachuri, Sebti Foufou, Yung-Tsun Tina Lee, and Abdelaziz Bouras. Simulating a virtual machining model in an agent-based model for advanced analytics. *Journal of Intelligent Manufacturing*, 30(4):1937–1955, 2019.

[Lu17] Yang Lu. Industry 4.0: A survey on technologies, applications and open research issues. *Journal of Industrial Information Integration*, 6:1–10, 2017.

[LWG+19] Li-Lan Liu, Xiang Wan, Zenggui Gao, Xiaolong Li, and Bowen Feng. Research on modelling and optimization of hot rolling scheduling. *Journal of Ambient Intelligence and Humanized Computing*, 10(3):1201–1216, 2019.

[LXH+20] Qiuchen Lu, Xiang Xie, James Heaton, Ajith Kumar Parlikad, and Jennifer Schooling. From bim towards digital twin: Strategy and future development for smart asset management. In Theodor Borangiu, Damien Trentesaux, Paulo Leitão, Adriana Giret Boggino, and Vicente Botti, editors, *Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*, pages 392–404, Cham, 2020. Springer International Publishing.

[MAGD+16] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures*, 46:206–235, 2016.

[Man18] Korea - Manufacturing Technology - Smart Factory. `https://www.export.gov/article?id=Korea-Manufacturing-Technology-Smart-Factory`, 2018. Accessed: 2018-06-04.

[Mar18] Azat Mardan. Template Engines: Pug and Handlebars. In *Practical Node.js*, pages 113–163. Springer, 2018.

[MBS16] Martin Melik Merkumians, Matthias Baierling, and Georg Schitter. A Service-Oriented Domain Specific Language Programming Approach for Batch Processes. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–9. IEEE, 2016.

[MDT07] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1):12–31, 2007.

[MDWA17] Sara Mahdavi-Hezavehi, Vinicius H.S. Durelli, Danny Weyns, and Paris Avgeriou. A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. *Information and Software Technology*, 2017.

[MEE+11] Johannes Mathias, Tobias Eifler, Roland Engelhardt, Hermann Kloberdanz, Herbert Birkhofer, and Andrea Bohn. Selection of Physical Effects Based on Disturbances and Robustness Rations in The Early Phases of Robust Design. In *International Conference on Engineering Design*, pages 11–15, 2011.

[Mer13] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9), 2013.

[Mer18] Mercator Institute for China Studies. Made in China 2025. `https://www.merics.org/sites/default/files/2017-09/MPOC_No.2_MadeinChina2025.pdf`, 2018. Accessed: 2018-06-06.

[MFBC12] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling Modeling Modeling. *Software & Systems Modeling*, 11(3):347–359, 2012.

[MHF17] Brice Morin, Nicholas Harrand, and Franck Fleurey. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software*, 34(1):30–36, January 2017.

[MHWM17] Mathias Mormul, Pascal Hirmer, Matthias Wieland, and Bernhard Mitschang. Situation model as interface between situation recognition and situation-aware applications. *Computer Science-Research and Development*, 32(3-4):331–342, 2017.

[Mic14] Patrice Micouin. *Model Based Systems Engineering: Fundamentals and Methods*. John Wiley & Sons, 2014.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge university press, 1999.

[MJ13] Abid Mehmood and Dayang N.A. Jawawi. Aspect-oriented model-driven code generation: A systematic mapping study. *Information and Software Technology*, 55(2):395 – 411, 2013. Special Section: Component-Based Software Engineering (CBSE), 2011.

[MKBZ16] Ellina Marseu, Dennis Kolberg, Max Birtel, and Detlef Zühlke. Interdisciplinary engineering methodology for changeable cyber-physical production systems. *IFAC-PapersOnLine*, 49(31):85–90, 2016.

[MKG+15] Georg Moeser, Christoph Kramer, Martin Grundel, Michael Neubert, Stephan Kümpel, Axel Scheithauer, Sven Kleiner, and Albert Albers. Fortschrittsbericht zur modellbasierten Unterstützung der Konstrukteurstätigkeit durch FAS4M. In *Tag des Systems Engineering*, pages 69–78. Carl Hanser Verlag GmbH & Co. KG, 2015.

[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.

[Mok98] Joel Mokyr. The second industrial revolution, 1870-1914. *Storia dell'economia Mondiale*, pages 219–245, 1998.

[Mol17] Michael F. Molnar. The U.S. Advanced Manufacturing Initiative. `https://www.nist.gov/system/files/documents/2017/04/28/Molnar_091211.pdf`, 2017.

[Mos90] Peter D. Mosses. Denotational semantics. In *Formal Models and Semantics*, pages 575–631. Elsevier, 1990.

[MPPU19] Claudio Mandolla, Antonio Messeni Petruzzelli, Gianluca Percoco, and Andrea Urbinati. Building a digital twin for additive manufacturing through the exploitation of blockchain: A case analysis of the aircraft industry. *Computers in Industry*, 109:134 – 152, 2019.

[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.

[MSO18] M Mayani Gholami, M Svendsen, and S.I. Oedegaard. Drilling Digital Twin Success Stories the Last 10 Years. In *SPE Norway One Day Seminar*. Society of Petroleum Engineers, 2018.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[MW00] Radmehr P. Monfared and Richard H. Weston. A method to develop semi-generic information models of change-capable cell control systems. *Computers in Industry*, 41(3):279–294, 2000.

[MW17] Stefan Mätzler and Martin Wollschlaeger. Interchange Format for the Generation of Functional Elements for Industrie 4.0 Components. In *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 5453–5459. IEEE, 2017.

[NAY17] Phu H Nguyen, Shaukat Ali, and Tao Yue. Model-based security engineering for cyber-physical systems: A systematic mapping study. *Information and Software Technology*, 83:116–135, 2017.

[NHW14] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A survey on domain-specific languages in robotics. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 195–206. Springer, 2014.

[NHWW16] Arne Nordmann, Nico Hochgeschwender, Dennis Wigand, and Sebastian Wrede. A Survey on Domain-Specific Modeling and Languages in Robotics. *Journal of Software Engineering for Robotics*, 7:75–99, 2016.

[NMR+18] Derek Nadhan, Maryam Gholami Mayani, Rolv Rommetveit, et al. Drilling with digital twins. In *IADC/SPE Asia pacific drilling technology conference and exhibition*. Society of Petroleum Engineers, 2018.

[Noz78] A. Nozaki. Practical Decomposition of Automata. *Information and Control*, 36(3), 1978.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, volume 1118 of *CEUR Workshop Proceedings*, pages 15–24, 2013.

[Ogr00] Ingmar Ogren. On principles for model-based systems engineering. *Systems engineering*, 3(1):38–49, 2000.

[Oqu04] Flavio Oquendo. π-adl: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[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).

[PB08] Rialette Pretorius and David Budgen. A Mapping Study on Empirical Evidence Related to the Models and Forms Used in the UML. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 342–344, New York, NY, USA, 2008. ACM.

[PBFG07] Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Engineering Design - A Systematic Approach*. Springer, London, 3 edition, 2007.

[PBFS17] Guillaume Prévost, Jan Olaf Blech, Keith Foster, and Heinrich-Wilhelm Schmidt. An Architecture for Visualization of Industrial Automation Data. In *ENASE*, pages 38–46, 2017.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[PC16] Jaehui Park and Su-young Chi. An implementation of a high throughput data ingestion system for machine logs in manufacturing industry. In *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 117–120. IEEE, 2016.

[Pet13] Marian Petre. UML in Practice. In *Proceedings of ICSE '13*, pages 722–731. IEEE Press, 2013.

[PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic Mapping Studies in Software Engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pages 68–77, Swinton, UK, UK, 2008. British Computer Society.

[PJIZ18] Davy Preuveneers, Wouter Joosen, and Elisabeth Ilie-Zudor. Robust digital twin compositions for Industry 4.0 smart manufacturing systems. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 69–78. IEEE, 2018.

[Plo81] Gordon D Plotkin. A structural approach to operational semantics. In *STACS'87: Proc. Fourth Annual Symposium on*. Computer Science Department, Aarhus University Aarhus, Denmark, 1981.

[PM18] Klaus Pohl and Andreas Metzger. *Software Product Lines*, pages 185–201. Springer International Publishing, Cham, 2018.

[PNBL+19] Luca Pivano, Dong Trong Nguyen, Kristine Bruun Ludvigsen, et al. Digital twin for drilling operations–towards cloud-based operational planning. In *Offshore Technology Conference*. Offshore Technology Conference, 2019.

[PNL+19] Kyu Tae Park, Young Wook Nam, Hyeon Seung Lee, Sung Ju Im, Sang Do Noh, Ji Yeon Son, and Hyun Kim. Design and implementation of a digital twin application for a connected micro smart factory. *International Journal of Computer Integrated Manufacturing*, 32(6):596–614, 2019.

[PP15] Boris Plotkin and Tatjana Plotkin. Decompositions and complexity of linear automata. *arXiv preprint arXiv:1506.06017*, 2015.

[PRB+09] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. *Human-Computer Interaction. Novel Interaction Methods and Techniques*, pages 889–898, 2009.

[PRCO16] Ricardo Silva Peres, Andre Dionisio Rocha, Andre Coelho, and Jose Barata Oliveira. A highly flexible, distributed data analysis framework for industry 4.0 manufacturing systems. In *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*, pages 373–381. Springer, 2016.

[Pto14] Claudius Ptolemaeus. System Design, Modeling, and Simulation using Ptolemy II, 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.

[QCT+16] Jean Quilbeuf, Everton Cavalcante, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, and Axel Legay. A logic for the statistical model checking of dynamic software architectures. In *International Symposium on Leveraging Applications of Formal Methods*, pages 806–820. Springer, 2016.

[Rai05] Chris Raistrick. Applying MDA and UML in the Development of a Healthcare System. In *UML Modeling Languages and Applications*, pages 203–218. Springer, 2005.

[RBK18] Anro Redelinghuys, Anton Basson, and Karel Kruger. A six-layer digital twin architecture for a manufacturing cell. In *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*, pages 412–423. Springer, 2018.

[RFB11] Ana Luísa Ramos, José Vasconcelos Ferreira, and Jaume Barceló. Model-based systems engineering: An emerging approach for modern systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(1):101–111, 2011.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RL10] Frank H. Riddick and Yung-Tsun Lee. Core Manufacturing Simulation Data (CMSD): A standard representation for manufacturing simulation-related information. *National Institute of Standards and Technology*, 2010.

[Rot94] Karlheinz Roth. *Konstruieren mit Konstruktionskatalogen - Band I: Konstruktionslehre.* Springer, Berlin, 2 edition, 1994.

[Rot96] Karlheinz Roth. *Konstruieren mit Konstruktionskatalogen - Band III: Verbindungen und Verschlüsse - Lösungsfindung.* Springer, Berlin, 2 edition, 1996.

[Rot11] Karlheinz Roth. Selection of Physical Effects Based on Disturbances and Robustness Rations in The Early Phases of Robust Design. In *International Conference on Engineering Design*, 2011.

[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.

[RRIG09] Jan Recker, Michael Rosemann, Marta Indulska, and Peter Green. Business process modeling-a comparative analysis. *Journal of the association for information systems*, 10(4):1, 2009.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton.* Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW16] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*, pages 1–12. fortiss, An-Institut TU München, Technical Report, March 2016.

[RSDT19] Michael Riesener, Günther Schuh, Christian Dölle, and Christian Tönnes. The Digital Shadow as Enabler for Data Analytics in Product Life Cycle Management. *Procedia CIRP*, 80:729–734, 2019.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods.* Springer International, 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring.* Springer International, 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.

[SA95] Robert Shishko and Robert Aster. Nasa systems engineering handbook. *NASA Special Publication*, 6105, 1995.

[SAOI18] Gurtej Saini, Pradeepkumar Ashok, Eric van Oort, and Matthew R Isbell. Accelerating Well Construction Using a Digital Twin Demonstrated on Unconventional Well Data in North America. In *Unconventional Resources Technology Conference, Houston, Texas, 23-25 July 2018*, pages 3264–3276. Society of Exploration Geophysicists, American Association of Petroleum Geologists, Society of Petroleum Engineers, 2018.

[SB17] Stephen Samuel and Stefan Bocutiu. *Programming kotlin*. Packt Publishing Ltd, 2017.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[SBRB16] Günther Schuh, Matthias Blum, Jan Reschke, and Martin Birkmeier. Der Digitale Schatten in der Auftragsabwicklung. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 111(1-2):48–51, 2016.

[SCD+12] Mike Shafto, Mike Conroy, Rich Doyle, Ed Glaessgen, Chris Kemp, Jacqueline LeMoigne, and Lui Wang. Modeling, simulation, information technology & processing roadmap. *National Aeronautics and Space Administration*, 2012.

[SDLJ19] Chang-Su Shim, Ngoc-Son Dang, Sokanya Lon, and Chi-Ho Jeon. Development of a bridge maintenance system for prestressed concrete bridges using 3D digital twin model. *Structure and Infrastructure Engineering*, 15(10):1319–1332, 2019.

[SDT18] Günther Schuh, Christian Dölle, and Christian Tönnes. Methodology for the derivation of a digital shadow for engineering management. In *2018 IEEE Technology and Engineering Management Conference (TEMSCON)*, pages 1–6. IEEE, 2018.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, Sept 2003.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, Sept 2003.

[Sel06] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, April 2006.

[SF18] Bruno Scaglioni and Gianni Ferretti. Towards digital twins through object-oriented modelling: a machine tool case study. *IFAC-PapersOnLine*, 51(2):613–618, 2018.

[SFA17] Chantal Steimer, Jan Fischer, and Jan C. Aurich. Model-based design process for the early phases of manufacturing system planning using SysML. *Procedia CIRP*, 60:163–168, 2017.

[SGBvB12] Yu Sun, Jeff Gray, Karlheinz Bulheller, and Nicolaus von Baillou. A Model-Driven Approach to Support Engineering Changes in Industrial Robotics Software. In *International Conference on Model Driven Engineering Languages and Systems*, pages 368–382. Springer, 2012.

[SHB+17] Partha Sharma, Hamed Hamedifar, Aaron Brown, Richard Green, et al. The dawn of the new age of the industrial internet and how it can radically transform the offshore oil and gas industry. In *Offshore Technology Conference*. Offshore Technology Conference, 2017.

[SHH+05] Dag I. K. Sjoberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, September 2005.

[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.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE, 2006.

[SLF17] Claudio Santo Longo and Cesare Fantuzzi. Simulation and Optimisation of Production Lines in the Framework of the IMPROVE Project. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2017.

[SP07] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *OWLED*, volume 258, 2007.

[SSP+16] Greyce Schroeder, Charles Steinmetz, Carlos Eduardo Pereira, Ivan Muller, Natanael Garcia, Danubia Espindola, and Ricardo Rodrigues. Visualising the digital twin using web services and augmented reality. In *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, pages 522–527. IEEE, 2016.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, 1973.

[Sta06] Miroslaw Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In *Model Driven Engineering Languages and Systems*, pages 57–72. Springer, 2006.

[Sto11] Tijs van der Storm. *The Rascal Language Workbench*. CWI. Software Engineering [SEN], 2011.

[SUN+17] Bahram Lotfi Sadigh, Hakki Ozgur Unver, Shahrzad Nikghadam, Erdogan Dogdu, A. Murat Ozbayoglu, and S. Engin Kilic. An ontology-based multi-agent virtual enterprise system (OMAVE): part 1: domain modelling and rule management. *International Journal of Computer Integrated Manufacturing*, 30(2-3):320–343, 2017.

[SvB02] Thorsten Sturm, Jesco von Voss, and Marko Boger. Generating code from UML with velocity templates. In *International Conference on the Unified Modeling Language*, pages 150–161. Springer, 2002.

[SW99] Frank Strobl and Alexander Wisspeintner. Specification of an Elevator Control System. Technical Report TUM-I9906, SFB Bericht Nr. 342/04/99 A, U Munich, 1999.

[SWL+16] Günther Schuh, Pia Walendzik, Melanie Luckert, Martin Birkmeier, Anja Weber, and Matthias Blum. Keine Industrie 4.0 ohne den Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 111(11):745–748, 2016.

[SZ16] Michael Szvetits and Uwe Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling*, 15(1):31–69, 2016.

[TC16] Kleanthis Thramboulidis and Foivos Christoulakis. UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry*, 82:259–272, 2016.

[TCQ+18] Fei Tao, Jiangfeng Cheng, Qinglin Qi, Meng Zhang, He Zhang, and Fangyuan Sui. Digital twin-driven product design, manufacturing and service with big data. *The International Journal of Advanced Manufacturing Technology*, 94(9-12):3563–3576, 2018.

[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.

[TM02] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*, volume 2. Springer, 2002.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons, Inc., 1 edition, 2009.

[TP97] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 280–291. IEEE, 1997.

[TQWN19] Fei Tao, Qinglin Qi, Lihui Wang, and Andrew Y. C. Nee. Digital Twins and Cyber–Physical Systems toward Smart Manufacturing and Industry 4.0: Correlation and Comparison. *Engineering*, 5(4):653 – 661, 2019.

[TTG+16] Amy J. C. Trappey, Charles V. Trappey, Usharani Hareesh Govindarajan, John J. Sun, and Allen C. Chuang. A Review of Technology Standards and Patent Portfolios for Enabling Cyber-Physical Systems in Advanced Manufacturing. *IEEE Access*, 4:7356–7382, 2016.

[TTR+13] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry. *Journal of Systems and Software*, 86(8):2110–2126, 2013.

[TX00] Jeffrey J.P. Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 2000.

[TZLN18] Fei Tao, He Zhang, Ang Liu, and Andrew Y. C. Nee. Digital twin in industry: State-of-the-art. *IEEE Transactions on Industrial Informatics*, 15(4):2405–2415, 2018.

[UE03] Karl Ulrich and Steven Eppinger. *Product Design and Development*. McGraw-Hill, New York, 3 edition, 2003.

[UE09] Darian W. Unger and Steven D. Eppinger. Comparing product development processes and managing risk. *International Journal of Product Development*, 8(4):382–402, 2009.

[UML20] UML Specification version 1.1 (OMG document ad/97-08-11). `https://www.omg.org/cgi-bin/doc?ad/97-08-11`, 2020. Accessed: 2020-05-01.

[Ung96] Stephen H. Unger. *The Essence of Logic Circuits.* Wiley-IEEE Press, 1996.

[US13] Gürkan Uygur and Sebastian M. Sattler. Parallel decomposition for safety-critical systems. In *2013 3rd International Electric Drives Production Conference (EDPC)*, pages 1–8, 2013.

[Val14] Antonio Vallecillo. On the Industrial Adoption of Model Driven Engineering. Is your company ready for MDE? *Journal of Information Systems and Software Engineering for Big Companies (IJISEBC)*, 1(1):52 – 68, 2014.

[VBD+13] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[VCGP19] Igor M. Verner, Dan Cuperman, Sergei Gamer, and Alex Polishuk. Training robot manipulation skills through practice with digital twin of baxter. *International Journal of Online and Biomedical Engineering (iJOE)*, 15(09):58–70, 2019.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[VHH16] Birgit Vogel-Heuser and Dieter Hess. Guest Editorial Industry 4.0 - Prerequisites and Visions. *IEEE Transactions on Automation Science and Engineering*, 13(2):411–413, April 2016.

[VLG15] George-Christopher Vosniakos, Emmanuel Levedianos, and Xenofon V Gogouvitis. Streamlining virtual manufacturing cell modelling by behaviour modules. *International Journal of Manufacturing Research*, 10(1):17–44, 2015.

[VOSC14] Edoardo Vacchi, Diego Mathias Olivares, Albert Shaqiri, and Walter Cazzola. Neverlang 2: a framework for modular language implementation. In *Proceedings of the companion publication of the 13th international conference on Modularity*, pages 29–32, 2014.

[VSB+13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley Software Patterns Series. Wiley, 2013.

[VV10] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–304. ACM, 2010.

[WB17] Timothy D. West and Mark Blackburn. Is Digital Thread/Digital Twin Affordable? A Systemic Assessment of the Cost of DoD's Latest Manhattan Project. *Procedia computer science*, 114:47–56, 2017.

[WBCW20] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study. *Software and Systems Modeling*, 19(1):67–94, January 2020.

[WHG+09] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE Software*, 26(4), 2009.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, 31(3):79–85, 2014.

[Wil01] David S. Wile. Supporting the DSL Spectrum. *Computing and Information Technology*, 4:263–287, 2001.

[WKV14] Guido H. Wachsmuth, Gabriël DP Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014.

[Wor19] Andreas Wortmann. Towards Component-Based Development of Textual Domain-Specific Languages. In Luigi Lavazza, Herwig Mannaert, and Krishna Kavi, editors, *International Conference on Software Engineering Advances (ICSEA 2019)*, pages 68–73. IARIA XPS Press, November 2019.

[WS09] Stefan Wölkl and Kristina Shea. A Computational Product Model for Conceptual Design Using SysML. In *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2009.

[WWB17] Denis Wohlfeld, Viktor Weiss, and Bernd Becker. Digital Shadow–From production to product. In *17. Internationales Stuttgarter Symposium*, pages 783–794. Springer, 2017.

[WWM+07] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin Van Den Berg, Kim Fleer, David Nelson, et al. Experiences in Deploying Model-Driven Engineering. In *SDL 2007: Design for Dependable Systems*, pages 35–53. Springer, 2007.

[WYG+19] Jinjiang Wang, Lunkuan Ye, Robert X. Gao, Chen Li, and Laibin Zhang. Digital twin for rotating machinery fault diagnosis in smart manufacturing. *International Journal of Production Research*, 57(12):3920–3934, 2019.

[WZD+19] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–31, 2019.

[YTM08] Hong Yul Yang, Ewan Tempero, and Hayden Melton. An empirical study into use of dependency injection in Java. In *19th Australian Conference on Software Engineering (aswec 2008)*, pages 239–247. IEEE, 2008.

[ZAMM12] Christian Zingel, Albert Albers, Sven Matthiesen, and Michael Maletz. Experiences and Advancements from One Year of Explorative Application of an Integrated Model-Based Development Technique Using C&C^2-A in SysML. *International Journal of Computer Science*, 34-39, 2012.

[ZLX18] Cunbo Zhuang, Jianhua Liu, and Hui Xiong. Digital twin-based smart production management and control framework for the complex product assembly shop-floor. *The International Journal of Advanced Manufacturing Technology*, 96(1-4):1149–1163, 2018.

[ZLZ17] Qiuchong Zhang, Yuqi Liu, and Zhibing Zhang. A new method for automatic optimization of drawbead geometry in the sheet metal forming process based on an iterative learning control model. *The International Journal of Advanced Manufacturing Technology*, 88(5-8):1845–1861, 2017.

[ZM17] Andrew J. Zakrajsek and Shankar Mall. The development and use of a digital twin model for tire touchdown health monitoring. In *58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2017.

[ZMHK18] Victor Zhidchenko, Iuliia Malysheva, Heikki Handroos, and Alexander N. Kovartsev. Faster than real-time simulation of mobile crane dynamics using digital twin concept. In *Journal of Physics: Conference Series*, volume 1096, pages 10–1088, 2018.

[ZML10] Pengcheng Zhang, Henry Muccini, and Bixin Li. A Classification and Comparison of Model Checking Software Architecture Techniques. *Journal of Systems and Software*, 2010.

[ZR13] Linda L. Zhang and Brian Rodrigues. A Petri net model of process platform-based production configuration. *Journal of Manufacturing Technology Management*, 2013.

[ZZL+19] Guanghui Zhou, Chao Zhang, Zhi Li, Kai Ding, and Chuang Wang. Knowledge-driven digital twin manufacturing cell towards intelligent manufacturing. *International Journal of Production Research*, pages 1–18, 2019.

Appendix A

Author Contribution to Publications

This appendix describes the author's contribution to the publications presented in this thesis.

Table A.1: Contribution by the author to the publications presented in this thesis.

#	Publication Title	Thesis Chapter	Contribution to Publication Phase (%)			
			Research Design	Research Execution	Writing	Revision
1	Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study	2	80	60	70	65
2	Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features	3	75	80	60	50
3	Systematic Composition of Independent Language Features	3	80	50	70	50
4	Modeling Language Variability with Reusable Language Components	3	75	50	70	50
5	Towards Component-Based Development of Textual Domain-Specific Languages	3	100	100	100	100
6	A Compositional Framework for Systematic Modeling Language Reuse	3	100	100	80	70
7	Modeling Mechanical Functional Architectures	4	45	30	35	45
8	Automated semantics-preserving parallel decomposition of finite component and connector architectures	4	40	35	40	35
9	Semantic Differencing for Message-Driven Component & Connector Architectures	4	45	30	40	40
10	Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution	4	50	30	50	35
11	Effizientere Produktion mit Digitalen Schatten	5	65	50	35	40
12	Model-Driven Development of a Digital Twin for Injection Molding	5	70	30	55	50
13	Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems	5	30	15	35	20

Appendix B

Reprints of Selected Publications

This appendix comprises the selected publications summarized in this thesis, listed in their order of appearance.

Modeling Languages for Cyber-Physical Systems

This section comprises the publications summarized in Chapter 2.

Paper 1	A. Wortmann, O. Barais, B. Combemale, M. Wimmer. Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study. In: J. Gray and V. Kulkarni, editors, Software and Systems Modeling, 19(1), pages 67-94, Springer, 2020. Reference: [WBCW20]

[WBCW20] A. Wortmann, O. Barais, B. Combemale, M. Wimmer:
Modeling languages in Industry 4.0: an extended systematic mapping study.
In: Software and Systems Modeling, 2020.
www.se-rwth.de/publications/

SPECIAL SECTION PAPER

Modeling languages in Industry 4.0: an extended systematic mapping study

Andreas Wortmann[1] · Olivier Barais[2] · Benoit Combemale[3] · Manuel Wimmer[4]

Received: 2 July 2018 / Revised: 24 August 2019 / Accepted: 8 September 2019

Abstract
Industry 4.0 integrates cyber-physical systems with the Internet of Things to optimize the complete value-added chain. Successfully applying Industry 4.0 requires the cooperation of various stakeholders from different domains. Domain-specific modeling languages promise to facilitate their involvement through leveraging (domain-specific) models to primary development artifacts. We aim to assess the use of modeling in Industry 4.0 through the lens of modeling languages in a broad sense. Based on an extensive literature review, we updated our systematic mapping study on modeling languages and modeling techniques used in Industry 4.0 (Wortmann et al., Conference on model-driven engineering languages and systems (MODELS'17), IEEE, pp 281–291, 2017) to include publications until February 2018. Overall, the updated study considers 3344 candidate publications that were systematically investigated until 408 relevant publications were identified. Based on these, we developed an updated map of the research landscape on modeling languages and techniques for Industry 4.0. Research on modeling languages in Industry 4.0 focuses on contributing methods to solve the challenges of digital representation and integration. To this end, languages from systems engineering and knowledge representation are applied most often but rarely combined. There also is a gap between the communities researching and applying modeling languages for Industry 4.0 that originates from different perspectives on modeling and related standards. From the vantage point of modeling, Industry 4.0 is the combination of systems engineering, with cyber-physical systems, and knowledge engineering. Research currently is splintered along topics and communities and accelerating progress demands for multi-disciplinary, integrated research efforts.

Keywords Industry 4.0 · Modeling languages · Smart manufacturing

1 Introduction

Industrial revolutions always introduced step changes to manufacturing. The first industrial revolution (eighteenth–nineteenth century) advanced production from manual to machine-driven manufacturing, introduced factories, and enabled leveraging steam power for production [36]. The second industrial revolution (1870–1914) introduced electric power to enable the mass production of goods using the concept of interchangeable parts [108]. The third industrial revolution (ca. 1980–2010) describes the transition from analog to digital (mostly isolated) production systems. Industry 4.0 is the fourth industrial evolution focusing on integrating digitized cyber-physical production systems with processes and stakeholders to optimize the complete value-added chain. Originally, it has been announced as part of the high-tech strategy of the German Federal Ministry for Education and Research [27]. However, the essence of Industry 4.0 has become an international phenomenon as the Japanese

Communicated by Vinay Kulkarni and Jeff Gray.

Andreas Wortmann
wortmann@se-rwth.de

Olivier Barais
Olivier.Barais@irisa.fr

Benoit Combemale
benoit.combemale@irit.fr

Manuel Wimmer
manuel.wimmer@jku.at

[1] Software Engineering, RWTH Aachen University, Aachen, Germany

[2] University of Rennes 1, Rennes, France

[3] University of Toulouse, Toulouse, France

[4] Institute of Business Informatics - Software Engineering and CDL-MINT, Johannes Kepler University Linz, Linz, Austria

Published online: 20 September 2019

Industrial Value Chain Initiative [146], the Advanced Manufacturing Initiative in the USA [145], the Chinese Made in China 2025 strategy [96], Manufacturing 3.0 in South Korea [84], and the national Catapult research center on High Value Manufacturing [65] in the UK indicate.

This "fourth industrial revolution" raises new challenges for future manufacturing which are driven by four disruptions: (1) data volumes, computational power, and connectivity; (2) the emergence of analytics and business-intelligence capabilities; (3) new forms of human–machine interaction; (4) and improvements in transferring digital instructions to the physical world, such as advanced robotics and 3D printing. The interplay of these four disruptions led to recognizing four particular Industry 4.0 design principles [63]:

- Interoperability: connect production systems, devices, sensors, and people.
- Information transparency: query data and connect digital planning with the runtime data collected from sensors.
- Technical assistance: provide the right abstraction to understand the complexity of Industry 4.0 systems and processes.
- Decentralized decision making: enable autonomous systems.

All of these aim to enable more efficient production down to the individualized the mass production of "lot-size 1" [42].

Model-based software development is one of the key enablers for successfully engineering, integrating, and maintaining complex systems of systems, which is indicated by the increasing number of related publications in key conferences and journals investigating these challenges, e.g., see [29,49,62,97,129,143]. For successfully engineering Industry 4.0 systems of systems, fostering research in modeling is crucial to enable realizing the aforementioned design principles.

As a research area matures, there often is a significant increase in the number of related reports and results. Thus, it becomes important to summarize and to overview those results. There are different methods for structuring a scientific landscape, such as systematic literature reviews [24,76] or systematic mapping studies [115]. Systematic literature reviews are a "form of secondary study that use a well-defined methodology to identify, analyze, and interpret all available evidence related to a specific research question in a way that is unbiased and (to a degree) repeatable" [76]. They aim to summarize the existing evidence concerning the object of research (e.g., modeling languages) to identify gaps in the current research. To this end, systematic literature reviews follow an a priori defined review protocol of research questions and a documented (hence, reproducible) search strategy. Based on the obtained corpus of primary studies, the research questions are answered. Systematic literature reviews are common to software engineering [13,59], model-based engineering [25,37], software product lines [32,47], or domain-specific languages [53,82], etc., while mapping studies are less common. A systematic mapping study (SMS) structures a body of research through its reports by categorizing these. These often culminate in a visual summary, the map, of its results. Such a map supports understanding what has been addressed by the community for a particular domain and its corpus can serve as the basis to answer in-depth research questions of a subsequent systematic literature review.

We investigate modeling in Industry 4.0 through the lens of modeling languages and applied to the field's diverse challenges. Conducting a systematic mapping study on modeling languages for Industry 4.0, hence, enables providing guidance and feedback for the modeling community about challenges for and reception of their contributions in the domain of Industry 4.0. Moreover, it provides an overview for the automation systems community about the contributions to modeling languages and techniques in their domain and which challenges these modeling languages and techniques address. The resulting map enables identifying limitations and challenges, as well as best practices in the field. Also, it supports identifying new lines of research and provides a corpus for future investigation.

In this paper, we present an extension of our SMS on modeling languages in Industry 4.0 presented in [159]. The previous study [159] included 1466 papers that were published until April 2017. This contribution extends its investigation with 1878 additional papers published until February 2018 to describe the use of modeling languages in Industry 4.0. Out of these, 186 additional papers were included in the resulting map. With Industry 4.0 being a multi-disciplinary, heterogeneous challenge, we consider modeling and modeling languages in a broad sense, i.e., we include 3D modeling, knowledge representation, business process modeling, and other modeling techniques into our study.

Following a detailed search strategy involving six digital libraries, we initially identified 3344 unique publications. Out of these, 408 publications were selected and categorized using a particular classification scheme focusing on the contribution types, research types, Industry 4.0 concerns, and modeling contributions. We present the concerns addressed by research on modeling in Industry 4.0, how these concerns are investigated, when and where the results are published, and by whom. The resulting research landscape can help to understand, guide, and compare research in this field. In particular, this paper identifies the Industry 4.0 challenges addressed by the modeling community as well as the challenges that seem to be less investigated. Through this, we obtain a classification scheme and structure the research on modeling languages and techniques for Industry 4.0. In summary, the contributions of this paper, hence, are:

– Extension of the mapping study with 1878 novel and unique primary studies published until February 2018 in Sect. 4
– A detailed explanation of the research method used for this extended systematic mapping study presented in Sect. 3.3.
– Novel investigations on modeling for cyber and physical concerns as well as on the use of standards in Sect. 4.3.
– The discussion and investigation of trends in modeling in Industry 4.0 based on differences between the papers presented until April 2017 and the subsequently published papers in Sect. 5.
– A vision on model-based DevOps for Industry 4.0 and its relation to our findings in Sect. 5.3.

In the following, Sect. 2 discusses related mapping studies and literature reviews, before Sect. 3 details our research method. Afterward, Sect. 4 presents our findings before Sect. 5 reports insights into modeling in Industry 4.0 and discusses a vision of model-based DevOps for Industry 4.0 in the presence of our findings. Sect. 6 discusses threats to validity before Sect. 7 concludes.

2 Related studies

Mapping studies are a common method to investigate research trends in software engineering [115]. Current studies include, e.g., the classification of techniques for test-set generation and selection [73], software development effort and cost estimation [71], the use of experimental studies [135], object-oriented design [11], the use of patterns [162], the usage of UML diagrams [121], the empirical evaluation of software requirements specification techniques [33], on software product lines [46,85], and domain-specific languages [82]. Aside from investigating different concerns, these mapping studies vary in the level of analysis detail and in the number of included publications (between 35 and 400). However, we found only a single mapping study on model-driven engineering [104]. That study surveys existing research on aspect-oriented modeling and code generation. However, several literature reviews and surveys focus on the Industry 4.0 domain in general.

A recent systematic literature review of Industry 4.0 related research efforts [88] discusses the state of the art in Industry 4.0, deficiencies in current research, and potential research directions that culminate in a research agenda. In this context, modeling is mentioned as a frequently used technique for managing complex production systems as well as products for both: development of new artifacts and better understanding existing ones. XML, UML, and AutomationML are mentioned as frequently used modeling languages. However, a more in-depth study on the modeling aspect is not provided that review focuses on giving a general overview of Industry 4.0 literature.

Originally initiated in Germany in 2011, Industry 4.0 has attracted much attention in recent literature. In their perspective on Industry 4.0, Vogel-Heuser and Hess identify a set of challenges for the domain [154]. In particular, they identify four key challenges for software engineering that are well known to the modeling community:

1. Transition to modular and maintainable interfaces as a fundamental basis for adaptable and evolvable systems.
2. Tracking of changes in hundreds of heterogeneous and distributed machines or plants on different operation sites operated over decades.
3. Management while ensuring consistency of software variants and versions, including self-adaptation and reconfiguration at runtime.
4. Adaptation of big data algorithms and technologies.

Following this paper, Mosterman and Zander [109] discuss the needs and challenges of developing and operating cyber-physical systems (CPS) along with corresponding technologies to address the challenges and their potential impact. In the same trend, Turowski et al. identify the current challenges on Industry 4.0 faced by companies through a survey [79]. The survey aims to understand the stakeholders expectations, requirements and the potential challenges Industry 4.0 poses in real case studies. Complementary to these works, Trappey et al. [151] provide a consolidated review of the latest CPS literature. In this survey, they provide a complete review of international standards and an analysis of patent portfolios related to the CPS architecture model. Hermann et al. identify design principles of Industry 4.0 based on quantitative text analysis and a qualitative literature review [64]. Their paper illustrates how the identified design principles support practitioners in identifying Industry 4.0 scenarios.

A recent literature review on technologies and applications in Industry 4.0 investigated a corpus of 88 papers retrieved via Web of Science and Google Scholar [92]. In this study, the author uncovers three popular frameworks for the realization of Industry 4.0, presents key technologies (such as 5G or agent-based systems) and discusses popular applications (smart factories, smart products, and smart cities). Overall, that paper serves as a compact signpost guiding through a small subset of Industry 4.0 literature.

Similar studies have been conducted regarding the application and benefits of model-based software engineering in embedded systems [3,89,150]. The first study [89] surveyed 112 software developers from different companies on the reasons for applying model-based software engineering, its effects, and shortcomings. The authors conclude that development in embedded systems already leverages models as

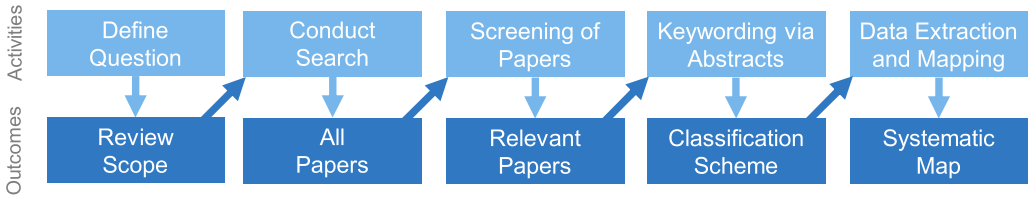


Fig. 1 The five phases of a systematic mapping study as proposed in [115]

primary development artifacts but that adopting MBSE still is challenging and that the tools are still challenging as well.

A study on the use of UML and model-driven techniques in the design of embedded software in Brazil surveyed 209 embedded software engineers and researchers [3]. Through the study, the authors identify a lack of knowledge about the application of UML and model-based techniques due to "the lack of skills" and "the lack of coherent tools". Moreover, the authors found that modeling is mainly used for documentation, whereas model-driven techniques, such as code generation, are hardly used. In contrast to this study, we investigate Industry 4.0 assuming that modeling is used. Through our search terms and exclusion criteria, we especially exclude sources not about modeling. Consequently, the research of our study differs not only on the subject but also on the focus.

However, with similar aspirations as [3], another study investigates the relevance of model-driven software engineering in the Italian industry [150]. The authors surveyed 155 Italian software professionals and inquired, inter alia challenges for the adoption of model-driven techniques, the use of code generators, interpreters, UML, and DSLs. In contrast to the results of [3] the authors uncovered that 68% of the surveyed professionals "produce models", whereas only 48% use model-driven techniques and almost all of the latter leverage code generation. Similar to the first study, they identified "easier maintenance" and "higher quality" as the main drivers for modeling. The study also finds the "typical anecdotal" challenges for adoption of modeling, such as requiring a high effort to create models and lack of supporting tools.

Thus, while there is already work on summarizing the research done in the field of Industry 4.0 and related fields, none of these studies is particularly concerned with the development or application of modeling languages.

3 Research method

A systematic mapping study identifies and classifies primary studies of the field under investigation. Through this, it aims to provide a systematic overview of the topics of research contributed to this area and the forms of contribution. We conducted this study following established guidelines [76,115] and included useful practices and suggestions from similar studies [26,44,80,82]. Ultimately, we employed the five-phase process for conducting this study proposed in [115] and depicted in Fig. 1: (1) define research questions; (2) search for primary studies; (3) identify inclusion and exclusion criteria and screen primary studies based on these criteria; (4) classify primary studies through keywording; and (5) extract and aggregate data.

In the first phase, we defined the scope of this study. In the second phase, we created the initial corpus of potentially relevant publications. In the third and fourth phases, we sanitized and reduced this corpus to include only relevant publications and classified according to research qualities derived from the research questions. In the fifth phase, we extracted data[1] from the publications to enable answering our research questions. This section describes the activities and decisions of these phases.

3.1 Research questions

We aim to identify relevant publications on development and use of modeling languages in Industry 4.0, which Industry 4.0 concerns are addressed with modeling techniques, how research addressing these concerns is conducted, and which modeling languages are used to contribute to these concerns. Moreover, we investigate who is contributing to modeling in Industry 4.0, where the contributions are published, and when they occurred. This manifests in the following research questions:

RQ1 *What are the expected benefits of applying modeling languages to Industry 4.0?* This question aims to uncover the high-level benefits expected by applying modeling languages to Industry 4.0.

RQ2 *Which Industry 4.0 concerns are addressed through modeling languages?* With this question, we investigate which concerns and challenges of Industry 4.0 are addressed through the different kinds of modeling languages.

RQ3 *Which kinds of modeling languages are used in Industry 4.0 and which concerns do they address?* This

[1] Available from companion website http://gemoc.org/modeling4Industry4.0/.

Fig. 2 Logical search clause defined to identify relevant literature

*("digital factory" **OR** "digital factories" **OR** "smart factory" **OR** "smart factories" **OR** "factory of the future" **OR** "factories of the future" **OR** "Industry 4.0") **AND** ("metamodel" **OR** "DSL" **OR** "UML" **OR** "domain-specific language" **OR** "modeling language" **OR** "modelling language")*

question investigates the use of modeling languages in Industry 4.0 and relates the findings of **RQ2** to the solutions contributed to the research field.

RQ4 *What are the most frequently applied research methods in the context of modeling languages for Industry 4.0?* This question aims to understand how research on modeling and modeling languages in Industry 4.0 is performed and how this relates to the concerns of **RQ2** and the tools of **RQ3**.

RQ5 *Who researches modeling languages in Industry 4.0?* This question investigates who has adopted this notion and contributes to modeling in Industry 4.0.

RQ6 *Where have the contributions been published?* Similar to the **RQ5**, we like to uncover which venues are relevant to publishing on modeling for Industry 4.0.

RQ7 *When did the contributions on modeling languages to Industry 4.0 occur?* With this question, we investigate when modeling started contributing to smart manufacturing.

To answer these questions, we conducted the literature search presented in the next section.

3.2 Search strategy and data sources

The search strategy guides the identification of relevant publications to answer the research questions. This includes conceiving an appropriate search query and identifying relevant libraries to apply this clause to. Industry 4.0, at its core, focuses on manufacturing, production processes, and ultimately the "factory of the future" [54,140] or the "smart factory" [86,118]. Thus, in contrast to [92], we included these terms in our search clause. Similarly, the second part of our search clause focuses on the objects of modeling research, its modeling language technology, instead of specific modeling languages. Thus, we search for publications mentioning metamodels, DSLs, modeling languages, or UML as relevant contributions to modeling in our context. This ultimately leads to the logical search clause depicted in Fig. 2.

Essentially, this is a conjunction of two disjunctions: The first part of the conjunction captures terms related to Industry 4.0. The second part captures terms representing the objects of modeling research. As we conducted a full-text search with this clause, we omitted including synonyms for "DSL" or "modeling language". Papers contributing to modeling should at least use these terms in either related work or the referenced literature. Although we cannot exclude omitting a small number of possibly relevant publications that do not provide such discussions, searching this way yields better results than just searching titles and abstracts. Moreover, we also did not enforce any inferior year-limit and included papers published until February 2018. Where such complex logical conditions were not supported, we searched for parts of the query and joined the results manually. For ACM Digital Library we could reuse the query as is (modulo minor changes to its concrete syntax). For Google Scholar we used its advanced search mode to separate to split the query into five queries, each containing one exact phrase of the modeling terms (i.e., "modeling language", "metamodel", etc.) and at least one of the domain terms (i.e., "Industry 4.0", "digital factory", etc.). We extracted the results using Harzing's Publish or Perish[2] software to extract results. Due to its limitation to ca. 1.000 citations per query, we downloaded the citations in multiple parts using inferior and superior year limits. We manually merged the resulting lists of citations and removed the Scholar-internal duplicates obtained by our process manually. Through this, we aim to minimize the issues of using Google Scholar for structured literature retrieval [21] (e.g., non-commutativity of logical disjunctions) while benefiting from its wealth of provided publications.

IEEE Xplore enforces a limit of 40 search terms, which did not affect our query and supports the use of nested Boolean queries through its advanced search, and hence data retrieval was straightforward. Similarly, retrieving citations from Scopus, SpringerLink, and Web of Science did not require any changes to the query as all three libraries support nested Boolean queries through their advanced search. Applying our query—with the explained operative modifications—to ACM Digital Library, Google Scholar, IEEE Xplore, Scopus, SpringerLink, and Web of Science yields the results presented in Table 1.

Due to including Google Scholar, this search includes documents unsuitable to answer our research questions, such as non-peer-reviewed publications, descriptions of curricula, or patents. These were removed in the next phases as illustrated in Fig. 3: First, we removed 1060 duplicate documents from the results, then we applied the criteria for inclusion and exclusion to remove additional 1369 documents based on their title, keywords, and abstracts in the screening phase (Sect. 3.3). Afterward, the results contain 1975 peer-reviewed, English, possibly relevant papers. We reviewed each of these papers during the classification phase

[2] Publish or Perish: https://harzing.com/resources/publish-or-perish.

Table 1 Search results returned from the different digital libraries

Digital Library	URL	Papers
ACM Digital Library	https://dl.acm.org	138
Google Scholar	https://scholar.google.com	3133
IEEE Xplore	https://ieeexplore.ieee.org	255
Scopus	https://www.scopus.com/	504
SpringerLink	https://link.springer.com	342
Web of Science	https://www.webofknowledge.com	32
Total (incl. duplicates)		4404

(Sect. 3.4) to understand whether these are relevant to our study and applied the criteria for inclusion and exclusion to the complete paper. In total, 408 papers remain in our corpus. The next sections detail our criteria for inclusion and exclusion as applied in the screening phase as well as in the classification phase.

3.3 Screening papers for inclusion and exclusion

The inclusion of a study into the classification phase of a systematic mapping study usually is decided on its title, abstract, and keywords. To reduce the corpus and enable reproduction of the study, we established the following inclusion criteria and exclusion criteria.

Inclusion criteria We identified potentially relevant documents based on the following three criteria:

1. Peer-reviewed studies published in journals, conferences, and workshops.
2. Studies are accessible electronically.
3. From title, abstract, and keywords, we can deduce that the paper focuses on developing or applying modeling languages in Industry 4.0.

Exclusion criteria Documents fulfilling the inclusion criteria may still be excluded based on the following four criteria:

1. Studies not available in English.
2. Studies not systematically peer-reviewed, such as books, slides, websites.
3. Teasers and short papers of less than two pages, such as calls for papers, editorials, or curricula.
4. Studies where Industry 4.0 is mentioned as a future application, related work, or broad context only, e.g., papers on the Internet of Things (IoT) or CPS mentioning Industry 4.0 as a possible use case only.

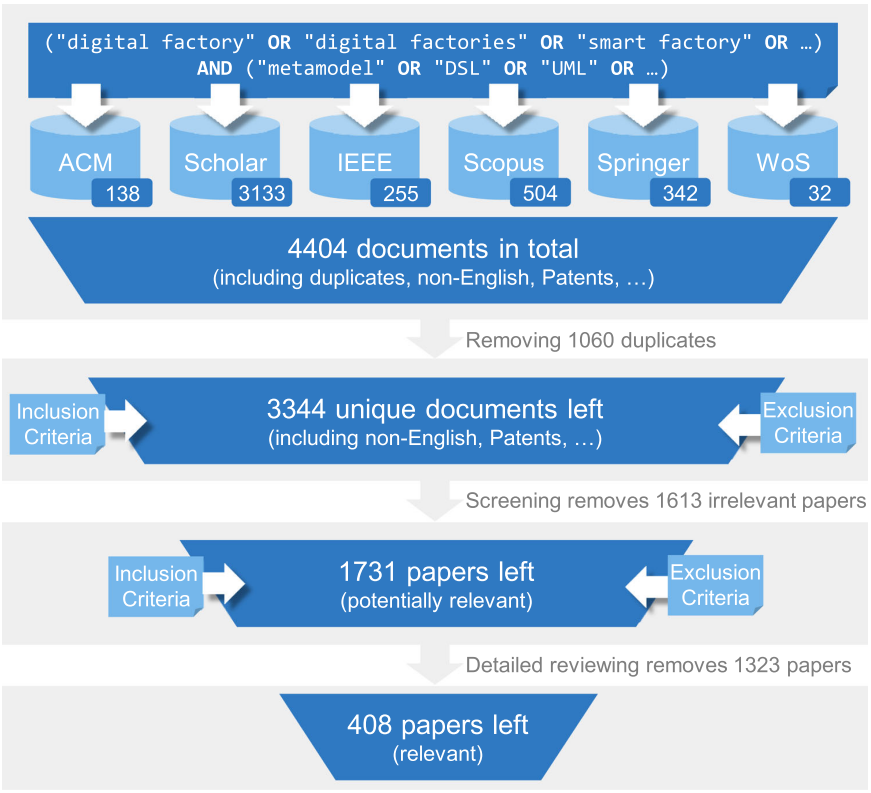


Fig. 3 Data collection initially produced 3344 unique documents, out of which 408 were identified as relevant for our study

To align our understanding of Industry 4.0 and the classification scheme, each of the authors reviewed the first 20 (about 1%) documents of the corpus of 3344 unique documents on his own. We discussed results and built a shared understanding of the documents as well as of our methodology and goals. As a next step, the remaining 3324 documents were filtered by the first author based on the unambiguous criteria of being non-English, non-peer-reviewed, or teasers only.

Removing 1593 documents left 1731 papers for review. These were split into three corpora of 430 papers and one corpus of 441 papers, which were reviewed and classified by a single author each. To continuously align our shared understanding of the topic and our classification scheme, inclusion, exclusion, and classification were discussed among the authors in bi-weekly teleconference sessions. During these, we excluded additional publications and refined our shared understanding of the classification scheme. We did, however, not discard papers based on their comprehensibility or venue alone. We also assigned each paper to the most suitable research type facet to yield a clear partitioning of the data set according to the categories in Table 3. Where the author reading a paper was uncertain about its inclusion or classification, we discussed this paper also among all authors. To prevent classification fatigue, we performed classification in blocks of at most one hour broken up by at least 15-minute breaks.

We then applied the criteria to titles, keywords, and abstracts. Where this did not suffice to determine inclusion, we temporarily included the publications for the classification phase to prevent excluding relevant, but suboptimally phrased publications. In that phase, the final inclusion or exclusion could be decided based on the publication's full text. Hence, this phase only eliminates publications obviously not within our study's scope and publications failing on formal requirements (such as not being available in English). In detail, we eliminated 1060 duplicates as well as 1369 publications outside this study's scope, including non-peer-reviewed publications (e.g., theses, technical reports, websites, patents, project deliverables, etc.), non-English publications, full proceedings (Google Scholar produces complete conference proceedings as results), and teasers (publications of two pages or less). Publications in languages other than English were excluded for this reason alone. Concurring with [81], we did not conduct any additional quality evaluation, such as including papers published at highly ranked conferences or workshops only. Hence, after the screening phase, 1975 potentially relevant papers remain in the corpus.

3.4 Classifying studies

In the classification phase, we reviewed the remaining 1975 papers to assign qualities of the dimensions derived from the research questions. To this end, we followed [115] in considering at least the introduction and the conclusion. However, for almost all most papers this was insufficient and we read the complete paper for proper classification. This also is the last phase in which publications were eliminated. Hence, after further elimination of 1567 irrelevant papers, a total of 408 publications remained. We classified these papers along the facets described in the following.

Contribution type facet

The first facet is inspired by [115] and classifies publications according to the type of research they contribute (**RQ4**). We adapted this to our study by employing the five *contribution types* presented in Table 2. These contribution types are disjoint and each paper was classified to provide exactly one contribution type. When a paper was suitable for more than one contribution type, we discussed this and assigned the most suitable contribution type.

Research type facet

Also inspired by [115], we classified the publications according to the *research type* they contribute. This enables addressing **RQ4** regarding the most frequently applied research methods contributed to modeling in Industry 4.0. Again, we adjusted these also to better fit to our study. In particular, we eliminated the category of philosophical papers as such papers did not occur. The resulting, disjoint, research types are depicted in Table 3. Each paper was classified to belong to exactly one research type. Papers suitable for more than one research type were discussed and assigned the most suitable research type.

Table 2 Contribution type facets inspired by [115] and adjusted to our research questions and corpus

Analyses	Papers contributing investigations *without constructive contributions*, such as [14,34,61]
Concepts	Papers suggesting ways of *thinking things*, such as new metamodels or taxonomies (this was titled "models" in [115], which is misleading in the context of this study), such as [113,124,144]
Methods	Papers suggesting ways of *doing things*, for instance, [126,141,165]
Metrics	Papers suggesting ways of *measuring things*, such as [72,152,155]
Tools	Papers presenting novel software tools related to modeling in Industry 4.0, e.g., [78,114,164]

Table 3 Research type facets also inspired by [115] and adjusted to our research questions and corpus as well

Evaluation	Papers evaluating *existing* techniques, e.g., [35,50,161]
Experience	Report of personal experiences, such as [17,18]
Solution	A *novel solution* is presented and argued for with case studies, for instance [23,39,117]
Validation	Papers presenting novel techniques and *experimenting with* them, such as [57,130,163])
Vision	Non-disruptive research agendas, such as the vision of model-based logistics engineering presented in [7,69,93]

Table 4 Industry 4.0 concern facets defined for the corpus of 408 papers

Digital representation	Publications on modeling systems, factories, or knowledge as well as the standardization of digital representations
Failure handling	Publications focusing on failure management or safety aspects
Human factors	Publications addressing the human side of Industry 4.0, such as worker localization or human–machine interaction
Information management	Publications on accessing and distributing information
Integration	Publications focusing on integrating CPS with something (other CPS, processes, the cloud) at design time and runtime
Processes	Publications on the modeling and management of processes
Product modeling	Publications contributing to modeling (smart) products
(Re-)configuration	Publications focusing on modeling configuration, monitoring, system resiliency, and self-* properties
Verification and validation	Publications employing modeling to simulation and testing
Visualization	Publications on using modeling to system visualization, such as 3D modeling, augmented reality, or virtual reality

Industry 4.0 concern facet

We also classified the publications along the Industry 4.0 concerns addressed by the various publications. This addresses **RQ2** and aims to uncover which concerns are investigated how often. During classification, keywording (cf. [115]) the abstracts, introductions, conclusions, and, if necessary, of the complete paper, we produced the following *Industry 4.0 concerns*. In contrast to contribution types and research types, these concerns are not disjoint and included papers can contribute to multiple concerns (Table 4).

Modeling technique facet

To find answers to **RQ3** regarding the modeling tools and languages used in Industry 4.0, we also classified the publications along this dimension. Overall, we found various modeling techniques (e.g., different CAD tools, UML dialects, DSLs, knowledge representation languages, etc.) and many papers addressed more than one modeling technique. To prevent dissipating the results we sorted the modeling techniques into groups (such as 3D modeling, architecture description languages, or business process modeling techniques) and isolated modeling techniques specific to Industry 4.0 (such as AutomationML). This produced the 15 groups presented in Table 5.

Moreover, we also investigated whether the included publications report on real-world industrial applications. Out of the 408 included publications, only 23 (5.64%) reported such applications. The industrial domains include automotive [38,68,78], avionics manufacturing [119,125], packaging [155], production of white goods [8], oil production [70,131], and production of windows and doors [6]. The next section presents our main findings along the four classification dimensions.

4 Findings

This section presents our findings on the expected benefits of applying modeling languages to Industry 4.0 as well as on the contribution types, research types, Industry 4.0 concerns, and modeling concerns for the included papers.

4.1 Expectations on the impact of modeling languages on Industry 4.0

With **RQ1** ("What are the expected benefits of applying modeling languages to Industry 4.0?"), we address the expected impact of contributing research in modeling to Industry 4.0 challenges. To this effect, we extracted these expectations whenever these were made explicit. Out of the 408 publications included after classification, only 55 (13.48%) papers explicitly described the authors' expectations on the impact of their contribution. The expectations include reducing the cost of production system integration [52], saving energy on production system reconfiguration [102], and remaining internationally competitive in high-wage countries [141]. We classified the expectations into expectations on

Table 5 Modeling language facets

3D Modeling	Techniques for representing geometric properties, e.g., for factory planning or augmented reality systems, including AutoCAD [43] and CATIA 3D [163]
ADL	Techniques employing architecture description languages [103], e.g., [30,31]
AML	Techniques employing the AutomationML [41] plant engineering data exchange format, such as [20,128]
BPM	Techniques for business process modeling in the context of Industry 4.0, for instance [75,142]
CMSD	Approaches based on Core Manufacturing Simulation Data (CMSD), such as [106]
DSL	Domain-specific languages, e.g., EDDL [127] or SDL [136]
ER	Entity-relationship modeling, cf. [9,125]
Formal Modeling	Automata-based and mathematical modeling approaches, including Petri Nets [90] or Priced Timed Automata [102]
GPL	Techniques employing general programming languages (GPLs), for instance, to model the services provided by a robotic manufacturing system [67]
KR	Knowledge representation languages, using, for instance, OWL [91]
Meta	Various metamodeling techniques, such as [31,83]
Simulink	Approaches using MATLAB/Simulink, e.g., [48,98]
SysML	Techniques employing SysML, including [12,138]
UML	UML and UML profiles, such as [95,123]
XML	XML-based modeling techniques, for example [99,148]

- reducing time (development time, time-to-market),
- reducing costs (of development, integration, (re-) configuration),
- improving sustainability, and
- improving international competitiveness.

Overall, the included publications explicated 59 expectations. Out of these, most publications expected modeling to either reduce cost (26x mentioned) or time (22x). Only a few publications propose modeling to improve sustainability (4x), increase international competitiveness (3x), to facilitate learning (2x), or to enhance the quality of products (2x). However, as the number of papers making the expectations of contributing modeling in Industry 4.0 explicit is rather small, these motivations cannot be generalized.

4.2 Industry 4.0 concerns addressed with modeling languages

With **RQ2** ("Which Industry 4.0 concerns are addressed through modeling languages?"), we investigate which concerns of Industry 4.0 are addressed using modeling techniques and how they are addressed in terms of contribution types (Table 2) and research methods (Table 3).

Investigating this, we found that most publications on modeling in Industry 4.0 contribute *methods* to challenges in digital representation (considered by 120 publications), integration (113), and processes (73). Out of the 614 concerns addressed by the included publications, these three combinations of contribution types and concerns make up 49.84% of concerns addressed by papers of our corpus. Overall, the majority of contributions are methods (69.71%) or concepts (13.52%), whereas tools (9.61%), or metrics (0.49%) are contributed significantly less often.

With contributions claiming to reduce costs and time (cf. Sect. 4.1), the lack of papers contributing metrics to track these claims is surprising. However, the new papers included after April 2017 do not investigate metrics at all. The results concerning contribution types—as inquired by **RQ4** ("What are the most frequently applied research methods in the context of modeling languages for Industry 4.0?")—are depicted on the left part of Fig. 4 and these findings are reflected by the research type contributions on its right part. Most contributions are solution proposals (i.e., application of existing techniques to solve particular problems) that focus either on digital representation or on integration challenges. It is also surprising that only a few publications investigate modeling for the (smart) product, which is supposed to control its production processes in many visions of Industry 4.0.

With respect to the publications' research types, we found that solution proposals make up 123.04% of the publications. These also most often address digital representation (addressed in 141 publications), integration (123), and processes (84). Out of the 614 concerns addressed by the publications included in our corpus, these three combinations of research types and concerns make up 49.84% of addressed concerns (Fig. 5).

Other research contributions are significantly less common. Evaluation reports contribute only 12.25% of the included papers, validation papers only 6.86%, vision papers only 5.88%, and experience reports only 2.45% of the included papers are contributed significantly less often. That most solution papers also are method papers might reflect the very constructive research typical to modeling. However,

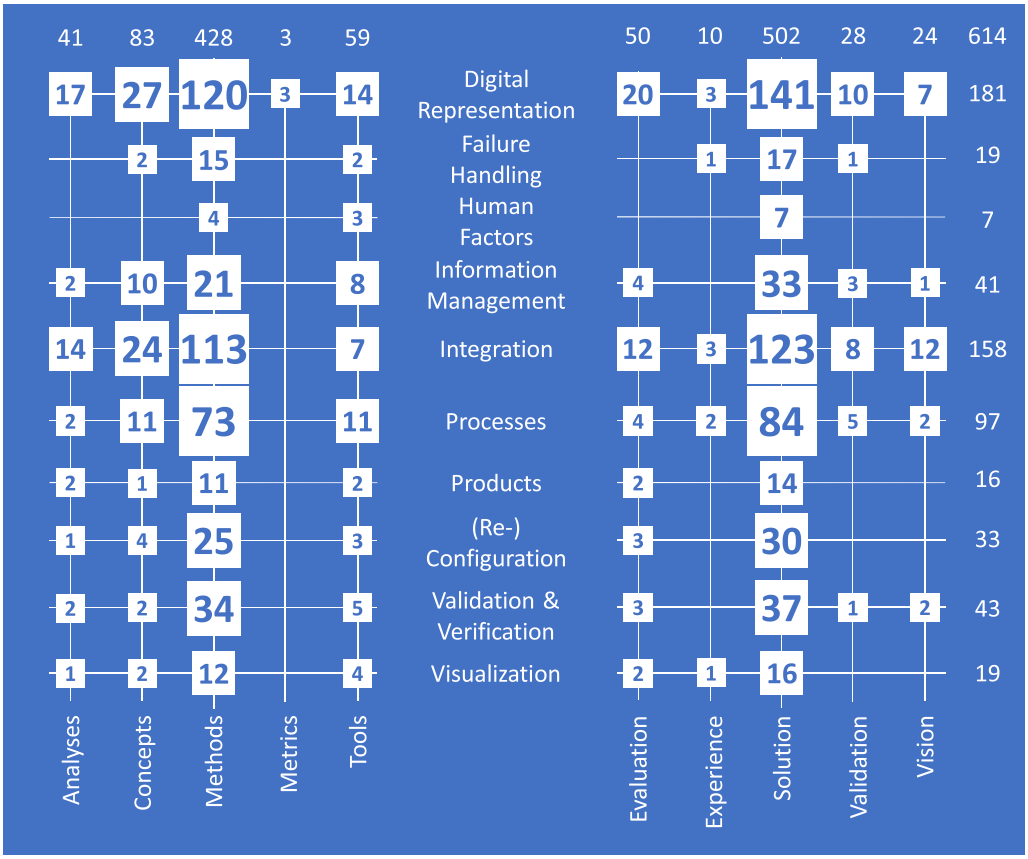


Fig. 4 Industry 4.0 concerns by research type and contribution type

the large number of method papers over papers contributing new concepts, validating new techniques, or proposing visions implies that research mainly approaches Industry 4.0 with established methods and techniques. This is supported by our findings on the modeling techniques contributed to Industry 4.0 presented in the next section.

Moreover, we investigated whether research on modeling languages in Industry 4.0 focuses more on the cyber (i.e., software) elements of automation or on its physical elements. To this end, we noted whether the publications explicitly mention which kind of parts the contributions are applied to. We found that 252 (61.76%) explicate this. They provide contributions focusing on cyber elements, physical elements, activities, or a combination thereof. Overall, 59 (14.46%) publications focus solely on cyber elements, 36 (8.81%) focus solely on physical elements, and 29 (7.11%) focus on activities that are not specified whether being cyber or physical. Of the remaining publications, 106 (25.98%) focus on cyber-physical elements, 10 (2.45%) on purely physical activities, and 12 (2.94%) on activities incorporating cyber and physical elements.

While research on modeling languages for Industry 4.0 is very balanced between contributing to handling cyber elements and physical elements, modeling physical elements or processes operating with them is important to modeling in Industry 4.0.

4.3 Modeling languages applied to Industry 4.0

Regarding **RQ3** ("Which kinds of modeling languages are used in Industry 4.0 and which concerns do they address?"), out of the 408 publications included in our classification, a total of 86 (86.03%) publications explicitly specified the (meta)modeling technique the authors applied to Industry 4.0. Examining these publications produced 124 different modeling techniques. Most notably among these are:

- Variants of UML, such as DiSpa [16], Mechatronic UML [134], UMM [100], and UML4IoT [149];
- The systems modeling language (SysML) [147] and its variants, such as SysML4Mechatronics [48] and SysML4Modelica [19];

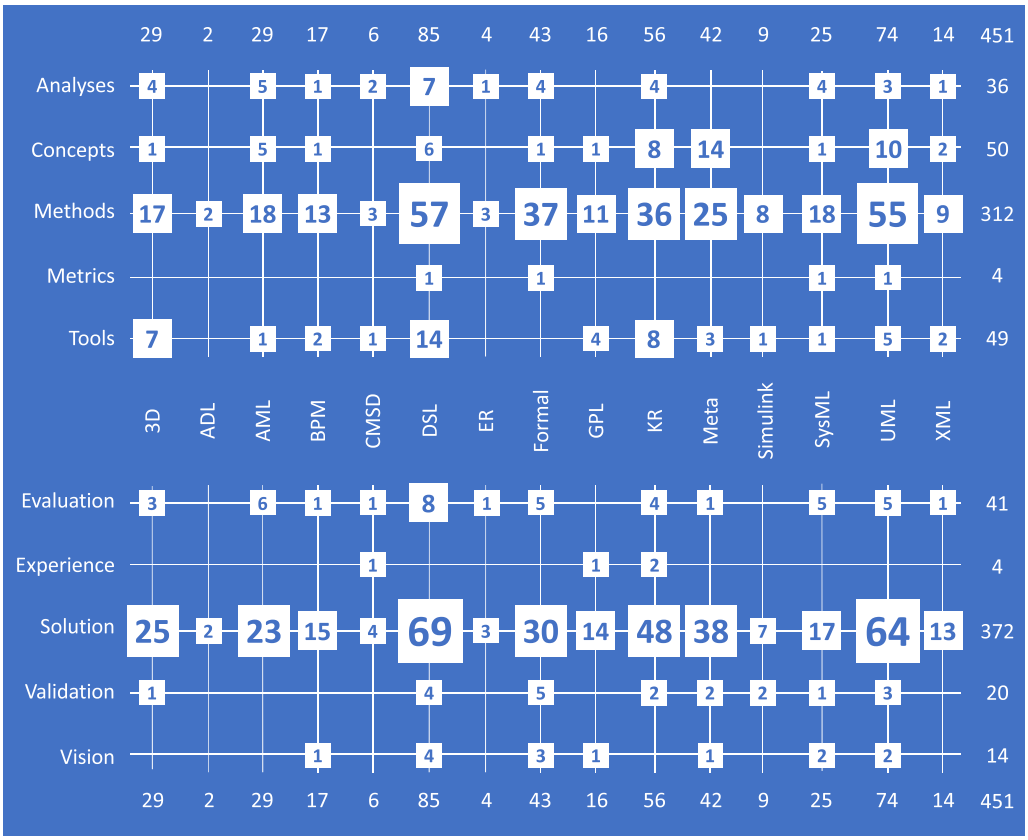


Fig. 5 Modeling language facet by research type and contribution type

- Knowledge representation techniques, mostly employing the Web Ontology Language (OWL) [60,110] or the Semantic Web Rule Language (SWRL) [66,129].
- Metamodels specific to Industry 4.0 challenges, such as the industrial metamodel for automation systems [107] or AutomationML [83].
- Metamodeling techniques, such as ADOxx [45,156], MetaEdit+ [30,31], or Xtext [55,77].
- Various DSLs, such as the EXPRESS DSL for product data modeling [39], the virtual factory data model [74], the Industry 4.0 process modeling language [116], the graphical modeling language for value networks [133], or the graphical modeling framework for production processes [94].

Overall, out of the 408 classified papers, 85 (20.83%) contribute or apply DSLs to specific to Industry 4.0 challenges and total of 74 (18.14%) papers employ UML (including variants). We also observed that leveraging UML and DSLs is not mutually exclusive in Industry 4.0 as 8 of the publications, such as [5,56,139], employ both. A total of 74 (18.14%) papers employ knowledge representation techniques, 29 (7.11%) papers use AutomationML [41], and 25 (6.13%) papers use SysML to address Industry 4.0 challenges. We also found 42 (10.29%) publications that discuss some form of conceptual metamodeling, i.e., describing the entities and their relations, of a specific aspect of Industry 4.0. Out of these only eight papers explicitly identified the metamodeling techniques used to define software languages for Industry 4.0 challenges. These either employed language workbenches, such as Xtext [77,105,112] or MetaEdit+ [30,31], or generic metamodeling frameworks, such as MOF [86], and Ecore [83]. Overall, 127 (31.13%) of the overall contributions address Industry 4.0 challenges with new DSLs or metamodeling techniques. This could hint at modeling challenges that cannot be properly addressed by established modeling techniques.

To answer **RQ3**, we also investigate which modeling languages are applied to address the different Industry 4.0 concerns. The results, depicted in Fig. 6, show that UML is used mostly to solve challenges in digital representation (39 publications) and integration (35 publications), which is consistent with identifying these as the most important challenges addressed by included publications. Consequently,

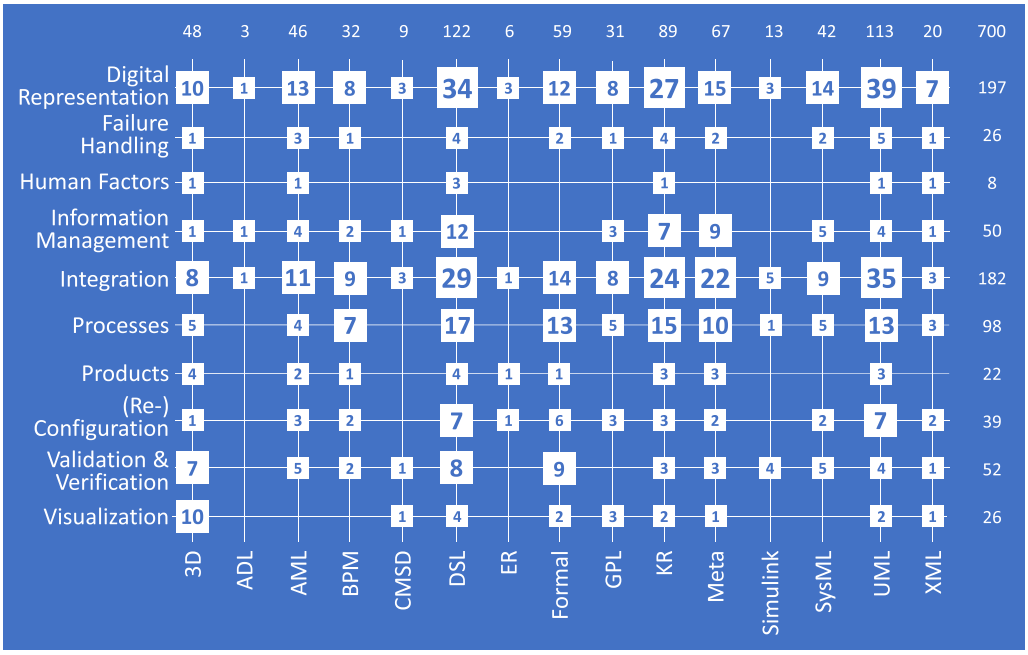


Fig. 6 Individual Industry 4.0 concerns and modeling techniques addressing these

these also are the two concerns most often addressed with knowledge representation techniques, DSLs, SysML, and AutomationML as well. For process modeling, another important aspect if Industry 4.0, DSLs are most popular (17 publications), followed by the application of knowledge representation techniques (35) publications), formal methods (13) publications), and UML (also 13) publications)

Overall, the concerns digital representation and integration—addressed by either AutomationML, various DSLs, knowledge representation techniques, SysML, or UML—represent 33.43% of the 700 concerns addressed with modeling languages. Together, these are a major focus of the field's research activities. While the usage of UML and DSLs is almost equally distributed between both concerns, knowledge representation techniques lean toward digital representation challenges.

The results also show that neither validation and verification, nor the human factors crucial to the success of Industry 4.0 or product modeling are investigated as much as integration and digital representation. Whereas the former might require solving digital representation and integration (to some degree) first, the lack of research on the latter two is elusive. Unless the smart factory of the future is fully automated, human interaction and control are necessary and should be considered appropriately.

We also observe that standards are crucial bases on shared understanding in the context of Industry 4.0. And while many papers apply techniques implementing standards to Industry 4.0, out of the 408 papers, 66 (5.64%) papers explicitly discuss, relate to, or challenge 54 different standards defined by the International Organization for Standardization (ISO), the American National Standards Institute (ANSI), the International Electrotechnical Commission (IEC), the Simulation Interoperability Standards Organization (SISO), the American Society of Mechanical Engineers (ASME), and the National Institute of Standards and Technology (NIST). The standards are addressed in the context concerns identified as research contributions of the corpus, including digital representation, human factors, integration, metamodeling, processes, and visualization. But they also address cross-cutting concerns, such as the environment, quality issues, safety, and security.

With integration being one of the Industry 4.0 concerns investigated most often, the most popular standards regarding modeling for Industry 4.0 also focus on integration as well. The standard for the exchange of product model data ("STEP", ISO 10303) is considered most often and discussed 12 times. It is followed by the standard for the integration of lifecycle data for process plants including oil and gas production facilities (ISO 15926), which is mentioned 9 times, and the standard on enterprise-control system integration, mentioned 7 times (IEC 62264). The standard defining a data model for computerized numerical controllers ("STEP-NC", ISO/TS 14649) also was mentioned 5 times. Other standards discussed at least once in the context of integration include the standards on data element types with associated

classification scheme (IEC 61630), the parts library standard ("PLIB", IEC 13584), Core Manufacturing Simulation Data ("CMSD", SISO-STD-008), manufacturing message specification ("MMS", ISO 9506), industrial manufacturing management data ("MANDATE", ISO 15531), metadata registries (ISO/IEC 11179), or the metamodel framework for interoperability (ISO/IEC 19763). Overall, integration is the main driver for the standardization of modeling techniques in Industry 4.0.

Other important drivers for discussing, challenging, or relating to standards are (1) processes and process modeling which is addressed by the standards ANSI/ISA-88, ISO/DIS 18828-5, IEC 61499, ISO 22400, ISO 60848, ISO 6983, ISO/IEC 19510, and ISO/IEC 6523; (2) digital representation in the context the standards IEC 61346, ASME B5.59-2, ISO 42010, ISO/IEC 10746 ISO/IEC 14662, and ISO/IEC 19501; and (3) visualization with the related standards ISO 10628, ISO 15519, ISO 3511, ISO 1219, ISO/PAS 17506, ISO 14306, ISO 14739. Overall, this indicates that standardization is in line with the general research direction in the field.

Out of the 54 standards, at least 13 standards address topics of direct interest for the modeling community in software engineering, as these directly specify, imply, require, or constrain (meta) modeling techniques. These include standards prominent in software engineering, such as Unified Modeling Language (UML) in version 1.3 (ISO/IEC 19501), the Business Process Model and Notation (BPMN, ISO/IEC 19510), or the Meta Object Facility (MOF, ISO/IEC 19502), or architecture description (ISO/IEC 42010). The majority of modeling-related standards in Industry 4.0, however, appear to be less prominent in the modeling community in software engineering. We assume that this indicates a gap between both communities, modeling in automation systems engineering and modeling in software engineering.

4.4 Countries and institutions contributing to the field

Investigating **RQ5** ("Who researches modeling languages in Industry 4.0?"), we found that 184 (45.1%) of the publications were contributed by teams including German authors, followed by teams including authors from the USA (35 publications), Austria (29 publications), and France (28 publications) as depicted in Fig. 7. Overall 53 countries contributed to research on modeling languages for Industry 4.0 in 521 contributions (papers with authors from multiple countries count as multiple country contributions). Out of these, the 10 most actively publishing countries produce 392 (75.24%). Among these 392 contributions, 325 (82.91%) contributions are from Europe. This suggests that modeling in Industry 4.0 still largely is a European research project despite starting related initiatives in many countries across the globe.

Aside from the contributing authors' countries, we also identified the institutions most actively engaging in research on modeling languages for Industry 4.0. Overall 358 institutions contribute to the field. Due to Industry 4.0 being coined in Germany and 45.1% of the included publications having German co-authors, it is unsurprising that out of the 10 most active institutions in this field, 6 are from Germany (as depicted in Fig. 8). It is, however, interesting that among these most active institutions are two national research institutions, the USA's National Institute of Standards and Technology (NIST) and the National Research Council of Italy, whereas for Austria, Germany, and Spain the most active institutions are universities or companies. Multi-national institutions were assigned the country of their headquarter.

Out of the 358 overall contributing institutions, 235 (65.64%) are universities, 72 (20.11%) are companies, and 51 (14.25%) are other kinds of research institutes, such as the Department of Energy of the USA, the Greek ATHENA Research and Innovation Centre, or the German Fraunhofer institutes. While this might indicate that—despite being a business-driven paradigm (cf. Sect. 4.1)—research on modeling in Industry 4.0 could be driven by academic researchers, our initial data collection also produced 235 (7.03% out of the 3344 potentially relevant unique publications) patents via Google Scholar. These indicate that there is industrially driven research on Industry 4.0 that does not necessarily lead to scientific publications.

4.5 Popular venues for publications on modeling languages for Industry 4.0

Regarding **RQ6** ("Where have the contributions been published?"), we found that most papers are published at conferences (249, 61.03%), followed by journals (137, 33.58%), and workshops (22, 5.39%). We also identified the most popular journals, conferences, and workshops of this particular field of research, to answer **RQ6** on the most popular venues for modeling research in the context of Industry 4.0.

Figure 9 presents the 10 most popular journals, where (15.26%) of the related journal papers are published. Where journals produced the same number of publications, they are represented in alphabetical order according to their name. Notably, no publications of the Transactions on Industrial Informatics (no. 4) or the International Journal of Production Research (no. 5) were included in the dataset until April 2017. However, the small numbers of publications in these most popular journals, do not support conclusions over their importance. As the Industry 4.0 matures, future studies maybe could draw such conclusions based on larger corpora of relevant publications.

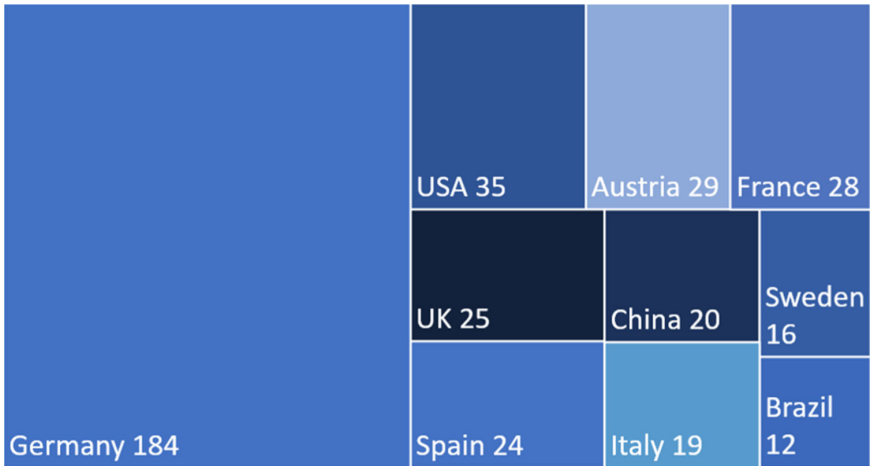


Fig. 7 The 10 most actively publishing countries with authors contributing to modeling in Industry 4.0 are largely from Europe and contribute 82.91% of the publications

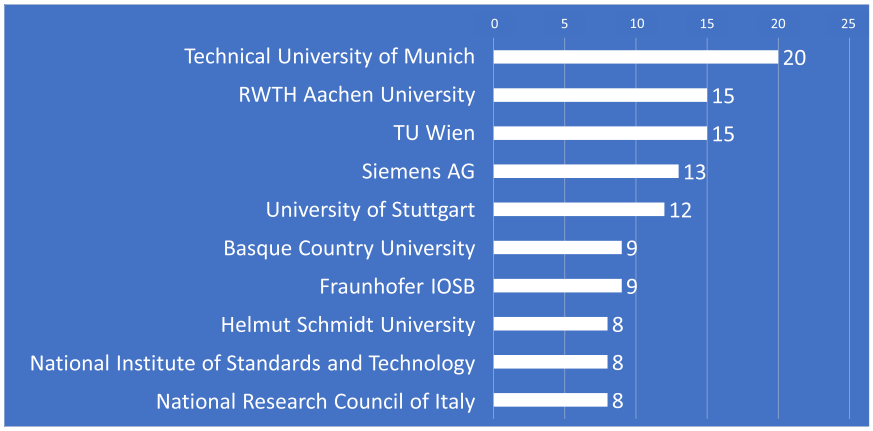


Fig. 8 The 10 most active institutions engaging in research on modeling for Industry 4.0

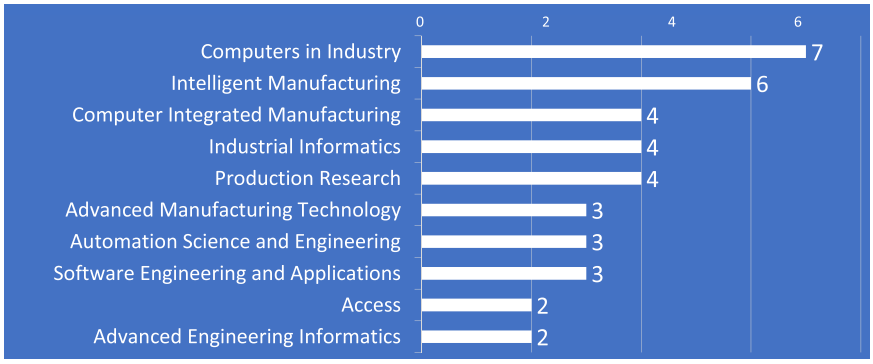


Fig. 9 Most popular journals for publications on modeling for Industry 4.0

The 10 most popular conferences regarding modeling in Industry 4.0, depicted in Fig. 10, publish, with 38.15%, also a large part of the related conference publications. Again, conferences yielding the same number of publications are represented in alphabetical order according to their full name. The large number of conference publications supports the conclusion that the conference on Emerging Technologies and Factory Automation (ETFA)—publishing 31 (12.45%) of included conference papers—is the most important conference for publications on modeling in Industry 4.0. The nine other most popular conferences published between 4 and 15 papers on the topic. With some distance to ETFA, the International Conference on Industrial Informatics (INDIN), the CIRP Conference on Manufacturing Systems (CIRP CMS),

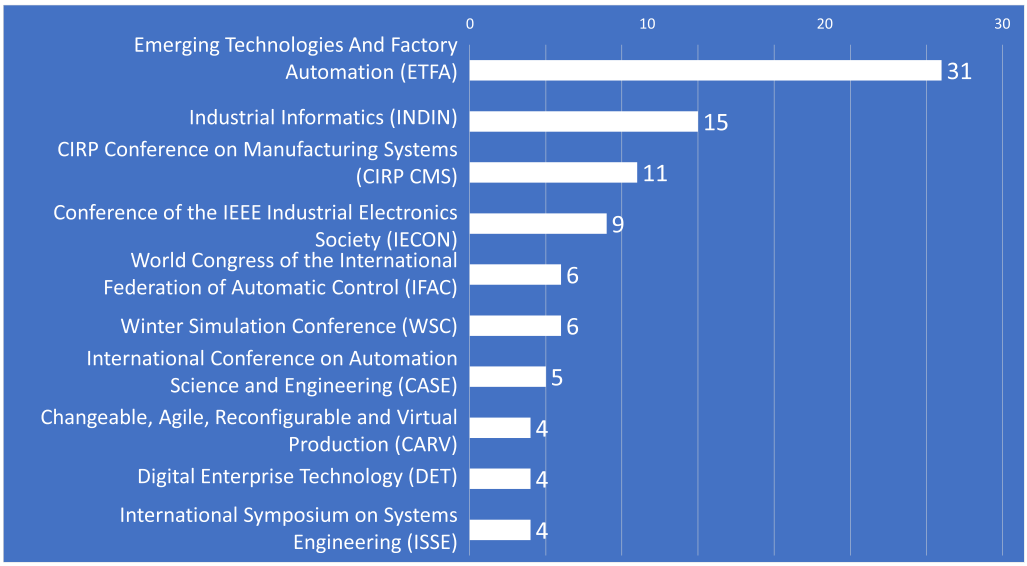


Fig. 10 Most popular conferences for publications on modeling for Industry 4.0

and the conference of the IEEE Industrial Electronics Society (IECON) are the next most popular conferences for modeling in Industry 4.0. Together, they publish a similar share (14.05%) of related papers.

Overall, the 10 most popular journals and conferences publish 32.6% of the included papers, which hints at a healthy distribution of publications over multiple venues. This is reflected by the 22 workshop papers included in the classification, which were published at 20 different workshops. In this context, no trends on workshop popularity can be observed.

4.6 Publication activities over time

Regarding **RQ7** ("When did the contributions on modeling languages to Industry 4.0 occur?"), we found that modeling for Industry 4.0 was already addressed as early as 1991 [158], although the term "Industry 4.0" was not coined yet. Over half (219, 84.56%) of related publications were published starting in 2016 and 345 (84.56%) of the publications are from 2011 (the year the term "Industry 4.0" was coined) or later (cf. Fig. 11) and later. We also observe that the number of papers increased by (31.82%) on average per year since 2011. Whether this trend will continue requires future investigation.

5 Trends and perspectives on modeling for Industry 4.0

Updating our previous mapping study provided the unique opportunity to investigate publication trends between April 2017 and February 2018. While we are aware of this compact time frame, comparing both data sets produced interesting observations. Subsequently, this section presents perspectives on potential future trends of research in modeling for Industry 4.0.

5.1 Trends in modeling for Industry 4.0

We extended the mapping study with papers published between April 2017 and February 2018. Through this, we included 186 (an increase of 83.78%) additional papers into our observations, which corresponds to the increase of addressed concerns of 76.94%. Comparing both data sets yields insights into differences between publications until April 2017 and afterward.

Considering changes in contribution types with respect to addressed Industry 4.0 concerns, we found significant increases regarding methods (450%) for product modeling, validation and verification (183.33%), and failure handling (150%), as well as regarding tools for validation and verification (150%). Concerning the research types of the publications in the updated corpus, we found that solutions for product modeling (183.37%), for validation and verification (146.67%), and for information management (135.67%) increased the most. The absolute numbers of increases regarding contribution types and research types are depicted in Fig. 12.

Nonetheless, we also found that analyses (54.55%), methods (53.85%), tools (27.27%) for digital representation showed a disproportionately lower increase. This might indicate that some techniques for digital representation have

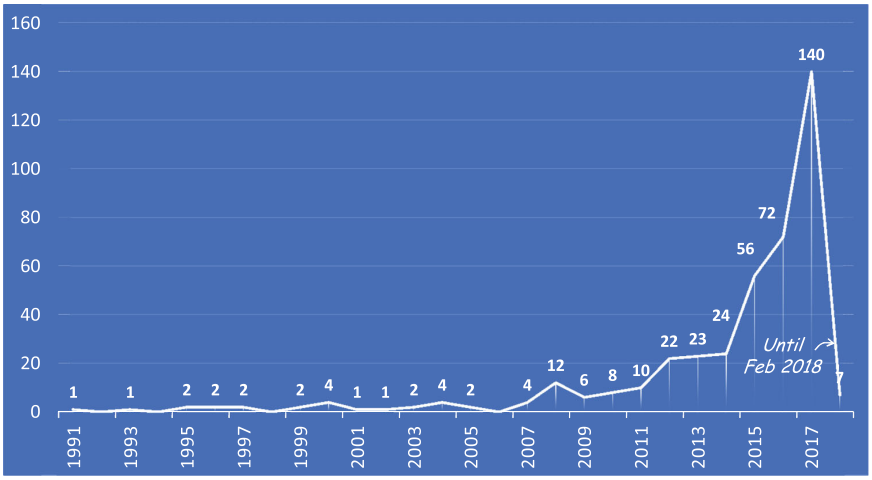


Fig. 11 Number of publications per year until February 2018 (missing numbers identify years without related publications)

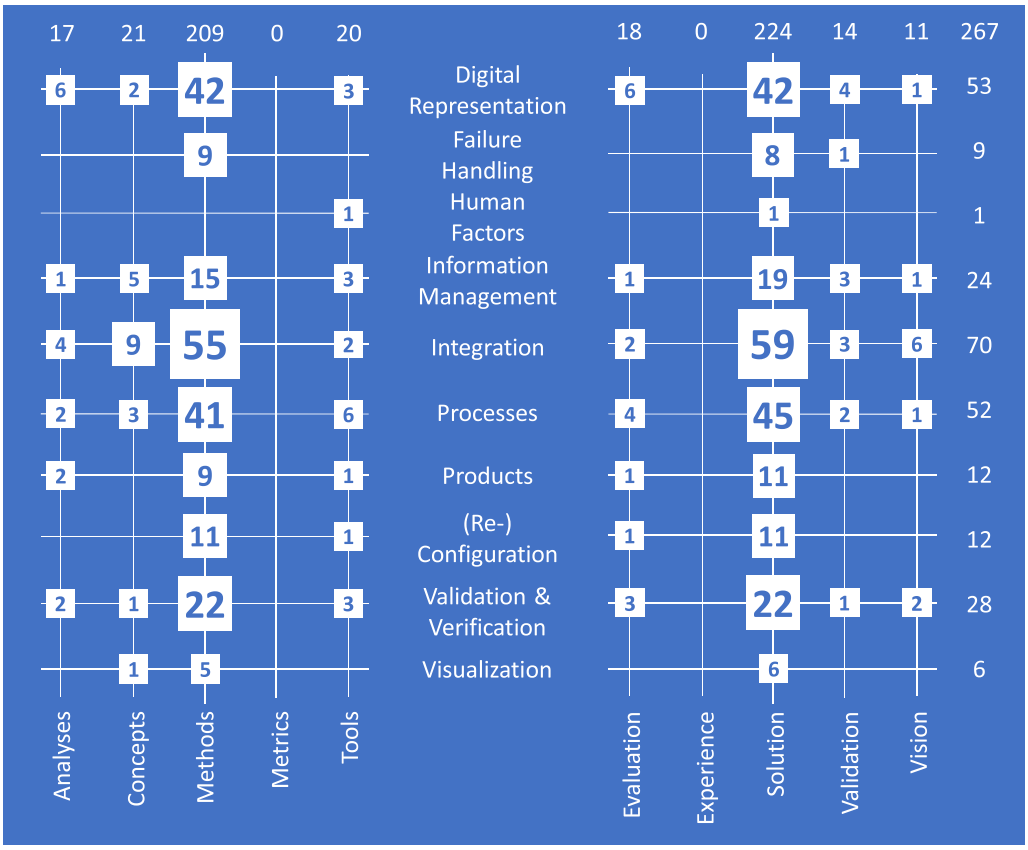


Fig. 12 Numbers of addressed Industry 4.0 concerns relative to the different contribution types and research types contributed by the 186 publications included since April 2017

become a stable basis for other research to build upon. Overall, contributions investigating product modeling (300%), validation and verification (186.67%), and information management (135.70%) increased the most. This change of focus from digital representation and integration, as found in [159], to validation and verification and processes could be in line with building on top of established representation and integration techniques.

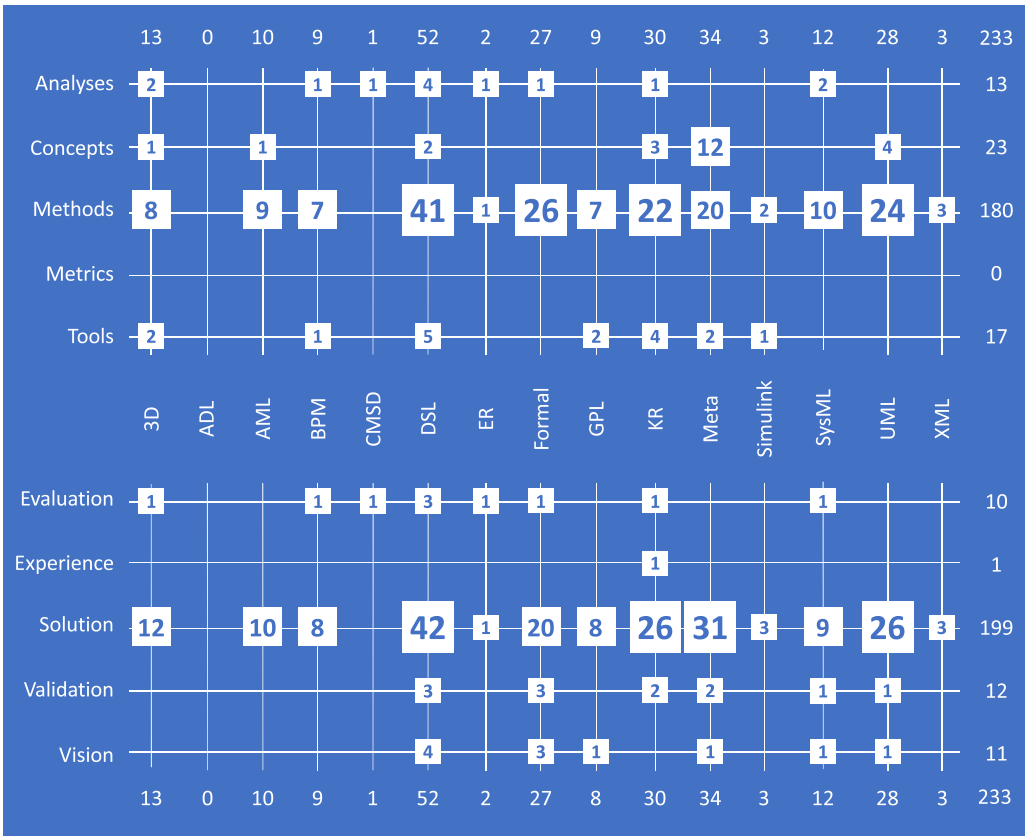


Fig. 13 Numbers of applied modeling techniques relative to the different contribution types and research types contributed by the 186 publications included since April 2017

Regarding the technologies addressed by the included publications, we found that, relatively, the use of metamodeling techniques (425%), formal methods (169%), and domain-specific languages (158%) for Industry 4.0 increased the most. This could be an effect of a wave of early approaches investigating applying more general or established modeling techniques, based, e.g., on UML or XML [159], abating. Moreover, this underlines the importance of modeling knowledge in Industry 4.0.

In contrast, the use of pure XML or Simulink increased by 27% and 50%, respectively, only. However, with their absolute numbers—as presented in Fig. 13—of publications considering pure XML or Simulink being low to begin with, their relative small increases might not imply any trends (Fig. 14).

Investigating trends regarding modeling techniques applied to Industry 4.0 concerns, we found that the application of the different metamodeling techniques to processes (800%) and integration (633%), as well as the application of knowledge representation techniques to information management (600%) increased the most. In contrast, the overall application of ADLs (0%) and CMSD (13%), XML (25%), Simulink (44%), and AutomationML (48%) increased the least. However, the low number of publications on metamodeling found initially [159] explains their relatively sharp increase, whereas the low numbers of publications applying ADLs, UML, or XML-based techniques emphasize the trend toward novel and specific modeling languages for Industry 4.0.

Considering the different venues relevant to publishing on modeling for Industry 4.0, it is notable that no publications of the International Journal of Production Research or the Transactions on Industrial Informatics were included in the dataset until April 2017.

5.2 Different perspectives on modeling

In line with our findings of standardization activities, the identified publication venues also indicate a gap between the different modeling communities (automation engineering, software engineering, etc.) related to Industry 4.0. From our experience, this also is visible in some of the topics relevant to modeling in Industry 4.0, such as 3D modeling, knowledge representation, or simulation that seem to attract fewer

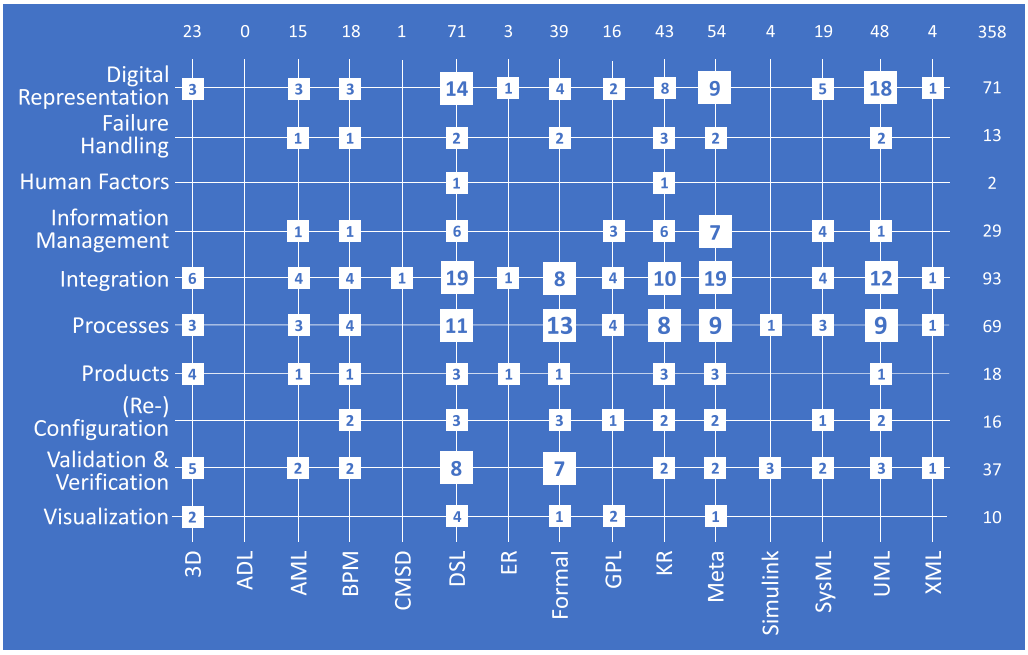


Fig. 14 Numbers of Industry 4.0 concerns addressed by the different modeling techniques as contributed by the 186 publications included since April 2017

publications in the software engineering modeling community (e.g., the MODELS or ECMFA conferences). This also is reified by the different standardization or specification bodies of the different communities. While the OMG considers automation engineering standards, as well as software engineering standards, or cross-cutting standards (e.g., on environment, safety, or quality), there are modeling-related standards by standardization bodies not primarily considering software engineering, such as the Core Manufacturing Simulation Data (CMSD) standardized by SISO (SISO-STD-008) or the related standards by IEC (e.g., IEC 61630 or IEC 62424).

Moreover, there are various standards addressing issues relevant to the software engineering modeling community, such as (1) the standard for the "exchange of product model data" ("STEP" [120]) reified in ISO 10303, which comprises the EXPRESS [120] modeling language, the standard data access interface, or the STEP-NC [160] machine tool control language; (2) the standard for "industrial automation systems and integration - Parts library" ("PLIB") reified in ISO 13569, which defines the OntoML ontology markup language; or the (3) the process specification language of ISO 18629. Hence, we suggest for software engineering researchers to consider these standards when contributing to modeling in Industry 4.0.

Moreover, with AutomationML [14], research and industry have started a promising initiative on modeling automation systems for Industry 4.0 that features research groups for all participating communities. However, the underlying technologies that define models and languages (e.g., XML) can significantly improve from research conducted in the community around model-driven software development.

We also found that research on modeling languages for Industry 4.0 to a large extent addresses challenges either typical to software and systems engineering, such as digital system representation and integration, or typical to artificial intelligence, such as representing knowledge about processes and resources and reasoning about these (cf. Fig. 6). Despite these challenges being central to computer science research, the most popular venues (cf. Sect. 4.5) suggest that this research is not discussed in computer science, but in journals and conferences related to automation engineering instead. Whether this is due to the contributing researchers' backgrounds is subject to ongoing research and cannot be answered from the data on contributing institutions alone (cf. Sect. 4.4).

Research on modeling languages for Industry 4.0 focuses on constructive contributions, i.e., methods solving specific problems (cf. Sect. 4.2), while there are few experience reports, validation research, or evaluation papers. Similarly, metrics and analyses, expected to be prominent for such a business-driven research agenda, are very rare. This is in line with the observation that only a few papers conduct empirical evaluations in industrial settings (5.64%) and might suggest

that the majority of solutions provided to the field are not mature enough for to be evaluated in the field, or that there is a significant amount of research not targeted at industrial needs. Also, there is a noticeable lack of vision papers on modeling for the newly coined research agenda of Industry 4.0, which might contrast the hypothesis that research on modeling for Industry 4.0 is of insufficient maturity.

5.3 Looking ahead on modeling for Industry 4.0

We are currently striving for new opportunities, but at the same time facing a dramatically increasing complexity in the development and operation of systems with the emergence of *Cyber-Physical Production Systems (CPPS)* [153] in Industry 4.0. This demands for more comprehensive and systematic views on all aspects of systems (e.g., mechanics, electronics, software, and network) not only in the engineering process but in the operation process as well [22]. Moreover, flexible approaches are needed to adapt the systems' behavior to ever-changing requirements and tasks, unexpected conditions, as well as structural transformations [87]. Modeling languages are traditionally more focused on the development phases as also indicated by our literature study. However, the reference architecture of Industry 4.0 explicitly targets the management of the complete lifecycle, going from development (i.e., type level) to operation (i.e., instance level) in addition to vertical and horizontal integration requirements. In this context, the later phases of the lifecycle may become a new playground for existing modeling languages. Although some of the surveyed languages already provide some support for type and instance level such as UML, typically the instance level modeling did not receive much attention compared to the type level.

To tackle the challenges of Industry 4.0, such as the flexible and resilient adaption of CPPS to changing requirements, the operation processes of CPPS, as well as their interplay with the engineering processes and vice verse, has to be taken into consideration also by the employed modeling languages. This raises the question of how model-based DevOps practices for CPPS can be achieved. Such practices are currently highly needed to reduce the time between identifying the necessity for a change and putting the appropriate change into production. Definitely, we have to go beyond the current support offered by current Product Lifecycle Management (PLM) tools [1].

Furthermore, current DevOps practices have to be completed to be applicable not only for code-based artifacts but for a larger variety of artifacts such as models, engineering documents, CAD drawings, simulation data, etc.

In the following, we present a vision for model-based DevOps as well as challenges related to the development of the next-generation modeling languages that have to be tackled to realize model-based DevOps for Industry 4.0. Finally, we conclude with the potential benefits of model-based DevOps but also enumerate potential barriers to model-based DevOps.

5.3.1 Model-based DevOps: a vision

While current DevOps practices apply to code integration, deployment and delivery, we envision the application of the very same practices at the model level. In such a vision, the various domain-specific development models are seamlessly integrated with operations, either models at runtime (e.g., model-based MAPE-K loop or digital twins) or a combination of software and hardware components within a given environment. In the last two decades, the MDE community developed a rich and useful toolset for implementing such a vision through the efficient development, usage, maintenance, and evolution of modeling languages. Figure 15 presents some of the modeling techniques that can be used across the DevOps cycle.

5.3.2 Model-based DevOps: What is needed from modeling language research?

Integration of the MDE Technologies with DevOps Technologies In the past decade, a plethora of different modeling languages for design, validation, verification, evolution, and transformation of models have been proposed. However, how these languages may be bundled into a pipeline for continuously integrating, building, testing, and deploying models into production environments is less explored. The only exception is the work by García and Cabot [51] who married continuous deployment technologies and model-driven technologies. Some approaches toward leveraging MDE in the context of PLM that might serve as a vantage point for moving from PLM with MDE to model-based DevOps include model- and standards-based data integration [101], increasing virtualization [2,30], or domain-specific languages tailored to the industry's processes [105].

Integration of different artifact kinds While current model-based technologies provide common services for model-based artifacts, other artifact kinds such as software components or hardware descriptions cannot be directly integrated with models. However, this demands integration techniques on the language level that support a progressive integration of models starting in the engineering process and extending into the deployment process even going to the operation processes. Activities in this direction include, e.g., integrating geometric tolerance information into STEP (ISO 10303) [132] or integrated representations of products, processes, and resources [10] to model system changes over its lifecycle.

Aligning operational data and design models A major challenge is the back-propagation of operational data (e.g.,

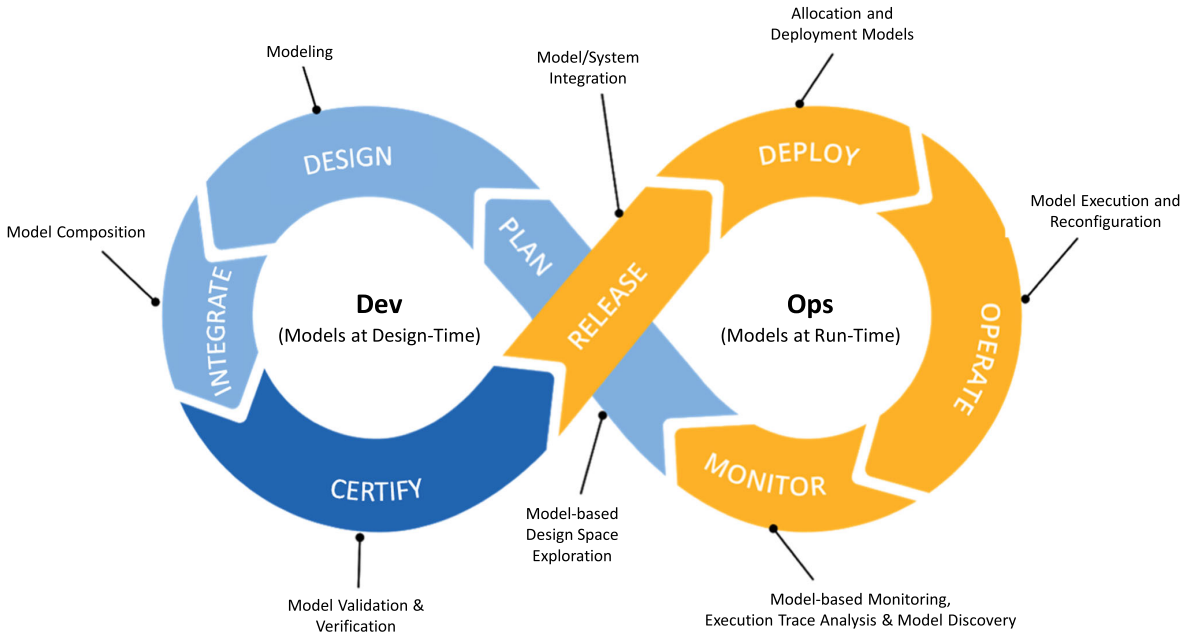


Fig. 15 A vision of model-based DevOps that aims to facilitate addressing the challenges of the CPPS of Industry 4.0 through pervasive modeling across their complete lifecycle

measures about performance, energy consumption, masses, costs, etc.) into the documentation provided on top of heterogeneous design models—which may be software engineering models, formal models of physical processes, knowledge bases, CAD, or something else. Currently, most of the modeling languages identified in our study lack a dedicated viewpoint for operations. Extensions to these languages are required to link to operational data or to store summaries of operational data in models (cf. [54,122]).

Visualizing operational data in design models Operational data is becoming huge in size for complex systems. Even if operational data is aligned with design models, current modeling languages most often fail short in visualization support for non-2D-diagram-based data. Additional requirements for visualization of design models occur such as how to visualize the underlying quality of the data such as uncertainties. Integrating sophisticated visualization techniques [4] are required to provide an understanding of operation-enriched design models.

Exploiting runtime models for continuous improvement of design models Runtime models have gained considerable attention in MDE, mostly in the context of self-* systems. Interpreting runtime models for continuous improvement of the design models (possibly through additional predictive models) would enable reasoning about the next versions of a system. Runtime models would indeed be very helpful here: for instance, assume the transform of the runtime models back into traces which can be replayed by simulators for animation, exploration, etc., on the design models. We, however, found that most publications focus on modeling languages to describe design-time models or runtime data is analyzed without a deeper connection to the design models.

5.3.3 Perspectives of model-based DevOps for Industry 4.0

The path toward model-based DevOps for Industry 4.0 yields specific benefits and challenges. For instance, considering business concerns, as presented in the BizDevOps approach [58], requires reasoning over the global system at the business level - the highest vertical level in the reference architecture of Industry 4.0. This level would benefit from the application of the DevOps principles at the model level as models are closer to the application domain and can provide a comprehensive representation of the system, including its environment and possible extra-functional properties related to business concerns.

Moreover, promoting DevOps principles at the model level enables leveraging it earlier in the development process. Hence, DevOps principles would not only apply to the integration, deployment, delivery, and operation of the global system, but could also apply at a finer level of granularity for the different concerns addressed during the development processes of plants, production systems, and products their various abstraction levels. This could lead to powerful development processes where automation and continuous feedback are not only available at the level of the global sys-

tem, but also at the level of the different concerns and across the various levels of abstraction. This, for instance, could facilitate operating (partly) virtual factories [28,74] earlier and support factory and CPPS integration planning as well as simulation of manufacturing novel products.

Based on our findings, obstacles to the adoption of model-driven DevOps in Industry 4.0 might arise from a gap between the modeling communities of Industry 4.0 and software engineering. As DevOps is a set of software engineering practices, it largely focuses on cyber (i.e., software) elements, a DevOps for Industry 4.0 must also incorporate its physical parts (cf. Sect. 4.2) and leverage associated modeling languages. Where DevOps, i.e., introducing change, for pure software elements is manageable, e.g., by over-the-air updates at runtime, changing physical elements at runtime is complicated to impossible. Similarly, software generally can be released and monitored more easily than the physical elements of Industry 4.0. Moreover, a DevOps for Industry 4.0 must comply with relevant industry standards and regulations (see Sect. 4.3). This is especially critical where (manual) certification prior to deployment is required as this can hamper the DevOps loop. Finally, while modern software engineering tools are providing open APIs to be integrated into DevOps pipelines which allow for automation and traceability, classical PLM tools for managing the lifecycle of physical components by virtual representations are often closed environments with proprietary data formats.

To sum up, with both communities generally leveraging different modeling techniques, standards, and tools, realizing DevOps for Industry 4.0 demands for concentrated efforts to bridge this gap. Otherwise, realizing the reference architecture of Industry 4.0 becomes a utopia.

6 Threats to validity

For identifying the threats to the validity of our SMS, we follow the four basic types of validity threats according to Wohlin et al. [157]. Our study is subject to threats to construct validity (research design), internal validity (data extraction), and conclusion validity (reliability). Threats to external validity (generalizability) are irrelevant as the results of this study can neither be generalized to problems domains other than Industry 4.0 nor to solution domains other than modeling.

Regarding threats to research design, the presented findings are valid only for our sample of papers. Hence, it is crucial to ensure the inclusion of as many relevant papers as possible. To achieve this, we included the Google Scholar digital library and only very carefully excluded publications. We are aware that a great number of subsequent exclusions for formal reasons (e.g., non-peer-reviewed materials) are due to querying Google Scholar. However, its inclusion was useful to capture venues not published in the other libraries. Overall, using Google Scholar led to including 207 papers that would have otherwise been omitted.

Moreover, we did not restrict our search to publications mentioning "Industry 4.0" explicitly, but also included the related terms of the search clause's first disjunction. Similarly, the search clause's second disjunction included terms closely related to modeling, without narrowing it to the exact terms. Instead, we used terms one can expect from relevant contributions to be included in the full text. This enabled capturing related publications without focusing on the very specific, partly ambiguous, modeling terminology. Our search clause also might entail a bias toward European research by explicitly mentioning "Industry 4.0", i.e., the name of a European initiative on smart manufacturing, whereas the names of other national initiatives (e.g., the Japanese "Industrial Value Chain" or the "Advanced Manufacturing Initiative" of the USA) are not part of the search clause.

Another threat to research design validity arises from the definition of the criteria of inclusion and exclusion. During the screening, only title, abstract, and keywords were considered. To prevent excluding relevant publications based on the lack of investigation, we included papers we were uncertain of temporarily. In the subsequent classification phase, the complete papers were read and inclusion or exclusion were decided ultimately.

Of course, our mapping study also is subject to the so-called publication bias, i.e., it can report on published results only. As publications focus on positive results, we cannot derive which modeling languages are not applicable from our data sources. Also, we restrict our research to work applied to Industry 4.0, instead of also considering *potential* applications to it. Due to its diversity, a study on the latter must include at least publications focusing on robotics, the Internet of Things, production planning, enterprise systems, human–computer interaction, and much more. However, including all these fields would dilute the validity of such a study.

Threats to conclusion validity arise drawing wrong conclusions and from the study's replicability. Regarding the former, we have discussed various issues that could lead to wrong conclusions in the context of threats to internal validity. For replicability, we detailed the complete research method in Sect. 3, which enables replicating every phase of this mapping study. Regarding some conclusions, such as the most active institutions or most relevant journals, our study is by construction biased toward institutions and journals publishing in English.

7 Conclusion

We conducted a systematic mapping study to investigate the state of research on modeling languages for Industry 4.0. The

study revealed that digital representation of cyber-physical production systems, i.e., their interfaces and data models, and their integration and (re-)configuration are the prime Industry 4.0 concerns addressed through modeling languages.

The number of papers explicating the authors' expectations of applying modeling languages to Industry 4.0 is rather small. There also are no papers or benchmarks that investigate evaluating the expected benefits through experiments. This is in line with uncovering a lack of experience papers and experience reports. Moreover, there appears little published research on metrics and benchmarks to test the explicated expectations. This might hint that research on modeling languages for Industry 4.0 is still focusing on foundational challenges and maturing the discipline could produce these necessary validations. This also is indicated by the high number of publications focusing on methods and solutions, instead of validation research. Where evaluation research is reported, it mostly focuses on case studies or lab-sized systems at universities possibly using industrial components. To fully investigate the benefits of modeling on Industry 4.0, more evaluation research in industrial settings is necessary.

It is also startling that—despite the huge costs that production system failures might entail—there is relatively little research on validation and verification. However, with Industry 4.0 being business-driven and aiming to reduce cost and time, such contributions might arise once the field has matured more. Recent trends indicate that validation and verification already are becoming a more important concern for the field.

We found that domain-specific languages and UML (including variants) are the modeling languages applied most often, followed by knowledge representation techniques. The use of metamodeling and DSLs, as well as UML profiles or other extension mechanisms, might suggest that specific challenges are not supported by current modeling languages. However, in the 23 publications reporting an industrial evaluation of their contributions, this assumption is not reflected. As expected, these publications report on applying more established modeling languages, such as AutomationML (cf. [40]), OWL (cf. [129,137,139]), or UML (cf. [119,129]). Nonetheless, even among the publications with industrial evaluation, we found contributions introducing novel metamodels (cf. [70]) or extensions of established ontologies (cf. [111]).

Also, with the majority 76% of contributions related to DSLs being published since 2014—and in an increasing number since then—we expect more research contributing modeling techniques specifically tailored to Industry 4.0 in the future. The significantly growing number of papers related to metamodeling and DSLs in the last year alone suggests that the community on modeling for Industry 4.0 invests increasing efforts in tailoring specific modeling tools. This matches the number of modeling standards and extensions to these standards as well. Most notably, Computer Aided Engineering Exchange (CAEX) [15] acts as a modeling language and metamodeling language which is used by AutomationML [41] and enables its extension with domain-specific concepts.

While integration still is a major challenge in Industry 4.0, there seem to be trends to shift research from a mostly digital representation of CPPS toward information management and process modeling. Moreover, research shifts away from applying established (e.g., UML-based) modeling techniques toward specific and tailored modeling techniques. The latter might be an effect of increasing adoption of modeling standards (such as ISO 10303 or IEC 62264) specific to Industry 4.0, which are worthwhile to investigate for everybody aiming to contribute to the field.

Especially modeling knowledge about processes operating on established digital representations seems to become increasingly important. To this end, integration of software engineering and knowledge representation (e.g., on the integration of ontologies and class diagrams or SPARQL and OCL) demands further research that supports its deployment in the Industry 4.0 field. Moreover, there also is less research on modeling for (smart) products of Industry 4.0 than expected.

Future work on investigating the contribution of modeling languages to Industry 4.0 should investigate details on the modeling techniques applied to Industry 4.0, such as their forms, integration, and usage. The dataset produced through our systematic mapping study enables this.

Acknowledgements This work has been partially supported by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and by the FWF in the Project LEA-xDSML under the Grant Number P 30525-N31.

References

1. Abramovici, M.: Future trends in product lifecycle management (PLM). In: The Future of Product Development, pp. 665–674. Springer, Berlin (2007)
2. Affonso, R.C., Cheutet, V., Ayadi, M., Haddar, M.: Simulation in product lifecycle: towards a better information management for design projects. J. Mod. Project Manag. **1**(1) (2013)
3. Agner, L.T.W., Soares, I.W., Stadzisz, P.C., SimãO, J.M.: A Brazilian survey on UML and model-driven practices for embedded software development. J. Syst. Softw. **86**(4), 997–1005 (2013)
4. Aigner, W., Miksch, S., Schumann, H., Tominski, C.: Visualization of Time-Oriented Data. Human–Computer Interaction Series. Springer, Berlin (2011)
5. Al-Fedaghi, S., Al-Shahin, F.: Control software modeling in production systems. Open Autom. Control Syst. J. **7**(1), 184–198 (2015)
6. Aleksić, D.S., Janković, D.S., Stoimenov, L.V.: A case study on the object-oriented framework for modeling product families with

the dominant variation of the topology in the one-of-a-kind production. Int. J. Adv. Manuf. Technol. **59**(1), 397–412 (2012)
7. Alenazi, M., Niu, N., Wang, W., Gupta, A.: Traceability for automated production systems: a position paper. In: 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), pp. 51–55. IEEE (2017)
8. Alexopoulos, K., Makris, S., Xanthakis, V., Sipsas, K., Chryssolouris, G.: A concept for context-aware computing in manufacturing: the white goods case. Int. J. Comput. Integr. Manuf. **29**(8), 839–849 (2016)
9. Back, M.G., Lee, D.K., Shin, J.G., Woo, J.H.: A study for production simulation model generation system based on data model at a shipyard. Int. J. Naval Archit. Ocean Eng. **8**(5), 496–510 (2016)
10. Backhaus, J., Reinhart, G.: Digital description of products, processes and resources for task-oriented programming of assembly systems. J. Intell. Manuf. **28**(8), 1787–1800 (2017)
11. Bailey, J., Budgen, D., Turner, M., Kitchenham, B., Brereton, P., Linkman, S.: Evidence relating to object-oriented software design: a survey. In: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM '07, pp. 482–484. IEEE Computer Society, Washington, DC, USA (2007)
12. Bareiß, P., Schütz, D., Priego, R., Marcos, M., Vogel-Heuser, B.: A model-based failure recovery approach for automated production systems combining sysml and industrial standards. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–7. IEEE (2016)
13. Beecham, S., Baddoo, N., Hall, T., Robinson, H., Sharp, H.: Motivation in Software Engineering: a systematic literature review. Inf. Softw. Technol. **50**(9–10), 860–878 (2008)
14. Berardinelli, L., Biffl, S., Lüder, A., Mätzler, E., Mayerhofer, T., Wimmer, M., Wolny, S.: Cross-disciplinary engineering with AutomationML and SysML. at-Automatisierungstechnik **64**(4), 253–269 (2016)
15. Berardinelli, L., Drath, R., Maetzler, E., Wimmer, M.: On the evolution of CAEX: a language engineering perspective. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8. IEEE (2016)
16. Bergert, M., Diedrich, C., Kiefer, J., Bar, T.: Automated PLC software generation based on standardized digital process information. In: IEEE Conference on Emerging Technologies and Factory Automation. ETFA, pp. 352–359. IEEE (2007)
17. Bergmann, S., Strassburger, S.: Challenges for the automatic generation of simulation models for production systems. In: Proceedings of the 2010 Summer Computer Simulation Conference, SCSC '10, pp. 545–549. Society for Computer Simulation International, San Diego, CA, USA (2010)
18. Bergmann, S., Straßburger, S.: On the use of the Core Manufacturing Simulation Data (CMSD) standard: experiences and recommendations. In: Fall Simulation Interoperability Workshop 2015 (SIW) (2015)
19. Berndt, O., von Lukas, U.F., Kuijper, A.: Functional modelling and simulation of overall system ship-virtual methods for engineering and commissioning in shipbuilding. In: ECMS, pp. 347–353 (2015)
20. Bigvand, P.G., Drath, R., Scholz, A., Schüller, A.: Agile standardization by means of PCE Requests. In: 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–8. IEEE (2015)
21. Boeker, M., Vach, W., Motschall, E.: Google Scholar as replacement for systematic literature searches: good relative recall and precision are not enough. BMC Med. Res. Methodol. **13**(1), 131 (2013)
22. Broy, M., Schmidt, A.: Challenges in engineering cyber-physical systems. Computer **47**(2), 70–72 (2014)
23. Bscher, C., Voet, H., Krunke, M., Burggrf, P., Meisen, T., Jeschke, S.: Semantic information modelling for factory planning projects. Procedia CIRP **41**, 478–483 (2016)
24. Budgen, D., Brereton, P.: Performing systematic literature reviews in software engineering. In: Proceedings of the 28th International Conference on Software Engineering, pp. 1051–1052. ACM (2006)
25. Budgen, D., Burn, A.J., Brereton, O.P., Kitchenham, B.A., Pretorius, R.: Empirical evidence about the UML: a systematic literature review. Softw. Pract. Exp. **41**(4), 363–392 (2011)
26. Budgen, D., Turner, M., Brereton, P., Kitchenham, B.: Using mapping studies in software engineering. In: Proceedings of PPIG, vol. 8, pp. 195–204. Lancaster University (2008)
27. Bundesministerium für Bildung und Forschung: Zukunftsprojekt Industrie 4.0. https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html. Accessed 20 Apr 2017
28. Cândea, G., Cândea, C., Radu, C., Terkaj, W., Sacco, M., Suciu, O.: A practical use of the Virtual Factory Framework. In: 14th International Conference on Modern Information Technology in the Innovation Process of the Industrial Enterprises, Budapest, Hungary (2012)
29. Chavarra-Barrientos, D., Batres, R., Wright, P.K., Molina, A.: A methodology to create a sensing, smart and sustainable manufacturing enterprise. Int. J. Prod. Res. **56**(1–2), 584–603 (2018)
30. Chen, D., Maffei, A., Ferreirar, J., Akillioglu, H., Khabazzi, M.R., Zhang, X.: A virtual environment for the management and development of cyber-physical manufacturing systems. IFAC-PapersOnLine **48**(7), 29–36 (2015)
31. Chen, D., Panfilenko, D.V., Khabbazi, M.R., Sonntag, D.: A model-based approach to qualified process automation for anomaly detection and treatment. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8. IEEE (2016)
32. Chen, L., Ali Babar, M., Ali, N.: Variability management in software product lines: a systematic review. In: Proceedings of the 13th International Software Product Line Conference, pp. 81–90. Carnegie Mellon University (2009)
33. Condori-Fernandez, N., Daneva, M., Sikkel, K., Wieringa, R., Dieste, O., Pastor, O.: A systematic mapping study on empirical evaluation of software requirements specifications techniques. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (2009)
34. Constantinescu, C., Matarazzo, D., Dienes, D., Francalanza, E., Bayer, M.: Modeling of system knowledge for efficient agile manufacturing: tool evaluation, selection and implementation scenario in SMEs. Procedia CIRP **25**, 246–252 (2014). 8th International Conference on Digital Enterprise Technology—DET 2014 Disruptive Innovation in Manufacturing Engineering towards the 4th Industrial Revolution
35. Constantinescu, C., Matarazzo, D., Dienes, D., Francalanza, E., Bayer, M.: Modeling of system knowledge for efficient agile manufacturing: tool evaluation, selection and implementation scenario in SMEs. Procedia CIRP **25**, 246–252 (2014)
36. Deane, P.M.: The First Industrial Revolution. Cambridge University Press, Cambridge (1979)
37. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 31–36. ACM (2007)
38. Díaz-Madroñero, M., Mula, J., Peidro, D.: A mathematical programming model for integrating production and procurement transport decisions. Appl. Math. Model. **52**, 527–543 (2017)

39. Divoux, T., Rondeau, E., Lepage, F.: Using the EXPRESS language as a reference interface to define MMS communication. J. Intell. Manuf. **8**(1), 59–66 (1997)
40. Dorofeev, K., Cheng, C.H., Guedes, M., Ferreira, P., Profanter, S., Zoitl, A.: Device adapter concept towards enabling plug&produce production environments. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8. IEEE (2017)
41. Drath, R., Luder, A., Peschke, J., Hundt, L.: AutomationML—the glue for seamless automation engineering. In: IEEE International Conference on Emerging Technologies and Factory Automation. ETFA 2008, pp. 616–623. IEEE (2008)
42. Dregger, J., Niehaus, J., Ittermann, P., Hirsch-Kreinsen, H., ten Hompel, M.: The digitization of manufacturing and its societal challenges: a framework for the future of industrial labor. In: 2016 IEEE International Symposium on Ethics in Engineering, Science and Technology (ETHICS), pp. 1–3 (2016)
43. Du, J., He, Q., Fan, X.: Automating generation of the assembly line models in aircraft manufacturing simulation. In: 2013 IEEE International Symposium on Assembly and Manufacturing (ISAM), , pp. 155–159. IEEE (2013)
44. do Nascimento, L.M., Viana, D.L., Neto, P.A.S., Martins, D.A., Garcia, V.C., Meira, S.R.: A systematic mapping study on domain-specific languages. In: The Seventh International Conference on Software Engineering Advances (ICSEA 2012), pp. 179–187 (2012)
45. Efendioglu, N., Woitsch, R.: A modelling method for digital service design and intellectual property management towards Industry 4.0: CAxMan case. In: International Conference on Serviceology, pp. 153–163. Springer, Berlin (2017)
46. Engström, E., Runeson, P.: Software product line testing—a systematic mapping study. Inf. Softw. Technol. **53**(1), 2–13 (2011)
47. Engström, E., Runeson, P.: Software product line testing-a systematic mapping study. Inf. Softw. Technol. **53**(1), 2–13 (2011)
48. Feldmann, S., Herzig, S.J., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C.J., Vogel-Heuser, B.: Towards effective management of inconsistencies in model-based engineering of automated production systems. IFAC-PapersOnLine **48**(3), 916–923 (2015)
49. Foradis, T., Thramboulidis, K.: From mechatronic components to industrial automation things—an IoT model for cyber-physical manufacturing systems. J. Softw. Eng. Appl. **10**(08), 734 (2017)
50. Francalanza, E., Borg, J., Constantinescu, C.: A knowledge-based tool for designing cyber physical production systems. Comput. Ind. **84**, 39–58 (2017)
51. García, J., Cabot, J.: Stepwise adoption of continuous delivery in model-driven engineering. In: DEVOPS (2018)
52. García, M.V., Irisarri, E., Pérez, F., Estévez, E., Marcos, M.: OPC-UA communications integration using a CPPS architecture. In: Ecuador Technical Chapters Meeting (ETCM), IEEE, vol. 1, pp. 1–6. IEEE (2016)
53. García-Borgoñon, L., Barcelona, M., García-García, J., Alba, M., Escalona, M.J.: Software process modeling languages: a systematic literature review. Inf. Softw. Technol. **56**(2), 103–116 (2014)
54. Gisbert, J.R., Palau, C., Uriarte, M., Prieto, G., Palazón, J.A., Esteve, M., López, O., Correas, J., Lucas-Estañ, M.C., Giménez, P., et al.: Integrated system for control and monitoring industrial wireless networks for labor risk prevention. J. Netw. Comput. Appl. **39**, 233–252 (2014)
55. Givehchi, O., Landsdorf, K., Simoens, P., Colombo, A.W.: Interoperability for industrial cyber-physical systems: an approach for legacy systems. IEEE Trans. Ind. Inform. **13**(6), 3370–3378 (2017)
56. Göring, M., Fay, A.: Automation systems—formal modeling of temporal change of physical structure. In: IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society, pp. 6160–6165. IEEE (2012)
57. Gregor, M., Hromada, J., Matuszek, J.: Digital Factory supported by simulation and metamodelling. Appl. Comput. Sci. **4**, 63–74 (2008)
58. Gruhn, V., Schäfer, C.: BizDevOps: because DevOps is not the end of the story. In: International Conference on Intelligent Software Methodologies, Tools, and Techniques, pp. 388–398. Springer, Berlin (2015)
59. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. **38**(6), 1276–1304 (2012)
60. Harcuba, O., Vrba, P.: Ontologies for flexible production systems. In: 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–8. IEEE (2015)
61. Hasan, B., Wikander, J.: A review on utilizing ontological approaches in integrating assembly design and assembly process planning. Int. J. Mech. Eng. (SSRG-IJME) **4**(11), 5–16 (2017)
62. Heineck, T., Gonçalves, E., Sousa, A., Oliveira, M., Castro, J.: Model-driven development in robotics domain: a systematic literature review. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 151–160. IEEE (2016)
63. Hermann, M., Pentek, T., Otto, B.: Design principles for Industrie 4.0 scenarios. In: 2016 49th Hawaii International Conference on System Sciences (HICSS), pp. 3928–3937. IEEE (2016)
64. Hermann, M., Pentek, T., Otto, B.: Design principles for Industrie 4.0 scenarios. In: Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS), HICSS '16, pp. 3928–3937. IEEE Computer Society, Washington, DC, USA (2016)
65. High Value Manufacturing Carapult. https://hvm.catapult.org.uk/. Accessed 5 June 2018
66. Hildebrandt, C., Glawe, M., Müller, A.W., Fay, A.: Reasoning on engineering knowledge: applications and desired features. In: European Semantic Web Conference, pp. 65–78. Springer, Berlin (2017)
67. Hoffmann, A., Angerer, A., Schierl, A., Vistein, M., Reif, W.: Service-oriented robotics manufacturing by reasoning about the scene graph of a robotics cell. In: Proceedings of ISR/Robotik 2014; 41st International Symposium on Robotics, pp. 1–8. VDE (2014)
68. Holz, D., Topalidou-Kyniazopoulou, A., Rovida, F., Pedersen, M.R., Krüger, V., Behnke, S.: A skill-based system for object perception and manipulation for automating kitting tasks. In: 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), , pp. 1–9. IEEE (2015)
69. Hummel, B., Braun, P.: Towards an integrated system model for testing and verification of automation machines. In: Proceedings of the 2008 International Workshop on Models in Software Engineering, MiSE '08, pp. 51–56. ACM, New York, NY, USA (2008)
70. Irisarri, E., García, M.V., Pérez, F., Estévez, E., Marcos, M.: A model-based approach for process monitoring in oil production industry. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–4. IEEE (2016)
71. Jorgensen, M., Shepperd, M.: A systematic review of software development cost estimation studies. IEEE Trans. Softw. Eng. **33**(1), 33–53 (2007)
72. Jung, K., Kulvatunyou, B., Choi, S., Brundage, M.P.: An overview of a smart manufacturing system readiness assessment. In: IFIP International Conference on Advances in Production Management Systems, pp. 705–712. Springer, Berlin (2016)

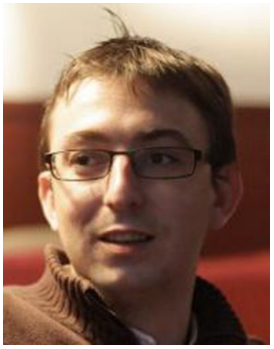
73. Juristo, N., Moreno, A.M., Vegas, S., Solari, M.: In search of what we experimentally know about unit testing. IEEE Softw. **23**(6), 72–80 (2006)
74. Kádár, B., Terkaj, W., Sacco, M.: Semantic Virtual Factory supporting interoperable modelling and evaluation of production systems. CIRP Ann. Manuf. Technol. **62**(1), 443–446 (2013)
75. Kannengiesser, U., Müller, H.: Towards agent-based smart factories: a subject-oriented modeling approach. In: 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), vol. 3, pp. 83–86. IEEE (2013)
76. Keele, S.: Guidelines for performing systematic literature reviews in software engineering, vol. 5. Technical report, Ver. 2.3 EBSE Technical Report (2007)
77. Kern, H., Stefan, F., Dimitrieski, V.: Intelligent and self-adapting integration between machines and information systems. IADIS Int. J. Comput. Sci. Inf. Syst. **10**(1), 47–63 (2015)
78. Khaleel, H., Conzon, D., Kasinathan, P., Brizzi, P., Pastrone, C., Pramudianto, F., Eisenhauer, M., Cultrona, P.A., Rusinà, F., Lukáč, G., et al.: Heterogeneous applications, tools, and methodologies in the car manufacturing industry through an IoT approach. IEEE Syst. J. **11**(3), 1412–1423 (2017)
79. Khan, A., Turowski, K.: A survey of current challenges in manufacturing industry and preparation for Industry 4.0. In: Proceedings of the First International Scientific Conference on Intelligent Information Technologies for Industry (IITI'16), pp. 15–26 (2016). https://doi.org/10.1007/978-3-319-33609-1_2
80. Kitchenham, B., Brereton, O.P., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic literature reviews in software engineering—a systematic literature review. Inf. Softw. Technol. **51**(1), 7–15 (2009). Special Section—Most Cited Articles in 2002 and Regular Research Papers
81. Kitchenham, B.A., Budgen, D., Brereton, O.P.: The value of mapping studies: a participant-observer case study. In: Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering, EASE'10, pp. 25–33. BCS Learning & Development Ltd., Swindon, UK (2010)
82. Kosar, T., Bohra, S., Mernik, M.: Domain-specific languages: a systematic mapping study. Inf. Softw. Technol. **71**, 77–91 (2016)
83. Kovalenko, O., Wimmer, M., Sabou, M., Lüder, A., Ekaputra, F.J., Biffl, S.: Modeling AutomationML: semantic web technologies vs. model-driven engineering. In: 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–4. IEEE (2015)
84. Korea-Manufacturing Technology-Smart Factory. https://www.export.gov/article?id=Korea-Manufacturing-Technology-Smart-Factory. Accessed 4 June 2018
85. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring. Sci. Comput. Program. **78**(8), 1010–1034 (2013)
86. Lahire, P., Parigot, D., Tundrea, E.: SMARTFACTORY—an implementation of the domain driven development approach. In: SACI2004, 1st Romanian-Hungarian Joint Symposium on Applied Computational Intelligence, p. 6 (2004)
87. Lee, E.A.: Cyber physical systems: design challenges. In: Proceedings of the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), pp. 363–369 (2008)
88. Liao, Y., Deschamps, F., de Freitas Rocha Loures, E., Ramos, L.F.P.: Past, present and future of Industry 4.0—a systematic literature review and research agenda proposal. Int. J. Prod. Res. **55**(12), 3609–3629 (2017). https://doi.org/10.1080/00207543.2017.1308576
89. Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Assessing the state-of-practice of model-based engineering in the embedded systems domain. In: International Conference on Model Driven Engineering Languages and Systems, pp. 166–182. Springer, Berlin (2014)
90. Long, F., Zeiler, P., Bertsche, B.: Potentials of coloured petri nets for realistic availability modelling of production systems in Industry 4.0. In: Proceedings of the ESREL 2015 Conference, 07.09.-10.09. 2015, Zürich, Switzerland, pp. 4455–4463 (2015)
91. Loskyll, M., Heck, I., Schlick, J., Schwarz, M.: Context-based orchestration for control of resource-efficient manufacturing processes. Future Internet **4**(3), 737–761 (2012)
92. Lu, Y.: Industry 4.0: a survey on technologies, applications and open research issues. J. Ind. Inf. Integr. **6**, 1–10 (2017)
93. Lütjen, M., Kreowski, H.J., Franke, M., Thoben, K.D., Freitag, M.: Model-driven logistics engineering-challenges of model and object transformation. Procedia Technol. **15**, 303–312 (2014)
94. Lütjen, M., Rippel, D.: GRAMOSA framework for graphical modelling and simulation-based analysis of complex production processes. Int. J. Adv. Manuf. Technol. **81**(1–4), 171–181 (2015)
95. Ma, Z., Hudic, A., Shaaban, A., Plosz, S.: Security viewpoint in a reference architecture model for cyber-physical production systems. In: 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pp. 153–159. IEEE (2017)
96. Made in China 2025. https://www.merics.org/sites/default/files/2017-09/MPOC_No.2_MadeinChina2025.pdf. Accessed 6 June 2018
97. Mahdavi-Hezavehi, S., Durelli, V.H., Weyns, D., Avgeriou, P.: A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems. Inf. Softw. Technol. **90**, 1–26 (2017)
98. Matei, M.M., Popescu, D.: Extend IT services in process control domain for onshore oilfields. In: 10th International Conference on Dynamical Systems and Control (CONTROL15), December, pp. 12–14 (2015)
99. Mätzler, S., Wollschlaeger, M.: Interchange format for the generation of functional elements for industrie 4.0 components. In: Industrial Electronics Society, IECON 2017-43rd Annual Conference of the IEEE, pp. 5453–5459. IEEE (2017)
100. Mazak, A., Huemer, C.: A standards framework for value networks in the context of Industry 4.0. In: 2015 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), pp. 1342–1346. IEEE (2015)
101. McMillan, A.J., Swindells, N., Archer, E., McIlhagger, A., Sung, A., Leong, K., Jones, R.: A review of composite product data interoperability and product life-cycle management challenges in the composites industry. Adv. Manuf. Polym. Compos. Sci. **3**(4), 130–147 (2017)
102. Mechs, S., Grimm, S., Beyer, D., Lamparter, S.: Evaluation of prediction accuracy for energy-efficient switching of automation facilities. In: Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE, pp. 6928–6933. IEEE (2013)
103. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)
104. Mehmood, A., Jawawi, D.N.: Aspect-oriented model-driven code generation: a systematic mapping study. Inf. Softw. Technol. **55**(2), 395–411 (2013). Special Section: Component-Based Software Engineering (CBSE) (2011)
105. Merkumians, M.M., Baierling, M., Schitter, G.: A service-oriented domain specific language programming approach for batch processes. In: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–9. IEEE (2016)
106. Michaloski, J., Proctor, F., Arinez, J., Berglund, J.: Toward the ideal of automating production optimization. In: ASME 2013 International Mechanical Engineering Congress and Exposition,

p. V02AT02A089. American Society of Mechanical Engineers (2013)
107. Miguel Gutierrez-Guerrero, J., Antonio Holgado-Terriza, J.: iMMAS an industrial meta-model for automation system using OPC UA. Elektronika ir Elektrotechnika **23**(3), 3–11 (2017)
108. Mokyr, J.: The second industrial revolution, 1870–1914. Storia delleconomia Mondiale, pp. 219–45 (1998)
109. Mosterman, P.J., Zander, J.: Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems. Softw. Syst. Model. **15**(1), 5–16 (2016)
110. Negri, E., Fumagalli, L., Garetti, M., Tanca, L.: Requirements and languages for the semantic representation of manufacturing systems. Comput. Ind. **81**, 55–66 (2016)
111. Negri, E., Perotti, S., Fumagalli, L., Marchet, G., Garetti, M.: Modelling internal logistics systems through ontologies. Comput. Ind. **88**, 19–34 (2017)
112. Niggemann, O., Maier, A., Jasperneite, J.: Model-based development of automation systems. In: MBEES, pp. 45–54 (2010)
113. Onori, M., Semere, D., Barata, J.: Evolvable assembly systems: from evaluation to application. In: Innovation in Manufacturing Networks, pp. 205–214. Springer, Berlin (2008)
114. Pedrazzoli, P., Alge, M., Bettoni, A., Canetta, L.: Modeling and simulation tool for sustainable MC supply chain design and assessment. In: IFIP International Conference on Advances in Production Management Systems, pp. 342–349. Springer, Berlin (2012)
115. Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M.: Systematic mapping studies in software engineering. EASE **8**, 68–77 (2008)
116. Petrasch, R., Hentschke, R.: Process modeling for Industry 4.0 applications: towards an Industry 4.0 process modeling language and method. In: 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 1–5. IEEE (2016)
117. Pfouga, A., Stjepandić, J.: Leveraging 3D geometric knowledge in the product lifecycle based on industrial standards. J. Comput. Des. Eng. **5**, 54–67 (2017)
118. Pisching, M.A., Junqueira, F., Filho, D.J.S., Miyagi, P.E.: Service on the Industry 4.0, pp. 65–72. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16766-4_7
119. Polacsek, T., Roussel, S., Bouissiere, F., Cuiller, C., Dereux, P.E., Kersuzan, S.: Towards thinking manufacturing and design together: an aeronautical case study. In: International Conference on Conceptual Modeling, pp. 340–353. Springer, Berlin (2017)
120. Pratt, M.J.: Introduction to ISO 10303 the STEP standard for product data exchange. J. Comput. Inf. Sci. Eng. **1**(1), 102–103 (2001)
121. Pretorius, R., Budgen, D.: A mapping study on empirical evidence related to the models and forms used in the UML. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '08, pp. 342–344. ACM, New York, NY, USA (2008)
122. Prévost, G., Blech, J.O., Foster, K., Schmidt, H.W.: An architecture for visualization of industrial automation data. In: ENASE, pp. 38–46 (2017)
123. Priego, R., Agirre, A., Estévez, E., Orive, D., Marcos, M.: Middleware-based support for reconfigurable mechatronic systems. IFAC-PapersOnLine **48**(10), 81–86 (2015)
124. Ranky, P.G., Lonkar, M., Chamyvelumani, S.: eTransition models of collaborating design and manufacturing enterprises. Int. J. Comput. Integr. Manuf. **16**(4–5), 255–266 (2003)
125. Rashid, M.A., Qureshi, H., Khan, N.: ERP life-cycle management for aerospace smart factory: a multi-disciplinary approach. Int. J. Comput. Appl. **26**(11), 55–62 (2011)
126. Ren, G., Hua, Q., Deng, P., Yang, C., Zhang, J.: A multi-perspective method for analysis of cooperative behaviors among industrial devices of smart factory. IEEE Access **5**, 10882–10891 (2017)
127. Runde, S., Wolf, G., Braun, M., Siemens, A.: EDDL and semantic web From field device integration (FDI) to Future Device Management (FDM). In: 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–8. IEEE (2013)
128. Sabou, M., Ekaputra, F., Kovalenko, O., Biffl, S.: Supporting the engineering of cyber-physical production systems with the AutomationML analyzer. In: 2016 1st International Workshop on Cyber-Physical Production Systems (CPPS), pp. 1–8. IEEE (2016)
129. Sadigh, B.L., Unver, H.O., Nikghadam, S., Dogdu, E., Ozbayoglu, A.M., Kilic, S.E.: An ontology-based multi-agent virtual enterprise system (OMAVE): part 1: domain modelling and rule management. Int. J. Comput. Integr. Manuf. **30**(2–3), 320–343 (2017)
130. Sadlauer, A., Hehenberger, P.: Using design languages in model-based mechatronic system design processes. Int. J. Agile Syst. Manag. **10**(1), 73–91 (2017)
131. Saraeian, S., Shirazi, B., Motameni, H.: Towards an extended BPMS prototype: open challenges of BPM to flexible and robust orchestrate of uncertain processes. Comput. Stand. Interfaces **57**, 1–9 (2017)
132. Sarigecili, M.I., Roy, U., Rachuri, S.: Enriching step product model with geometric dimension and tolerance information for one-dimensional tolerance analysis. J. Comput. Inf. Sci. Eng. **17**(2), 021004 (2017)
133. Schneider, M., Mittag, T., Gausemeier, J.: Modeling Language for Value Networks. In: 25th International Association for Management of Technology Conference Proceedings, 25th International Association for Management of Technology Conference, vol. 25, pp. 94–110. International Association for Management of Technology (IAMOT), IAMOT, Orlando, Florida (2016)
134. Schubert, D., Heinzemann, C., Gerking, C.: Towards safe execution of reconfigurations in cyber-physical systems. In: 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), pp. 33–38. IEEE (2016)
135. Sjoberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N.K., Rekdal, A.C.: A survey of controlled experiments in software engineering. IEEE Trans. Softw. Eng. **31**(9), 733–753 (2005)
136. Soares, A.L., Ferreira, J.P., Mendonça, J.: Organizational behaviour analysis and information technology fitness in manufacturing. In: Balanced Automation Systems, pp. 319–326. Springer, Berlin (1995)
137. Soylu, A., Kharlamov, E., Zheleznyakov, D., Jimenez-Ruiz, E., Giese, M., Skjæveland, M.G., Hovland, D., Schlatte, R., Brandt, S., Lie, H., et al.: Optiquevqs: a visual query system over ontologies for industry. Semantic Web (Preprint) 1–34 (2018)
138. Steimer, C., Fischer, J., Aurich, J.C.: Model-based design process for the early phases of manufacturing system planning using SysML. Procedia CIRP **60**, 163–168 (2017)
139. Steinegger, M., Melik-Merkumians, M., Zajc, J., Schitter, G.: A framework for automatic knowledge-based fault detection in industrial conveyor systems. In: 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–6. IEEE (2017)
140. Stemmler, S., Reiter, M., Abel, D.: Model predictive control as a module for autonomously running complex plastics production processes. Int. Polym. Sci. Technol. **41**(12), T1 (2014)
141. Strang, D., Anderl, R.: Assembly process driven component data model in cyber-physical production systems. In: Proceedings of the World Congress on Engineering and Computer Science, vol. 2 (2014)

142. Sungur, C.T., Breitenbücher, U., Leymann, F., Wieland, M.: Context-sensitive adaptive production processes. Procedia CIRP **41**, 147–152 (2016)
143. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Softw. Syst. Model. **15**(1), 31–69 (2016)
144. Takahashi, K., Ogata, Y., Nonaka, Y.: A proposal of unified reference model for smart manufacturing. In: 2017 13th IEEE Conference on Automation Science and Engineering (CASE), pp. 964–969. IEEE (2017)
145. The U.S. Advanced Manufacturing Initiative. https://www.nist.gov/sites/default/files/documents/2017/04/28/Molnar_091211.pdf. Accessed 6 June 2018
146. The Industrial Value Chain Initiative. https://iv-i.org/wp/en/about-us/whatsivi/. Accessed 4 June 2018
147. Thoma, A., Kormann, B., Vogel-Heuser, B.: Fault-centric system modeling using SysML for reliability testing. In: 2012 IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–8. IEEE (2012)
148. Thomalla, C.S.: Interoperability in manufacturing by semantic integration. In: 2011 International Conference on System Science, Engineering Design and Manufacturing Informatization (ICSEM), vol. 2, pp. 146–149. IEEE (2011)
149. Thramboulidis, K., Christoulakis, F.: UML4IoT—a UML-based approach to exploit IoT in cyber-physical manufacturing systems. Comput. Ind. **82**, 259–272 (2016)
150. Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., Reggio, G.: Relevance, benefits, and problems of software modelling and model driven techniques—a survey in the Italian industry. J. Syst. Softw. **86**(8), 2110–2126 (2013)
151. Trappey, A.J.C., Trappey, C.V., Govindarajan, U.H., Sun, J.J., Chuang, A.C.: A review of technology standards and patent portfolios for enabling cyber-physical systems in advanced manufacturing. IEEE Access **4**, 7356–7382 (2016)
152. Van Stein, B., Van Leeuwen, M., Wang, H., Purr, S., Kreissl, S., Meinhardt, J., Bäck, T.: Towards data driven process control in manufacturing car body parts. In: 2016 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 459–462. IEEE (2016)
153. Vangheluwe, H., Amaral, V., Giese, H., Broenink, J., Schätz, B., Norta, A., Carreira, P., Lukovic, I., Mayerhofer, T., Wimmer, M., Vallecillo, A.: MPM4CPS: Multi-paradigm modelling for cyber-physical systems. In: Proceedings of the Project Showcase @ STAF 2015, pp. 1–10 (2016)
154. Vogel-Heuser, B., Hess, D.: Guest Editorial Industry 4.0—prerequisites and visions. IEEE Trans. Autom. Sci. Eng. **13**(2), 411–413 (2016)
155. Vogel-Heuser, B., Rösch, S., Fischer, J., Simon, T., Ulewicz, S., Folmer, J., et al.: Fault handling in PLC-based Industry 4.0 automated production systems as a basis for restart and self-configuration and its evaluation. J. Softw. Eng. Appl. **9**(1), 1 (2016)
156. Walch, M.: Knowledge-driven enrichment of cyber-physical systems for industrial applications using the KbR modelling approach. In: 2017 IEEE International Conference on Agents (ICA), pp. 84–89. IEEE (2017)
157. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B.: Experimentation in Software Engineering. Springer, Berlin (2012)
158. Wollert, J., Lehne, M.: Modeling for ship design and production. In: 1991 Ship Production Symposium Proceedings: Building the Ships and Boats of 2010-The Way Forward, p. 1 (1991)
159. Wortmann, A., Combemale, B., Barais, O.: A systematic mapping study on modeling for Industry 4.0. In: Conference on Model Driven Engineering Languages and Systems (MODELS'17), pp. 281–291. IEEE (2017)
160. Xu, X.: Realization of STEP-NC enabled machining. Robot. Comput. Integr. Manuf. **22**(2), 144–153 (2006)
161. Zadeh, N.S., Lindberg, L., El-Khoury, J., Sivard, G.: Service oriented integration of distributed heterogeneous IT systems in production engineering using information standards and linked data. Model. Simul. Eng. **2017**, 9814179 (2017)
162. Zhang, C., Budgen, D.: What do we know about the effectiveness of software design patterns? IEEE Trans. Softw. Eng. **38**(5), 1213–1231 (2012)
163. Zhang, Q., Liu, Y., Zhang, Z.: A new method for automatic optimization of drawbead geometry in the sheet metal forming process based on an iterative learning control model. Int. J. Adv. Manuf. Technol. **88**, 1845–1861 (2016)
164. Zhao, W.B., Park, Y.H., Lee, H.Y., Jun, C.M., Do Noh, S.: Design and implementation of a PLM system for sustainable manufacturing. In: IFIP International Conference on Product Lifecycle Management, pp. 202–212. Springer, Berlin (2012)
165. Zhiwei, X., Yongxian, L.: Mechanical production line simulation and optimization analysis. In: 2008 IEEE International Conference on Automation and Logistics, pp. 2925–2930 (2008). https://doi.org/10.1109/ICAL.2008.4636677

Andreas Wortmann received his Ph.D. in Computer Science from RWTH Aachen University in 2016, where he is a senior researcher and leads a group on model-driven systems engineering. He is a member of the IEEE Technical Committee on Software Engineering for Robotics and Automation and a board member of the European Association for Programming Languages and Systems. His research interests include model-driven engineering, systems engineering, software language engineering, and robotics. For more information, please visit http://www.se-rwth.de/~wortmann/.

Olivier Barais is an associate professor at the University of Rennes 1, where he is a member of the Triskell INRIA research team. He received an engineering degree from the Ecole des Mines de Douai, France, in 2002 and a Ph.D. in computer science from the University of Lille 1, France, in 2005, in the Jacquard INRIA research team. His research interests include component-based software design, model-driven engineering and aspect-oriented modeling. Olivier Barais has coauthored 8 journal articles and 35 international conference papers, in conferences and journals such as Sosym, IEEE Computer, ICSE, MoDELS, SPLC and CBSE.



Benoit Combemale is full professor of Software Engineering at the University of Toulouse, and a Research Scientist at Inria. His research interests are in the field of software engineering, including model-driven engineering, software language engineering and validation and verification; mostly in the context of (smart) cyber-physical systems and Internet of Things. He is also teaching worldwide in various engineering schools and universities. More information at http://combemale.fr.

Manuel Wimmer is full professor leading the Institute of Business Informatics—Software Engineering at the Johannes Kepler University Linz, and he is the head of the Christian Doppler Laboratory CDL-MINT. His research interests comprise foundations of model engineering techniques as well as their application in domains such as tool interoperability, legacy modeling tool modernization, model versioning and evolution, and industrial engineering. For more information, please visit https://www.se.jku.at/manuel-wimmer/.

Modeling Language Engineering

This section contains the publications summarized in Chapter 3.

Paper 2	A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. 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, 2018. Reference: [BEK+18a]
Paper 3	A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann. Systematic Composition of Independent Language Features, In: Rafael Capilla Sevilla, Lidia Fuentes, Malte Lochau, editors, Journal of Systems and Software, 152, pages 50-69, Elsevier, 2019. Reference: [BEK+19]
Paper 4	A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann. Modeling Language Variability with Reusable Language Components, In: International Conference on Systems and Software Product Line (SPLC'18), September, ACM, 2018. Reference: [BEK+18b]
Paper 5	A. Wortmann. Towards Component-Based Development of Textual Domain-Specific Languages, In: Luigi Lavazza, Herwig Mannaert, Krishna Kavi, editors, International Conference on Software Engineering Advances (ICSEA 2019), pages 68-73, IARIA XPS Press, 2019. Reference: [Wor19]
Paper 6	A. Butting, J. Pfeiffer, B. Rumpe, and A. 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, 2020. Reference: [BPRW20]

[BEK+18] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. 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), pg. 75-82. ACM, Feb. 2018.
www.se-rwth.de/publications

Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features

Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
<lastname>@se-rwth.de

ABSTRACT

"Software languages are software too", hence their creation, evolution, and maintenance is subject to the same challenges. Managing multiple stand-alone variants of similar DSLs raises the related maintenance and evolution efforts for the languages and their associated tooling (analyses, transformations, editors, *etc.*) to a higher power. Software variability management techniques can help to harness this complexity. Research in software language variability focuses on metamodels and consequently mainly supports managing the variability of abstract syntaxes, omitting concrete syntax variability management. We present an approach to manage controlled syntactic variability of extensible software language product lines through identification of dedicated syntax variation points and specification of variants from independently developed features. This fosters software language reuse and reduces creation, maintenance, and evolution efforts. The approach is realized with the MontiCore language workbench and evaluated through a case study on architecture description languages. It facilitates creating, maintaining, and evolving the concrete and abstract syntax of families of languages and, hence, reduces the effort of software language engineering.

CCS CONCEPTS

• **Software and its engineering** → *Source code generation*; *Domain specific languages*; *Software product lines*;

KEYWORDS

Language Variability, Language Product Lines, Software Language Engineering

ACM Reference Format:
Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann. 2018. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3168365.3168368

VAMOS 2018, February 7–9, 2018, Madrid, Spain

ACM ISBN 978-1-4503-5398-4/18/02…$15.00
https://doi.org/10.1145/3168365.3168368

1 INTRODUCTION

Model-driven development (MDD) leverages (domain-specific) modeling languages (DSMLs) to reduce the conceptual gap between problem domains and the solution domain of software engineering [8]. Hence, efficient engineering, customization, and reuse of DSMLs has become a prime concern in MDD characterized as software language engineering (SLE) [13]. But "software languages are software too" [7] and as such are subject to the same challenges regarding creation, evolution, and maintenance. Consequently, SLE has produced a multitude of solutions to create languages based on metamodels or grammars, interpreters or generators, well-formedness rules in metalanguages or programming languages. Metamodels encode the abstract syntax (*i.e.,* structure) of languages as classes and their associations without providing means to instantiate models. Grammars also describe the structure of a language, but can support integrated definition of concrete syntax as well [9]. From these, model processing infrastructure to translate textual models into abstract syntax instances can be derived automatically, which greatly facilitates the efficient usage of DSMLs.

Research and industry have contributed a wealth of different DSMLs for different application domains and scenarios. A study [18] on architecture description languages (ADLs), for instance, discovered over 120 different ADLs for various domains. This requires creating, evolving, and maintaining independent languages and tooling for each of these individually. Research in software product lines (SPLs) has produced means to capture and manage variability of similar software in product lines. From these, different products can be derived through selection of features. Leveraging SPLs to DSMLs can facilitate engineering and maintaining product lines of similar languages. Research to DSML SPLs focuses on metamodels [22, 29] or grammars [16]. Both requires to maintain variability of the semantic mapping separately. Moreover, approaches to DSML variability either are restricted to fixed 150% models, where every possible feature must be known a priori [29], or support arbitrary language feature extension, which allows to break the structure imposed by feature diagrams easily.

We present a concept of controlled language variability that facilitates a posteriori extensibility with additional features, considers concrete syntax, and enables (re-)using languages as features without explicitly foreseeing this usage at language design time. It is realized through a family of integrated MontiCore [9] DSMLs and a composition mechanism based on well-defined language extension points. The realization builds upon existing language composition techniques of MontiCore and enables *controlled* language composition to support, *e.g.,* validation on product line level. The individual languages are independent of each other, which enables these to be developed by different language engineers.

The contributions of this paper, hence, are:

- A concept of controlled, extensible language product lines (LPLs) leveraging composition of independent language features that can be composed post hoc without invasion.
- Its realization with the MontiCore language workbench.
- A case study in architecture description languages.

To this end, Section 2 introduces our running example, before Section 3 introduces preliminaries. Afterwards, Section 4 presents our problem space concept for managing DSML variability and Section 5 presents its solution space implementation. Section 6 describes the application of our approach to managing features of an ADL. Section 7 highlights related work and Section 8 discusses observations.

2 EXAMPLE

Consider developing software architectures for different domains. To prevent the efforts of creating, maintaining, and evolving multiple stand-alone ADL variants tailored to the specific domains, language engineering starts with a core ADL with specific extension points and independently developed language components that provide modeling elements required for architectures of the different domains. A feature model associates the different language features (LFs) with extension points of the core ADL and of other features. Based on a feature configuration, the language components are combined such that an integrated ADL is created that allows the different domain experts to use precisely the modeling elements required. This enables a separation of concerns where language engineers develop LFs independent of each other. The arrangement of these features to a feature diagram is performed by a LPL manager. A language product manager is a domain expert who selects all features of a LPL that are relevant to the domain to generate language-processing tooling. A modeler then uses such a tool to implement models that conform to this language.

An example of a compact LPL for ADLs used for cloud systems and embedded systems is depicted in Figure 1. Based on independent language components (depicted right) with explicit extension points provided by different language engineers, the LPL manager defines the feature model governing, which features are available and how these relate (depicted left). Besides a common base feature, the LPL described by the feature model includes features typical to ADLs for embedded systems (such as automated connection of ports based on their types or names or component behavior models) as well features related to scalable and secure cloud systems (such as replicating components and encrypted communication). Each feature contains a language component that may yield further extension points. The relation between two features defines how their language components can be integrated. This decoupling enables reusing the language components in different feature models. Based on a feature configuration defined by the product manager (middle left), a software tool establishes the connections between the selected features' language components. Based on the selection of features, their respective language components are integrated into the extension points of the language components of the respective feature's parent. For instance, the language component of feature `InputOutputAutomata` is integrated into the

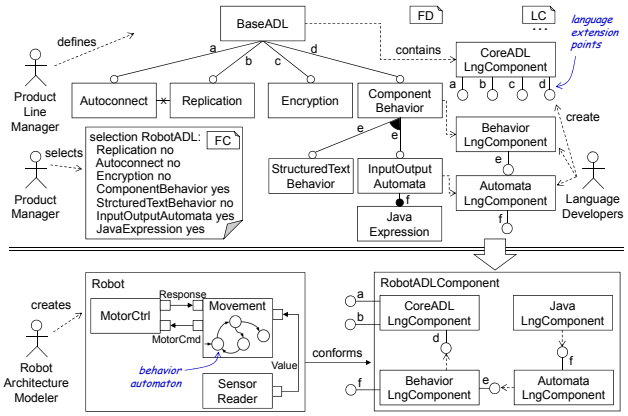


Figure 1: A LPL defined as feature model over language components. Given a feature configuration (top), the variant is transformed into a new language component (bottom).

extension point *e* of the language component contained in the feature `ComponentBehavior`. After integrating all referenced language components of the selected features, a new language component is generated, which can be used by the respective domain experts to model corresponding software architectures using the modeling elements selected through the feature configuration (in this case automata models describing component behavior).

Being able to reuse language components without modification enables to reuse the associated tooling (analyses, transformations) with the generated language component as well. Changes to a language component and its tooling are immediately available in the generated language modules as well. Both reduces the effort in creating, maintaining, and evolving modeling languages. The loose coupling between features and language components also enables to easily integrate new features into the feature diagram – integrating a new feature below `ComponentBehavior`, for instance, does not influence other features and language components. As the resulting language component can yield extension points again, the creation of intermediate products that require further refinement also is supported. Where multiple similar domains are addressed, creating refined domain-specific LPLs enables restricting a large base LPL accordingly.

3 PRELIMINARIES

While our concept for syntactical language variability can be applied to metamodel-based languages (*e.g.,* via abstract metamodel classes) as well, its realization is based upon the MontiCore language workbench [9]. MontiCore employs extended context-free grammars (CFGs) supporting integrated definition of concrete and abstract syntax [9] of DSMLs. From a DSML's CFG, MontiCore generates its abstract syntax tree (AST) classes and parsers that translate textual models into AST instances. To validate well-formedness constraints not expressible with CFGs, MontiCore features compositional *context conditions* (CoCos). Template-based code generators realize the DSMLs' semantics. MontiCore also supports compositional DSML integration via inheritance, embedding, and

aggregation [9]. Inheritance enables DSMLs to extend and override productions of their (possibly multiple) parent DSMLs. From inheriting DSMLs, MontiCore produces refined AST classes that inherit from the AST classes of the overridden productions. MontiCore also features *interface* productions, which enable underspecification in grammars that can be leveraged through inheritance to contribute new productions at well-defined extension points as depicted in Figure 2.

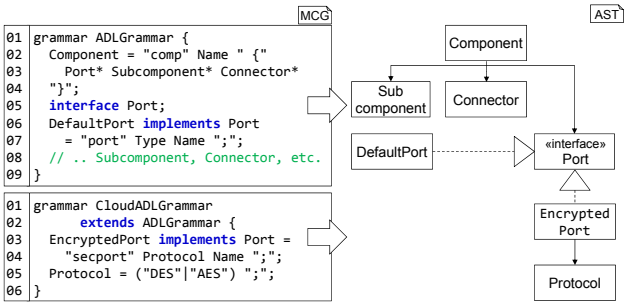


Figure 2: Example: Grammar Inheritance in MontiCore.

Here, the grammar `ADLGrammar` (top left) describes the quintessential elements of an ADL [19], *i.e.,* components that yield interfaces of typed ports and subcomponents that exchange messages through connectors between their ports (ll. 2-4). The production `Port` (l. 5) is an interface production that does not describe concrete or abstract syntax. Instead, it acts as extension point, which can be used in the defining language, such as the production `DefaultPort` (ll. 6-7), which consists of a data type and a name. From this, MontiCore generates five AST classes (depicted top right), out of which `Port` is an interface implemented by the AST class `DefaultPort`.

Interface productions can also be used in inheriting grammars, such as illustrated with grammar `CloudADLGrammar` depicted bottom. The grammar extends `ADLGrammar` (l. 2) and defines another implementation of `Port` that features security properties and omits port data types (ll. 3-4). Accordingly, MontiCore generates the two AST classes `EncryptedPort` and `Protocol`. Through this, `CloudADLGrammar` can reuse all modeling elements of the extended grammar and introduce new ones where foreseen by the developers of `ADLGrammar`.

To leverage interface productions as extension points for DSML features, we specify their use and relations through feature models [12]. We use a textual representation of feature trees, with the usual relations (mandatory, optional) between parent- and sub features and feature groups (alternative, exclusive). Besides this, the possible feature configurations can be restricted via cross-tree constraints (requires, excludes). For visualization purposes, we sometimes provide the graphical representation.

4 MODELING DSML VARIABILITY

Generally, languages are characterized as "the set of sentences" [13] that constitute the language, which also applies to modeling languages. With this definition being hardly accessible to investigation, a common refinement [4] is that DSMLs comprise (1) a concrete syntax (its sentences); (2) a minimal abstract syntax (structuring its sentences); (3) a semantic domain (typically a well-defined mathematical theory); and (4) a semantic mapping (giving meaning to the abstract syntax by mapping it to the semantic domain).

Software languages can be constructed from a variety of different constituents. Abstract syntaxes can be defined through metamodels (*cf.* EMF's Ecore [24], MPS [28]) or grammars (*cf.* Neverlang [25], Xtext [1]). Concrete syntaxes can be implemented through parsing textual models [14] or graphical editors [27]. Semantic mappings can be implemented through interpretation [5] or code generation [1]. Depending on the selected language constituents, various different forms of language composition are possible as well [6]. To manage variability and to explain composition, the definition above needs refinement. In the following, we define *language components* to comprise of (1) a grammar defining its abstract syntax (AS) and concrete syntax (CS) in an integrated fashion, (2) a (possibly empty) set of dedicated interface productions acting as abstract syntax extension points, and (3) a set of well-formedness rules. The dedicated set of abstract syntax extension points is foreseen by the language engineer to enable extending the language with new capabilities. The language itself might provide (default) implementations for its extension points. This, for instance, enables to define a Statechart language with extension points for guard transition expressions that already yields built-in expression but is open for future extension as well. The notion of AS extension points also applies to metamodel-based AS definitions, where these can be realized similarly.

As language components are unaware of being used with features, the composition of LFs and their composition enables reusing independently developed language components in different feature models. This facilitates extending the feature model post-hoc with new features, and prevents uncontrolled composition (*cf.* [25]). The composition operator we use for composing language components prevents invalidation of feature models through adding additional features. This property, called *conservative extension*, holds because it is impossible to remove syntactical language concepts by adding new features. The property holds only for the language's syntax and guarantees tooling stability, but cannot guarantee semantical correctness of analyses. The next section presents a realization of language components, LFs, and the composition operator based on MontiCore.

Our variability concept aims at enabling a well-defined integration of language components through definition of LPLs over features using these components. To this end, we describe language variability as trees of LFs supporting the usual relations [12] between feature diagram elements. Each feature contains a reference to its parent feature, a language component, and a set of mappings. Each of these mappings relates a production of its grammar to an interface production of the grammar contained by its parent feature (relative to a specific feature model). The concept also allows to refine an extension point with another extension point, *i.e.,* interface production. For the top-level feature, the set of mappings is empty. Through the relation to a parent feature, LFs are specific to the feature model they are used in, which also governs the restrictions between LFs (such as being optional, mandatory, exclusive, *etc.*). In particular, this enables reusing language components in different features and with different extension points. We reuse the *requires* relationship between two features to denote that a feature

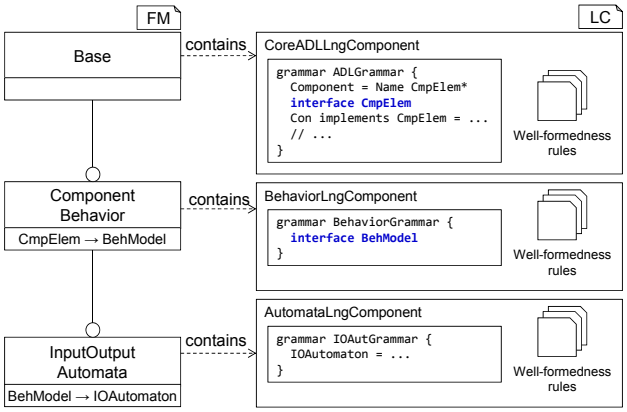


Figure 3: An illustration of LFs based on language components comprising grammars with dedicated extension points.

relies on the presence of another feature, which, *e.g.,* reflects in an inheritance relationship between the grammars of these features.

Individual LFs are not required to implement all extension points of their parent feature, which enables to refine LPLs by fixing the binding of some extension points and leaving other to be bound through features. Moreover, this enables using any suitable language as base language without modifying the language itself. Creating a LPL requires identifying and arranging LFs in a feature model based on their extension points. Language product managers then can derive language products from the LPL according to specific requirements (*e.g.,* to derive a robot ADL). How language components are related to LFs is depicted in Figure 3, which highlights the contents of some of the features depicted in Figure 1.

Here, the `Base` LF contains the `ADLGrammar` grammar, which describes quintessential elements of a component & connector ADL. This includes the extension point `CmpElem`, which is realized as an interface production of `ADLGrammar` and for which the `ADLGrammar` might provide its own (default) implementations. The feature `ComponentBehavior` contains the `BehaviorGrammar` comprising an interface `BehModel` that refines the extension point `CmpElem`. The grammar of the feature `InputOutputAutomata` describe the syntax for input output automata, and the feature maps these to the interface production `BehModel` of its parent feature. Each feature can contribute well-formedness rules operating on its abstract syntax: for instance, the `AutomataLngComponent` might yield a well-formedness ensuring that names of automaton states are unique. Based on the selected feature configuration, the features are composed to ultimately produce an integrated language component that enables the modeler to describe precisely what she needs to express without facing the accidental complexities of superfluous modeling elements. This composition is directed, as features lower in the feature models' hierarchy implement extension points of their parent features.

Language features are composed pairwise one after another, such that ultimately, a single, composed LF remains. The order in which sibling features are composed has no influence on the overall result. All context conditions are contained in the composed feature, and a joined grammar is generated. The composed LF retains all extension points.

To this effect, the composition is monotonically increasing in number of the extension points as features cannot remove extension points of their parents' grammars. The notion of a *mandatory* extension point only reflects in the feature model, but not in a single LF. Moreover, if grammar extension is allowed in the implementation, the approach only permits the grammars of a LPL to inherit from another if this is indicated as a requires relation between the features. As the composition relies on grammar extension, the newly added inheritance relations might interfere with existing ones and create a circular inheritance relation. To overcome this, we check the validity of a feature selection also with regard to potential circular inheritance relationships. The next section presents a realization of language components, language features, and composition based on the MontiCore languages.

5 INTEGRATING DSML SYNTAXES

With MontiCore [9], languages are defined in terms of extended context-free grammars (CFGs) that integrate concrete syntax with abstract syntax and use well-formedness rules implemented in Java, called context conditions (CoCos), to add restrictions not expressible through CFGs. Interface productions describe grammar extension points for which the grammar itself may provide (default) implementations (*cf.* Figure 2). Our solution space variability mechanisms, hence, are based on composing language components provided as MontiCore CFGs and context conditions.

Language components are developed independent of how they are used within a LPL. Nonetheless, language engineers have to foresee potential extension points in terms of interface productions. In our realization of the concept, language components are defined by a MontiCore CFG (MCG) and a list of context conditions. The approach generally permits the MCG of a language component to extend another MCG, unless this grammar is also part of the LPL. MontiCore supports dedicated interface productions that do not have a right-hand side as illustrated in l. 5 of Figure 2. Interfaces can be used in other productions (*cf.* `Component` in ll. 2ff). In another production, which is not necessarily in the same grammar model, interfaces can be implemented by other productions (*cf.* ll. 6f). The resulting structure of the AS is depicted on the right of Figure 2.

A *language component* encapsulates all artifacts of the definition of a single language into one dedicated artifact by holding explicit links to the grammar and a set of links to CoCo classes. Figure 4 depicts an exemplary language component `ADLLC` that references the ADL grammar (see Figure 2) and two CoCos. The *exports* keyword starts a list of explicit extension points via the names of the respective MontiCore interface productions. As the implementation exploits MontiCore interface productions as extension points, it might be that several interface productions are exposed as extension points undesiredly. To overcome this, we require language engineers to explicate all language extension points in the language component.

We separate LFs from language components to foster decoupled development of a language component and its context in a LPL. A LF defines (1) the language component it is based on, (2) the parent feature in the feature tree, and (3) a binding of grammar

```
01 language ADLLC {                                   LC
02   grammar com.ma.ADLGrammar;
03   cocos {
04     com.ma.cocos.CompNameLowerCase,
05     com.ma.cocos.PortNamesUnique
06   }
07   exports { CmpElem }
08 }
```

Figure 4: The language feature `ADLLC`.

```
01 feature BehaviorLF {                               LF
02   parent ADLLF;
03   language BehaviorLngComponent;
04   bindings { CmpElem -> BehModel; }
05 }
```

```
01 feature ADLLF {                                    LF
02   // no parent
03   language ADLLC;
04   // no bindings
05 }
```

Figure 5: The LFs `BehaviorLF` and `ADLLF`.

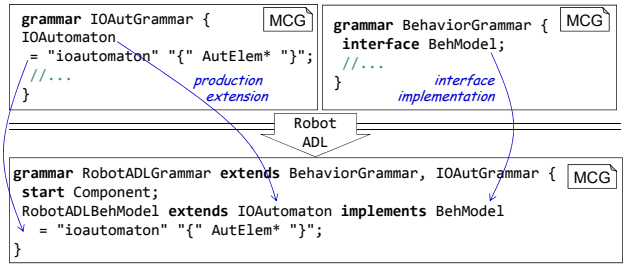


Figure 6: The composition (bottom) of two grammars (top).

productions to extension points of the parent feature. Language features are the building blocks of the feature model of a LPL. Each LPL requires a root feature, which has an empty parent feature and must not have bindings. For all other features, we require the parent to be present. If any bindings are present, the composition operator realizes language embedding (*cf.* Section 2). Otherwise, if no bindings are present, it realizes language aggregation.

Figure 5 depicts two LF models. `BehaviorLF` (left) references `ADLLF` as parent feature (l. 2) and `BehaviorLngComponent` as the implementing language component (l. 3). The binding relates the extension point `CmpElem` of the parent feature with the `BehModel` extension point of the grammar of the `BehaviorLngComponent` (l. 4). `ADLLF` (right) neither yields a parent feature, nor bindings.

The features are arranged in a feature model. We reuse a language and tooling for textual feature diagrams and selected variants (*i.e.,* feature configurations) implemented with MontiCore. With the LPL defined by the LPL manager at hand, a product manager can define a language variant by selecting a set of features. If the selection is valid with regard to the feature model and with regard to the approach's constraints, a new, composed language component is generated. From this, MontiCore generates language processing infrastructure such as a parser and an AST data structure on a push-button basis and iteratively composes two features from the leaves of the feature tree to the root as explained in Section 4. Figure 6 (top) depicts the grammars of the two LFs `BehaviorLF` and `InputOutputAutomata`. The bottom of Figure 6 depicts the grammar that results from the composition of these features, with

```
01 language RobotADLLC {                              LC
02   grammar com.ma.RobotADLGrammar;
03   cocos {
04     com.behavior.SingleBehaviorModel,
05     com.ioaut.SingleInitialState,
06     //...
07   }
08   exports { CmpElem, BehElem}
09 }
```

Figure 7: Language component of the composed LFs.

the binding applied. The name of the generated grammar is derived, with the feature configuration name as prefix before the name of the grammar of the root feature. The resulting grammar uses MontiCore's inheritance mechanism to extend both grammars. For technical reasons, the generated grammar has to reference the start production of the grammar of the root feature, to be used as top-level element for the generated parser. Further, the generated grammar comprises a new, generated production for every binding that has been applied. The left-hand side of the generated production is a derived non-terminal name, that states that the production extends the production of the extension and that it implements the interface production of the extension point. The effect of implementing an interface production has been explained in Section 3, the effect of extending another production is that the production can be applied wherever the extended production can be applied. Additionally, the generated abstract syntax class extends the generated abstract syntax class of the extended production. This has the advantage that all algorithms and tooling that are applicable to the extended AS element can be applied to the new one. The right-hand side of the generated production usually equals the right-hand side of the extended production. An exception is that if the right-hand side of a generated production contains a non-terminal symbol that has been bound as part of mapping. This is replaced with the left-hand side of the production generated from this mapping.

If an extension point is refined, both the extension and the extension point are realized as interface productions. In this case, a new interface (with a derived name) is generated that extends both interfaces. MontiCore generates parsers such that they are capable of parsing, despite this ambiguity in the concrete syntax.

Figure 7 depicts the language component resulting from the compositions of the two language features `InputOutputAutomata` and `BehaviorLF`. The referenced grammar is the grammar depicted at the bottom of Figure 6, which combines the abstract syntaxes and concrete syntaxes of the composed grammars. The context conditions of both features are joined. As the context conditions are checked against certain AS classes (and therefore, also their subclasses), all context conditions can be applied to the generated classes. It also comprises all exported extension points of the individual language components. Generally, the composition of two features f and g, where g is the parent of f, results in a composed LF with the parent feature of g, the composed language component, and the bindings of f.

6 EXTENDED EXAMPLE

This section demonstrates the application in context of an ADL to adapt it towards different domains. Developing and maintaining

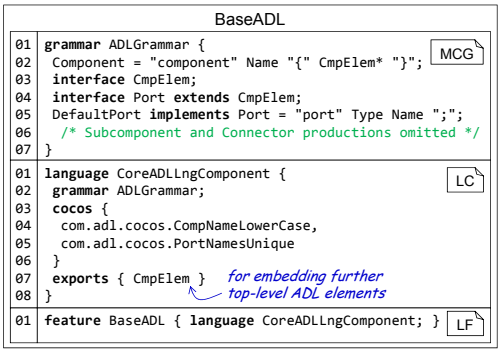


Figure 8: Parts of the `BaseADL` language feature.

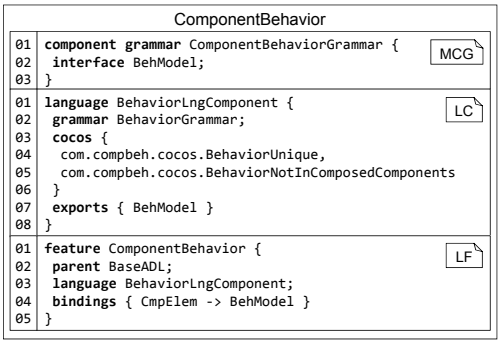


Figure 9: The `ComponentBehavior` language feature.

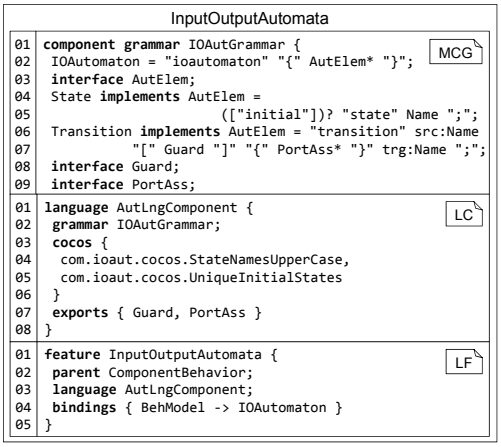


Figure 10: The `InputOutputAutomata` language feature.

domain-specific variants of an ADL is challenging [2]. To this effect, we applied the approach presented in this paper to derive domain-specific ADL variants from a domain-agnostic BaseADL. Figure 1 in Section 2 depicts the language feature model representing all possible ADL variants. The root feature `BaseADL` contains the basic elements such as components, ports, and connectors that are common to each variant. The `Autoconnect` feature adds syntax and transformations to realize an automatic connection of ports with either identical names or types. The feature `Encryption` enables to describe secure ports (`SecurePort`) and encrypted connections (`EncryptedConnector`) between them. The `Replication` feature provides elements for modeling systems where components are capable of replicating themselves when needed. This is useful in client-server architectures, for instance, where a client component is replicated on each request. The LPL manager considers component replication to be a threat for autoconnecting ports. Choosing one of the two corresponding features thus excludes the other. The `ComponentBehavior` feature introduces *behavior-blocks* to the ADL. Component behavior models are intended to be modeled in such blocks, only. The subfeatures `StructuredTextBehavior` and `InputOutputAutomata` contain different behavior languages. As it should not be possible to model empty behavior blocks, choosing the `ComponentBehavior` feature requires to choose at least one feature that defines a component behavior language. Automata use expressions on their transitions as guard conditions. For this purpose, the LPL currently only includes `JavaExpressions`, which are therefore marked mandatory.

Consider a product manager who aims at developing static software architectures where atomic components' behavior can be specified via input/output automata. She thus selects the configuration containing the following features `BaseADL`, `ComponentBehavior`, `InputOutputAutomata`, and `JavaExpression`. Parts of the configuration's constituent are depicted in Figures 8-11. The `BaseADL` LF (cf. Figure 8) neither has a parent feature nor defines a binding as it is the LPL's root feature. The feature's grammar defines language elements common to all ADL variants such as components, connector, and ports. The language component further defines two context conditions and exports the interface `CmpElem`. With this, it is possible to extend component definitions with further top level elements through LF composition. The `Subcomponent` and `Connector` productions of the `BaseADL` grammar are omitted. The grammar of LF `ComponentBehavior` (cf. Figure 9) defines a single interface `BehModel` where behavior models for atomic components are intended to be embedded. The feature's language component further comprises two context conditions. The first ensures that each component contains at most one behavior model. The second requires that composed components must not contain behavior models. The LF binds the `BehModel` interface to the `CmpElem` interface of its parent feature's grammar. As a result, it is possible to specify component behavior models as top-level elements in component definitions. However, the syntax of possible component behavior models is still underspecified. For this reason, the `ComponentBehavior` LF is connected to two further features via an `or`-node (cf. Figure 1). Thus, each valid configuration containing the `ComponentBehavior` feature also contains at least one of the two subfeatures. The product manager chose the `InputOutputAutomata` feature out of these two features. The feature's grammar (cf. Figure 10) enables to model input/output automata for specifying component behavior. Transitions of such automata consist of guards (l. 7) and port assignments (l. 7). The productions' implementation remain underspecified (ll. 8-9) and are exported by the feature's language component. Thus, the exported interfaces must be bound by the LF's sub-features. The language component further consists of two

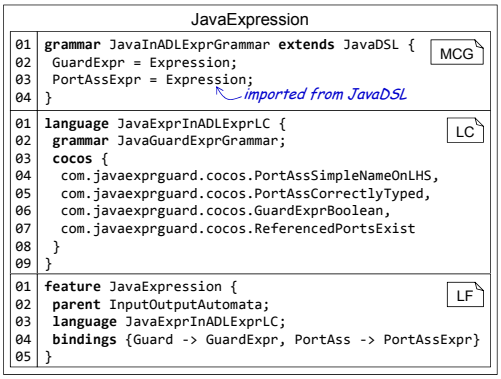


Figure 11: The `JavaExpression` language feature.

context conditions ensuring each input/output automaton contains exactly one initial state and that state names start with capital letters. The grammar's `IOAutomaton` production is embedded into the `BehModel` production of the LF's parent feature. Further, the `JavaExpression` feature (cf. Figure 11) has to be embedded, as it is marked as mandatory subfeature of `InputOutputAutomata`. Its grammar inherits the productions from a Java grammar (l. 1) and defines two new Productions (ll. 2-3). Using the new productions `GuardExpr` and `PortAssExpr` enables to specify Java expressions (The `Expression` production is part of the inherited Java grammar). The two productions are bound to the `Guard` and `PortAssExpr` interfaces exported by the `InputOutputAutomata` LF. The LF's grammar introduces two new productions and does not simply directly bind the Java `Expression` production to the `Guard` and `PortAssignment` productions to enable separate handling of guards and port assignments via their types. The first two context conditions of the feature's language component, for instance, only restrict the well-formedness of expressions used in port assignments, whereas the third context condition only restricts guard expressions, and the fourth context condition affects guards as well as port assignments. Composing the four features as described in Section 5 leads to the LF depicted in Figure 12 that models the composed language. The grammar is composed of the grammars of the selects LFs by iteratively applying the transformation described in Section 5. The new LF's CoCos are all CoCos of all selected LFs. The new LF exports each interface exported by any selected LF. A valid model of the new language is depicted in Figure 13. The component and port declarations (ll. 1-3) originate from the `ADLGrammar` (cf. Figure 8). The `InputOutputAutomata` LF's grammar (cf. Figure 10) provides the possibility to declare automata, states, and transitions (ll. 5-7) through extending the interface added by the `ComponentBehavior` LF (cf. Figure 9). The expressions `true` and `in = out` used in the transition's guard and port assignment originate from the `JavaExpression` LF (cf. Figure 11).

7 DISCUSSION AND RELATED WORK

Our notion of DSML features is based on unrestricted interfaces, *i.e.*, the interface productions do no prescribe parts of the required abstract syntax or concrete syntax. While this prevents lifting functionality considering these features to the LPL, it allows for great

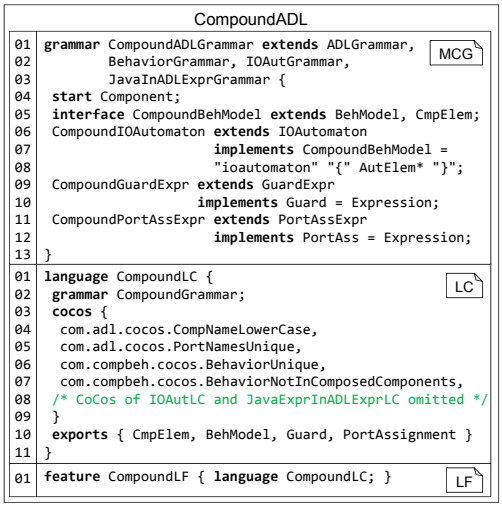


Figure 12: Result of composing the configuration's features.

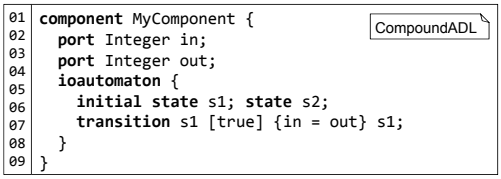


Figure 13: A valid model of the LF depicted in Figure 12.

extension flexibility. The inheritance relation between the feature grammars and the base grammar is established after feature selection. This can liberate feature developers from comprehending the base grammar at all, leading to fully independent DSML features. However, our concept also supports feature grammars aware of the base grammar to enable more specific features. Further, using the *requires* mechanism of feature diagrams, more dependencies between features can be described. Where most existing approaches use either bottom-up or top-down development of LPLs [15], we allow both directions as the feature model and the domain model are only loosely coupled. This supports agile extension of the LPL. The approach does not cover pure presentational variability [3] in the concrete syntax that does not affect the abstract syntax. As stated in [11], a usable language extension framework should have independent language extensions, which shall be automatically composable, and must not yield a corrupted composed compiler. Our approach satisfies these assumptions, because language components are (usually) independent of each other and can be composed using the composition mechanism described above. Our form of language (syntax) composition realizes the concept of conservative extensions known from formal languages. To this effect, all models that are conform to a language defined by a set of selected features, are still valid models of a language that is based on these features and arbitrary other additional selected features. Through a systematic literature review [20] comparing different approaches for LPLs, the authors identified 14 approaches realizing LPLs, where many

different concepts are involved. Some of the approaches support variability in abstract syntax only [10, 23, 29]. Most approaches use variability in metamodels, only few support variability in abstract syntax and concrete syntax on grammars, such as Neverlang [25], LISA [21], and FeatureHouse [17]. Our concept of LFs relates to the language components of Neverlang [25, 26], which contain syntax definitions in form of grammars and corresponding evaluation phases realizing its semantics. It differs in the way extension points are defined, which are realized in Neverlang using placeholders in the grammar, which are resolved via matching names. AiDE [16], built on top of Neverlang, also supports variability management of language components.

8 CONCLUSION

We have presented a concept for syntactic DSML variability that facilitates engineering, maintaining, and evolving product lines of related languages. The concept relies on modularly composable grammars encapsulated in DSML feature models related through a feature diagram model. The composition of the modular DSML features produces an integrated new feature that realizes the property of a conservative extension. With the generated DSML feature, a language workbench, such as MontiCore, can generate a parser and further tooling on a push-button basis. We lay this as the foundation for further research to be capable of covering not only the syntax, but all constituents of DSMLs.

REFERENCES

[1] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[2] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*. Springer International Publishing, 53–70.

[3] María Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within modeling language definitions. *Model Driven Engineering Languages and Systems* (2009), 670–684.

[4] Tony Clark, Mark den Brand, Benoit Combemale, and Bernhard Rumpe. 2015. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*. Springer, 7–20.

[5] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States.

[6] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. ACM, New York, NY, USA.

[7] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. 2010. Empirical Language Analysis in Software Linguistics. In *SLE*. Springer, 316–326.

[8] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.

[9] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[10] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. 2015. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling* 14, 2 (2015), 905–920.

[11] Ted Kaminski and Eric Van Wyk. 2013. Creating and using domain-specific language features. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. ACM, 18–21.

[12] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[13] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels.* Pearson Education.

[14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2008. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*.

[15] Thomas Kühn and Walter Cazzola. 2016. Apples and oranges: comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 50–59.

[16] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference*. ACM, 71–80.

[17] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '13)*. ACM, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/2430502.2430518

[18] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering* (2013).

[19] Nenad Medvidovic and Richard N Taylor. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* (2000).

[20] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures* 46 (2016), 206–235.

[21] Marjan Mernik. 2013. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software* 86, 9 (2013).

[22] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. 2009. Weaving variability into domain metamodels. *Model driven engineering languages and systems* (2009), 690–705.

[23] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. 2009. Composing visual syntax for domain specific languages. *Human-Computer Interaction. Novel Interaction Methods and Techniques* (2009), 889–898.

[24] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2. ed.). Addison-Wesley, Boston, MA.

[25] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[26] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. 2014. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Product Line Conference*. ACM, 167–176.

[27] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, 233–238.

[28] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. 2013. *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[29] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE software* 26, 4 (2009).

Systematic Composition of Independent Language Features

Arvid Butting[a,*], Robert Eikermann[a], Oliver Kautz[a], Bernhard Rumpe[a], Andreas Wortmann[a]

[a]*Software Engineering, RWTH Aachen University, Aachen, Germany*

Abstract

Systematic reuse is crucial to efficiently engineer and deploy software languages to software experts and domain experts alike. But "software languages are software too", and hence their engineering, customization, and reuse are subject to similar challenges. To this effect, we propose an approach for composing independent, grammar-based language syntax modules in a structured way that realizes a separation of concerns among the participants in the life cycle of the languages. We present a refined concept of systematic and controlled syntactic variability of extensible software language product lines through identification of syntax variation points and derivation of variants from independently developed features. This facilitates reuse of software languages and reduces the efforts of engineering and customizing languages for specific domains. We realized our concept with the MontiCore language workbench and assessed it through a case study on architecture description languages. Ultimately, systematic and controlled software language reuse reduces the effort of software language engineering and fosters the applicability of software languages.

1. Introduction

Model-driven development (MDD) [2, 88] leverages (domain-specific) software languages to reduce the conceptual gap between problem domain challenges and the software engineering solutions [26, 80]. Thus, efficient engineering, customization, and reuse of software languages has become a prime concern in MDD and gave rise to the field of software language engineering (SLE) [46, 35]. SLE develops methods and techniques to engineer domain-specific modeling languages (DSMLs) within language workbenches. But "software languages are software too" [23] and as such are subject to the same challenges regarding engineering, customization, and reuse as other software. Consequently, research in SLE has produced a variety of solutions to engineer languages based on metamodels or grammars, interpreters or generators, well-formedness rules in metalanguages or programming languages. Metamodels [7, 41, 58, 72, 90, 94] describe the abstract syntax (*i.e.,* structure) of languages as graphs of associated classes without providing a concrete syntax enabling instantiation of models. Grammars also describe the structure of a language, but can support an integrated definition of concrete syntax as well [22, 47, 50, 67]. From these grammars, model processing infrastructure (*e.g.,* a parser that translates textual models into abstract syntax instances) can be derived automatically, which greatly facilitates the efficient usage of DSMLs.

Efficient reuse is crucial to the success of software engineering [34] and software languages [13]: language users and language engineers can greatly benefit from reusing common, established, and mature language concepts. For software language engineers, reuse reduces the effort in engineering new languages from scratch. For language users, reusing concepts reduces the effort required to comprehend a language. For instance, Java reuses many concepts of C++, which lowers the barrier of using Java for C++ developers. With the digitalization of all aspects of our lives, more and more domain experts (mechanical engineers, physicist, lawyers, *etc.*) become software developers to some degree. They must be able to reify their domain expertise in software, which can be integrated with other systems. For some domains and challenges, such as software architectures in general [54] or Industry 4.0 in particular [91], many specific languages have been developed already and even more are under development.

Software product line engineering has produced methods and means to capture variability and commonalities for increasing software reuse, *e.g.,* within feature models [14]. A software product line describes several variants of software in an integrated fashion to mitigate cloning and owning of the commonalities in independent software projects. Software product line engineering techniques have been applied to software language engineering to form language product lines in several approaches [56].

We present a refined concept of controlled language variability based on reusable definitions of language syntaxes as modules within language components that can be composed to produce new languages from established building blocks. This facilitates a posteriori extensibility with additional language concepts, supports concrete syntax, and enables (re-)using languages (parts) without explicitly foreseeing this usage at the respective languages' design time. To enable a systematic reuse

[☆]This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

[*]Corresponding author

Email addresses: butting@se-rwth.de (Arvid Butting), eikermann@se-rwth.de (Robert Eikermann), kautz@se-rwth.de (Oliver Kautz), rumpe@se-rwth.de (Bernhard Rumpe), wortmann@se-rwth.de (Andreas Wortmann)

[BEK+19] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann:
Systematic Composition of Independent Language Features.
In: Journal of Systems and Software, 152, pp. 50-69, 2019.
www.se-rwth.de/publications/

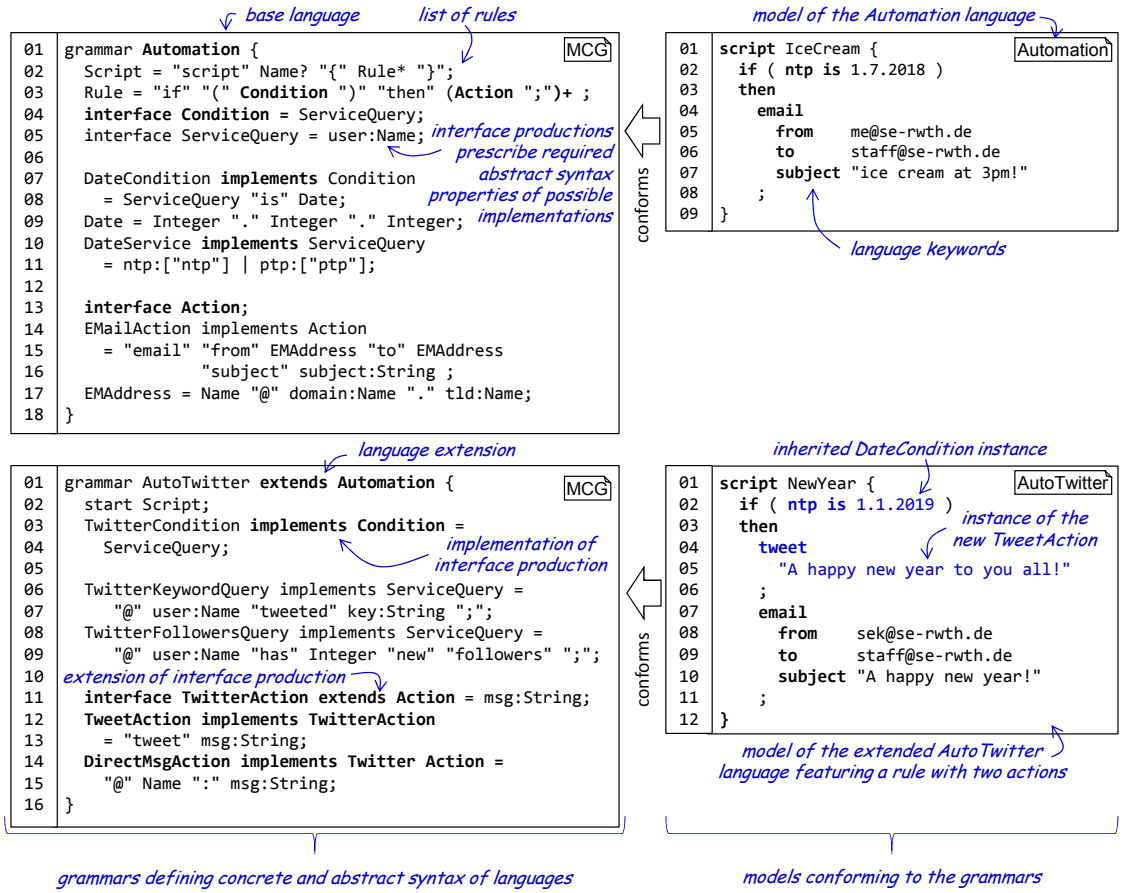


Figure 1: The `AutoTwitter` grammar (bottom left) extends the `Automation` grammar (top left) and implements its extension points (*e.g.,* interface `Condition`) to enable interacting with Twitter. Conforming models are depicted on the right.

that can ensure syntactically valid language variants, we propose explicating this variability in form of language product lines. Concrete language variants of the product line are derived from composing the incorporated language components and language-processing tooling for these can be derived on push-button basis. The key ideas of our approach are:

- to enable compositional language development by decomposing languages into composable language components,
- to enable automated language derivation via composing language components,
- to increase reuse of concrete and abstract syntax as well as tooling (*e.g.,* well-formedness check implementations) via the composable language components, and
- to decouple language development from language composition and language variant derivation.

Our concept is realized using the MontiCore [67] language workbench and leverages its composition mechanisms based on well-defined language extension points. This enables a *controlled* language composition supporting validation on product line level. The individual languages can be independent of each other, which enables these to be developed by different language engineers.

This paper is an extended version of [8], in which the basic principles of composable language components have been introduced, including a concept for systematic language reuse, its MontiCore realization, and a case study on architecture description languages. The contributions of the extended version include:

- An example product line for an imperative language.
- A concept for controlled abstract syntax extension through interface productions.
- Integration with FeatureIDE [74].
- Extended description of the involved stakeholders and more detailed discussion of related work.

In the remainder, Section 2 presents an example of an extensible base language for imperative event-based programming

and its configuration based on an explicit variability model. Afterwards, Section 3 introduces necessary preliminaries. Section 4 details our concept for systematic and controlled language reuse. Section 5 describes its realization, before Section 6 presents a comprehensive case study on architecture description languages. Thereafter, Section 7 debates observations and Section 8 discusses related work. Section 9 concludes.

2. Example

Consider engineering a modeling language for software automation that is supposed to be tailored to application-specific requirements by other developers. This language should provide general concepts and a framework for specifying automation rules consisting of extensible trigger conditions and extensible actions that are executed if the trigger condition holds. With models of this language, users can describe system-level rules (such as sending an email if a specific drive is full) as well as service-crossing rules (*e.g.,* whenever someone tweets about the hashtag `#SLE`, attach it to a file and save it locally).

Engineering such a language raises two requirements:

1. The base language must be extensible to support easy integration of new conditions and actions (*e.g.,* automatically check a weather web service on your phone to show whether you need an umbrella).

2. The extension must be restrictable to prevent interacting with undesirable trigger conditions and rules (*e.g.,* accessing a root drive of the server).

Regarding extensibility, we leverage controlled underspecification in the base language's abstract syntax (*i.e.,* the structure of a language) to enable a-posteriori integration of new language elements in a restricted fashion. Consequently, integration of new language elements should be possible in predefined places of the abstract syntax only and requires that the new language elements fulfill specific properties. The MontiCore [30, 67] grammar `Automation`, depicted in Figure 1 (top), illustrates this. It defines both concrete syntax (*i.e.,* the appearance of models of the language) and abstract syntax of conforming models and yields extension points that enable implementation and extension by inheriting grammars. The technical necessities for this are explained in Section 3.

Models conforming to this grammar are named scripts consisting of a list of rules. Hence, the grammar's start production `Script` (l. 2) begins with the concrete syntax keyword `script`, followed by an optional (denoted by "?") name and a block comprising a list of `Rule` instances. Rules (l. 3) begin with the keyword `if`, followed by a `Condition`, the keyword `then` and at least one (denoted by "+") `Action`. Both, `Condition` (l. 4) and `Action` (l. 13) are *interface* productions [67]. Other grammar productions can implement interface productions, which means that they can be applied in derivation wherever the interface production is expected on a right-hand side of another production. Technically, interface products are eliminated before handing the grammar to a parser generator. This

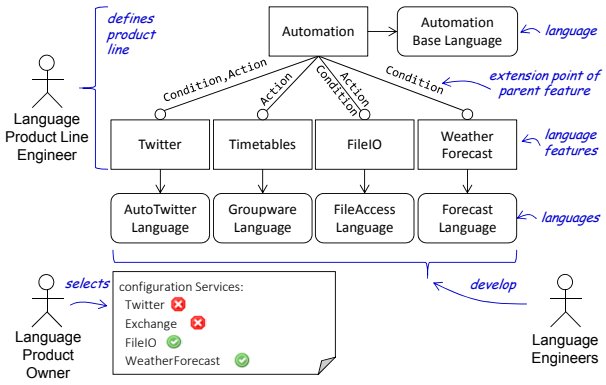


Figure 2: Example of a language product line using the `Automation` language a basis for language products featuring rules on twitter interaction, calendar services, file access, and weather forecasts.

is done by introducing an ordinary production with a left-hand side that is equal to the left-hand side of the interface production and a right-hand side that defines an alternative between all productions implementing the interface production. For example, before handing the `Automation` grammar (*cf.* Figure 1) to a parser generator, the interface production `Condition` is eliminated via adding a production `Condition = DateCondition`. Here, the right-hand side solely uses the nonterminal `DateCondition`, because the `DateCondition` production is the only production that implements the `Condition` interface production. Interface productions are translated into interface classes of the abstract syntax. Interface productions can prescribe (on their right-hand sides) abstract syntax properties required by possible implementations. For instance, `Condition` requires that every implementation yields a `ServiceQuery` (l. 5), which is another interface that requires implementations to yield a `Name`. The `Automation` base language enables conditions over dates (ll. 7-11) and actions to send emails (ll. 13-17). If additional language features are desired, `Automation` must be extended properly.

With MontiCore [30, 67], language extension can have the form of grammar inheritance between a base grammar and an inheriting grammar, *i.e.,* the inheriting grammar inherits production rules from the base grammar, such as the `AutoTwitter` grammar depicted at bottom left of Figure 1. The inheriting grammar imports all productions of the inheriting grammar and can optionally override or extend these. The `AutoTwitter` grammar reuses the start production of its parent grammar (l. 2) and defines the production `TwitterCondition` (ll. 3-4) as an implementation of `Condition`. To this end, the `TwitterCondition` introduces the concrete syntax keyword `@`, which is followed by a name and a `ServiceQuery`. The latter is necessary as it is required by the interface `Condition` of `Automation`. Moreover, the grammar introduces the two implementations `TwitterKeywordQuery` and `TwitterFollowersQuery` of `Automation`'s interface `ServiceQuery`. The grammar also extends the interface production `Action` of the `Automation` grammar with the interface production `TwitterAction`, which

prescribes that twitter actions always yield at least a `String` message. Moreover, `AutoTwitter` introduces two implementations of `TwitterAction` (and, hence, `Action`) that enable tweeting and sending direct messages.

Models conforming to both grammars are depicted in Figure 1 to their respective right. The script `IceCream` (top right) uses a date condition and an email action to describe a rule informing staff on the first of July about ice cream. To this end, it uses elements of the `Automation` grammar only. The script `NewYear` (bottom right) conforms to the `AutoTwitter` grammar and hence can use its elements as well as the elements inherited from `Automation`. The model, hence, can use a date condition of `Automation` (l. 2), a tweet action (ll. 4-6), and an email action (ll. 7-11) to send out New Year's greetings via email and twitter. Leveraging this form of grammar inheritance enables opportunistic reuse and extension, but lacks control regarding the languages that are combined, *i.e.,* which services are made available to the users. Thus, restricting the kinds of services that rules can interact with is impossible. Also, it requires comprehensive understanding of software language engineering concepts (such as grammar inheritance and interface productions) and that the inheriting languages are aware of inherited concepts (*e.g.,* the interfaces of their parent grammars). The latter also avoids reusing a language inheriting from a base language in a context different from the base language,

To enable controlled reuse, liberate domain experts from becoming language experts, and facilitate language composition, we leverage the concepts of software product lines: Available trigger conditions and actions should be provided in form of independent features arranged into a feature diagram by a language product line engineer. After ensuring the validity of feature combinations, the responsible product line engineer can use this to configure language products by selecting which features to include. Based on this selection, for instance, a concrete `Automation` language product can be derived that contains only the trigger conditions and actions desired by the application context. The language modules in the features are independent of a concrete context and can be reused for different language product lines. For the `Automation` language, a possible language product line is depicted in Figure 2. Here, the product line engineer (*e.g.,* the service provider) establishes the language product line by relating the generally available language modules in features in a meaningful way. To this end, she ensures that the available feature combinations lead to meaningful languages with respect to their syntax (*e.g.,* prevent duplicate keywords). This requires language engineering expertise and comprehending the available features, but liberates the product line engineers and the modelers from this. Each of the related features contains a complete language that can be independent of other languages, *e.g.,* of the base language. Through establishing the language product line, the engineer also defines which features implement which extension points of their parent feature (or of the base language) in the language product line. Through this, she ensures that, for instance, no actions implement the `Condition` interface. Here, the feature `Twitter` contributes conditions and actions, whereas the feature `WeatherForecast` contributes conditions only.

Based on a feature configuration, we derive a specific language product comprising only the selected features. Throughout the next sections, we describe how to decouple language components of each other such that they can be developed independently, *i.e.,* how to mitigate the inheritance relationship between a base language and an extension. Furthermore, we describe how to explicate extension points of language components, how to compose independent language components while being able to reuse tooling for a language component, and how to separate the concerns involved in the process of developing and using language product lines.

3. Preliminaries

Languages, in general, are characterized as "the set of sentences" [12, 46] that constitute the language, which also applies to software languages. As software languages – compared to natural languages – typically have a simpler structure to be processable by machines, this definition can be refined. This also is necessary for these languages to be better accessible to investigation. A common refinement [28, 31] is that DSMLs comprise

- a *concrete syntax*, describing the sentences of the language, which are build from words (textual languages), diagram elements (graphical language), or other representations,
- a (minimal) *abstract syntax*, describing the (essential) structure of sentences of the languages,
- a *semantic domain* (typically a well-defined mathematical theory) that can express the meaning of sentences, and
- a *semantic mapping*, which gives each well-defined sentence a meaning within the semantic domain.

The definition of the abstract syntax typically uses either metamodels (such as with EMF Ecore [73] or MPS [87]) or grammars (such as Neverlang [76] or Xtext [5]). The concrete syntax of textual models is usually also defined by grammars [67], whereas the concrete syntax of graphical models is usually defined via graphical or projectional editors, *e.g.,* with Sirius [82]. Textual models are usually parsed to translate the concrete syntax into abstract syntax. Whereas some approaches do not allow to create models that are not well-formed through robust projectional editors [71], most approaches require explicit checks to ensure well-formedness.

The language workbench MontiCore [67, 48] leverages extended context-free grammars (CFGs) for the integrated definition of concrete and abstract syntax [30] of DSMLs. From a CFG, MontiCore generates the corresponding (Java) abstract syntax classes, a parser for models of the language based on ANTLR [62], a model checking infrastructure that facilitates developing well-formedness rules realized in Java, and a model-to-text code generation infrastructure based upon the FreeMarker [25] template engine [1].

The generated parsers translate textual models into instances of the abstract syntax classes, the abstract syntax trees (ASTs), which are processed by handcrafted well-formedness rules registered with the generated model checking infrastructure that is realized with the visitor pattern [27]. Well-formed models are processed further and can, ultimately, be transformed into arbitrary target language artifacts by employing code generators. To validate well-formedness constraints not expressible with CFGs, MontiCore features compositional *context conditions* (CoCos). The target language artifacts then can realize the DSLs semantics and be subject to further analyses.

An exemplary MontiCore grammar `ADLGrammar` is depicted in Figure 3 (top left). It describes the quintessential elements of an ADL [55], *i.e.,* components that yield interfaces of typed ports and subcomponents that exchange messages through connectors between their ports (ll. 2-4). Nonterminals in MontiCore grammars start with an upper case letter and terminals are surrounded by quotation marks. The right-hand sides of grammar productions contain references to other nonterminals and terminals and use cardinalities ('?','*','+') known from regular expressions. To distinguish references to (non)terminals on the right-hand side of a grammar production, they can optionally be named. For instance, the reference to the nonterminal `Name` in l. 5 is named `id`. Besides 'usual' grammar productions, MontiCore supports abstract productions and interface productions. These do not directly influence the parser, *i.e.,* they cannot be derived. Instead, they influence the abstract syntax: interface productions translate to interfaces of the abstract syntax data structure and abstract productions are translated into abstract classes. Interface productions can prescribe required abstract syntax elements of possible implementations on their right-hand side. The interface production `Port` (l. 5), for instance, prescribes the presence of exactly one nonterminal `Name` with the name `id` and the presence of a `Type`. This mechanism enables underspecification of the ordering of prescribed abstract syntax elements as well as potential further (non)terminals in any rule that implements the interface production. Interface productions and abstract productions can be used on any right-hand side in the same way as normal grammar productions. A grammar production implementing an interface production, *e.g.,* the `EncryptedPort` implementing the `Port`, has to provide the required abstract syntax elements of the interface productions in the prescribed cardinality. If this is not the case, MontiCore detects it and aborts generation of language-processing tooling. In the abstract syntax this is reflected by the abstract syntax class (*e.g.,* the class EncryptedPort) implementing the abstract syntax interface (*e.g.,* the interface Port). During parsing, a production implementing an interface production can be applied at any point where the interface production is expected. For example, an `EncryptedPort` can be part of a component, as it implements the `Port`. Interface productions and abstract productions are translated into ordinary productions before handing the grammar to a parser generator. The `interface` and `abstract` modifiers in MontiCore grammars, however, effect the modifiers of the generated abstract syntax classes: Interface productions are translated to (Java) interfaces and abstract productions are translated to abstract (Java) classes. Detailed documentation about MontiCore is available [67].

MontiCore also supports compositional language integration via extension, embedding, and aggregation [30, 67]. Through extension of one or more parent grammars, a grammar inherits the grammar rules and terminal of its parent grammar(s) and can extend and override these. This, for instance, enables to eliminate rules, introduce new alternatives for inherited interfaces, or extend individual inherited rules. From inheriting grammars, MontiCore produces refined AST classes that inherit from the AST classes of the extended or overridden rules.

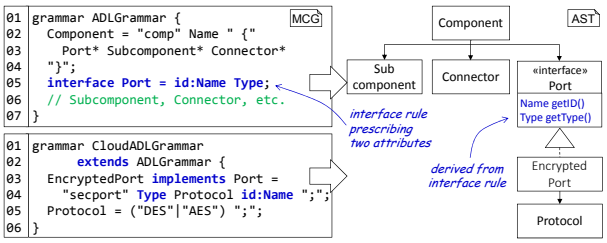


Figure 3: Example of grammar inheritance in MontiCore.

Software product line engineering (SPLE) conceives methods to handle similar software products in an integrated fashion within software product lines. Each software product, also referred to as variant, of the product line has certain commonalities and differences with regard to other variants. Developing commonalities independently is usually costly, time-consuming, error-prone, and subject to co-evolution of the independent parts. The aim of SPLE, therefore, is a reduction of developing commonalities independent of each other, but rather to reuse commonalities across the variants. Feature diagrams [3, 14, 44] group common parts of software within features. Features are arranged in feature trees. In our approach, we employ feature diagrams with the usual relations (mandatory, optional) between parent features and sub features as well as feature groups (alternative, exclusive). Besides this, the possible feature configurations can be restricted via cross-tree constraints (requires, excludes). Based on the restrictions in the feature model, a selection of a set of features determines a certain variant. The selection can be validated according to the constraints of the feature model.

4. Modeling Language Variability

In software engineering in general, and in component-based software engineering in particular, bundling of software into reusable components has proven to be helpful [39, 59]. Component-based software engineering pursues the idea of "black-box" components that can be reused off-the-shelf. This requires independent components with explicit interfaces for their composition. As software languages are software too [23], this also applies to software languages. Consequently, modularization and variability of software languages are investigated as well. Previous work [36, 67, 68, 86] conceived concepts and methods to facilitate composition of software languages.

For instance, the MontiArc [10] architecture description language [55] composes a host component & connector language with an embedded automata language and embedded statements of a Java DSL, and it is aggregated [30] with the UML/P [66] class diagram (CD) language to enable architectures with behavior models operating in the context of a CD. From applying this language and its variants to various domains including cloud systems [60], automotive [29], and robotics [33], we identified the following research questions that a concept for software language variability should solve:

RQ1: *How can variability of the syntax within a language (module) be realized?*
Answering this question requires a dedicated notion of variability that applies to the syntax of languages. One possibility to realize variability in general is the usage of underspecification. The question should answer how elements of a language syntax that are expected at a certain place can be properly underspecified. While underspecification inherently abstracts from expected elements, a certain typing of such elements is desirable for composition of syntaxes.

RQ2: *How to compose independent language modules?*
Composition integrates the syntax of two language modules. There needs to be additional information on which exact parts of the syntaxes, *i.e.,* which extension and which extension point, should be composed. This is typically realized via "glue". The solution to this question, therefore, requires a notion for this glue and dedicated composition operators for the constituents of a language module. Furthermore, proper handling of ambiguities that can arise in the composition of two language syntaxes must be assured.

RQ3: *How can language composition be guided to obtain meaningful combinations of language modules?*
To control, which language modules can be combined in a meaningful way, we explicate certain combinations of language components within families of similar languages. Modeling a family of similar languages requires selecting suitable modeling techniques to reflect the constituents realizing a language (solution space) as well as to represent the variability using appropriate techniques (problem space). Apart from this, an approach for guided combinations requires a notion of validity of the composition, *i.e.,* a member of the family of languages.

RQ4: *How can development of language modules and their composition be decoupled?*
In practice, revealing all technical details of language modules for their composition complicates their reuse and requires the language engineer composing the modules to have detailed knowledge about these. Instead, it should be sufficient to communicate the underlying conceptual model of each individually developed language module, so that the composition could be maintained by another person. This person can arrange language modules such that only meaningful compositions are allowed. This alleviates a domain engineer who selects a subset of these meaningful compositions of modules from requiring software language engineering expertise.

In the remainder, Section 4.1 presents a concept to model variability of a single language component. Subsequently, Section 4.2 explains a concept for modeling variability of multiple language components and for engineering language product lines. Then, Section 4.3 presents the roles involved in developing and using a language product line.

4.1. Reusable Language Components

The basis for our proposed variability mechanism are *language components*, which are language modules comprising (1) a grammar providing an integrated description of abstract syntax and concrete syntax; (2) a (possibly empty) set of dedicated grammar productions acting as extension points of the abstract syntax; and (3) a set of well-formedness rules. A language component is an independent, standalone *model* that is not aware of being used with an underlying variability model. To foster maturing in the sense of component-based software engineering [59], language components should be developed independent of each other, to be reusable in different contexts whenever possible. The aim of this is to enable reusing language components "off-the-shelf" by only requiring limited knowledge on the intricacies of a language component. We use grammars as foundational description mechanism for language syntaxes and show how to describe extension points and extensions of language syntax through grammars. The notion of extension points in the abstract syntax, however, can be transferred to metamodel-based syntax definitions as well.

We explicate the extension points within language components to denote the reduced effort of understanding the intricacies of a grammar when a language component is reused. For instance, in the exemplary language product line in Figure 2, the language component `AutomationBaseLanguage` can be used mainly by communicating the extension points `Condition` and `Action`. The language product line engineer has to understand implementation details on other grammar productions (*e.g.,* `EMAddress`) only on a more abstract level, unless the composition is invalid, *e.g.,* due to ambiguities. A further advantage of explicating the extension points is that language engineers who develop language components can hide certain parts of the language that must not be extended as, *e.g.,* known from private methods in Java. In general context-free grammars, our notion of extension points within a language's syntax applies to arbitrary grammar productions. This means any grammar production can serve as extension point. A grammar production used as extension point can provide a default implementation (that is its right-hand side). In context-free grammars in general, all productions must provide a right-hand side and with these, all extension points must provide a default implementation. Variability is realized through underspecification of alternatives to the default implementation. In other words, extending an extension point is realized by adding the extension as alternative to the right-hand side of the extension point. To foster only meaningful combinations of languages and enable reusing a language component based on knowing its extension points, language engineers have to decide carefully which grammar productions should act as extension points.

We prescribe that the cardinality of an extension point is only realized via the variability model, and not via the language component itself. For example, the grammar should not restrict that an extension point must be extended at all, once, or multiple times. This increases the reusability of a language component within different contexts. An extension is a grammar production that provides syntactical elements to resolve the underspecification of an extension point. To this effect, extensions provide the right-hand side that was underspecified in an extension point. Typically, the extension is contained in a different grammar than the extension point it implements. More precisely, connecting two grammars via an extension point and an extension is realized within two steps: (1) Merge all productions of both grammars to form a new grammar. The start production of the grammar containing the extension point becomes the start production of the new grammar. (2) Add the left-hand side of the production of the extension as alternative to the right-hand side of the extension point production.

In the realization of the concept, leveraging the powerful language composition techniques of MontiCore, we represent grammar extension points with *interface productions* (*cf.* Section 3). These have the additional advantage, that a concrete right-hand side can be underspecified (*i.e.,* no default implementation is required), but assumptions on the abstract syntax of possible productions implementing the interfaces can be made. This enables controlled extension and reasoning over possible implementations already on the interface level. In this sense, interface productions resemble interfaces or abstract classes in metamodels (depending on the formalism). Consequently, for instance, abstract classes in a metamodel would also enable to represent (controlled) extension points. The concept of language components realizes **RQ1**.

In MontiCore, language components are not required to provide default implementations of their extension points. MontiCore enables a grammar to define an interface production without providing a production implementing the interface production. However, the components themselves are abstract in the sense that it will not be possible to concretize all possible sentences (models) of the language. Nevertheless, language components may provide certain default implementations by providing grammar productions that implement the interface production (or classes implementing the interfaces in metamodel-based approaches). For example, a Statechart language component with an extension point for guard conditions on transitions can optionally include built-in default expressions but still be extensible with additional kinds of expressions (*e.g.,* OCL, LTL, ...) through different language components.

However, explicating extension points via special grammar constructs requires that language engineers have to foresee each extension point during development of a language component. The realization of our concept in MontiCore, for instance, considers all interface productions as extension points, which liberates language engineers in maintaining explicit extension points as separate artifact(s).

Besides the context-free grammar, a language component contains a set of well-formedness rules. These may check the well-formedness of any abstract syntax element of the grammar of the enclosing language component. Where the abstract syntax is arranged as a tree, well-formedness rules should check the most specific syntax element that is possible. This increases the reusability of the well-formedness rule if only parts of the language's syntax are reused. Additionally, it reduces side effects of the well-formedness rule that arise if it is applied for post-hoc added language constructs for which it was not designed to apply for. For example, consider a rule checking the well-formedness of dates (*e.g.,* 30.02.2019 is not considered well-formed) for the `Automation` grammar depicted in Figure 1. The well-formedness rule should be implemented against the abstract syntax element introduced by the nonterminal `Date` as the most specific possible element. If it was implemented against more general elements, such as the `DateCondition` nonterminal, and only the `Date` nonterminal was reused in another context, then the well-formedness rule would not apply there. With MontiCore, well-formedness rules can be implemented against interface productions to assert well-formedness of their right-hand sides. It is possible to check all implemented context conditions via a checking infrastructure. This infrastructure is automatically generated by MontiCore using the grammar [67].

The composition of language components requires different composition operators for the different constituents of a language component. A language component with a grammar and well-formedness rules requires composition operators for these. A language component based on metamodels would require a composition operator for metamodels [17, 19]. Two language components are composed by composing all constituents of these language components.

Multiple language components can be composed at a time, but the composition operator for grammars is directed. Applying all required composition operators produces an integrated, new language component. This integrated language component contains all details necessary to synthesize language-processing tooling for the contained language. Furthermore, it can serve as a language component within different language product lines, *e.g.,* by engineering a new language product line with the generated language component at its base, describing the commonalities of all possible variants.

Our notion of syntactic language components requires composition operators for grammars and for context conditions. The composition operator for grammars takes an ordered list of input grammars and a binding from extensions to extension points (of other grammars). Using these, it embeds extending implementations into the extension points. For MontiCore grammars, this is realized as implementing interface productions, for other context-free grammar by introducing additional alternatives at the extension point. For metamodels, this can be realized by introducing new interface implementations or subclasses of abstract metamodel classes. For grammars, the result would be a composed grammar that includes all grammar productions of the input grammars. For metamodels, this would be a joined metamodel including all concepts and relations of participating metamodels. The composition operator does not resolve potential conflicts, such as, through grammar productions with identical nonterminal names. Instead, this has to be

resolved via additional transformations. All context-free grammars have a single start production that defines the start of the derivation process. The start production of the composed grammar is the start production of the first input grammar[1].

Independent of the abstract syntax realization mechanism, the extension points of the composed language components are the union of extension points of the input language components. In other words, all extension points are preserved in the composed language component.

Similarly, the sets of well-formedness rules are preserved. The composition operator merges these such that the composed language component contains the union of all participating well-formedness rules. Again, the composition operator does not check if well-formedness rules contradict. If this is the case, the set of well-formed models may be empty. For instance, consider a well-formedness rule assuming that automation models (*cf.* Section 2) are well-formed if these contain exactly one action and another rule considering well-formed models to have more than one twitter action. Apart from that, composing sets of well-formedness rules is straightforward. The concept for composing language components (*i.e.,* composing grammars, their extension points, and well-formedness rules) realizes the identified **RQ2**.

4.2. Language Product Lines

With language components and a notion of their composition, new languages can be built up from independent, reusable modules. However, it is cumbersome and error-prone to identify, which language components can be reused for a certain purpose and how their composition is arranged, *i.e.,* which extension points and which extensions are connected.

To remedy this, our approach leverages a variability model to organize language components and establish bindings between these. For the realization, we identified feature models with their usual relations [44] and constraints as a suitable modeling technique. In the feature model of a language product line, each feature of a feature model instantiates exactly one language component. The concept, however, can be adapted and applied to different variability modeling techniques as well. As language components are unaware of being used with features, they can be used with different feature models and even with different features within a single feature model. For instance, two features could instantiate the same language component but embed it differently, which is useful when reusing more generic language components – such as expression languages – with different extension points. Moreover, the loose coupling between feature model and language components facilitates extending the language product line with new features and language components to support their evolution.

Using a feature model, bindings, and language components, a language product line can be constructed (*cf.* **RQ3**). Each possible variant (product) of the language product line is described by a valid feature configuration of the feature model. This prevents uncontrolled composition of language components, *i.e.,* a form of composition that is neither foreseen or nor intended by the language product line engineer and leverages language users from comprehending internals and composition of different language components.

Moreover, properties of extension points, such as being mandatory, optional, or exclusive with another extension point are solely realized via the variability model. This yields the advantage of the language components to be better reusable in different contexts, with different applied extension points, and to support staged configuration [15]. A *requires* relationship between two features (denoting that one feature relies on the presence of another feature), enables to indicate that a language component of one feature must have knowledge about the language component of the other feature. This reflects, *e.g.,* inheritance between the respective grammars or dependencies between the metamodels. It enables modeling that, for instance, a feature providing an expression language requires another feature providing a type system.

A binding between extension point and extension is specific to two features of a concrete feature model and the language components they instantiate. Therefore, we organize all bindings of a language product line within a single dedicated binding model. The model describes for each feature of the feature model which language component it instantiates by stating a mapping from a feature name to a language component name. Additionally, it defines binding rules that connect extension points and extension via their nonterminal names. To this effect, it contains one name mapping per feature of the feature model and one binding rule per edge between two features of the feature model. A binding model is specific to a language product line and its related language components and is the only connection between these, otherwise decoupled, parts as it indicates that a language component is instantiated in each feature it is bound to. Consequently, reusing the feature model of a language product line, *e.g.,* with language components based on another language definition formalism, becomes feasible. Each binding rule constitutes (1) the parent feature and the extension point that the rule applies to and (2) the child feature and the extension that is bound to the extension point.

Uncontrolled composition could occur if the connection between extension point and extension would be based upon, *e.g.,* both grammar productions to have the same name or by "autoconnecting" an extension to all extension points. To mitigate this, we explicate bindings. As the root feature does not have a parent feature, there is no binding rule that binds extensions of the root language feature to other extension points. To this effect, our approach allows extensions to be extension points (therefore in MontiCore, interface productions) as well. Extending an extension point *refines* the extension point and enables concretizing its required abstract syntax elements, as well as to enable more sophisticated structuring of features of the language product line. This is demonstrated by an example in Section 6.

Extending a selected set of features with further features does not invalidate the syntax of models that were valid before, as

[1]This appears to be a limitation at this point, but in combination with a feature model as described in Section 5, the first input grammar is always the one used in the root feature.

adding new language components cannot eliminate syntax elements. This *conservative extension* [67] of the syntax, enables reusing tooling for multiple language variants of the product line. However, it cannot be used to guarantee semantical correctness of analyses. Through adding new language elements, several analyses might produce different results. Consider, for instance, an analysis counting actions defined in a model conforming to the `Automation` grammar (*cf.* Figure 1). If a language extension would add nested actions, these would not be taken into account individually unless the analysis is modified.

Deriving a language variant begins with selecting features to compile a feature configuration. The feature configuration has to be valid with regard to the feature model and its constraints. Then, our language composition tool composes the language components of all selected features at once. As the composition operator is ordered, the order of language components is computed by a depth-first search starting from the root feature. Here, the order of siblings is irrelevant.

Composition produces an integrated grammar, a joined set of context conditions, and all extension points of the input language component are retained. Furthermore, a new language component containing these is generated that can be used, *e.g.,* as root for a new language product line. A language component should contain all information necessary to produce language-processing tooling for the defined language. This activity can be performed on the composed language component as last step of language variant derivation.

Developing a language product line by organizing which language components can be composed does not necessarily require to understand in-depth details of the language components' internals. This facilitates reusing language components that the language product line engineer did not develop on her own. Thus, she only has to consider all details of a language component if side effects of composing the grammars (such as ambiguities in abstract or concrete syntax) arise. To avoid language compositions that cause invalid language variants, she can, for instance, establish cross-feature relations preventing these (*e.g.,* by mutual exclusion of features contributing ambiguous grammars). This fosters a separation of concerns between language engineers developing language components and language product line engineers arranging these meaningfully (*cf.* **RQ4**).

4.3. Roles

Our method to enable modeling language reuse through variability rests on a separation on related concerns along different roles:

- *Language engineers* develop language components that encapsulate abstract syntax and concrete syntax.
- *Language product line engineers* are language engineers that create language families as feature models that describe possible characteristics of the family's language products. To this end, they collect relevant language components, assign these to features, and define how these realize extension points of language components of their parent features.
- *Language product owners* configure the language family to obtain a language product for a specific domain, context, or application. Based on this configuration, the tool chain generates model-processing tooling for the selected product.
- *Modelers* employ the generated modeling processing tooling to analyze models and transform these into GPL artifacts.

The separation of concerns between the participating roles frees the individual roles to be involved in all phases of conceiving language product lines and language products. Figure 4 details the activities of the different stakeholders: Creating a language product line begins with *language engineers* developing the modeling languages that the product line combines. We assume the languages are defined using grammars defining the concrete and abstract syntax and its possible extension points. Extension points of the grammar become extension points of the language component. After the grammar is defined, language engineers define rules describing the well-formedness of models of the language. Grammar and well-formedness rules then are explicated within a language component model.

The *language product line engineer* collects the different language components and arranges these into a feature model describing the product line. To this end, she first defines the desired features within a feature model and models their constraints according to the domain's needs. Then, within the binding model, she instantiates a language component for each feature or the feature model. Afterwards, she creates binding rules for each connection between extension point and extension. In the resulting product line, features denote selectable language components and the language components of child features are available for embedding into extension points of the language components of their parent features. Optionally, language product line engineers can create glue that is specific to the connection between two instances of language components that are connected via the feature model. This glue comprises *e.g.,* well-formedness rules and further analyses or tooling specific to the composition of two languages. Apart from these, it is possible to perform handcrafted adaptations of, *e.g.,* grammars that cause ambiguities when they are composed through creating a grammar that inherits from the original one and overrides conflicting productions.

Based on the language family, the *language product owner* selects features of interest for a specific domain, context, or application and uses the language variability infrastructure to automatically compose grammars and well-formedness rules into new model processing tooling for the specific configuration. With this, the *modelers* can process models using the selected features transparently as if developed for a single monolithic language.

To this end, the separation of concerns among the described roles alleviates experts to have capabilities in all fields of expertise among the process of developing language product lines as depicted in Figure 5. This reflects in the different kinds of artifacts created and used throughout the process. The capability

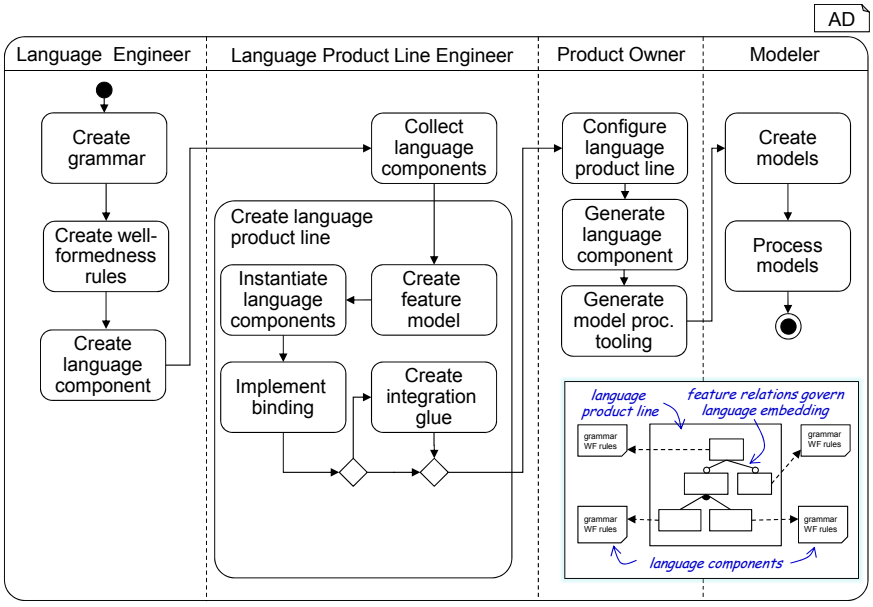


Figure 4: Activities and associated roles participating in creating, configuring, and using language product lines.

C = Create
U = Understand

	Syntax	Well-Formedness Rules	Language Components	Feature Tree	Binding Rules	Feature Configuration
Language Engineer	C	C	C	-	-	-
Language Product Line Engineer	U/C	U/C	U	C	C	-
Language Product Owner	-	-	U	U	-	C
Modeler	(U)	(U)	-	-	-	-

Figure 5: The roles involved in developing language product lines and their required capabilities.

to create artifacts related to syntax and well-formedness of languages is solely required by language engineers, and partially by language product line engineers. The latter only require to create syntax or well-formedness artifacts if conflicts arising in combinations of syntaxes or well-formedness rules result in problems that are to be resolved manually. Language product line engineers, however, have to understand the artifacts realizing syntaxes and well-formedness rules of the involved languages. Modelers are only required to understand these in a more abstract way, *e.g.,* through documentation. Language product owners do not need to understand syntax and well-formedness rules. Language components are created by language engineers and have to be understood by language product line engineers and product owners to create and configure language product lines. Modelers do not have to be aware of the existence of language components. The feature tree is created by the language product line engineer and used by language product owners for configuring language variants through feature configurations. The language product line engineer creates a binding model for each feature tree, in which she connects features to language components and binds extension points of language components with extensions of other language components. Language engineers and modelers do not have to be aware of the presence of feature models, binding rules, and feature configurations.

5. Integrating Languages Syntaxes

This section describes the MontiCore realization of the concept introduced in the previous chapter. The concrete and abstract syntax of a MontiCore [30] language is defined within an extended context-free grammar. Well-formedness checking is realized via context conditions implemented as Java classes (*cf.* Section 3). To define extension points of a grammar, our approach leverages interface grammar productions (interface nonterminals) that the grammar itself can provide default implementations for by containing productions that implement the interface production. As depicted in l. 13 of the top grammar in Figure 1, interface production rules are not required to have a right-hand side. If an interface production has a right-hand side, it indicates nonterminal symbols required to be provided by grammar productions that implement the interface production. Similar to ordinary nonterminals, interfaces can be referenced on the right-hand sides of other production rules (*cf.* `Component` in ll. 2ff). Nonterminals can implement interfaces (ll. 6f), resulting in an abstract syntax structure as depicted by example on the right of Figure 3. Our approach supports composition of *independent* language components. This mitigates the necessity to use grammar composition via inheritance as explained in Section 3 (*cf.* `CloudADLGrammar` in Figure 3), which requires

```
01 language ADLLangComp {                                   LC
02   grammar com.ma.ADLGrammar;
03   cocos {
04     com.ma.cocos.CompNameLowerCase,
05     com.ma.cocos.PortNamesUnique
06   }
07 }
```

Figure 6: The language component ADLLangComp.

```
01 binding for ADL {                                        Binding
02  feature BaseADL uses ADLLangCmp;
03  feature Behavior uses BehaviorLangComp;
04  feature Automaton uses AutomatonLangComp;
05  bind Behavior.BehModel to BaseADL.ADLElement;
06  bind Automaton.IOAutomaton to Behavior.BehModel;
07 }
```

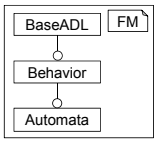


Figure 7: The binding model for the composition of exemplary language components.

that the inheriting grammar cannot be used without the inherited grammar and therefore reduces reusability.

Composition of language components realizes the variability in the solution space of our approach. The composition of language syntaxes is performed by composing their context-free grammars and context conditions. MontiCore grammars used in language components can, in general, use grammar inheritance as supported by MontiCore. Grammar inheritance between grammars contained in different language components of a language product line should be performed carefully, as it can yield unintended side effects. Therefore, we recommend indicating such a relationship via a "requires" constraint in the feature model.

Figure 6 depicts a *language component* definition. It holds a reference to a MontiCore grammar and a set of corresponding context conditions. The language component has the name `ADLLangCmp`, references the `ADLGrammar` presented in Figure 3, and references two context conditions. All interface nonterminals defined in the referenced grammar are automatically exported as extension points of the language component.

Consider a small language product line for ADLs with a base feature BaseADL, an optional behavior feature, and an optional Automaton feature for the realization of the behavior. The corresponding feature model is depicted in the right of Figure 7. The binding model depicted in the left of Figure 7 contains the relation between features in the feature model (depicted right) and language components (ll. 2-4) that these instantiate. Further, it defines binding rules between extension points and extensions(ll. 5-6). Thess rules define which nonterminals of a child feature are bound to which extension points of a parent feature. For example, in every language variant comprising the `Automata` feature, the `IOAutomaton` nonterminal of the grammar contained in the feature's language component is bound to the extension point `BehModel` exported by the language component of feature `Behavior` (l. 6). Extension points and extensions have to be identified via names comprising feature and grammar production to be uniquely identifiable within a product line. A single language component, for instance, could be instantiated more than once in a product line. The top of Figure 8 depicts two MontiCore grammars that are part of lan-

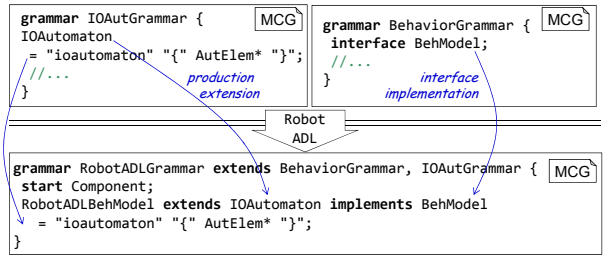


Figure 8: The composition (bottom) of two MontiCore grammars (top).

guage components to be composed. The grammar `IOAutGrammar` is part of the language component `AutomatonLangComp` and, inter alia, contains the grammar production `IOAutomaton`. The `BehaviorGrammar` is part of the language component `BehaviorLangComp` and provides the interface production `BehModel`. The grammar `RobotADLGrammar` depicted at the bottom is the result of the composition of the upper two grammars with the binding model as described above. The relevant part of the binding for this composition is contained in l. 6. The production `IOAutomaton` of `IOAutGrammar` uses the extension point (= interface production) `BehModel` of the grammar `BehaviorGrammar`. The name of the synthesized grammar is derived from the name of the language variant as prefix and `Grammar` as suffix. The grammar inheritance mechanism of MontiCore is used to extend both grammars, which enables to reuse all of their grammar productions. For technical reasons, the generated grammar has to reference the start production of the grammar of the root feature, to produce the same top-level abstract syntax element for each generated parser of the product line. For each applied binding rule (= implemented interface production), a new nonterminal is generated in the synthesized grammar. The new production extends the extension nonterminal with `extends IOAutomaton` and implements the extension point interface with `implements BehModel`. The effect of implementation of an interface has been explained in Section 3. Extending a production has a similar effect, it can be used in every place of the grammar where the extended production has been applied. With the abstract syntax tree data structure MontiCore generates from a grammar, the corresponding abstract syntax tree classes reflect the same extension and implementation on the level of Java classes and interfaces. Consequently, the abstract syntax class of `RobotADLBehModel` extends from the abstract syntax class of the `IOAutomaton` and implements the interface `BehModel`. This bears the advantage that all algorithms and tooling applicable to the base element can also be applied to the extension. For example, a well-formedness rule checking that an IOAutomaton has a single initial state can also be applied to the generated RobotADLBehModel. With our approach in general, the right-hand side of the generated production equals the right-hand side of the extended production. If the right-hand side contains a nonterminal that was extended during composition, it is replaced with the newly generated nonterminal name. This transformation is especially necessary for recursive productions and refining in-

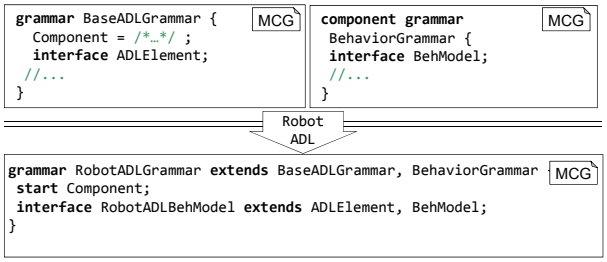


Figure 9: Composition with extension point refinement

terfaces to realize the expected behavior. Without this replacement, parsers would create instances of the abstract syntax class derived from the extension nonterminal instead of the abstract syntax class of the newly generated one. Therefore, the connection between extension point and extension would break, because the extension nonterminal alone has no connection to the extension point. With refinement of extension points, both extension and (as always) extension point are interface productions. Here, a new name is derived in the same style as above in the nonterminal case, but it extends both interfaces. This is visualized in Figure 9.

The language component resulting from the compositions of the two language components `AutomatonLangComp` and `BehaviorLangComp` is depicted in Figure 10. The referenced grammar is the newly generated grammar depicted at the bottom of Figure 8. It combines the abstract syntaxes and concrete syntaxes of the composed grammars. The set of context conditions of the integrated language component is the set union of all sets of context conditions of input language components. Checking all context conditions of the composed language component is possible, because of the compatibility of the abstract syntax tree data structure – that MontiCore relies on for checking context conditions – between the composed grammar and the input grammar that the context condition has been defined for. As no interface productions are removed in the composed grammar, all extension points are joined and the resulting language component comprises all extension points of the input language components. The language component synthesized for a certain language variant can optionally be extended with handwritten grammars and new context conditions before MontiCore is executed to produce tooling for the language variant.

6. Case Study

The following presents an extended case study of our approach in the context of architecture description languages based on the extended example in [8]. Architecture description languages (ADLs) [55] combine MDD with component-based software engineering for the description of software architectures. There are over 120 stand-alone ADLs [54], each tailored towards a specific application domain, such as automotive [16], avionics [24], consumer electronics [81], or robotics [69]. Developing and maintaining domain-specific variants of an ADL is challenging [10].

```
01 language RobotADLLangComp {                    LC
02   grammar com.ma.RobotADLGrammar;
03   cocos {
04     com.behavior.SingleBehaviorModel,
05     com.ioaut.SingleInitialState,
06     //...
07   }
08 }
```

Figure 10: The resulting composed language component.

6.1. ADL Language Product Line Motivation and Overview

Our approach enables to prevent the efforts of creating, maintaining, and evolving multiple stand-alone ADL variants tailored to specific domains individually. Using feature-driven language composition enables to start with a core ADL exhibiting extension points and independently developing language components that provide modeling elements required for architectures of the different domains. Modifying one language component automatically updates all language variants that contain the component if the variant derivation is executed again. In the context of ADLs, our approach facilitates to produce tailored ADL variants for new domains, *e.g.,* architectures for machine learning. Common ADL parts can be reused from the language product line, and new features specific to the new domain can easily be added.

Excerpts of a complete language product line for ADLs, called `MyADL`, capable of describing software components for modeling both cloud systems and embedded systems are depicted in Figure 11. Different language engineers contribute language components (depicted right) with explicit extension points defined via their interface productions in the grammar. The language product line engineer defines the feature model and therefore defines which features are available and how selecting a feature may depend on the selection of other features (depicted left). The product line comprises a common base feature (`BaseADL`) and features that are typical to ADLs for embedded systems (*e.g.,* automated connection of ports based on their types or names) as well as features related to scalable and secure cloud systems (*e.g.,* replicating components and encrypted communication). Each feature instantiates exactly one language component that might define further extension points. The relation between two features in the feature model describes how their language components are integrable (parent-child relationship) or whether a feature's language component relies on or conflicts with another language component (cross-tree constraints).

The decoupling between features of the language product line and language components, as well as the decoupling among different language components, enable to reuse the language components in different feature models and to instantiate a language component within different features of a single feature model. Based on a feature configuration defined by the language product owner (middle left), a software tool (implementing the mechanisms described in the previous sections) establishes the connections between the selected features' language

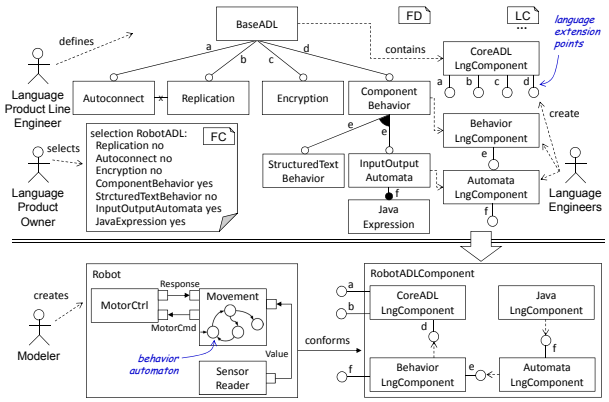


Figure 11: A language product line defined as feature model over language components. Given a feature configuration (top), the variant is transformed into a new language component (bottom).

components. A feature's language component is integrated into the extension points of the language component of the respective parent feature. For instance, the language component of feature `InputOutputAutomata` is integrated into the extension point *e* (the label of the edge between the features `ComponentBehavior` and `InputOutputAutomata` in Figure 11) of the language component contained in the feature `ComponentBehavior`. Integrating all language components of the selected features yields a new language component, which can be used by the respective domain experts to model corresponding software architectures using exactly the modeling elements selected through the feature configuration (in this case automata models describing component behavior).

Being able to reuse language components without modification enables reusing the associated tooling (analyses, transformations) with the generated language component as well. This is possible because the generated AST classes of each individual language component are reused by the tooling of a generated language component. Reuse is possible because the result of the integration of each feature's language component with the language component of the feature's parent stands in a "conservative extension" relationship (in the sense of [67]) with the parent feature's grammar. With this, changes to a language component and its tooling are immediately reflected in all generated language components as well. Therefore, the effort of creating, maintaining, and evolving modeling languages is minimized. The loose coupling between features and language components also simplifies integration of new features into the feature diagram. Integrating a new feature below `ComponentBehavior`, for instance, does not influence other features and language components. The language product line engineer, however, has to ensure potential cross-tree constraints (excludes, implies) of the new feature to existing features.

A generated language component can yield extension points. Thus, the creation of intermediate products that require further refinement is also possible. Where multiple similar domains are addressed, creating refined domain-specific language product lines enables to restrict a large base product line accordingly.

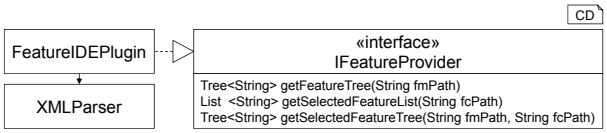


Figure 12: Integration of FeatureIDE [74] into the language composition tool.

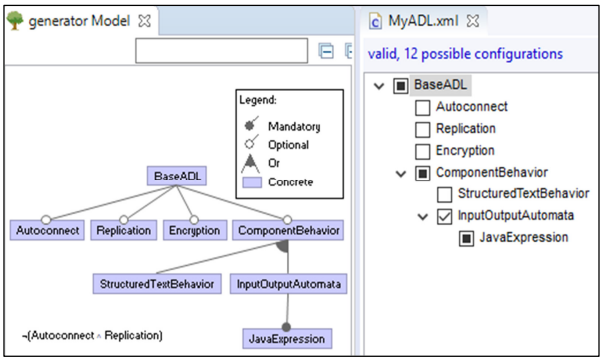


Figure 13: The feature model and the feature configuration in the employed FeatureIDE editors.

The tooling to process language product lines and derive variants is extensible into different directions: it enables to use different languages for the definitions of language components and bindings, and it supports different feature modeling tools. For example, the plug-in integration of FeatureIDE [74] for modeling feature diagrams and feature configurations is depicted in Figure 12. Our language composition tool provides an interface `IFeatureProvider` that has to be implemented by plug-ins for feature modeling tools. The plug-in for FeatureIDE employs the class `FeatureIDEPlugin`, which uses an `XMLParser` to parse the XML artifacts of feature models and feature configurations produced with FeatureIDE.

6.2. ADL Language Product Line Details

The following describes the language product line, its features and language components depicted in Figure 11, and presents a language component derived from a configuration as well as valid and invalid models with regard to the derived language component. The feature model and the selected configuration in FeatureIDE are depicted in Figure 13. The `BaseADL` feature must be domain-agnostic as it is part of all possible product line configurations. It thus only contains the basic elements of ADLs such as components, ports, and connectors that are common to each language variant. The feature `Autoconnect` adds syntax and transformations to realize an automatic connection of ports with either identical names or types. The `Encryption` feature enables describing secure ports (`SecurePort`) and encrypted connections (`EncryptedConnector`) between these. The `Replication` feature provides syntax for modeling systems with components that are capable of replicating themselves if a replication condition is satisfied. This is useful, for instance, in client-server architectures where a client component replicates on a large

```
01 bindings for MyADL {                                          Binding
02   feature BaseADL uses ADLGrammar;
03   feature ComponentBehavior uses ComponentBehaviorGrammar;
04   feature InputOutputAutomata uses IOAutGrammar;
05   feature JavaExpression uses JavaInADLExprGrammar;
06   feature DynamicReconfiguration uses DynReconBehGrammar;
07   feature ModeAutomata uses ModeAutGrammar;
08   //…
09   bind BaseADL.CmpElem to ComponentBehavior.BehModel;
10   bind ComponentBehavior.BehModel to InputOutputAutomata.IOAutomaton;
11   bind InputOutputAutomata.Guard to JavaExpression.GuardExpr;
12   bind InputOutputAutomata.PortAss to JavaExpression.PortAssExpr;
13   bind BaseADL.CmpElem to DynamicReconfiguration.ReconBehModel;
14   bind DynamicReconfiguration.ReconBehModel to ModeAutomata.ModeAut;
15   //…
16 }
```

Figure 14: The binding rules for the language product line.

BaseADL

```
01 grammar ADLGrammar {                                          MCG
02   Component = "component" Name "{" CmpElem* "}";
03   interface CmpElem;
04   interface Port extends CmpElem;
05   DefaultPort implements Port = "port" Type Name ";";
06    /* Subcomponent and Connector productions omitted */
07 }
```

```
01 language CoreADLLngComponent {                                 LC
02   grammar ADLGrammar;
03   cocos {
04     com.adl.cocos.CompNameLowerCase,
05     com.adl.cocos.PortNamesUnique
06   }
07 }
```

Figure 15: Parts of the `BaseADL` language component.

number of requests. The language product line engineer considers component replication to be a threat for autoconnecting ports. Choosing one of the two corresponding features thus excludes the other. The `ComponentBehavior` feature (*cf.* Figure 16) introduces *behavior blocks* to the ADL through the interface `BehModel`. The language product line engineer intends component behavior models to be modeled in such blocks. The child features `StructuredTextBehavior` and `InputOutputAutomata` contain different behavior languages. It should not be possible to model empty behavior blocks and thus, choosing the `ComponentBehavior` feature requires to choose at least one feature that defines a component behavior language. Automata use expressions on their transitions as guard conditions. For this purpose, the language product line currently only includes the `JavaExpressions` feature, which is therefore marked mandatory.

The *binding* model depicted in Figure 14 describes the mapping from feature name to the name of the language component instantiated by the feature (ll. 2-7) and the binding of extension point of a feature to an extension of another feature (ll. 9-14). The binding model belongs to the language product line `MyADL` (l. 1). The layer of indirection between feature name and language component name enables to use the same language component within two different features (which must have a unique name within a feature model) of a feature model. The explicit binding between extension point and extension prevents uncontrolled composition (*cf.* Section 4).

ComponentBehavior

```
01 component grammar ComponentBehaviorGrammar {                  MCG
02   interface BehModel;
03 }
```

```
01 language BehaviorLngComponent {                               LC
02   grammar BehaviorGrammar;
03   cocos {
04     com.compbeh.cocos.BehaviorUnique,
05     com.compbeh.cocos.BehaviorNotInComposedComponents
06   }
07 }
```

Figure 16: The `ComponentBehavior` language feature.

Consider a language product owner who aims at developing software architectures in which atomic components' behavior can be specified via input/output automata. She thus selects the configuration containing the features `BaseADL`, `ComponentBehavior`, `InputOutputAutomata`, and `JavaExpression`.

InputOutputAutomata

```
01 component grammar IOAutGrammar {                              MCG
02   IOAutomaton = "ioautomaton" "{" AutElem* "}";
03   interface AutElem;
04   State implements AutElem =
05                        (["initial"])? "state" Name ";";
06   Transition implements AutElem = "transition" src:Name
07             "[" Guard "]" "{" PortAss* "}" trg:Name ";";
08   interface Guard;
09   interface PortAss;
```

```
01 language AutLngComponent {                                    LC
02   grammar IOAutGrammar;
03   cocos {
04     com.ioaut.cocos.StateNamesUpperCase,
05     com.ioaut.cocos.UniqueInitialStates
06   }
07 }
```

Figure 17: The `InputOutputAutomata` language feature.

Figures 15-18 depict parts of the configuration's constituents. The `BaseADL` language component (*cf.* Figure 15) is the product line's root feature. The feature's grammar defines language elements common to all ADL variants such as components, connector, and ports. The language component further defines two context conditions and the interface `CmpElem`. With this, it is possible to extend component definitions with further top-level elements through language component composition via binding productions to the interface `CmpElem`. The `Subcomponent` and `Connector` productions of the `BaseADL` grammar are omitted. The grammar of the `ComponentBehavior` feature (*cf.* Figure 16) defines a single interface `BehModel`. Behavior models for atomic components are intended to be embedded into the `BehModel` interface. The feature's language component further comprises two context conditions. The first ensures that each component contains at most one behavior model. The second requires that composed components must not contain behavior models. The language component binds the `BehModel` interface to the `CmpElem` interface of its parent feature's grammar. The language component defines the `BehModel` interface as extension point. With this, the `BehModel` extension

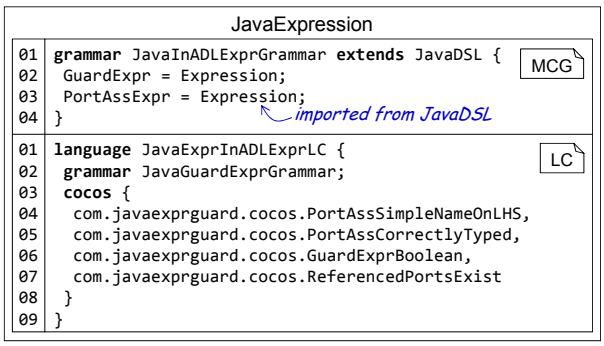


Figure 18: The `JavaExpression` language feature.

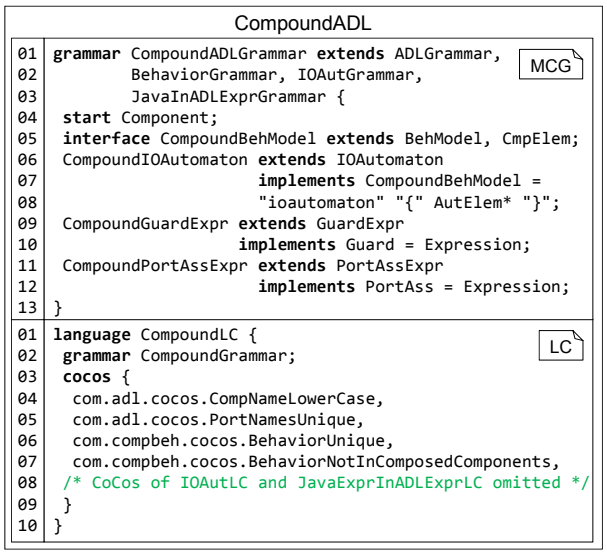


Figure 19: Result of composing the configuration's features.

point refines the `CmpElem` extension point (*cf.* Section 4.2): Every `BehModel` model can be embedded as a `CmpElem`, but the opposite does not hold. Integrating the language components of the `BaseADL` and `ComponentBehavior` features yields a new feature that enables specifying component behavior models as top-level elements in component definitions. However, the syntax of possible component behavior models is still underspecified. For this reason, the `ComponentBehavior` language component is connected to two further features via an `or`-node (*cf.* Figure 11). Thus, each valid configuration containing the `ComponentBehavior` feature also contains at least one of the two child features. The language product owner chooses the `InputOutputAutomata` feature to obtain a valid configuration. The language component's grammar (*cf.* Figure 17) enables to model input/output automata for specifying component behavior. Transitions (ll. 6-7) of such automata consist of guards (l. 8) and port assignments (l. 9). The production's implementations remain underspecified (ll. 8-9). Thus, these are modeled as interfaces by the feature's language component and must be bound by the `InputOutputAutomata` feature's child features. The language component further defines two context conditions, which ensure each input/output automaton contains exactly one initial state and that state names start with capital letters. The grammar's `IOAutomaton` production is embedded into the `BehModel` production of the language component's parent feature. The `JavaExpression` (*cf.* Figure 18) feature is a mandatory child feature of `InputOutputAutomata` and therefore has to be selected by the language product owner. Its grammar inherits the productions from a Java grammar (l. 1) and defines two new productions `GuardExpr` and `PortAssExpr` (ll. 2-3). These enable modeling guards and port assignments with Java expressions. The `Expression` production is part of the inherited Java grammar. The two productions are bound to the `Guard` and `PortAssExpr` interfaces exported by the `InputOutputAutomata` language component. Introducing two new productions in contrast to directly binding the Java `Expression` production to the `Guard` and `PortAssignment` productions enables to separately handle guards and port assignments as they are distinguishable via their types. This is necessary, for instance, if a context condition either only restricts the syntax of port assignments or of guards. The first two context conditions of the feature's language component, *e.g.,* only restrict the well-formedness of expressions used in port assignments (*cf.* Figure 18, ll. 4-5), whereas the third context condition only restricts guard expressions (*cf.* Figure 18, l. 6). The fourth context condition affects guards as well as port assignments. Composing the four features as described in Section 5 yields the language component depicted in Figure 19 that represents the composed language. The grammar is composed of the grammars of the selected features' language components by applying the transformation described in Section 5. The new language component's context conditions are all context conditions of all selected language components. The new language component retains all extension points defined by at least one selected language component. Figure 20 depicts the three models `valid` (ll. 1-8), `invalid1` (ll. 9-17) and `invalid2` (ll. 18-24). The model `valid` is a well-formed model of the new language. The productions for modeling component and port declarations (ll. 1-3) originate from the `ADLGrammar` (*cf.* Figure 15). The `InputOutputAutomata` language component's grammar (*cf.* Figure 17) adds the possibility to declare automata, states, and transitions (ll. 5-7) through extending the interface added by the `ComponentBehavior` language component (*cf.* Figure 16). The expressions `true` and `in = out` used in the transition's guard and port assignment (l. 7) originate from the `JavaExpression` language component (*cf.* Figure 18). The models `invalid1` and `invalid2` are no well-formed models of the new language. Model `invalid1` is an element of the language defined by the new grammar but not well-formed because it violates the three context conditions `PortNamesUnique`, `StateNamesUpperCase`, and `GuardExprBoolean`, which are defined by the three language features `BaseADL`, `InputOutputAutomata`, and `JavaExpression`, respectively. The model `invalid2` is not well-formed because the embedded `ioautomaton` model is not possible in the `In-`

CompoundADL valid

```
component MyComponent {
  port Integer in;
  port Integer out;
  ioautomaton {
    initial state S1; state S2;
    transition S1 [true] {in = out} S1;
  }
}
```

CompoundADL invalid1

```
component MyComponent {
  port Integer in;
  port Integer in; // PortNamesUnique
  ioautomaton {
    initial state s1; // StateNamesUpperCase
    transition S1 [1+1] // GuardExprBoolean
                  {in = out} S1;
  }
}
```

CompoundADL invalid2

```
component MyComponent {
  port Integer in;
  port Integer out;
  ioautomaton {
    init state S1; // parse error
  }
}
```

Figure 20: A valid model and two invalid models of the language component depicted in Figure 19.

`putOutputAutomaton` feature's grammar.

Based on the integrated language component, MontiCore generates model-processing infrastructure to parse models and perform well-formedness checks. Via handwritten extensions, this infrastructure can be further customized.

7. Discussion

The presented approach relies on grammars as descriptions of concrete syntax and abstract syntax. This limits the processable models to be textual. However, building a graphical concrete syntax on top of a textual one is possible [65]. Also, our approach currently only realizes *language embedding* for composition of language components, *i.e.,* a form of composition that produces abstract and concrete syntaxes integrated into a single model. Supporting further forms of language composition, such as language aggregation [67], is subject to further research. The realization of our concept with MontiCore uses *all* interface productions of a grammar as extension points. It is possible to reduce this to a subset of these by explicating the extension point that should be "exported" [8] within a language component model. We introduced this concept to mitigate accidental or unintended extension points when using interface productions for technical reasons. However, our experiences have shown that defining exported extension points explicitly, in practice, was cumbersome and using all interface productions as extension points did not produce problems. To this effect, we omitted the explicit statement of extension points but delegate the decision of when to use interface productions over alternatives to the language (component) engineer. Using disjunctions instead of interface productions can be used to avoid creating unintended extension points. The current realization of our concept does, however, prescribe designers of language components to foresee all extension points by explicating these through interface productions in the grammar. For example in Figure 1, `Condition` and `Action` are the only extension points of the grammar `Automation`. Therefore, extension of `Automation` is restricted to new trigger conditions and actions in the two foreseen places. Technically, MontiCore enables to override any grammar production in order to add new alternatives to their right-hand side [67], which can be leveraged to realize extension points as well. However, we currently do not make use of this to obtain a cleaner abstract syntax and to distinguish which productions are extension points and which are not to foster hiding of internal details of language components. Our concept to syntactic language variability relies on underspecification in the abstract syntax through qualified extension points. These can, for instance, be realized through abstract classes or interfaces in metamodel-based languages [73], through controlled merging of abstract syntax elements [17], or underspecification in grammars, such as binding elements of different languages by name [76].

Designing language components in an appropriate granularity is challenging: If the components are too fine-grained, the feature model becomes complex even for small language product lines. Further, the constituents of languages are scattered across many different language components, which also complicates the understandability and maintainability of these individually. Furthermore, it is unlikely that all language components can be developed independently. However, fine-grained components facilitate the reuse of components in different contexts. Coarse-grained language components, on the other hand, produce feature models that are better readable and reduce scattering of language components, but become more complex. The appropriate granularity is subject to the language engineers. Using a grammar production that is not the start production of the grammar as extension to an extension point cuts off all parts of the language's concrete and abstract syntax that are not reachable as child elements of the abstract syntax induced by the new start production.

Ultimately, our approach flattens the tree structure of the feature model and produces a single composed grammar that directly inherits from the grammars of all selected features and, thereby does not introduce new inheritance relationships between the grammars of the selected features. To this effect, we can allow inheritance dependencies between the grammars contained in different features of a selected variant. Such a dependency should be indicated in the feature model as a *requires* constraint between the feature of the extending grammar and the feature of the extended grammar. As with the composition no additional inheritance relationships are introduced to the grammars of the selected features, inheritance is acyclic if the *requires* constraints in the feature model are acyclic.

Our approach synthesizes a new grammar that integrates the individual grammars of the language components via inheritance. This layer of indirection complicates the readability and understandability of the language syntax. Therefore, the generated grammar is not useful as documentation of the syntax. Through a model-to-model transformation of these grammars, the inheritance relations can easily be flattened to produce a single integrated grammar with improved readability.

The separation of concerns in our approach usually alleviates

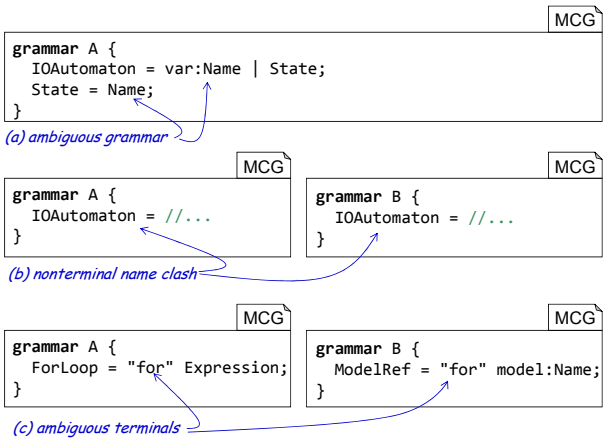


Figure 21: Grammars may be ambiguous (a). Composition of grammars can yield (b) name clashes of nonterminals with the same name and (c) terminals that make parsing ambiguous.

language product owners from being SLE experts and decouples engineering of language components from their composition. In our current work, it is necessary that the language product owner is an SLE expert only if she performs handcrafted, variant-specific customizations. This can be the case, *e.g.,* to customize further tooling such as editors. It is, however, impossible to completely alleviate the language product line engineer, who maintains the composition of language components, from being a software language engineer [78, 77].

MontiCore uses the mechanisms of ANTLR [62] to handle ambiguous grammars, *e.g.,* the grammar depicted in Figure 21 (a). Besides the automatic mechanisms, MontiCore grammars can contain semantic predicates [62] of ANTLR to manually control handling of ambiguities. Furthermore, the composition of independent language components can raise several issues, which are taken care of by the implementation of our approach. As the grammars of different language components typically are developed independent of each other, there might be conflicts in the nonterminals that share the same name unintentionally (*cf.* Figure 21 (b)). With grammar languages that have a hierarchical name space of nonterminals, the qualified names of the nonterminals would differ. For grammar languages with a flat name space (such as the MontiCore grammar language), a transformation of the grammars has to rename conflicting nonterminals to resolve name clashes. As renaming a nonterminal also influences tooling written against this nonterminal, the language composition engineer has to adapt this manually. The realization of our approach based on MontiCore checks whether such name clashes of nonterminals exist and aborts derivation of the language variant on name clashes. Furthermore, the generated parsers may be confronted with ambiguities caused by terminals with the same name that both can occur at a certain place in the model (*cf.* Figure 21 (c)). The parser has to be aware of this, as otherwise two different valid abstract syntax trees could be instantiated from the same model. Currently, our realization with MontiCore aborts generation of a parser if such terminal ambiguities occur. Cross-tree constraints in the feature model restrict valid variants. In our approach, an *excludes* constraint can indicate that the composition causes ambiguities or it indicates that the language product line engineer forbids the composition due to other reasons. This is similar to *requires* constraints, which can either have technical reasons or design intentions. Future work should investigate how to derive such constraints that are due to technical reasons or cause ambiguities from the grammars.

Composition of context conditions can prevent other context conditions to be applicable if the syntactical elements that these operate on are forbidden by another context condition. Further, through extension of the right-hand side of the grammar production of a nonterminal N with a further alternative, a context condition for N also applies to the new alternative. This might be unintentional. For example, the language component `ComponentBehavior` of the case study presented in Section 6 could include a context condition checking that each component behavior declares a name starting with an upper case letter. Later, the language product line is extended with a further language component arranged as child feature of `ComponentBehavior` that enables a different way to specify component behavior. The context condition then also applies to this new language component, which might be unintended. This situation requires to either adapt the context condition to exclude the newly added alternative or to replace it with a new one.

For language product lines with manageable size of involved language components, the approach helps to structure these and the feature model is understandable and well extensible. For large feature models, and language product lines, respectively, it is subject to further research how well the approach scales up.

Through conservative extension implemented by the grammar composition operator, reusability of tooling is increased. If this condition would not hold, tooling written for a certain variant could only guarantee to operate on models of this single language variant. With conservative extension, the set of models that are valid with regard to a language variant A are a subset of the valid models with respect to a language variant B if the set of selected features of A is a subset of the set of selected features of B. As depicted in Figure 22, through conservative extension, tooling of B can be used on models valid in A, because new language syntax can only be integrated by adding further alternatives to existing syntax elements. This property breaks when well-formedness rules are taken into account, for instance, if a well-formedness rule considers those models that do not use a new alternative as invalid. Considering only well-formedness rules (and not the underlying syntax), the relation between valid models A and B can be carried out into the other direction: If through additional language components in the language variants no existing rules can be eliminated, only new rules can be added. As with each new rule, the set of valid models can only decrease in size, valid models of B are a subset of valid models in A. However, well-formedness rules are checked after parsing and therefore, each activity realized before well-formedness checking benefits from the conservative extension. With subsequent activities in the model-processing pipeline, such as interpretation or code generation, the prop-

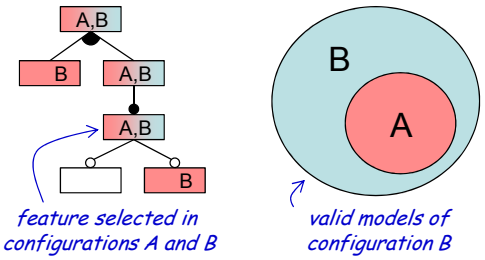


Figure 22: If the set of features contained in configuration *A* is a subset of the features contained in configuration *B*, the valid models of the language variant derived from *A* are a subset of valid models of the language variant *B*.

erty of conservative extension for language components does not hold anymore. With parsers, well-formedness checks, and further analyses and transformations that operate on the abstract syntax of (potentially ill-formed) models, the conservative extension property ensures proper reusability.

The presented approach has no explicit types for different kinds of language components. Besides that, it is questionable in which dimension typing languages should be carried out (*e.g.,* imperative or declarative languages, expression language or behavior language, language with code generators translating to the same language, ...), it could limit reusability of a language component for a different purpose. The advantage of a type system of language components is obvious: if many language components are available for similar purposes, typing these would limit the choice between finding a suitable one. Further, typing could reduce the knowledge about a language component's content required by language product line engineers. Instead of an explicit type system for language components, the extension points defined in a language component make assumptions about the language components that can be employed to deliver possible extensions.

Many other approaches imply that either the feature model is developed before the assets (top-down approach) or the assets are available and the feature model is built or derived for existing assets (bottom-up approach) [49]. Due to the loose coupling between the feature model and the language components of our approach, both paradigms are supported. While in our current work, we focus on a bottom-up approach, future work should investigate developing a language product line top-down.

The realization of our approach leverages MontiCore grammars as integrated descriptions of the concrete and abstract syntax of language components. Due to the inherent coupling between these in being defined within the same grammar rules, the realization does not support pure presentational variability [11], *i.e.,* variability within the concrete syntax only.

Another challenge arises from composing not only syntax but also the languages' behavior realizations (if available), which usually have the form of code generators [5, 14] or interpreters [6, 80]. Composition of both has been achieved for specific languages [64, 66] or under various restrictions [4, 40]. We presented a first approach for an integration of code generator composition into our concept for language product lines in [9]. However, this concept has limitations that have to be mitigated and a general approach towards code generator composition yet remains to be conceived.

According to [43], a good language extension framework must support independent language extensions and automated composition of these. Our approach supports both, but is even more powerful than a language extension framework, as it leaves the choice of a base language open to the language product line engineers, instead of being restricted to a fixed base language. Further, [43] states that composition must not produce a corrupted compiler. As with the transformations of conflicting grammar productions described above, and the conservative extension property, our approach guarantees to produce either an uncorrupted parser or no parser. With context conditions and the translation of a compiler (which would be carried out to MontiCore code generators in the realization of our approach), we cannot guarantee this anymore. Future work should investigate extracting language product lines from existing handcrafted, cloned-and-owned language variants.

8. Related Work

Variability within software languages has been investigated in a wealth of different approaches [56]. Modeling languages are usually engineered by means of language workbenches, for which different techniques of reusing and composing languages exist [21]. Further, language workbenches differ in the language constituents they process and the constituents of the languages they produce. Several language workbenches, including Rascal [79], MontiCore [30], Neverlang [76], Spoofax [45], and Xtext [5] define syntax via grammars. Other language workbenches employ metamodels for syntax definitions, including EMF [73], GEMOC Studio [17], or MetaEdit+ [75], or employ projectional editors, such as MPS [89]. Well-formedness rules are usually either implemented with OCL [32] or as GPL statements [30]. The following considers related work for each research question (*cf.* Section 4) individually.

RQ1: *Variability within language module*
With language definitions that employ metamodels for syntax definitions, concrete syntax often plays a minor role. To this effect, metamodel-based approaches typically support variability in abstract syntax only [38, 63, 90] while grammar-based approaches often consider concrete syntax and abstract syntax [53, 57, 76]. The language development framework Neverlang [76] supports modular language definitions. A language module comprises a grammar as syntax definition and a corresponding ordered list of evaluation phases. These evaluation phases realize the semantics of a language module and include type checking, well-formedness checking, and code generation. Extension points in Neverlang grammars are realized as placeholders, which are unused nonterminal names on the right-hand sides of grammar productions. Compared to our approach, these cannot prescribe the presence of certain abstract syntactical elements, enabling easier reusability, but bearing higher complexity in finding a module providing a suitable extension.

Further, explicating extension point through, *e.g.,* interface nonterminals reduces the risk of defining extension points unintendedly (*e.g.,* by misspelling a nonterminal).

The revisitor approach [52, 51] uses Ecore metamodels for describing the syntax of language modules. It enables to describe variability within a language's metamodel and the realization of its semantics by using the revisitor pattern. Extension points in a metamodel are realized as required metamodel elements.

The set of languages mbeddr [84, 85] is built upon MPS [83] and thereby, uses its projectional editors and further sophisticated tooling. In MPS, abstract syntax elements can extend other abstract syntax elements to perform language extension. Therefore, every abstract syntax element is a potential extension point.

Action-semantics modules [18] leverage context-free grammars to describe the syntax of such modules. To realize extension points, context-free grammars can define unused nonterminals. LISA [57] uses attribute grammars to describe the concrete syntax, abstract syntax, and semantics of language modules. LISA uses inheritance as known from object-oriented programming to realize language inheritance on attribute grammars. In principle, every rule of the grammar can be extended by new rules. The combination of SDF and FeatureHouse realizes compositional language modules [53] containing grammar rules, typing rules, and evaluation rules. Variability is carried out into two dimensions: the dimension of extension with new language concepts and the dimension of extension with new tooling (*e.g.,* new typing rules). For realizing the variability in the grammar rules, the SDF modularization is used and Spoofax [45] is employed to generate parsers. Silver [93] is an extensible attribute grammar system. It uses attribute grammars modules whose syntax and semantics can be extended.

RQ2: *Composition of language modules*
Our approach uses composition to derive a language variant, and therefore the related work focuses on compositional approaches as well. However, there are approaches that define 150% metamodels and use negative variability to reduce these it to obtain language variants [90].

Neverlang supports several forms of composition for the syntax of language components [76]. Language extension relies on composition of the grammars, where one component provides all implementations that another component requires. Language unification employs glue code to match required and provided implementations that do not match originally. The approach for development of languages with action semantics modules [18] relies on composition of the grammar productions for composing syntaxes. In this, the unused nonterminals serving as extension points can be implemented by importing language modules that realize these unused nonterminals. To the best of our knowledge, the approach does not have a mechanism to compose independent language modules for building language product lines with an explicit variability model.

SugarJ [20] enables to specify syntactic extensions to Java. These extensions are contained in syntactic sugar libraries. By "desugaring", the extended syntax is transformed into the base syntax. SugarJ uses parsers that are capable of detecting ambiguities, on which they report an error.

The language framework ableC [42] is an extensible C language implemented in Silver [93] that uses Copper as parser generator. ableC uses attribute grammars for describing the syntax of independent language extensions and provides different composition mechanisms for these extensions. The mechanism satisfies several criteria that make it expressive enough to provide a solution to the expression problem. Their reliable and automatic composition mechanism guarantees correct composition of attribute grammars and, therefore, also related analyses and transformations. The base language C, however, cannot be exchanged. Silver [93] uses an "import with syntax" mechanism to compose the syntax of attribute grammars. The semantics of the import is that rules of imported grammars are as if they were specified in the importing grammar.

Wyvern [61] is a further extensible language system that guarantees reliable composition. It uses delimiters (*e.g.,* braces, whitespace) to coordinate parsing between base language fragments and language extension fragments. Our approach investigates language composition on a more abstract level as the above-mentioned approaches. It is based on MontiCore that itself generates parsers by employing ANTLR [62]. The detected ambiguities are therefore limited to those directly accessible in the grammar (*cf.* Section 7).

In the revisitor approach [52], extension points do not have to be foreseen by language developers at design time and composition leverages the revisitor pattern. Further, the several extensions can be independent of each other. A binding [51] realizes an adapter functionality by connecting two metamodel elements of different metamodels. Mbeddr [84] includes an extensible set of language modules that have C as their base language. Therefore, it does not support to use a different base language than C, which limits its applicability for, *e.g.,* pure model-driven development. mbeddr supports three types of language composition: language modules can be loosely coupled by remaining in separate artifacts with language referencing. In this form of composition, a language module references a part of another language module only. Embedding realizes a syntactic composition between independent language modules similar to our approach. Language extension is realized by a language module extending the syntax of a host language module, which it must be aware of. mbeddr mostly relies on language extension to realize composition of language modules. LISA [53] uses an inheritance mechanism to compose attribute grammars. SDF+FeatureHouse [53] employs techniques of FeatureHouse to compose language modules with superimposition, weaving, or inheritance as composition operators.

RQ3: *Meaningful combinations of language modules*
Copper uses modular analyses [70] to verify that the composition of grammars will result in a valid grammar and a deterministic parser. These analyses are carried out independently on each extension to a base grammar and, if they are fulfilled, guarantee several properties.

Several approaches propose to employ feature diagrams to arrange language modules in a form that restricts possible com-

binations [37, 49, 50, 53, 78, 90]. There is an extension to Neverlang [78] that uses the common variability language to organize language modules and their interrelations. Another extension, AiDE [50], builds upon Neverlang and derives variability models from an existing landscape of interrelated Neverlang modules. The addressed use case of this is to post-hoc derive a language product line from an existing set of language modules used for a specific purpose. To do so, AiDE first extracts all dependencies between employed input language modules and then synthesizes a feature model. AiDE can be leveraged to describe language product lines but is not capable of developing language modules independently, and then build up a language product line of these.

To the best of our knowledge, the organization of language modules in mbeddr [84], LISA [57], action-semantics modules [18], the revisitor approach [51], and ableC [42] do not support building dedicated language product lines with controlled arrangement and interrelations between employed language modules.

RQ4: *Separation of concerns*
A separation of concerns between different actors or roles that are involved in developing a language product line, as described in Section 4.3 is only described for some of the approaches. Neverlang [78] distinguishes between language developers and domain experts who are involved in the process of developing language product lines. While language developers create language.

Wyvern [61] and Copper [92] completely alleviate composition engineer from understanding the details of the components. in mbeddr [84], any language engineer familiar with MPS can create language extensions. The decision which language modules should be developed and when these are engineered should be coordinated for rather central extensions used by many people of, *e.g.,* an organization. For small extensions, this is less relevant. In Argyle [37], DSLs are constructed from language assets. A feature model is created during domain analysis. The DSL user is a programmer and selects necessary functions to fulfill requirements of the target DSL.

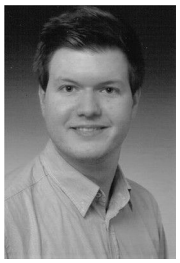
9. Conclusion

We presented a concept to engineer and maintain syntactic language features independent of another within language components. The language components inherit the extension points of the grammar they contain and enable controllable and systematic composition through these. This facilitates engineering new languages by reusing existing components instead of recreating their concepts and related artifacts from scratch. To guide composition of language components and liberate language engineers and modelers from comprehending the internals of all participating components, we propose to relate the language components through feature models. This enables defining product lines of languages that can be arranged by dedicated language product line engineers, which ensure that only valid (in a subjective sense) language products can be derived. Based on a feature configuration, language product owners can easily compose a new language from existing components without language expertise. This, ultimately, facilitates the engineering, maintenance, and evolution of software languages.

Vitae

Arvid Butting received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2014 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, software architectures, and model-driven development.

Robert Eikermann received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2012 and 2014. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software language engineering, behavior languages, and model-driven software development.



Oliver Kautz received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2014 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Department of Software Engineering at RWTH Aachen University. His research interests cover software engineering, software language engineering, software architectures, model-driven software development, and modeling language semantics.

Bernhard Rumpe is chair of the Department for Software Engineering at the RWTH Aachen University, Germany. His main interests are software development methods and techniques that benefit from both rigorous and practical approaches. This includes the impact of new technologies such as model-engineering based on UML-like notations and domain-specific languages and evolutionary, test-based methods, software architecture as well as the methodical and technical implications of their use in industry. He has furthermore contributed to the communities of formal methods and UML. Since 2009 he started combining modeling techniques and cloud computing. He is author and editor of eight books and editor-in- chief of the Springer International Journal on Software and Systems Modeling. See `http://www.se-rwth.de/topics/` for more.

Andreas Wortmann received his Ph.D. from RWTH Aachen University in 2016. Currently, he is a tenured researcher at the Department for Software Engineering at RWTH Aachen University. His research interests cover software engineering, software language engineering, model-driven development, and robotics. He is a member of IEEE and its Technical Committee on Software Engineering for Robotics and Automation and serves on the board of the European Association for Programming Languages and Systems (EAPLS).

References

[1] Kai Adam, Arvid Butting, Oliver Kautz, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Type-safe Interfaces into Template-based Code Generators. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*, pages 179 – 190. SciTePress, January 2018.

[2] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, sep 2003.

[3] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[4] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software*, 11(5):89–94, September 1994.

[5] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[6] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 84–89. ACM, 2016.

[7] 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.

[8] 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.

[9] 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.

[10] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017*, pages 53–70. Springer International Publishing, 2017.

[11] María Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within modeling language definitions. *Model Driven Engineering Languages and Systems*, pages 670–684, 2009.

[12] 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.

[13] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018.

[14] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In *International Conference on Software Product Lines*, pages 266–283. Springer, 2004.

[16] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. An Architecture Description Language. In *Architecture Description Languages*, pages 181–195. Springer, 2005.

[17] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015.

[18] Kyung-Goo Doh and Peter D Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.

[19] Matthew Emerson and Janos Sztipanovits. Techniques for Metamodel Composition. In *OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[20] Sebastian Erdweg, Lennart CL Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Library-based Model-driven Software Development with SugarJ. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 17–18. ACM, 2011.

[21] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In *Software Language Engineering*. Springer International Publishing, 2013.

[22] Moritz Eysholdt and Heiko Behrens. Xtext - Implement your Language Faster than the Quick and Dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.

[23] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In *SLE*, pages 316–326. Springer, 2010.

[24] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[25] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing Ltd, 2013.

[26] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, 2007.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[28] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In *FMOODS/FORTE*, pages 152–166, 2009.

[29] 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.

[30] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, Angers, France, 2015. Scitepress.

[31] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[32] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating ocl and textual modelling languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2010.

[33] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1793–1798, 2015.

[34] C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[35] 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.

[36] Andreas Horst and Bernhard Rumpe. Towards Compositional Domain Specific Languages. In *Proceedings of the 7th Workshop on Multi-Paradigm Modeling (MPM'13)*, pages 1–5. Citeseer, 2013.

[37] C. Huang, A. Osaka, Y. Kamei, and N. Ubayashi. Automated dsl construction based on software product lines. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 1–8, 2015.

[38] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, 14(2):905–920, 2015.

[39] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-Based Software Engineering. In *International Colloquium on Theoretical Aspects of Computing*, pages 70–95. Springer, 2005.

[40] Sven Jörges. *Construction and evolution of code generators: A model-driven and service-oriented approach*, volume 7747. Springer, 2013.

[41] Frédéric Jouault and Jean Bézivin. Km3: a dsl for metamodel specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[42] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.*, 1(OOPSLA):98:1–98:29, October 2017.

[43] Ted Kaminski and Eric Van Wyk. Creating and using domain-specific language features. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 18–21. ACM, 2013.

[44] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[45] Lennart CL Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *ACM sigplan notices*, volume 45, pages 444–463. ACM, 2010.

[46] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages using Metamodels*. Pearson Education, 2008.

[47] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Softwware Engineering Methodology*, 14(3):331–380, July 2005.

[48] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[49] Thomas Kühn and Walter Cazzola. Apples and oranges: comparing top-down and bottom-up language product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 50–59. ACM, 2016.

[50] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Software Product Line Conference*, pages 71–80. ACM, 2015.

[51] Manuel Leduc, Thomas Degueule, and Benoit Combemale. Modular Language Composition for the Masses. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 47–59. ACM, 2018.

[52] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs Van Der Storm, and Olivier Barais. Revisiting Visitors for Modular Extension of Executable DSMLs. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, Austin, United States, September 2017.

[53] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[54] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 2013.

[55] Nenad Medvidovic and Richard N Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[56] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. Leveraging software product lines engineering in the development of external dsls: A systematic literature review. *Computer Languages, Systems & Structures*, 46:206–235, 2016.

[57] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9), 2013.

[58] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. Weaving variability into domain metamodels. *Model driven engineering languages and systems*, pages 690–705, 2009.

[59] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[60] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J.-M. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, volume 1118 of *CEUR*, pages 15–24, Miami, Florida, USA, 2013. CEUR-WS.org.

[61] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely Composable Type-Specific Languages. In *European Conference on Object-Oriented Programming*, pages 105–130. Springer, 2014.

[62] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[63] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. Composing visual syntax for domain specific languages. *Human-Computer Interaction. Novel Interaction Methods and Techniques*, pages 889–898, 2009.

[64] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014)*, volume 1319 of *CEUR Workshop Proceedings*, pages 66 – 77, York, Great Britain, July 2014.

[65] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In Stefan Wagner and Horst Lichter, editor, *Software Engineering 2013 Workshopband*, volume 215 of *LNI*, pages 155–170. GI, Köllen Druck+Verlag GmbH, Bonn, 2013.

[66] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, 2016.

[67] Bernhard Rumpe and Katrin Hölldobler. *MontiCore 5 Language Workbench. Edition 2017*. Aachener Informatik-Berichte, Software Engineering Band 32. Shaker Verlag, 2017.

[68] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[69] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In *Introduction to Modern Robotics*. iConcept Press, 2011.

[70] August C Schwerdfeger and Eric R Van Wyk. Verifiable Composition of Deterministic Grammars. *ACM Sigplan Notices*, 44(6):199–210, 2009.

[71] Friedrich Steimann, Marcus Frenkel, and Markus Völter. Robust Projectional Editing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 79–90. ACM, 2017.

[72] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[73] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[74] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85, 2014.

[75] Juha-Pekka Tolvanen and Steven Kelly. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[76] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[77] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In *Proceedings of the 18th International Software Prod-*

uct Line Conference, pages 167–176. ACM, 2014.

[78] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. Variability support in domain-specific language development. In *International Conference on Software Language Engineering*, pages 76–95. Springer, 2013.

[79] Tijs van der Storm. *The Rascal Language Workbench*. CWI. Software Engineering [SEN], 2011.

[80] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[81] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[82] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference on*, pages 233–238. IEEE, 2014.

[83] Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1449–1450. IEEE, 2012.

[84] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[85] Markus Voelter, Jos Warmer, and Bernd Kolb. Projecting a Modular Future. *IEEE Software*, 32(5):46–52, 2015.

[86] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. 2011. Shaker Verlag, 2011.

[87] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. {*DSL*} *Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[88] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013.

[89] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–304. ACM, 2010.

[90] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE software*, 26(4), 2009.

[91] 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.

[92] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 63–72, New York, NY, USA, 2007. ACM.

[93] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Electronic Notes in Theoretical Computer Science*, 2008.

[94] Steffen Zschaler, Dimitrios S Kolovos, Nikolaos Drivalos, Richard F Paige, and Awais Rashid. Domain-specific metamodelling languages for software language engineering. In *International Conference on Software Language Engineering*, pages 334–353. Springer, 2009.

Modeling Language Variability with Reusable Language Components

Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
<lastname>@se-rwth.de

ABSTRACT

Proliferation of modeling languages has produced a great variety of similar languages whose individual maintenance is challenging and costly. Reusing the syntax and semantics of modeling languages and their heterogeneous constituents, however, is rarely systematic. Current research on modeling language reuse focuses on reusing abstract syntax in form of metamodel parts. Systematic reuse of static and dynamic semantics is yet to be achieved. We present an approach to compose syntax and semantics of independently developed modeling languages through language product lines and derive new stand-alone language products. Using the MontiCore language workbench, we implemented a mechanism to compose language syntaxes and the realization of their semantics in form of template-based code generators according to language product line configurations. Leveraging variability of product lines greatly facilitates reusing modeling language and alleviates their proliferation.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; *Extensible languages*; *Software product lines*;

KEYWORDS

Language Variability, Language Product Lines, Software Language Engineering

ACM Reference Format:
Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *Proceedings of 22nd International Systems and Software Product Line Conference, Gothenburg, Sweden, September 10–14, 2018 (SPLC '18),* 11 pages.
https://doi.org/10.1145/3233027.3233037

1 INTRODUCTION

Modeling to understand and shape the world is an essential human abstraction technique that has already been used in ancient Greece and Egypt. Scientists model to understand the world and engineers model to design parts of the world. Whilst humans employed modeling for ages and in virtually all disciplines, it is recent that the form of models is made explicit in modeling languages. Computer science has invented this approach to enable a precise understanding of what is a well-formed model in the communication between humans and machines. The general aspiration of such languages creates a conceptual gap between the problem domains and the solution domains that raises unintended complexities [11]. Consequently, research in industry produced a large body of domain-specific languages (DSLs) [37] to match domain-specific needs. With the ongoing digitization of virtually every domain in our life, work, and society, the need for even more DSLs raises. This proliferation raises three questions:

(1) How to create new DSLs that fit specific purposes?
(2) How to engineer DSLs from predefined components?
(3) How to efficiently derive DSLs from other DSLs?

In this paper, we address the second question through reusable language components arranged as a product line of software languages. From these, product owners can configure language products, *i.e.,* variants of the product line, for specific purposes (*e.g.,* domains, applications) without being forced to understand the intricate details of each participating language. Based on the selected language features, its grammars, well-formedness rules, and code generators are composed automatically, such that the result can be used transparently by the modelers. This extends our previous work on syntactic language reuse [4] with composition of code generators. The contributions of this paper, hence, are:

- A concept for syntactic and semantic modeling language variability based on language product lines over language components.
- An extension to our modeling technique for language components combining grammars and well-formedness rules [4] with code generators. Resulting language components are decoupled from a specific language product line and, hence, can be reused in different contexts as well.
- A composition mechanism for code generators of the participating independent languages.
- A realization of our concept with the MontiCore [15] language workbench [10].

With this extensible language variability mechanism in place, new languages can be configured using existing components more efficiently. Hence, the mechanism reduces the effort in engineering software languages for specific contexts as well as the proliferation of modeling languages.

In the following, Section 2 motivates the benefits of our approach and Section 3 presents preliminaries. Afterwards, Section 4 presents

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3233037

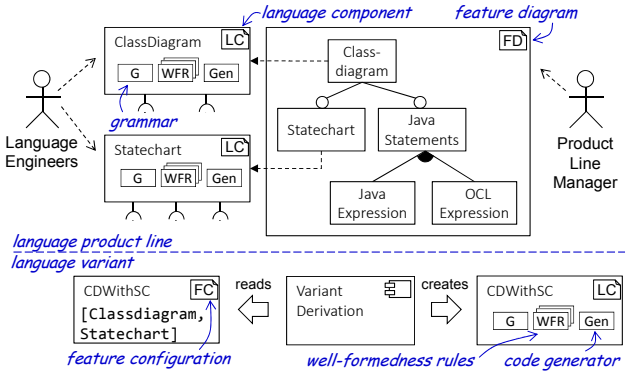


Figure 1: A language product line with a selected variant.

our language variability concept. Section 5 then describes code generators and our mechanism for their integration and Section 6 leverages these to describe language product lines. Section 7 presents an in-depth example, before Section 8 discusses observations and highlights related work. Section 9 concludes.

2 BACKGROUND AND EXAMPLE

Conceptually, there are various techniques to combine two languages, *e.g.*, through merging into embedded models [6] or integration of separate models [15]. Moreover, languages are rarely homogeneous artifacts, but often their definition requires the interaction of multiple meta-languages and programming languages. Popular combinations of these technological spaces, for instance, are ECore [28] metamodels to describe a languages' abstract syntax with OCL [14] for its well-formedness rules and Xtend [2] for code generators. Alternatives are different forms of grammars [30, 39] with GPL well-formedness rules [22] and dedicated languages for model-to-model transformations [19]. There are almost as many technological spaces as there are language workbenches [10] and most support different language composition mechanisms [9].

Consequently, systematic reuse of languages in a meaningful [16] fashion is nearly as complicated as engineering new languages from scratch and capturing the dependencies and relations between the loosely coupled language constituents requires in-depth language engineering expertise. Leveraging dedicated language components that capture these relations and structuring reuse through variability modeling techniques can greatly facilitate language engineering.

Consider developing a language product line for modeling applications, in which the application structure is modeled with class diagrams and there are different options how method bodies are realized with embedded behavior languages. Explicating the variability of such a language product line through feature models over consolidated language components reduces the complexity of identifying and integrating language constituents. Composition of these is performed systematically and does not require an in-depth understanding of the individual language components. An overview of a language product line, the employed language components, and the derived language variant for the example scenario is visualized in Figure 1. The top depicts the language product line comprising a feature model, where each feature references a language component, *i.e.*, constituents of a language definition with respect to a specific language workbench [10]. Each language component contains a grammar (G), well-formedness rules (WFR), and a code generator (Gen), which together realize the syntax and semantics of the language. Moreover, a language component can define named and typed extension points by underspecifying certain parts of its syntax and semantics. Language components are independent of each other and can be developed by different software language engineers individually. The feature model defines combinations of related language components considered valid by a product line manager. Product line managers are language engineers who create language families as feature models that describe possible characteristics of the family's language products. To this end, they collect relevant language components, assign these to features, and define how these realize extension points of language components of their parent features. The latter is realized by mapping an extension point of a parent feature to an extension in the child feature. Further, product line managers ensure that the employed code generators translate to the same or a compatible target language.

In this example, every language variant of the product line uses class diagrams to model the application's structure. The behavior of method bodies can be modeled with embedded Statecharts or statements of the Java/P [7] action language. These Java/P statements rely on expressions, which are either realized as Java expressions, OCL expressions, or both. The bottom left part of Figure 1 depicts the feature configuration `CDWithSC` defining a language variant. The feature configuration is selected by a language product owner, who is an expert in the domain that the language variant is to be used in. The variant includes the `Classdiagram` and the `Statechart` feature. Given this feature configuration as input, the variant derivation tool derives a language variant by composing the language components of all selected features, resulting in a new language component (depicted at the bottom right). This language component has a composed grammar, aggregated well-formedness rules, and a composed code generator. Afterwards, a language workbench, given this language component as input, produces tooling capable of processing models conforming to the language variant. This typically includes a parser, abstract syntax data structure, a checking infrastructure for the well-formedness rules, and coordination of code generation. The tailored tooling can be used by modelers to develop models conforming to the language variant.

3 PRELIMINARIES

This section presents the language workbench MontiCore [15, 27] and the concept for composition of independent MontiCore grammars as presented in [4].

3.1 MontiCore Language Workbench

Our concept for language variability is realized with the MontiCore language workbench [15, 27]. MontiCore supports development of modular modeling languages. It comprises a grammar modeling language and a tool chain for the efficient engineering of textual languages and their infrastructure (parsers, analyses, transformations, code generators). MontiCore employs context-free grammars

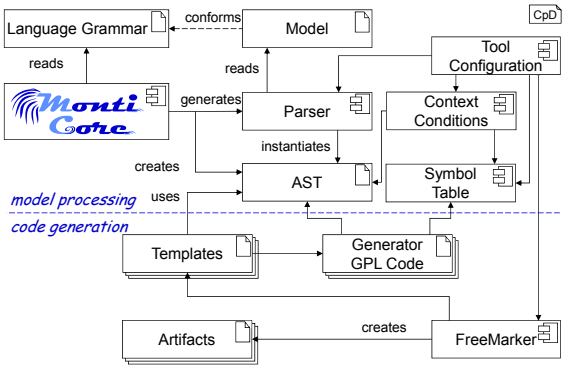


Figure 2: The quintessential components and artifacts of a MontiCore language: From a grammar, MontiCore generates a parser, abstract syntax classes as well as infrastructure for context condition checking and code generation.

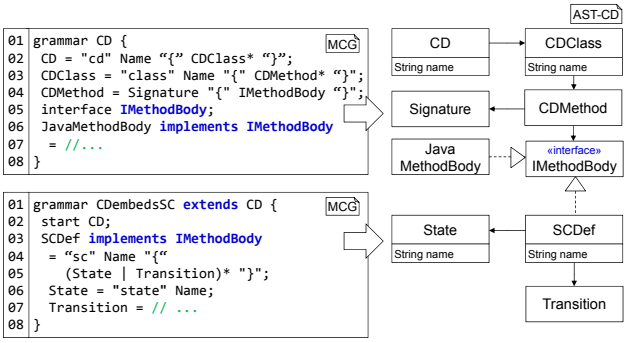


Figure 3: Grammar inheritance in MontiCore.

for integrated definition of abstract and concrete syntax. The grammars describe, which models are principally possible and Java well-formedness rules restrict these. Each grammar contains production rules, may extend other grammars, and yields a dedicated start rule. From the grammars, MontiCore generates model processing infrastructure to parse textual models into abstract syntax trees (ASTs), which store the content of models, such as their elements and their relations. MontiCore supports compositional Java context conditions checking the models' well-formedness that a language-specific, generated visitor applies to the ASTs. Template-based code generators realize the DSMLs' semantics. To this end, MontiCore provides an extensible code generation framework based on the FreeMarker template engine [1]. Figure 2 illustrates the quintessential components and artifacts of MontiCore and their relations.

MontiCore also supports compositional integration of modeling languages through inheritance, embedding, and aggregation [15]. *Inheritance* enables modeling languages to extend and override production rules of their (possibly multiple) parent languages. From inheriting DSMLs, MontiCore produces refined AST classes that inherit from the AST classes of the overridden production rules. MontiCore also features *interface* production rules, which enable underspecification in grammars by prescribing only the required abstract syntax elements of implementations. We leverage this through inheritance to integrate new production rules into these well-defined extension points as depicted in Figure 3.

The grammar `CD` (top left) is an excerpt of a grammar describing textual class diagrams. Each of these class diagram comprises classes that have a name and can contain methods (ll. 2-3). Methods have a signature and a method body, where the latter is realized as an interface production rule that underspecifies a concrete production rule body (l. 5). The interface production rule can be implemented by other production rules, *e.g.*, by the production rule `JavaMethodBody` (ll. 6-7). From this grammar, MontiCore generates six AST classes (depicted top right), out of which `IMethodBody` is an interface implemented by the AST class `JavaMethodBody`. Interface production rules can be used through grammar inheritance. For instance, the grammar `CDembedsSC` (Figure 3, bottom) extends the grammar `CD` (l. 1) and provides further implementation of the interface `IMethodBody`, which in addition to the Java method bodies, features Statecharts as method bodies (ll. 3-5). Accordingly, MontiCore generates new AST classes for newly introduced production rules and reuses all modeling elements of `CD`. The start production of a grammar determines the root of the generated AST and, thereby, also the return type of a parser. MontiCore uses the first production rule of a grammar as start production by default. With the keyword `start` (*cf.* l. 2), MontiCore can be set to use a different production rule than the one of a grammar as start production. In this example, MontiCore is set to use the start production `CD` of the grammar `CD` for the grammar `CDembedsSC`.

3.2 Composing Grammars and Context Conditions

The composition of grammars for achieving language embedding as explained in the last section requires that the embedded grammar has a dependency to the embedding grammar. As one of our concerns is *independent* development of language components, this mechanism is not feasible. Composition of independent grammars relies on syntactic extension points (interface production rules) of a base grammar and a binding indicating, which production rules of the implementing grammar connect to which extension point [4]. To achieve the composition of two grammars, we employ a specific variant of language embedding [15], which combines the syntaxes of the two languages through multiple-inheritance in a generated, third grammar. This new grammar leverages MontiCore's production rule extension and production rule implementation mechanisms to implement extension points of the base grammar with production rules of the embedded grammar. This mechanism especially enables to integrate arbitrary production rules of the embedded grammar into the base grammar and does not require any language developers being aware of this possible interaction. Extending the embedded production rule causes the generated abstract syntax classes of the extending production rules to become subclasses of the classes generated from the extended production rule. Consequently, all model analyses and transformations implemented against the original abstract syntax class can be reused without additional effort.

Embedding, for instance, Statecharts into class diagrams as depicted in Figure 4 requires that the `CD` grammar provides an interface

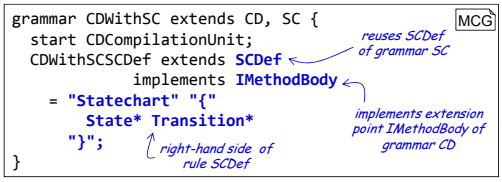


Figure 4: The composition of two independent grammars.

production rule as extension point (here `IMethodBody`) and a binding from a Statechart production rule (here `SCDef`) to the interface production rule extension point. In the example of `CDWithSC`, the binding realizes the integration between method bodies and the definition of Statecharts. This composition enables using the syntax of Statecharts in method bodies of class diagrams to describe their behavior. The grammar generated from this composition extends both the base grammar `CD` and the embedded grammar `SC`. The grammar `CD` is as depicted in Figure 3 and the grammar `SC` is similar to the grammar `CDembedsSC` depicted in the same figure. In contrast to `CDembedsSC`, `SC` does not require to extend the `CD` grammar as it is independent of this. Further, it has no reference to the start production of `CD` and `SCDef` does not implement the interface `IMethodBody`. By generating a new, composed grammar extending both individual grammars, all production rules from these become available in the new grammar. Our grammar composition does not prohibit dependencies between the base grammar and embedded grammar per se [4]. However, embedding production rules of a grammar into a base grammar on which the embedded grammar depends may break the composition, which MontiCore detects.

The integration of well-formedness rules (context conditions) is less complicated. These are realized as Java classes implementing an interface specific to the AST class of the production rule they operate on. As the synthesized production rules extend from production rules of the embedded grammar, the context conditions generally are applicable to these as well. Our integration hence collects the context conditions from both language components and registers these to a generated visitor that applies these accordingly. Through integration of handcrafted code, MontiCore also supports adding inter-language context conditions specific to the integration of both languages that cannot be defined for one language alone [13].

4 A CONCEPT FOR CODE GENERATOR VARIABILITY

Creating a language product line begins with language engineers developing the modeling languages that the language product line combines. We assume the languages are defined in terms of grammars, well-formedness rules, and code generators. Consequently, development begins with the grammars, which prescribe the languages *AS* and *CS*, as well as its possible extension points using, *e.g.*, underspecification, in the grammar description mechanism [4]. Afterwards, the language engineers create well-formedness rules to enable rejection of models not considered well-formed. This usually is necessary as the meta-languages used with common grammar-based language engineering methods lack sufficient mechanisms to describe well-formedness without additional rules. We assume

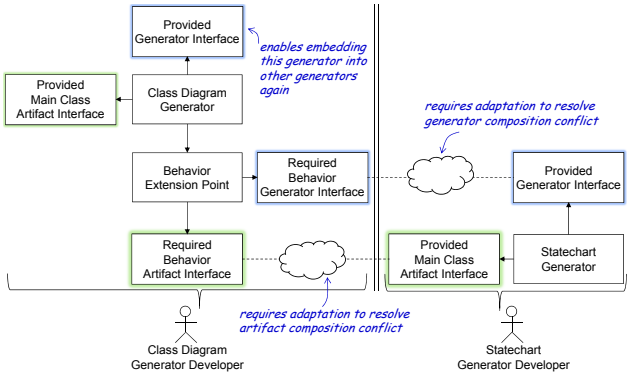


Figure 5: Conceptual representation of composable generators with required and provided interfaces on the example of class diagrams that embed Statecharts.

that the dynamic semantics of a language are realized through code generators that translate models into executable GPL artifacts. Variability of language components consisting of grammar-based languages using Java well-formedness-rules is presented in [4] and briefly recapitulated in Section 3.2.

To extend this notion to code generators, we include a reference to a code generator class into language component models. Based on selected features, we compose the related generators by *embedding* these into another. However, if the language components' generators were not developed for usage in a language product line, the product line manager's feature model raises two kinds of conflicts between features and their immediate parent features:

- *Generator composition conflict:* The code generator of a feature implements an interface not expected for embedding with the generator of the parent feature.
- *Artifact composition conflict:* The code generator of a feature produces artifacts of a type the embedding generator of the parent feature is unaware of.

The language variability infrastructure presented in this paper synthesizes adapters together with a lookup and instantiation framework for both kinds of conflicts based on the feature model and their language components' generator properties. However, the specific adaptation generally is inaccessible to automation as the interfaces on both levels are completely unknown at design time and, hence, may differ significantly. Consequently, the infrastructure generates a framework expecting handcrafted implementation of generator adaptation and artifact adaptation by the product line manager. Adding these two implementations per pair of composed generators enables automated execution of code generation and ensures structural compatibility of generated code by construction.

To resolve the generator composition conflict, generators for grammars with extension points, *i.e.*, underspecified elements, must support similar extension points for responsible code generators. For instance, a code generator for a class diagram grammar supporting embedding various behavior languages for method body implementations must yield an *extension point for generators* responsible for translating behavior models. As the class diagram

generator generally is unaware of any concrete behavior generator it prescribes a code *generator interface* for compatible behavior generators. One possible realization of behavior models could be Statecharts. Statechart generators, however, are generally unaware of the generator interfaces prescribed by the class diagram generator. To compose both generators nonetheless, we also require that each code generator implements a dedicated *code generator interface* itself. Making both, the required and the provided interfaces, explicit enables *automated construction of code generator adapters*. As the specific adaptation cannot be derived, the implementations of the generated adapters must be handcrafted and extend the generated abstract adapter base classes. Our framework then uses these for actual adaptation. Through this, *e.g.,* the class diagram generator can delegate generation of embedded Statechart models to the Statechart generator. This includes to translate available information into parameters that the Statechart generator requires.

This, however, does not resolve the artifact composition conflict, *i.e.,* that the jointly generated code is structurally compatible. The target language of code generation is typically a general purpose language, *e.g.,* an object-oriented programming language. These generally are not expressive enough to define contracts on arbitrary statements, expressions, or blocks. Therefore, we must rely on *structural compatibility between classes* through contracts, *e.g.,* through abstract base classes, implementation of interfaces, *etc.* Enabling class-wise compatibility between code produced by different generators requires that

- each embedded generator produces a *dedicated main class* (which may interact with other classes produced by the same generator);
- embedding generators specify the contract required by possible implementations (*e.g.,* behavior implementations); and
- embedded generators specify the contract provided by the generated main class as its *artifact interface*.

Explicating these provided and required contracts enables generating abstract *artifact adapters* between, *e.g.,* the interface expected for behavior implementations by the class diagram generator and the interface provided by Statechart behavior implementations by the Statechart generator. Similar to the generator interfaces, the actual adaptation requires specific handcrafted implementations to extend the generated adapter base classes, which then are incorporated by the framework automatically. Figure 5 illustrates the generators and interface related to embedding Statecharts into class diagrams. Our method to black-box composition of code generators therefore raises the following requirements:

RQ1 Each participating code generator implements a dedicated *provided generator interface* that describes its usage.

RQ2 Code generators for grammars with extension points support registration of other code generators responsible for translating implementations of these extension points. For each extension point, they describe the *required generator interface* and the *required artifact interface* related to the generated code.

RQ3 Each code generator produces a *single main class* and specifies its *provided artifact interface*.

With this in place, the interfaces of *generator adapters* and of *artifact adapters* (one of each per generator pair to be composed) can

```
01 language ClassDiagram {                    LC
02   grammar CD;                  // selected grammar
03   root CDCompilationUnit;      // start rule of reused language part
04   cocos {
05     CDDiagramNameUpperCase,    // selected context conditions
06     CDClassNameUnique
07   }
08   generator CD2JavaGen;        // code generator for root syntax element
09 }
```

```
01 language Statechart {                       LC
02   grammar SC;
03   root SCDef;
04   cocos {
05     UniqueStateNames,
06     // ...
07   }
08   generator SCJavaGen;
09 }
```

Figure 6: Two language component aggregating class diagram and Statechart language constituents, respectively.

be derived automatically. Provided handcrafted implementations are integrated into the composed generators automatically. We lift this composition to language product lines by applying it to all embedding (*i.e.,* in a parent feature) and embedded (from its child feature) code generators automatically. Hence, the product line manager must implement twice the number of participating code generators as adapters only.

The next section presents the realization of this variability concept that addresses both generator composition conflicts. The subsequent section explains how the product line manager leverages this to create language product lines.

5 COMPOSING INDEPENDENT CODE GENERATORS

The quintessential building blocks for language integration are language components. These aggregate the syntax and the realization of the semantics of a MontiCore language in terms of a grammar (concrete and abstract syntax), Java context conditions (static semantics), and a code generator (realizing dynamic semantics). For instance, Figure 6 illustrates a language component for class diagrams, which references its grammar (l. 2), its well-formedness rules (ll. 4-7), and its generator (l. 8). Optionally, a root production (l. 3) of the grammar can be selected. This enables to reuse only a subtree of the related grammar's abstract syntax (*e.g.,* to reuse only Java expressions of a grammar describing the syntax of Java classes). If no root production is specified, the default start production of the grammar is used. Each interface rule of the grammar becomes an extension point of the language component, where the name of the extension point is the name of the respective interface rule. Interfaces in the grammars therefore are used for typing extension points. Deriving a variant of the language product line entails composing the language components of all selected features. The different language's constituents require different composition mechanisms. The composition mechanisms for grammars and context conditions have been explained in Section 3, this section presents the realization of the black-box composition mechanism for code generators.

Composing code generators is an ongoing challenge that raises the questions of syntactic and semantic conformance. Our approach to code generator composition enables syntactic integration of code generators and generated code, which, by construction, ensures that generators and generated code interact syntactically. Whether behavior of generators or generated code generally is meaningful [16] is as complex as the halting problem [29] and not part of our composition.

Our general approach is to employ adaptation between the explicit interfaces of code generators to enable their interaction at generation time. Through the language components, it is clear which generators must be combined and the grammars of their language components prescribe the generators' extension points (*e.g.,* the generator for `CD` must be able to invoke generators responsible for translating `IMethodBody` instances. Through architectural constraints (such as implementing a single execution interface per generator), abstract adapter classes for the composition of two language components' generators can be generated. Using the generation gap pattern [33], the developer composing two language components with their code generators must provide a proper implementation of the interface imposed by the generated adapter. By construction, the generator of the base language then can call the generators of the embedded languages, pass model parts to these and invoke code generation. Leveraging the assumption that each generator produces at least a main GPL artifact (*e.g.,* a Java class) responsible for interacting with the generated code, the base language generator can produce code instantiating this and using it as intended.

For the integration of generated code artifacts, the embedding generator prescribes for each extension point what it expects from the code produced by generators registered for this extension point within a required artifact interface. For instance, the generator translating `CD` instances could prescribe that code produced for realizations of `IMethodBody` must implement a specific GPL interface. As language components can be developed independent of each other, this expectation, however, rarely is fulfilled. Hence, we impose that all generators also explicate the interface of the main GPL artifact they generated. Generators are mapped transitively to language component extension points for which the generator of the embedding language component prescribes a specific interface. The code produced by the embedded generator provides a mapping between expected GPL artifact interface and GPL artifact interface. Thus, we also can leverage adaptation between these interfaces and generate abstract adapters accordingly. For these, usually one per pair of generators, also a proper implementation must be provided. This also is integrated through the generation gap pattern [33].

For adaptations to work, all generators must be implemented in the same GPL and produce code of the same GPL. The GPLs for code generator implementation and for generated code may differ. Future work on cross-GPL code generator invocation and platform-independent artifact interfaces can mitigate this. Consequently, we make the following assumptions for a realizing the generator composition:

(1) All code generators are implemented in the same GPL and all artifacts are implemented in the same GPL.
(2) Each code generator produces at least one main GPL artifact for which it explicates its GPL interface. The generator ensures that all generated main artifacts comply to this interface.
(3) All extension points of generators have to be explicated at design time of the generator. Generator extension points are typed with the main abstract syntax element they translate (*e.g.,* `IMethodBody`) and the interface expected from a generator registered to translate instances of this element.
(4) Operation of code generators may not rely on assumptions regarding code generated by other generators that are not made explicit through their interfaces.

We assume that each code generator constitutes a generator class (*e.g.,* `CD2JavaGen` and `SC2JavaGen` in Figure 7) realizing the actual code generation. This class performs the code generation, *e.g.,* by invoking a template engine. Each generator class implements an interface describing types of the input and output of the generator (*e.g.,* `ICDGenerator` and `ISCGen` in Figure 7) as presented in [1]. Further, each generator references the type of an interface *typing* the main artifact of the generated code (*e.g.,* `IJavaClassArtifact` and `IStatechart` in Figure 7). For each extension point that is foreseen in the generated code of a generator, an additional class (*cf.* `ExtensionPointInfo`) describes the type of the main artifact in the code that is generated by a generator (*e.g.,* `IBehaviorArtifact`). Further, the additional class describes the interface of the generator producing the main artifact of the generator implementing the extension point (*e.g.,* `IBehaviorJavaGen`). Additionally, an extension point info has a reference to the abstract syntax type (*e.g.,* `IMethodBody`) that is being translated. To this effect, extending an extension point is realized by adapting an extension point interface of the generator defining the extension point to an interface of a generator realizing the extension.

These adapters are necessary to enable decoupled development of the involved code generators. While the interface of the adapter can be generated, it is impossible to automatically generate the implementation of the adapter as it requires in-depth understanding of the behavior and meaning of the generated code. The same mechanism is applied at artifact interface level, where the expected artifact interface of an extension point is adapted to the provided artifact interface of the embedded generator. We combine the classical adapter pattern [12] with a mechanism to integrate a handwritten implementation of the concrete adapter (the TOP mechanism [27]). Each generated adapter is an abstract class that has a *target* (or adaptee) that it implements, which is the (artifact or generator) interface required by the extension point. Further, each generated adapter yields an attribute *delegate* of the (artifact or generator) interface provided by the embedded generator. The class realizing the adapter has to be handcrafted and extends the generated abstract adapter class. The name of the generated adapter, therefore, is fix and can be derived automatically by the embedding generator to interact with (a) the registered, embedded generators, and (b) artifacts produced by these generators. The names of handcrafted adapters are also fix, as these have to be identical to the generated adapter names without the suffix `TOP`. This property simplifies instantiation of the correct adapters, which is explained in Section 7. The above mechanism is applied to all combinations of extension point and extension that are possible within the generators of a language product line to relieve product owners from being language engineers. The developer combining two language components, hence, is responsible for registering the generator of the embedded language component with the generator of the embedding language component. Also, the developer of the embedding generator must be aware of the general existence of adapters.

Considering the example of embedding `SC` into `CD`, a class diagram describes the architecture of an application and Statecharts

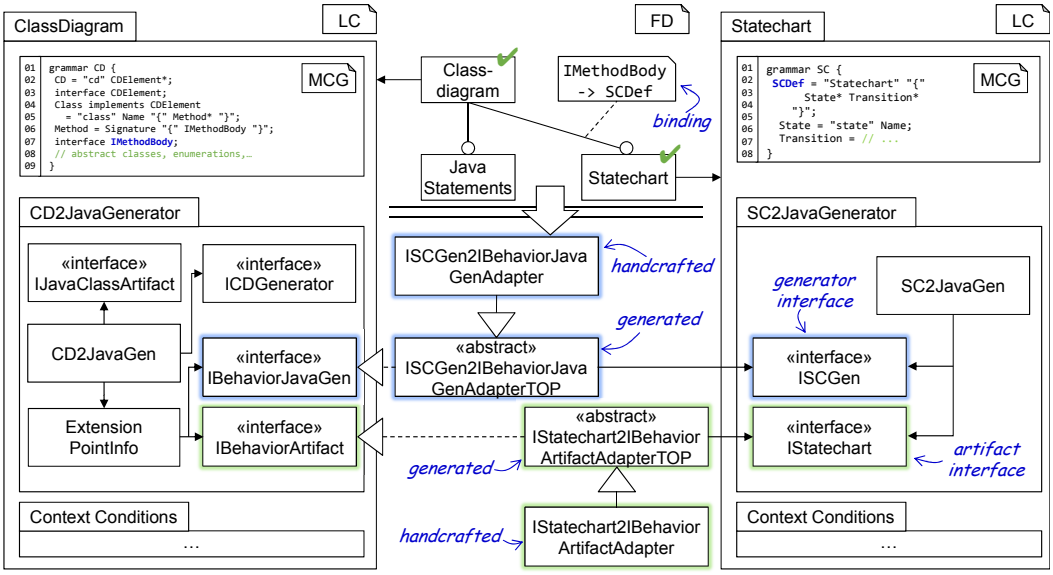


Figure 7: Constituents for a composition of two language components `ClassDiagram` and `Statechart`: The left depicts grammar, context conditions, and a code generator for class diagrams and the right respective constituents for Statecharts. The middle depicts the connection between these independent language components: The top visualizes the feature model and the binding between extension point and extension. The bottom presents the generated adapters and their handcrafted implementations.

can be employed to describe the behavior of methods modeled within the class diagram's classes. Figure 7 depicts the constituents of the individual generators of the class diagram language component and the Statechart language component whose functionality has been explained above. The generator for class diagrams has to be aware of the existence of adapters adapting any (generator or artifact) interface to the required (generator or artifact) interface of each extension point. From the mapping within the language product line, the code generator receives a map from the type of each extension point (*e.g.,* `IMethodBody`) to the concrete type of the extension (*e.g.,* `SCDef`). This information is used to invoke the correct generator. For instance, parsing a class diagram model containing a method body with a statechart results in an abstract syntax tree containing an object of type `CDWithSCSCDef` (*cf.* Figure 4). This type is a subtype of `SCDef` and therefore, the class diagram code generator invokes the generator for Statecharts via the registered generator adapter.

6 FEATURE-ORIENTED LANGUAGE ENGINEERING

This section explains how the variability and composition of language components as explained in Section 5 (the solution space [5]) integrates with the variability model and the derivation of variants in the problem space. With the feature model at hand, all possible compositions of language components are described at product line level. The composition mechanism and the definition of language components are loosely coupled to an employed feature modeling tool, which is therefore easily exchangeable. The set of bindings for a concrete feature model is realized as a model conforming to a dedicated small-scale DSL. The feature model restricts the cardinality of the bindings between several extensions and the extension point. To this effect, it also realizes the differentiation between optional and mandatory extension points. Although in general, dependencies between different language components may lead to unpredicted issues, such dependencies sometimes can be very useful. Hence, the feature model may indicate that a certain feature *requires* another feature, which allows the respective language component to have (*e.g.,* grammar inheritance) dependencies to other language components. Further, the composition of two specific language components can be forbidden using *excludes* in the feature model.

The language product line manager assigns language components to features and aligns the feature model meaningfully. After completing the product line, she generates adapters for all generators and artifacts of language components, which are directly related and implements the adaptations accordingly using the generation gap pattern.

Afterwards, a product owner can configure a language product by selecting desired features. After validating the feature configuration against the feature model, the language components and their constituents are resolved. Based on these, the new grammar that extends from the grammars of all related features is synthesized and for each binding, a new production rule is created that realizes this binding through grammar rule extension and implementation. Based on this grammar, MontiCore generates a parser, AST classes, context condition interfaces, and visitors. Further, all context conditions are collected and a wrapper for the generated context condition checking visitor is generated that parametrizes the latter with the collected context conditions. Thus, parsing and checking models of the language product already is possible.

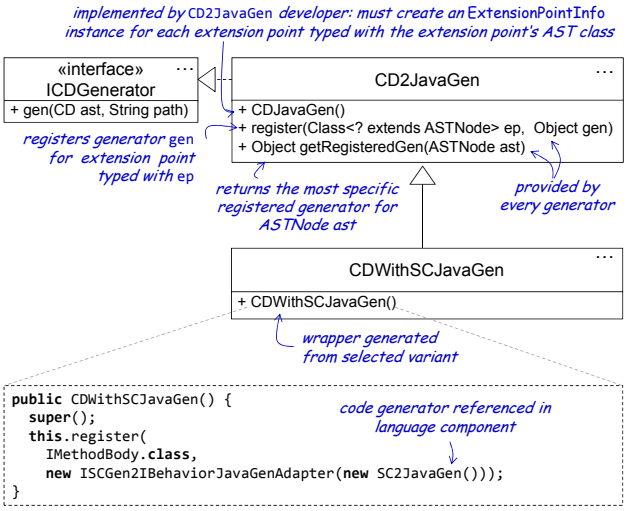


Figure 8: The generator `CDWithSCJavaGen` is generated from the selected variant.

Generators are composed by generating a wrapper for the generator of the feature model's root feature that parametrizes that generator with the other selected generators. This is realized by instantiating the respective (handcrafted) generator adapter class. Where multiple levels of generators are selected, the parametrization is nested accordingly. This wrapper then is registered with the new language model processing framework generated by MontiCore. On artifact level, adapters typically are instantiated within the templates of the embedding generator. As the name of artifact adapters is fixed (*cf.* Section 5) and derived from provided and required artifact interfaces, the name can be calculated within templates. Finally, a new language component is generated that uses the synthesized grammar, the context condition checker wrapper, and the synthesized code generator to process and transform models. This composed language component can be reused, *e.g.,* in a new language product line.

7 EXAMPLE REVISITED

The previous sections explained composition mechanisms for the grammars, context conditions, and generators of language components. A more detailed explanation of the composition of grammars is given in [4]. This section provides an insight into the composition of code generators alongside of the example with the composition of the Statechart language component and class diagrams (Figure 6).

Figure 7 depicts the involved classes of the `CD2JavaGen`, the `SC2JavaGen`, and the adaptation, where the abstract adapters are generated at product line level. The `CD2JavaGen` initializes its extension points in its constructor and yields a method to obtain a registered generator for an extension point based on a given node of the AST as depicted in Figure 8. For a single extension point there might be multiple registered generators, but for a concrete type of ASTNode (*e.g.,* `SCDef`) there must be only a single generator translating it. To use a generator with other generators embedded via generator adapters, a wrapper is generated (*cf.* Section 6). This

```
01 <#-- requires: ASTClass ast, String package, CD2JavaGen cdGen -->          FTL
02 class ${ast.getName()} {
03 <#list ast.getMethods() as met>                          find most
04 <#assign name = met.getMethodHead().getName()>           specific      generate
05 <#assign fqName = package + name>                        generator     artifact
06                                                              ↓             ↓
07 <#assign g = cdGen.getRegisteredGen(met.getMethodBody())>
08 <#assign genClassName = g.generate(met.getMethodBody(), fqName)
09                          .getGenClassName()>
10 IBehaviorArtifact ${name}Body =
11   new ${g.getTargetInterface()
12        .getSimpleName()}2IBehaviorArtifactAdapter(new ${genClassName}());
13                                                                  ↑
14 public void ${name}(Map<Object,Object> _input) {
15   ${name}Body.compute(_input);
16 }                                  method of                    instantiate
17 </#list> <#-- ... -->          IBehaviorArtifact          generated artifact
18 }                                  interface              through adapter
```

Figure 9: Excerpt of a template called by `CD2JavaGen`.

wrapper extends the host generator and in its constructor, registers the employed handwritten extensions of the generator adapters. Each generator adapter has to be parametrized with the concrete instance of its adaptee. In the example, the employed generator adapter adapts the `SC2JavaGen` as realization of an `ISCGen` to an `IBehaviorJavaGen` and registers the adapter at the extension point `IMethodBody`.

If invoked to translate a concrete model, the `SC2JavaGen` executes its main template as depicted in Figure 9. The template produces a new Java class (l. 2) for each class of the class diagram. For each method of the class to translate, it generates an artifact adapter attribute (ll. 10-12) and a Java method (ll. 14-16). The correct artifact adapter is determined according to the embedded generator that translates the specific type of method body (*i.e.,* in the example only `SCDef`) and the fix naming scheme for adapters. The generated method delegates the execution of the method of the class diagram method to the respective artifact adapter. For generating an artifact for an embedded `IMethodBody`, the template retrieves the method-specific generator (l. 7) and executes it to synthesize the artifact (ll. 8-9). Afterwards, the template instantiates the artifact adapter and parametrizes it with a new instance of the generated artifact it delegates to (ll. 11-12).

Figure 10 overviews the artifact adapter infrastructure. The artifact interfaces of the code generators, in the example `IBehaviorArtifact` of the `CD2JavaGen` and `IStatechart` of the `SC2JavaGen`, are handwritten. The notion behind the `IBehaviorArtifact`, for instance, is that the `CD2JavaGen` expects all generators corresponding to productions bound to the `IMethodBody` extension point to produce classes that implement the `IBehaviorArtifact` interface. To this effect, it provides an implementation of the method `compute()` that has the given parameters and return type. `IStatechart2IBehaviorArtifactAdapterTOP` is a generated, abstract adapter class. It implements the interface it adapts to and holds an attribute for the adaptee. When initialized, each adapter has to specify the concrete instance of the adaptee. Concrete adapter implementations have to be handcrafted (cf. Section 5). For instance, the class `IStatechart2IBehaviorArtifactAdapter` is a concrete adapter implementation that extends the abstract adapter and follows the naming scheme. In this concrete adapter, all methods that are required by the interface that the adapter adapts to, have to be implemented while delegating to the methods provided by the adaptee

```java
public interface IBehaviorArtifact {
  public void compute(Map<Object,Object> _input);
}
```
Java

```java
public abstract class IStatechart2IBehaviorArtifactAdapterTOP
                                    implements IBehaviorArtifact {
  private IStatechart adaptee;
  public IStatechart2IBehaviorArtifactAdapterTOP(IStatechart adaptee) {
    this.adaptee = adaptee;
  }
}
```
generated

```java
public class IStatechart2IBehaviorArtifactAdapter extends
                IStatechart2IBehaviorArtifactAdapterTOP {
  public IStatechart2IBehaviorArtifactAdapter(IStatechart adaptee) {
    super(adaptee);
  }

  @Override
  public void compute(Map<Object,Object> _input) {
    String scInput = (String)_input.get("trigger");
    boolean wasFinal = false; int i = 0;
    while(i<scInput.length() || !isFinal)
      isFinal = getAdaptee().updateCurrentState(entry.charAt(i));
    }
  }
}
```
handcrafted

```java
public interface IStatechart {
  public boolean updateCurrentState(char trigger);
}
```

Figure 10: Parts of the artifact adaption infrastructure.

interface. In this example, the compute method has to be implemented to adapt the execution of a method of the class diagram to the execution of a Statechart. The implementation retrieves a parameter `trigger`, which it assumes is present and of type `String`. Then it iterates over the concrete String and updates the state of the Statechart by invoking the method `updateCurrentState()` with the next character of the String, until either the Statechart is in a final state or the input word is processed completely.

The presented approach relieves product owners from being software language engineers completely. A product owner should know the language concepts required for a certain product, but does not require to implement adapters or any form of "glue code". Further, all composition mechanisms applied during derivation of a variant are completely automated. Product line engineers have to be software language engineers as these have to connect extension points to extensions and realize, *e.g.,* the adapters for code generators. Nonetheless, product line managers do not have to be aware of intricate details of the implementations within individual language components, which greatly facilitates reusability of these.

8 DISCUSSION AND RELATED WORK

Our work is a first approach to realize parts of the VCU (variability, customization, use) model of reuse [21] for software languages as sketched in [18]. To this end, we support aggregating language concerns – including syntax and semantic realizations – through features. We therefore currently investigate whether partial configuration of language products is suitable.

Our approach to compose code generators is limited to composing independent generators. Adding, *e.g.,* aspect-like functionality through a single feature is not supported and we are currently investigating this extension. Moreover, our approach limits the variability of language syntax, well-formedness, and dynamic semantics to a single dimension. While this reduces the effort of modeling a language product line, it may require to produce multiple language components that rely on common constituents. For example, if a language can be translated either to Java or C using two different code generators, our approach relies on two different language components. These have references to the same grammar and well-formedness rules, but each employs a different code generator. Our approach also leverages language embedding as the composition mechanism of choice. With this, a loose coupling of languages, in which their abstract syntaxes are not composed – such as language aggregation [15] – is not supported, but subject to ongoing research. Another open challenge is to make language components an active unit of systematic reuse, for instance through inheritance of language components. Some approaches consider code generation as the last step in a pipeline of tools that process a model. In this representation, the language's semantics is realized via applying several model-to-model transformations in a certain order and then translating the transformed model into text by employing a code generator. Currently, our approach does not explicate model-to-model transformations within language components. Considering model-to-model transformations as first phase of executing a code generator, however, is possible.

Our generator variability mechanism ensures compatibility between composed code generators, but cannot guarantee correctness of the generated code. However, the syntactical correctness (*e.g.,* avoiding that two generators generate a file with the same names) can be checked. Also, the restriction to code generators producing standalone artifacts enables a better investigation of their compatibility, but limits their modularity to be coarser grained. While many use cases of code generator composition can be realized following this premise, we are aware of its limitations. For instance, code generators producing, a return statement of method bodies only, are too fine-grained for our composition approach. Future work will investigate how the composition of generators and artifacts can be realized with finer grained modularity of generators.

For composition of generators and generated artifacts, the product line engineer has to provide two adapters per combination of code generators on product line level. If, however, we assume that the product manager is also a software language engineer, the adaptation could be performed later – while deriving the variant. This reduces the number of adapters to be created to twice the number of selected generators. Also, the product owner could perform variant-specific integration of handcrafted code to further customize the other constituents of the language components (such as integration of novel inter-language well-formedness rules). Ultimately, our composition mechanism also relies on the constituents of all language components to be implemented in the same technological space. This prevents, *e.g.,* defining the language components of a single product line in different language workbenches. Such inter-space language definitions also subject to ongoing research.

Future work on the benefits of code generator composition through feature must also investigate efficiency and usability considerations. Our approach requires only little overhead (a language component model per language and the interfaces per code generator) and leverages this to yield black-box composition of reusable code generators. These interfaces are very compact and the language composition merely aggregates existing artifacts. Code generator composition without explicit interfaces for generators and

artifacts requires sophisticated and costly white-box generator investigation. To uncover the benefits of generator variability, future work could investigate and apply metrics regarding size and complexity of artifacts as well as empirical metrics regarding usability.

Research and practice have produced a number of language workbenches, *i.e.,* software tools that support developing and (re)using modeling languages [10]. These language workbenches employ different language definition paradigms, *e.g.,* to (1) define concrete syntax and abstract syntax of languages (usually grammars [2, 15, 30, 32], metamodels [6, 28], or projectional editing [38]); (2) develop the well-formedness rules applied to the abstract syntax (typically OCL [17] or GPL rules [15]); and (3) describe the behavior of models (interpretation [3] or code generation [5]). Due to this wealth of technological spaces and fragmentation in different solution techniques for language development, support for reusing syntactic and semantic language components is rare [31]. Consequently, language reuse is an ongoing research challenge and different approaches [25] exist to address this challenge. Some approaches employ plain negative variability to derive variants of a 150% metamodel [40], which limits their extensibility. There are, however, few approaches addressing both syntax and semantics of modeling languages. The revisitor approach [24] is one of these. It uses a new pattern to enable independent extensions of executable DSMLs covering both metamodel and the realization of the semantics. It supports to extend a language without foreseeing explicit extension points at its design time and reusing language components without recompilation. To the best of our knowledge, it does not support to develop extending language and extended language independent of another.

Neverlang [30] is a language development framework that enables to develop compositional languages components comprising a grammar-based syntax definition and several evaluation phases realizing that, among other things, include type checking and code generation. Extension points in these grammars are placeholders, which are unused nonterminal names. To the best of our knowledge, there is no dedicated typing system for placeholders. AiDE [23], built on top of Neverlang, guides language developers in composition of the language components by extracting dependencies between language components. From these, AiDE synthesizes a feature model for a language product line fully automated. There is an extension to Neverlang using the common variability language for organizing the variability across language components [31]. However, both extensions to Neverlang require dependencies between the language components. Compared to our approach, this limits their reuse in different contexts as language components cannot be developed independently.

The approach presented in [8] enables developing programming languages gradually from independent language modules containing context-free grammars as their syntaxes and action semantics. Action semantics differ from the denotational semantics realized with code generators as presented in our approach. Action semantics modules are built from action notation symbols and as such limited to the expressiveness of the underlying symbols. This yields the advantage that composing semantics modules is more controllable compared to our approach. Further, such symbols are interpretable from different target GPL interpreters. To the best of our knowledge, the approach lacks an explicit variability model to build up product lines of languages.

mbeddr [35, 36] is an extensible set of language modules that builds upon C and the language workbench MPS [34]. It is, therefore, limited to use C as common base language and, to the best of our knowledge, lacks a variability model to properly manage available language modules and their conceptual interrelations. However, it has great usability in terms of editors through MPS. The MPS code generator is extensible by resolving all generator rules and mapping configurations of the individual languages and then building a generation plan. The order of executing the single code generators is specified via priorities. Similarly, ableC [20] is an extensible language framework that builds upon C. It uses Silver and Copper as underlying technologies for the definition of attribute grammars to describe the syntax, and provides several mechanisms to realize composition of these.

In [26], generator composition is realized similar to our approach. Here, each code generator implements one out of three predefined interfaces and provides basic information required by the main generator to call other generators. To this end, each generator yields a model describing its interface-related properties. This approach enables to compose generators, but the composed generators have to implement the predefined interfaces and possible extensions are restricted to existing generator interfaces. The runtime dimension is not covered as all possibly existing generator types are known a priori and, hence, generated source code matches by construction. However, this approach is limited to embedding behavior languages into architecture description languages.

9 CONCLUSION

We have introduced a method to reuse modeling language (parts) through syntactic and semantic embedding of language components. Based on this, modeling language product lines foster systematic reuse of languages and related tooling. Our method relies on abstract syntax descriptions that support underspecification and code generators that, yielding explicit interfaces, produce target GPL artifacts whose contracts (*e.g.,* interfaces), they make explicit. With this, language product line engineers can arrange features representing language components such that various domain-specific language variants can be derived easily. All language engineering efforts (such as integration of inter-language well-formedness rules or code generator adapters) are with the language product line manager. Hence, all composition is transparent to the product line users and modelers. Ultimately, this can facilitate reducing the proliferation of (domain-specific) modeling languages.

REFERENCES

[1] Kai Adam, Arvid Butting, Oliver Kautz, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2018. Retrofitting Type-safe Interfaces into Template-based Code Generators. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*. SciTePress, 179 – 190.

[2] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd.

[3] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 84–89.

[4] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. 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)*. ACM, 75–82.

[5] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[6] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States.

[7] Thomas Degueule, Tanja Mayerhofer, and Andreas Wortmann. 2017. Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support. In *Proceedings of MODELS 2017. Workshop EXE (CEUR 2019)*.

[8] Kyung-Goo Doh and Peter D Mosses. 2003. Composing programming languages by combining action-semantics modules. *Science of Computer Programming* 47, 1 (2003), 3–36.

[9] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. ACM, New York, NY, USA.

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*. Springer International Publishing.

[11] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[13] Timo Greifenberg, Katrin Hoelldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Mueller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. 2015. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[14] Object Management Group. 2010. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01). (2010).

[15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[16] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer* 37, 10 (2004), 64–72.

[17] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. 2010. Integrating OCL and textual modelling languages. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 349–363.

[18] Jean-Marc Jézéquel, Manuel Leduc, Olivier Barais, Tanja Mayerhofer, Erwan Bousse, Walter Cazzola, Philippe Collet, Sébastien Mosser, Benoit Combemale, Thomas Degueule, Robert Heinrich, Misha Strittmatter, Jörg Kienzle, Gunter Mussbacher, Matthias Schöttle, and Andreas Wortmann. 2018. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures* 54 (2018), 139 – 155.

[19] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 719–720.

[20] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages. https://doi.org/10.1145/3138224

[21] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. 2016. VCU: The Three Dimensions of Reuse. In *International Conference on Software Reuse*. Springer, 122–137.

[22] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: a Framework for Compositional Development of Domain Specific Languages. In *International Journal on Software Tools for Technology Transfer (STTT)*.

[23] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, 71–80.

[24] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs Van Der Storm, and Olivier Barais. 2017. Revisiting Visitors for Modular Extension of Executable DSMLs. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*. Austin, United States. https://hal.inria.fr/hal-01568169

[25] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures* 46 (2016), 206–235.

[26] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2014. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014) (CEUR Workshop Proceedings)*, Vol. 1319. York, Great Britain, 66 – 77.

[27] Bernhard Rumpe and Katrin Hölldobler. 2017. *MontiCore 5 Language Workbench. Edition 2017*. Shaker Verlag.

[28] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2. ed.). Addison-Wesley, Boston, MA.

[29] Alan Mathison Turing. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society* 2, 1 (1937), 230–265.

[30] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[31] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, 76–95.

[32] Tijs van der Storm. 2011. *The Rascal Language Workbench*. CWI. Software Engineering [SEN].

[33] John Vlissides. 1998. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. online at http://www.research.ibm.com/designpatterns/pubs/gg.html.

[34] Markus Voelter and Vaclav Pech. 2012. Language modularity with the MPS language workbench. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 1449–1450.

[35] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 121–140. https://doi.org/10.1145/2384716.2384767

[36] Markus Voelter, Jos Warmer, and Bernd Kolb. 2015. Projecting a Modular Future. *IEEE Software* 32, 5 (2015), 46–52.

[37] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. 2013. *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[38] Markus Völter and Eelco Visser. 2010. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 301–304.

[39] Guido H Wachsmuth, Gabriel DP Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *Software, IEEE* 31, 5 (2014), 35–43.

[40] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE software* 26, 4 (2009).

Towards Component-Based Development of Textual Domain-Specific Languages

Andreas Wortmann
Software Engineering, RWTH Aachen University, http://www.se-rwth.de/

***Abstract*—Software-intensive systems are developed with experts of different domains. This requires reifying their domain expertise in software, which raises the need for domain-specific languages (DSLs) to bridge the gap between the problem space of the experts' experience and software development. Developing suitable DSLs still is prohibitively complex due to the lack of pervasive concepts for DSL reuse. Existing concepts either give rise to a conceptual gap between their abstractions and language definition constituents or are tied to specific technological spaces. To mitigates this, we present a novel conceptual model for the systematic reuse of textual DSLs. This technology-independent model promotes modularity and reusability based on language families that exhibit specific reuse interfaces. To realize these concepts, we conceived an extensible modelling infrastructure that supports engineering reusable textual DSLs using the MontiCore language workbench. This enables systematic reuse of textual DSLs for compatible technological spaces from which DSL engineers in many domains can greatly benefit.**

I. Motivation

Society increasingly depends on systems developed by experts of various domains using domain-specific languages (DSLs) [1]. DSLs have become innovation drivers in many disciplines, including automotive, avionics, civil engineering, Industry 4.0, robotics, and software engineering itself. This, *e.g.,* led to the engineering of over 120 DSLs for software architectures [2] used in different domains and various technological spaces [3]. All of these need to be developed, maintained, and evolved on their own, which is costly, error-prone, and hinders progress in the multi-domain engineering of modern software-intensive systems.

Research in software language engineering (SLE) [4] investigates the efficient and reliable engineering, maintenance, deployment, use, and evolution of DSLs to support software engineers and domain experts in efficiently developing future systems. Despite attempts to a systematic SLE, many DSLs are engineered ad-hoc, for very specific challenges, and very limited purposes only [5]. Hence, research has produced a multitude of solutions to facilitate creating DSLs. These include on metamodels [6], grammars [7], or abstract data types [8], interpreters [9] or code generators [10], and well-formedness rules defined in metalanguages [8] or programming languages [11]. For these, the SLE community has proposed various reuse techniques, based on experiences from general software reuse (*e.g.,* polymorphic [12] and parametric [13] reuse, composition [7] or variability [14]). Although these techniques address a wide range of scenarios, most support specific parts of DSL definitions (*e.g.,* abstract syntax or code generators) only and are limited to specific technological spaces. This complicates the engineering and customization of real-world DSLs for different usage scenarios, which ultimately hinders systems engineering with domain experts.

To mitigate this, we present the COLD4TXT conceptual model for component-based language development of textual DSLs that realize behavior with code generators (txtDSLs). In this model, *language components* with explicit interfaces of required and provided grammar rules, well-formedness rules, and code generators are the principal elements of reuse. Feature models arrange these components following their required and provided extension points to language families. Thus selecting features governs how the language components are composed. Based on this model, we present a systematic method to describe and resolve the component's variability as well as their customization.

As the technical realizations of composing grammars, well-formedness rules, and code generators have been presented already [10], [15], this contribution illustrates their conceptual framework consisting of:

1) The COLD4TXT conceptual model for reusable txtDSL components featuring explicit interfaces of required and provided elements.
2) A systematic method for engineering languages based on reusable txtDSL components.
3) A realization of both with the MontiCore language engineering workbench.

With these, reusing language components in different languages families can greatly facilitate engineering DSLs.

In the following, Sec. II motivates our method by example. Afterwards, Sec. III presents txtDSL language components and Sec. IV our method to reuse theses for efficient txtDSL engineering. Ultimately, Sec. V debates observations, Sec. VI discusses related work, and Sec. VII concludes.

II. Example

Consider using architecture description languages (ADLs) [2] – DSLs for the specification of software architectures – for the different departments of a large corporation. In each of these departments, some developers occasionally, maybe once a week, (re-)model parts of a specific software architecture (*e.g.,* of a train, a factory, or a mobile service robot). Instead of learning overly generic ADLs and operating with complex modeling guidelines that describe how to properly model with these, modelers of each department should be able to use their specific terminology

[Wor19] A. Wortmann:
Towards Component-Based Development of Textual Domain-Specific Languages.
In: International Conference on Software Engineering Advances (ICSEA 2019), pp. 68-73, IARIA XPS Press, Valencia, Nov. 2019.
www.se-rwth.de/publications/

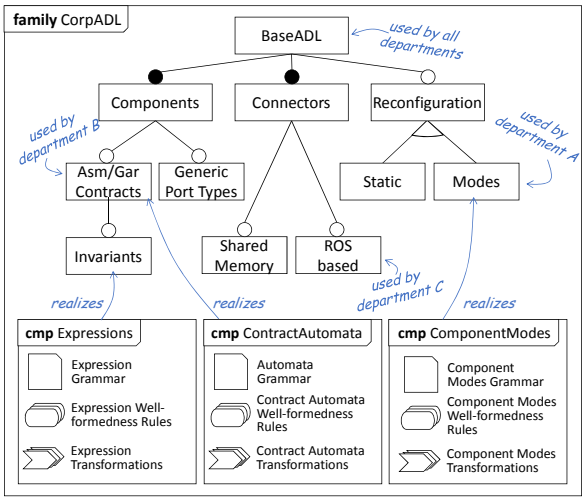


Fig. 1. A language family comprising features of language components that can fulfill the requirements of all three departments.

and learn only the modeling elements required for their specific application.

Hence, while in general, these ADLs require some notion of components, ports, and connectors, each department has domain-specific requirements for the ADLs to be used:

- Department A (trains) requires components that support dynamic reconfiguration via components modes [16] to enable switching components related to country-specific technology when the train crosses a border.
- Department B (smart factories) demands components with assumption/guarantee contracts [17] that facilitate correct integration of new components when the factory reconfigures.
- Department C (robotics) demands novel connectors that support bridging architecture models with the robot operating system (ROS) [18].

Developing a general ADLs that captures all of these concepts is not feasible as it complicates modeling in departments where only some of these modeling elements are not required. Alternatively developing three specific ADLs – each with their specific infrastructure (*e.g.,* parsers, model checkers, code generators) – independently is costly and inefficient.

Instead, building suitable language components and combining these as required can significantly reduce the effort of fulfilling the deparments' requirements. For our example, consider the language family of Fig. 1: this family contains the language features required by the different departments and each feature is realized by a language component comprising a combination of grammar, well-formedness rules, and code generators. By developing independent language components that realize the different features and by leveraging variability modeling techniques, the configuration of the base ADL for the different departments only requires selecting the appropriate language components and (semi-)automatically integrating these. If no appropriate features are available, developing and integrating novel language components and integrating these into existing language families reduces the effort of building a suitable ADL.

Our method to engineer and reuse language components considers both, planned variability and opportunistic reuse, and supports semi-automated composition of language component constituents in the technological space of the MontiCore [11] language workbench.

III. COLD4TXT Language Components

The conceptual model of COLD is a vision of language reuse that requires concretization. For txtDSLs, we have developed the COLD4TXT variant of COLD which realizes variability, explains how resolving variability affects the language components, how variability and customizability interact, how variability, customizability, the language facets' artifacts relate, and provides modeling techniques to realize this. At its core, COLD4TXT resolves variability and customizability through the additive composition of language components according to their explicitly provided and required extension points.

To enable this, COLD4TXT differs from COLD: In COLD4TXT, *language families* and *language components* replace language concerns and language facets of COLD, respectively: The language concerns of COLD provide both variability and customizability. This entails that they provide the complete customizability of their intrinsic language product line and express this towards the user despite only a small subset of customization options being available in the language product derived from the product line (namely these provided by the features selected for the product). In contrast, customizability should express means for tailoring languages that are not resolved by variability. Therefore, the language component comprising the derived language product provides customizability options instead. Moreover, to enable the proper composition of language components based on a feature selection, the COLD4TXT language components yield interfaces themselves. These interfaces guide and restrict their use in the variation interface's feature model and enable composing two language components (semi-)automatically, with only the implementation of adapters for generator composition requiring manual interaction [10]. To explain the effects of resolving variability and customizability in COLD4TXT, a *language component* consists of a

- one language component interface,
- one customization interface,
- up to one grammar artifact,
- arbitrary many well-formedness rule artifacts, and
- arbitrary many code generator artifacts.

The language component interfaces explicitly provide or require language grammar productions, well-formedness rules, or code generators. Also, they may yield constraints between these (*e.g.,* representing whether an extension point is optional or mandatory, or to express that selecting a provided code generator entails selecting a grammar production as well). The provided extension points for grammar rules identify productions of the contained CFG that are meant for reuse (*e.g.,* expressions of an imperative modeling language, method

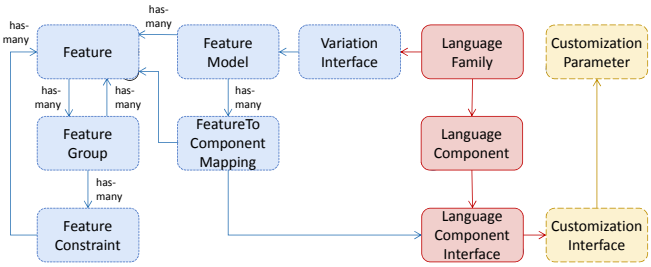


Fig. 2. Conceptual model for txtDSL reuse focusing on language families and their variation interfaces.

signatures of a class diagram language, ...). The required extension points for grammar productions explicate productions that demand (optional or mandatory) extension for the contained syntax to be completed.

Specifying required well-formedness rules within the interface either demands for giving complete specifications of the required well-formedness rules behavior (*i.e.,* their implementation) or specifying conditions under which an independently provided well-formedness rule is suitable for the required rule (*i.e.,* some form of acceptance tests). The former entails having a specification that is precise enough to become a implementation automatically and the latter testing rarely would be complete. Hence, we decided to consider the set of well-formedness rules of a language component as its extension point. Thus, a language component can provide arbitrary many well-formedness rules that may or may not be used by other components, but it cannot (yet) describe that it requires additional well-formedness rules. This is subject to ongoing work. For code generators, language components leverage the notions of producer interface and product interface as introduced in [10]. Hence, language components may provide and require extension points that declare exactly one producer interface and one product interface. The customization interfaces of language interfaces comprise parameters of well-formedness rules and generators that are not meant to be resolved through the closed variation of language families but enable open customization instead. Such customization could be the numbers of initial states supported in models of a language component for an automaton DSL or the path a generator should produce artifacts in.

The language interfaces ground their required and provided extension points the artifacts of their language components. Here, the red concepts (solid lines) represent the language components and the yellow concepts (dashed lines) highlight their customization interface parts. The language components are part of language families as depicted in Fig. 2: aside from at least one language component, a *language family* contains a variation interface comprising a single feature model and a mapping that relates features to language component interfaces. By transitivity of language interface extension points, this also identifies one language component per feature. The feature model of the variation interface is developed by a language family designer that intends to derive similar DSLs of joint buildings blocks. As such, she models how selecting

```
01 language family CorpADL {
02   components                                   // available language components
03     MontiArc, ContractAutomata, ComplexPortTypes, Expressions;
04   variation interface root BaseADL {
05     mandatory Components {                     // variation interface
06       optional AsmGarContracts { optional Invariants; }
07       optional GenericPortTypes;
08     }
09     // additional features relations
10   }
11   root feature BaseADL uses MontiArc;
12   abstract feature Components;
13   feature AsmGarContracts uses ContractAutomata {
14     binds production Automaton to Components.ArcElement;
15     binds generator Automaton2Java to Components.BehaviorGenerator;
16     binds wfrs NonHierarchical;
17   }                                            // feature definition
18   feature Invariants uses Expressions {
19     binds production Expression to AsmGarContracts.Expr.
20     binds Expression.All;
21     binds generator Expressions2POJO to AsmGarContracts.Guard2Java;
22   }                                            // feature to component mappings
23   // additional features definitions
24 }
```

Fig. 3. Textual model of the `CorpADL` language family of Fig. 1.

```
01 language component ContractAutomata {
02   grammar mc.automata.ca.ContractAutomata;
03
04   provides production ConractAutomatonMain;          // provided grammar productions
05   provides AsmAutomaton for production AssumptionAutomaton;
06   provides GarAutomaton for production GuaranteeAutomaton;
07   requires mandatory Expr for production IGuardExpression;
08 // provided generator                               // required grammar production
09   provides generator Automaton2Java for ContractAutomatonMain {
10     producer IAutomatonGen;
11     product IAutomatonPairRealization;
12   }                                                 // required generator
13   requires generator Guard2Java for IGuardExpression {
14     producer IGuardExpressionGenerator;
15     product IGuardExpression
16   }
17
18   provides wfrs Hierarchical {                      // provided sets of well-formedness rules
19     mc.automata.ca.coco.base.*;
20     mc.automata.ca.coco.hierarchical.*;
21   }
22   provides wfrs NonHierarchical {
23     mc.automata.ca.coco.base.*;
24     mc.automata.ca.coco.NoHierarchy;
25   }
26
27   parameters {                                      // customization parameters
28     int max for mc.automata.cocos.NumHierarchyLevels.initialize;
29     String prefix for IAutomatonGen.generate;
30   }
31 }
```

Fig. 4. Model of the `ContractAutomata` component sketched in Fig. 1.

a specific child feature implements the extension points of its parent feature and specifies constraints between features in the `Feature2ComponentMappting`.

The language components are composed based on the arragement of language components in the variation interfaces' feature model. From this, a new language component comprising their (possibly composed) artifacts together with a derived interface are synthesized. If there are required extension points not fulfilled by the selected features, these become part of the new component's interface.

COLD4TXT is realized in a language engineering framework using the MontiCore language workbench. To this end, we have developed modeling languages for language families, language components, feature configurations, and customization configurations as well as a toolchain that supports resolving variability and customizability.

```
01 language component Expressions {
02   grammar mc.basic.expressions.Expressions;
03   provides production Expression;
04   provides wfrs All { mc.basic.expressions.*; }
05
06   provides transformation Expressions2POJO for Expression {
07     producer IExpressionJavaGenerator;
08     product IExpression;
09   }
10 }
```

Fig. 5. Model of the `Expression` component sketched in Fig. 1.

The language family `CorpADL` of our example (*cf.* Fig. 1) can be represented as illustrated in Fig. 3. This family describes which language components it comprises (ll. 2-3), its variation interface in terms of a feature model (ll. 4-10), and defines its features (ll. 11-24). A feature either is a root feature (at most one), an abstract feature solely for grouping other features (such as the feature `Component`), or is realized by a language component. Each feature of the latter kind defines how the provided extension points of its language component are mapped to the required extension points of its parent feature. For instance, selecting the feature `Invariants` entails that (1) its production `Expression` will be embedded [15] into the extension point `Expr` of the language component `AsmGarContracts` (l. 21); (2) its well-formedness rules provided via the extension point `All` will be reused (l. 22); and (3) its code generator provided via the extension point `Expressions2POJO` will be embedded into the code generator `Guard2Java` of language component `AsmGarContracts` (l. 23). The well-formedness rules of the language family ensure that these mappings are valid w.r.t. the language components illustrated in Fig. 4 and Fig. 5.

The next section explains how these language components are combined.

IV. Deriving Languages

Modeling language families with COLD4TXT first demands its instantiation for a specific technological space by providing modules for (1) analysing the compatibility of COLD4TXT models with the referenced technology space artifacts and (2) composing these artifacts according to COLD4TXT specifications as depicted in Fig. 6. The former modules, for instance, check whether a well-formedness rule provided by a language component exists or whether a grammar production declared as an extension point indeed is an interface production. The latter modules take composition instructions (the binding mappings) and related artifacts, and compose these accordingly. For MontiCore, these modules are provided. Language engineers than can use this instance of COLD4TXT to engineer language components. Language family developers then can reuse these in different contexts through arranging these in the variation interfaces. Language family users then select the desired language features matching their requirements and use the COLD4TXT instance to synthesize a suitable language component. If this language component is incomplete w.r.t. its mandatory required extension points or parameters, it cannot be used as a DSL yet. Then, the language family user has to specify the missing customization configuration, before a fully configured language component and the artifacts for a DSL in the corresponding technological space are derived.

For MontiCore, these artifacts are a synthesized CFG the union of the selected well-formedness rules, and a code generator composed along its producer and product interfaces. These artifacts can be processed by MontiCore to produce a DSL that is completely independent of language families and language components. Moreover, the (possibly incomplete) language components derived from resolving variability and customizability can be used as parts of other language families again, which facilitates their reuse.

Based on a feature configuration, the COLD4TXT framework composes the language components associated with the selected features pairwise and top-down. The resulting component yields the provided extension points of the parent and child components. For each mandatorily required extension point (*e.g.,* `Expr` of language component `ContractAutomata`), if an implementation is defined by the binding mappings in the variation interface's feature model, then this extension point becomes optional and is copied to the interface of the new component as well. The sets of well-formedness rules from the parent component and the ones from the selected provided extension point of the child component are merged and provided as a new extension point in the new component. For the CFGs, COLD4TXT expects the responsible modules of the specific technology space to produce combined CFGs and adapters between the participating code generators accordingly.

For instance, selecting the features "Asm/Gar Contracts" and "Invariants" depicted in Fig. 1 with the variation interface specified in Fig. 3 entails combining the language components `ContractAutomata` (Fig. 4) and `Expressions` (Fig. 5) accordingly. The resulting language component is given in Fig. 7. This component uses a synthesized CFG featuring contract automata and expressions (l. 2), the union of selected well-formedness rules, and the composed code generators. Its interface reduces the cardinality of the required grammar extension point `Expr` to optional (l. 7), adds the provided extension point `Expression` (l. 8) as well as the code generator for expressions (ll. 14-17) from the `Expressions` language component of Fig. 5, and provides a new set of well-formedness rules (ll. 19-23). As this component does not require further extension, specifying values for its parameters enables MontiCore to derive a complete DSL from it.

V. Discussion

In contrast to the purely conceptual models of DSL reuse [19], [20], COLD4TXT on capturing all DSL definition constituents at a sufficient level of abstraction to support the precise explanation of the effects of composing these, binding their variability, and resolving their customizability on its own.

The conceptual model of COLD4TXT aims to be independent of technological spaces as long as these enable to (1) identify grammar extension points; (2) compose grammars, sets of well-formedness rules, and code generators without

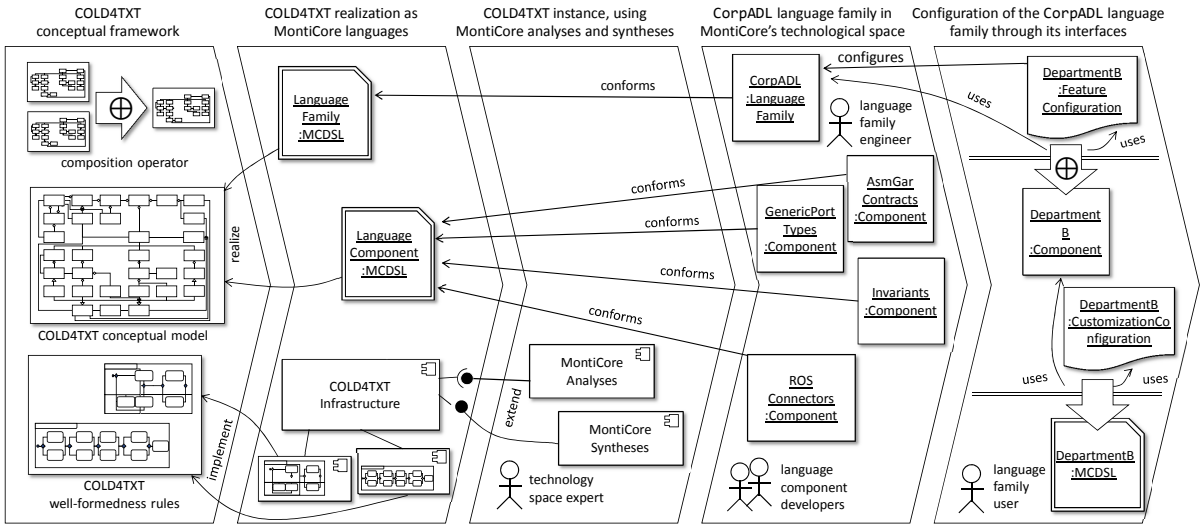


Fig. 6. After tailoring CORE4TXT for a specific technological space, developers can engineer language components to be used by language family developers to facilitate creating DSLs.

```
01 language component ContractAutomataWithExpresssions {
02   grammar mc.automata.ca.ContractAutomataWithExpressions;
03
04   provides production ConractAutomatonMain;                    // synthesized grammar
05   provides AsmAutomaton of production AssumptionAutomaton;
06   provides GarAutomaton of production GuaranteeAutomaton;
07   requires optional Expr for production IGuardExpression;     // added
08   provides production Expression;                              // became optional
09
10   provides transformation Automaton2Java for ContractAutomatonMain {
11     producer IAutomatonGen;
12     product IAutomatonPairRealization;
13   }                                                            // added
14   provides transformation Expressions2POJO for Expression {
15     producer IExpressionJavaGenerator;
16     product IExpression;
17   }
18                                                                // synthesized set of well-formedness rules
19   provides wfrs All {
20     mc.automata.ca.coco.base.*;
21     mc.automata.ca.coco.NoHierarchy;
22     mc.basic.expressions.*;
23   }
24
25   parameters {
26     int max for mc.automata.cocos.NumHierarchyLevels.initialize;
27     String prefix for IAutomatonGen.generate;
28   }
29 }
```

Fig. 7. Language component synthesized as result from selecting the features "Asm/Gar Contracts" and "Invariants" of Fig. 3.

eliminating the extension points in the process; (3) describe code generators and the generated products in terms of their interfaces; (4) identify parameters of well-formedness rules and code generators in an object-oriented fashion. While these are strong assumptions, we currently investigate applying COLD4TXT and its realization within the technological spaces of Neverlang [7] and Xtext [21]. Moreover, it currently only supports embedding in the sense of [10], whereas there are various other composition operators for txtDSLs. Whether and how supporting these is possible, also is ongoing research.

In the future, we aim to extend the notion of language components to feature additional constituents (*e.g.,* model-to-model transformations or editor fragments).

VI. Related Work

Research on Language product lines (LPLs) [15], [22], [23] is scattered across different kinds of DSL definition constituents and technological spaces. And while we developed a notion of LPLs for the technological space of MontiCore [15] in particular, there currently is no actionable understanding of the variability of complete txtDSLs (*i.e.,* encompassing all four kinds of constituents). Moreover, (closed) variation rarely is connected with (open) customization to systematically reuse DSLs in general. There are only a few solutions that consider either txtDSL variation or customization across different kinds of DSL definition constituents. These include a few language workbenches [24], such as Argyle [23], Neverlang [7], or the combination of SDF and FeatureHouse [22].

In Argyle [23], DSLs are constructed from language assets that resemble concerns and comprise syntax, data types, and code generation templates. A feature model arranges assets according to their dependencies, which demands their white-box apriori composition that hinders the reuse of facets. In contrast, COLD4TXT will be based on our exploratory work [15] that makes extension points of concerns explicit and supports the black box composition of their artifacts through the generation of suitable adapters between these.

SDF and FeatureHouse realize variability based on compositional language modules containing grammar rules, typing rules, and evaluation rules [22]. It also focuses on the white-box composition of artifacts and interpretation. Similar partial solutions towards variation or customization of selected kinds of DSL definition constituents are available from a variety of language workbenches. For instance, ableC [25] is an extensible C language that leverages attribute grammars to reuse syntax and semantics, MPS [21] enables reuse of projectional languages with views and model transformations, and Spoofax [8] supports reuse of textual, interpreted languages. All of these focus on specific technological spaces.

VII. Conclusion

We have presented the novel COLD4TXT conceptual framework to facilitate reusing textual DSLs through systematic variability and customizability. In COLD4TXT, language families capture txtDSL variability as feature models and realize it via composition of language components according to their interfaces. Composing language components yields new language components that may demand further extension or customization before these can be translated into complete DSLs for specific contexts. This facilitates engineering textual DSLs for different contexts and fosters the application DSLs.

References

[1] M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann, "Using language workbenches and domain-specific languages for safety-critical software development," *Software & Systems Modeling*, 2018.

[2] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What Industry Needs from Architectural Languages: A Survey," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, 2013.

[3] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," *CoopIS, DOA*, vol. 2002, 2002.

[4] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, 2018.

[5] J. Whittle, J. Hutchinson, and M. Rouncefield, "The State of Practice in Model-Driven Engineering," *Software, IEEE*, vol. 31, no. 3, 2014.

[6] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, no. 4, 2006.

[7] E. Vacchi and W. Cazzola, "Neverlang: A framework for feature-oriented language development," *Computer Languages, Systems & Structures*, vol. 43, 2015.

[8] G. H. Wachsmuth, G. D. P. Konat, and E. Visser, "Language Design with the Spoofax Language Workbench," *IEEE Software*, vol. 31, no. 5, 2014.

[9] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xDSMLs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ACM, 2015.

[10] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Modeling Language Variability with Reusable Language Components," in *International Conference on Systems and Software Product Line (SPLC'18)*, 2018.

[11] K. Hölldobler and B. Rumpe, *MontiCore 5 Language Workbench Edition 2017*. 2017.

[12] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Safe model polymorphism for flexible modeling," *Computer Languages, Systems & Structures*, vol. 49, 2017.

[13] J. de Lara and E. Guerra, "Generic Meta-modelling with Concepts, Templates and Mixin Layers," in *Model Driven Engineering Languages and Systems*, 2010.

[14] T. Kühn, W. Cazzola, and D. M. Olivares, "Choosy and picky: configuration of language product lines," in *Proceedings of the 19th International Conference on Software Product Line*, ACM, 2015.

[15] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic composition of independent language features," *Journal of Systems and Software*, vol. 152, 2019.

[16] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. 2012.

[17] M. Broy and K. Stølen, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. 2012.

[18] K. Adam, K. Hölldobler, B. Rumpe, and A. Wortmann, "Modeling Robotics Software Architectures with Modular Model Transformations," *Journal of Software Engineering for Robotics (JOSER)*, vol. 8, no. 1, 2017.

[19] T. Clark, M. v. d. Brand, B. Combemale, and B. Rumpe, "Conceptual Model of the Globalization for Domain-Specific Languages," in *Globalizing Domain-Specific Languages*, 2015.

[20] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann, "Concern-oriented language development (COLD): Fostering reuse in language engineering," *Computer Languages, Systems & Structures*, vol. 54, 2018.

[21] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. 2013.

[22] J. Liebig, R. Daniel, and S. Apel, "Feature-oriented language families: A case study," in *VaMoS*, 2013.

[23] C. Huang, A. Osaka, Y. Kamei, and N. Ubayashi, "Automated DSL construction based on software product lines," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, IEEE, 2015.

[24] S. Erdweg, T. Van Der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, *et al.*, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems & Structures*, vol. 44, 2015.

[25] T. Kaminski, L. Kramer, T. Carlson, and E. Van Wyk, "Reliable and automatic composition of language extensions to C: the ableC extensible language framework," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 2017.

A Compositional Framework for Systematic Modeling Language Reuse

Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
www.se-rwth.de

ABSTRACT

Many engineering domains started using generic modeling languages, such as SysML, to describe or prescribe the systems under development. This raises a gap between the generic modeling languages and the domains of experience of the engineers using these. Engineering truly domain-specific languages (DSLs) for experts of these domains still is too challenging for their wide-spread adoption. One major obstacle, the inability to reuse multi-dimensional (encapsulating constituents of syntax and semantics) language components in a black-box fashion, prevents the effective engineering of novel DSLs. To facilitate engineering DSLs, we devised a concept of 3D components for textual, external, and translational DSLs that relies on systematic reuse through systematic closed and open variability in which DSL syntaxes can be embedded, well-formedness rules joined, and code generators integrated in a black-box fashion. We present this concept, a method for its systematic application, an integrated collection of modeling languages supporting systematic language reuse, and an extensible framework that leverages these languages to derive novel DSLs from language product lines. These can greatly mitigate many of the challenges in DSL reuse and, hence, can advance the engineering of truly domain-specific modeling languages.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

DSL, Modeling Language, Reuse, Variability

ACM Reference Format:
Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, Andreas Wortmann. 2020. A Compositional Framework for Systematic Modeling Language Reuse. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3365438.3410934

1 INTRODUCTION

Our society thrives on Cyber-Physical System (CPS) that enable communication, education, healthcare, mobility, and more. These systems are engineered in collaboration with experts from multiple domains, such as mechanical engineering, electrical engineering, material sciences, jurisprudence, software engineering, and systems engineering. To cope with the complexity of engineering these systems, domain experts have begun to leverage the benefits of modeling languages [49] to, among others, describe product geometries [17, 37] physical properties [19, 30], or the integration of contributions from different domains [2, 40].

The efficient use of models by domain experts demands well-defined, Domain-Specific Languages (DSLs) that support automated analysis and synthesis of conforming models. Engineering DSLs is a complex endeavor that demands understanding the domain of interest, creating implementations capturing the DSL's syntax and semantics, integrating these properly, and providing tools supporting to their use. Due to these challenges, experts often have to use overly generic modeling languages, such as UML [25] or SysML [22], instead of DSLs precisely tailored to the concepts and notations of their respective domains. This hinders domain experts in employing these languages efficiently. Reusing components to engineer DSLs more efficiently can lead to more precise and specific languages that can foster the adoption of modeling techniques and, ultimately, facilitate engineering complex CPS. The contributions of this paper support the efficient engineering of DSLs through (1) a novel conceptual model of the reuse of 3D DSL components through closed variability of DSL families (product lines) and open customization; (2) a method for its systematic application; (3) a collection of integrated modeling languages to describe DSL families and their constituents; and (4) an extensible framework that supports engineering DSL families as well as deriving DSL components and complete DSLs from these.

The research results presented in this paper extend the findings presented in [6–8] by making extension points explicit on the component level, introducing different kinds of bindings between the DSL families and their components, and providing an integrated feature modeling language that describes how families relate components through features.

In the remainder, Sec. 2 motivates the benefits of systematic language reuse and Sec. 3 presents preliminaries. Sec. 4 introduces our conceptual model and a method for its systematic application. Sec. 5 describes the modeling languages and the framework realizing the conceptual model. Sec. 6 illustrates its application by example. Sec. 7 discusses observations and related research. Sec. 8 concludes.

2 MOTIVATING EXAMPLE

Consider a company engineering different kinds of CPS featuring state-based behavior, such as robotics systems and appliances for smart buildings. Instead of using the same generic modeling language for all three departments, engineers in each department should be enabled to use a DSL closely related to their domain of

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410934

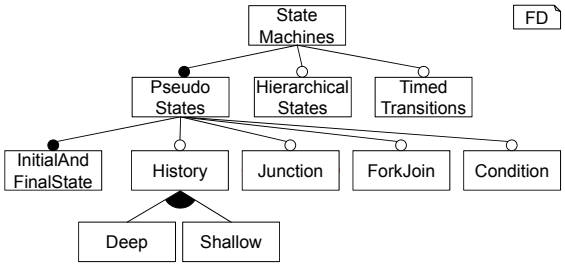


Figure 1: Feature model of an FSM Language Product Line (LPL) (adapted from [38])

expertise. A team of language engineers, therefore, decides to engineer a family of Finite-State Machines (FSMs) DSLs (*cf.* Figure 1). State machines are commonly used behavior descriptions in a multitude of application domains such as robotics [5, 9], aerospace software [21], web applications [23] or game development [35]. Thus, different variants for FSM notations have been brought forth.

For instance, the company has engineered a DSL for describing state-based behavior of robot arms with limited computational power. For this application, performance is crucial and managing state histories and concurrency as induced by join and fork nodes is not desired. However, the behavior of the robot arm should feature time-based triggers on transitions to ease its programming. In another department of the company, a variant of the FSM DSL is used to describe the behavior of web-based smart home appliances. For these, a deep state history can improve the user experience by continuing an interrupted procedure in the state it was interrupted. Furthermore, the language engineers decide that for web applications, junctions in FSMs should simplify user response handling.

Through domain analysis, the engineers of the FSM DSL product line consider these and a set of similarly fashioned applications and decide to create the feature model depicted in Figure 1. Each FSM DSL contains states and transitions that are not explicated in the feature model, as they are contained in every variant of the language. Further, each DSL must contain initial and final pseudo states. These are explicated in the feature model, as the language engineers plan to evolve the feature model in the near future by providing an alternative textual notation for initial and final pseudo states. Moreover, each DSL variant has the option of including deep or shallow history pseudo states, junction pseudo states, fork and join pseudo states, and condition pseudo states. Optionally, an FSM DSL may support modeling hierarchical states that themselves contain states and transitions. Timed transitions, also optional features, enable users to model a passage of time as a transition trigger.

The team developing the above DSL family has four requirements for the reuse of DSL components:

- **R1 Black-box reuse**: To foster DSL reuse across time and involved developers, it must be possible to reuse DSL parts in a black-box fashion without needing to become an expert in their internal implementation details.
- **R2 Structured reuse**: To support reusing DSL parts for building similar DSLs by domain experts without language engineering expertise, it should be possible to arrange the relevant DSL components in the DSL family definition according to their options for composition.
- **R3 Push-button reuse**: To enable domain experts to derive suitable languages with minimal effort, the composition of selected DSL components based on their arrangement in the DSL family should be automated.
- **R4 Open reuse:** To enable extending DSL components with capabilities unforeseen at time of their arrangement, it must be possible to customize these systematically through open variability [13].

A language engineering approach satisfying the above requirements can help the company to reduce cost and effort for engineering and maintaining modeling languages tailored to each kind of CPS they develop.

3 PRELIMINARIES

Our method for efficient DSL engineering relies on research in Software Language Engineering (SLE) [29, 32, 50] and leverages the MontiCore language workbench as technological space [34] for realization and for the case study.

3.1 Software Language Engineering

A modeling language usually is defined by the set of models it accepts. To make languages machine-processable, language definitions in terms of their constituents have been proposed. These usually require that (1) a language definition comprises a concrete syntax, an abstract syntax, a semantic domain, and a mapping from the abstract syntax to the semantic domain giving meaning [26] to the language's sentences [11]; or that (2) each language definition comprises a concrete syntax, an abstract syntax, static semantics (well-formedness rules), and dynamic semantics (behavior) [12]. The abstract syntax of a modeling language defines the structure of accepted models and is typically defined in terms of grammars [3, 10, 47] or metamodels [16, 41, 43]. The concrete syntax is the representation of models towards the user and can be, *e.g.,* textual, graphical, or mixed. Often, this is defined by the editor used to process models. Well-formedness rules can restrict the abstract syntax further to prevent undesired model properties not expressible through the abstract syntax formalism itself. Interpreters and model transformations can give meaning [26] (and possibly behavior) to models by translating these into other languages.

In the following, we assume language implementations that are

- textual: they feature an integrated definition of concrete and abstract syntax through a grammar;
- external: they are not defined in terms of a host language (in contrast to internal DSLs [15]); and
- translational: they give meaning to models through transformation (in particular through code generation).

3.2 MontiCore

We use the language workbench MontiCore [28] to realize our apporach as proof of concept. MontiCore is a language workbench for the development of textual, external DSLs. The integrated concrete and abstract syntax of a DSL is specified in the form of a context-free grammar. From this, MontiCore generates language

```
01 grammar FSM {                                          MCG
02  StateMachine = "sm" Name "{" (IState | ITrans)* "}";
03  interface ITrans;
04  interface IState = Name ;
05  State implements IState = "state" Name ";" ;
06  Transition implements ITrans = from:Name "->" to:Name ";" ;
07 }
```

concrete syntax only · iteration · interface productions · interface implementation

Figure 2: Example MontiCore grammar of an FSM DSL

tooling including an abstract syntax data structure, a parser that instantiates this data structure, a visitor infrastructure for traversing the abstract syntax, and infrastructures for defining and checking well-formedness rules as well as for generating code from models conforming to the grammar. Well-formedness rules in MontiCore are realized as Java classes called context conditions and are checked against the abstract syntax leveraging the generated visitor infrastructure. Code generation is realized through template-based code generators based on the FreeMarker [20] template engine.

Each MontiCore grammar begins with the keyword `grammar`, followed by the name of the grammar as depicted by example in Figure 2. The body of a grammar (ll. 2-7) contains grammar productions. By default, the first production is the start production of a grammar. On the left-hand side, each production defines a nonterminal, *e.g.,* `StateMachine` (l. 2). On the right-hand side, a production can contain terminals (in double quotes) and nonterminals (starting with upper case letter) as well as iterations ('`*`', '`+`' ,'`?`'), alternatives ('`|`'), and concatenations (' ') thereof. Interface nonterminals can underspecify a right-hand side completely (l. 3) or prescribe abstract syntax elements (l. 4). Other productions can implement interface productions (ll. 5-6). If the right-hand side prescribes abstract syntax elements, implementing nonterminals must provide these. The generated parser treats the usage of an interface nonterminal equal to an alternative over all nonterminals defined by productions implementing the interface nonterminal.

Moreover, MontiCore supports language inheritance [28], which enables reusing complete grammars by inheriting from them and using all inherited productions in the new grammar. We will leverage this to compose the grammars of DSL components according to their arrangement in the DSL family.

4 A METHOD FOR SYSTEMATIC LANGUAGE ENGINEERING

This section introduces the process of creating families of reusable DSL components and composing these to derive novel DSLs. It further presents a conceptual model describing DSL components, their properties, and their relation to feature models of DSL families. With this in place, it explains the effect of selecting two DSL features as the composition of the two related DSL components.

Our method for systematic language composition relies on encapsulating related language constituents in DSL components, making their provided and required extensions explicit, and composing the language components according to these and guided by a feature model. All these activities are related to roles with specific expertise as illustrated in Figure 3. First, language engineering experts create reusable DSL components for specific purposes, such as the features illustrated in Figure 1. Each of these contains a combination

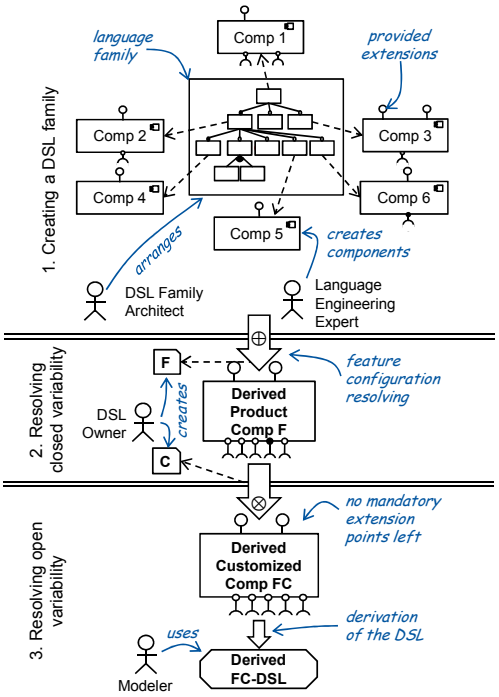


Figure 3: The composition of two DSL components processes all bindings, and updates the interface of the resulting DSL component accordingly. If all mandatory required extensions have been fulfilled, a new DSL can be derived.

of grammars, well-formedness rules, and code generators relating to these grammars. By making their provided extensions (*i.e.,* grammar productions, well-formedness rules, or code generators) and their required extensions (grammar productions or generator extensions) explicit, language family architects can arrange these into a feature model representing a family of DSLs.

In this feature model, each feature either is related to a language component or is an abstract feature [42] for logical grouping. By relating features to DSL components and to other features (through their parent-child relation), the language family engineer decides how the components will be composed if their respective features are selected. Once the DSL family is complete, DSL owners, who are experts of the application domains, can derive a suitable DSL by selecting appropriate features from the family. Based on the resulting feature configuration, the selected DSL components are composed and their provided and required extensions are updated accordingly. Through extension points of our framework, the composition of the language constituents (*i.e.,* grammars, well-formedness rules, code generators) is delegated to software modules of the specific technological spaces (such as Neverlang [44], MontiCore [28], or Xtext [18]). The result either is a new DSL component, if mandatory extensions were not provided through the family or a new DSL otherwise. In the former case, the DSL owner can specify additional customization information that was either not available during family creation (*e.g.,* the action language needed for automata transitions for a specific domain) or not suitable for configuring in a

feature model (such as numerical parameters). If the DSL family was well-defined, *i.e.,* options for all required extensions of its components were provided, the DSL owner does not need to have any expertise in SLE but can derive the most suitable DSL variant on a push-button basis.

To foster DSL reuse, we have conceived and integrated modeling languages for describing DSL components and DSL families. They are tailored to language engineering experts and support making provided and required DSL component extensions explicit. Their models form the basis of component composition. The latter language is an extension of features models that supports describing DSL families and the binding of features to extension points of DSL components. A customization language supports implementing required extensions of DSL components not provided by their language family. The modeling languages and the software modules processing these support extension with new language elements and analyses to support extending DSL component definitions and to address challenges of different technological spaces. However, conceptually, our approach assumes the following[1]:

- **A1** Composition leads to *conservative extension* [28], *i.e.,* it is purely additive in terms of language constituents, *i.e.,* composition cannot eliminate grammar productions, well-formedness rules, or generators. Otherwise, composition could eliminate extension points, which yields undesired complexities. Nonetheless, adding new well-formedness rules can restrict the accepted models of the resulting DSL.
- **A2** The grammar language must support identification of extension points. Otherwise, binding extensions to grammars is not possible. This identification, however, can be realized, *e.g.,* through dedicated forms of productions or naming conventions. Hence, many grammar specification formalisms can support this.
- **A3** The well-formedness rules of the technological space must be identifiable and applicable individually. Otherwise, selecting and reusing these rules in different contexts might not be possible. Whether these rules are implemented in OCL [27], a general-purpose programming language [18, 28], or another modeling language [44] then does not matter.
- **A4** The code generators (*producers*) and generated artifacts (*products*) must be defined in a language that supports the notion of object-oriented interfaces and both interfaces (producer and product) must be made explicit by the code generators. Otherwise, the form of adaptation between the generators (producers) or generated artifacts (products) that we propose, will not be possible [6, 8]. This prevents applying our approach to various kinds of target languages and formats (such as CSV, SQL, XML, *etc.*)
- **A5** Each code generator must create a main artifact adhering to the generator's product interface through which that artifact can be invoked during product runtime. If there is no such product, adapting between the required product and the provided product is not possible. While this does not limit the application of our approach technically, enforcing the existence of such a product can make the generated code less efficient. Mitigating this is subject to current research.

[1]The reasoning for the code generator assumptions is discussed in detail in [6, 8].

4.1 A Conceptual Model for Black-Box Language Reuse

Our conceptual model describes the properties of DSL components (**R1**) and DSL families (**R2**) relevant to their systematic reuse. For this purpose, the DSL components do not provide closed variability themselves, but support customization through their required extensions. DSL families comprise feature models to describe closed variability of potential DSLs by arranging DSL components (*cf.* Figure 4).

4.1.1 3D DSL components and interfaces. **DSL components** provide the constituents of a language definition. They are three-dimensional by comprising elements of each of the three essential language definition constituents: (1) syntax, (2) well-formedness rules, and (3) semantics-based code generators. To this end, each DSL component comprises at least one grammar and can comprise multiple sets of identifiable (**A3**) well-formedness rules, as well as multiple code generator specifications. As both, the well-formedness rules and the generator specifications rely on a grammar for the definition of the abstract syntax data types, it is mandatory for each component. The well-formedness rules are grouped in sets to facilitate their reuse in different contexts. The generator specifications identify a generator as a GPL code class that adheres to an explicit producer interface (**A4**) and creates at least a main GPL artifact that adheres to an explicit product interface (**A5**).

DSL interfaces expose (parts of) these constituents through explicit extensions with cardinalities (optional or mandatory) to the environment (*e.g.,* the language family). For grammars and generators, the interfaces support both, provided and required extensions, whereas for well-formedness rules, only provided extensions can be made explicit. Specifying what is required from a well-formedness rule is subject to ongoing research (*cf.* Sec. 7). For well-formedness rules and code generators, additional parameters can be defined that enable more fine-grained customization (such as numerical constraints, paths, *etc.*).

Provided extensions offer DSL functionality to be reused by other components. Provided grammar extensions reference a production in the grammar that can be reused by other components' grammars. Provided well-formedness rules extensions offer sets of well-formedness rules for a specific production that can be reused in different contexts. Provided generator extensions reference a production for which they provide a transformation, a reference to a GPL class, and the interfaces of producer and product.

Required extensions specify missing functionality of a DSL component–*e.g.,* an automaton DSL might need an expression DSL for specifying guards–and can be either optional or mandatory. Required extensions for grammars reference a production of a contained grammar that supports extension. Required generator extensions demand extension for a specific production (such as the guard expressions above), with specific product and producer interfaces as introduced in [6]. Required parameters also are either optional or mandatory and parameterize well-formedness rules or generator specifications, respectively.

Generator specifications describe code generators of components in terms of processed product rules, provided producer and product interfaces, and a set of extension points that follows the

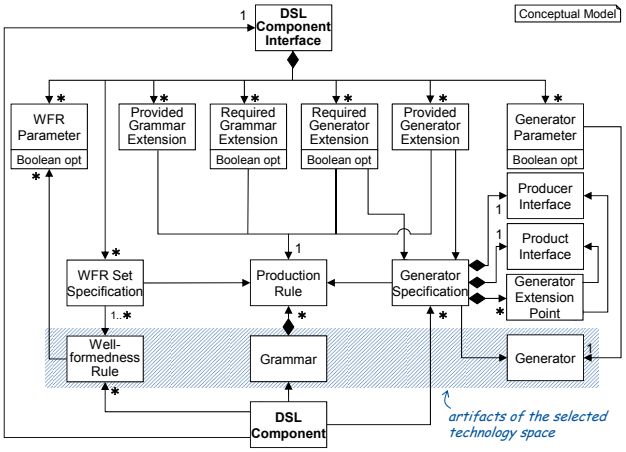


Figure 4: DSL components provide extensions, *i.e.,* parts of language constituents that are exposed by DSL component interfaces. DSL component interfaces also specify required extensions and parameters.

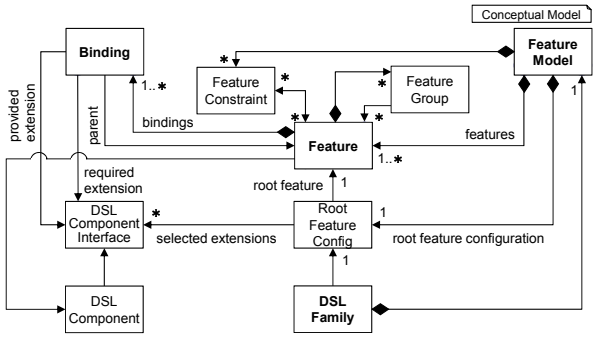


Figure 5: A language family has a feature model with features that reference and bind DSL components.

extension points of the processed grammar. For each required grammar extension, the generator specification provides a generator extension point that describes the required producer interface and a required product interface. The required producer interface prescribes the expected structure of a compatible generator being usable for translating productions embedded into the required grammar extension this extension point relates to. The required product interface prescribes the expected structure of a compatible main artifact produced for the required grammar extension this extension point relates to.

4.1.2 DSL families and bindings. **Language families** [48] describe closed variability through a central feature model [14]. Features relate to the extensions of DSL components and the arrangement of features in this model describes how the components will be composed if their respective features are selected. To this effect, DSL family architects select DSL components for specific purposes and arrange these carefully for DSL owners to use (**R2**).

Figure 5 depicts the conceptual model of DSL families. A family references one or more DSL component(s) and yields a single feature model [1, 4] consisting of features and bindings. A feature references a DSL component of the family that realizes it.

The root (top) feature of the DSL family defines the base DSL component into which the components related to all selected child features are embedded according to the family's feature model. As such, it might yield provided extensions as well, for which the **root feature configuration** can define bindings (*i.e.,* selections) already. This enables using a comprehensive DSL for the root feature while giving the flexibility of reusing only selected parts of it.

Bindings relate features to DSL components. Our concept supports three kinds of bindings, matching the different kinds of required and provided extensions (grammar, well-formedness rule, generator). Bindings are defined within features, *i.e.,* each feature describes how (a subset of) the provided extensions of its related DSL component will be bound to (a subset of) the required extensions of the DSL component related to its parent feature. To this end, each non-abstract feature must define at least one such binding and can define as many bindings as there are provided extensions in its DSL component. The different kinds of bindings are:

Grammar bindings map a provided grammar extension of the embedded component (*e.g.,* of a child feature) to a required grammar extension of the embedding component (*e.g.,* of a parent feature). The effect of such a binding is that everything producible from the provided grammar extension will become producible from the required grammar extension as well. For instance, when embedding the provided grammar extension for arithmetic expressions into a required grammar extension for Boolean expressions, arithmetic expressions become an alternative to the former. This can be realized through production inheritance [28] or adding an alternative supporting the provided productions to the requiring productions [5, 44]. This composition is supported by adhering to (**A1**) and (**A2**).

Generator bindings map a provided generator extension of the embedded component to the required generator extension of the embedding component. Such a binding entails that the provided generator will be used whenever the required generator is called. For instance, embedding a generator for translating arithmetic expressions to Java into another generator requiring that translation entails, per construction detailed in [8], that the embedding generator will call the embedded generator via an adapter between the required producer for arithmetic expressions and the provided producer interface of the generator for arithmetic expressions. This composition is enabled by (**A1**), (**A3**), and (**A4**).

Well-formedness rule embeddings join a well-formedness rule set of the embedded component into a well-formedness rule set of the embedding component. The result is a novel component with the same number of well-formedness rule sets than before, but more well-formedness rules in its sets. This enables refining DSLs by adding additional rules to its provided well-formedness rules.

Well-formedness rule addition adds a complete set of well-formedness rules of the embedded component en-bloc to the embedding component. Through this, a novel set of well-formedness rules becomes present in the resulting component. Both forms of well-formedness rule set composition rely on (**A1**) and (**A5**).

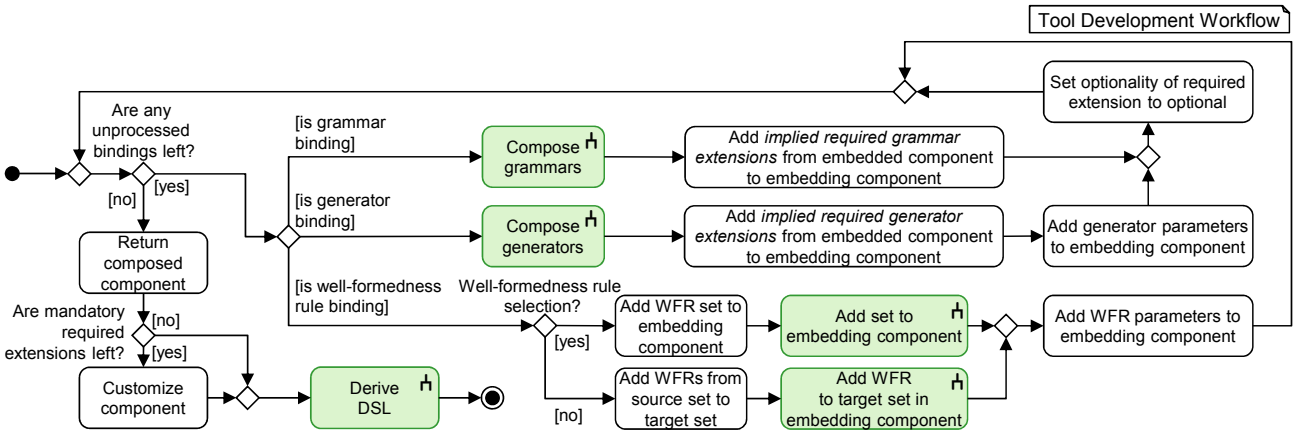


Figure 6: DSL components are composed according to the bindings defined between DSL family features.

4.2 Composing DSL Components

The composition of two DSL components is the directed application of bindings between these components. It produces a novel component resulting from adding selected provided extensions of the embedded component into the respective required extensions of the embedding component. This comprises two main activities: (1) Composition of the components' interfaces; and (2) Composition of the comprised language definition constituents (grammars, well-formedness rules, code generators);

Our method of reusing DSLs and DSL parts is independent of the actual composition of language constituents in the different technological cases as long as these adhere to (**A1**)-(**A5**). Consequently, the method and its realization anticipate extension with software modules specific to the technological space of choice that take care of the technical composition (*cf.* Sec. 5.4).

The process of composing two DSL components along their interface is illustrated in Figure 6: As long as there are unprocessed bindings, these and the related artifacts are passed to the technology space-specific composition components (used by green activities with fork icon) to perform the composition of DSL constituents. Afterwards, the required extensions of the language interface of the embedding component are updated accordingly by (a) setting fulfilled extensions to be optional and (b) adding implied required extensions of the embedded component. Provided extensions of the embedded components are not added to the interface of the embedding component as this would add options for reuse unintended by the DSL family. For well-formedness rules, either a set of the embedded component was meant to be reused en-bloc, in which case the complete set is added to the interfaces of the embedded component, or individual rules shall be reused. In this case, these are added to the set of well-formedness rules of the embedding component as indicated by the respective bindings.

Where an embedded component yields parameters, these are added to the interfaces of the embedding component. If all required extensions are fulfilled, the resulting DSL component can be translated into a new DSL automatically. Otherwise, it needs subsequent customization.

For a feature configuration relative to a language family, the feature tree of the language family is traversed bottom-up. If a feature is selected in the feature configuration, all associated bindings of this particular feature are applied, and the components are composed pairwise. The application of the bindings is similar to the composition process stated in the former part of this section. The traversing of the feature tree ends with the root feature configuration, if present. When applying the root feature configuration, all provided extensions and well-formedness rule sets not stated in the root configuration are removed. If the component has no mandatory required extension or component parameter, a usable DSL can be derived from it automatically (**R3**). Otherwise, customizing the component (**R4**) is necessary to obtain a usable DSL. To this effect, the bindings between the embedded and the customized component are applied and the components are composed as if they were related to a parent feature and its child.

5 MODELING LANGUAGES AND FRAMEWORK

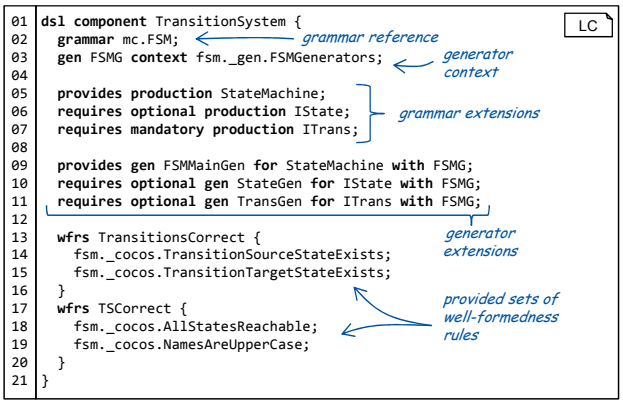
Based on the example of the FSM family of Figure 1, this section presents the modeling languages for DSL components and families.

5.1 DSL Components

The DSL component language reifies our conceptual model of DSL components and interfaces (**R1**) in form of a MontiCore modeling language. Following the conceptual model, each DSL component references exactly one grammar, zero to many generator contexts (describing producers and products), as well as various provided and required extensions and well-formedness rule sets.

Figure 7 illustrates this by example of the `TransitionSystem` DSL component It references a grammar via its fully qualified name `mc.FSM` (l. 2) and specifies the generator `FSMG` with context `FSMGenerators` (l. 3, Figure 8). The generator context is a class diagram describing the generator and its interfaces.

Afterwards, `TransitionSystem` defines a provided and two required grammar extensions of different optionalities (ll. 5-7) using productions from the `mc.FSM` grammar. This defines that the




```
01 dsl component TransitionSystem {
02   grammar mc.FSM;
03   gen FSMG context fsm._gen.FSMGenerators;
04
05   provides production StateMachine;
06   requires optional production IState;
07   requires mandatory production ITrans;
08
09   provides gen FSMMainGen for StateMachine with FSMG;
10   requires optional gen StateGen for IState with FSMG;
11   requires optional gen TransGen for ITrans with FSMG;
12
13   wfrs TransitionsCorrect {
14     fsm._cocos.TransitionSourceStateExists;
15     fsm._cocos.TransitionTargetStateExists;
16   }
17   wfrs TSCorrect {
18     fsm._cocos.AllStatesReachable;
19     fsm._cocos.NamesAreUpperCase;
20   }
21 }
```

Figure 7: A component representing a transition system DSL. It provides and requires extensions for the language's grammar, generator, and well-formedness rules.

component enables extension for the productions `IState` and `ITrans`. For code generation, the component defines a provided and two required generator extensions (ll. 9-11) relative to the generator context represented in Figure 8. The provided generator extension `FSMMainGen` enables reusing the component's generator. The required generator extensions `StateGen` and `TransGen` enable to extend code generation of this component for states and transitions accordingly. Ultimately, `TransitionSystem` also defines two provided sets of well-formedness rules of two rules each (ll. 13-20). The FSM grammar itself is illustrated in Figure 2 and comprises four productions: it defines two interfaces that can act as grammar extension points (ll. 3-4) and defines a transition system as a named collection of instances of these interfaces (l. 2). For states and transitions, it provides a default implementation (ll. 5-6).

Figure 8 depicts the generator context for the transition system component. The top three classes, `IFSMProducer`, `IFSM–Product`, and `IFSMSystemGenerator` define how this the `FSMGenerator` can be embedded into other components, *i.e.,* that it can act as an `IFSMProducer` and that its generated artifacts will adhere to the `IFSMProduct` interface. Moreover, `FSM–Generator` yields registration methods corresponding to its two extension points. For extension of states, *e.g.,* the `FSMGenerator` expects a producer of type `IStateProducer` and that this producer generates a main artifact of type `IStateProduct`, hence the corresponding code interfacing with implementations of this interface can be generated.

When bindings between two DSL components specify embedding `FSMGenerator` as `IFSMProducer` into a generator expecting another particular producer interface (as defined in the embedding component's generator context), an adapter between the expected producer interface and `IFSMProducer` and a factory for its injection is generated. When the adaptation is non-trivial, this generated adapter needs to be extended with handcrafted adaptation functionality using the generation gap pattern [24]. For the product interfaces, the same mechanism is applied. The classes of the generator context and signatures of the registration methods follow framework-wide naming conventions [6, 8].

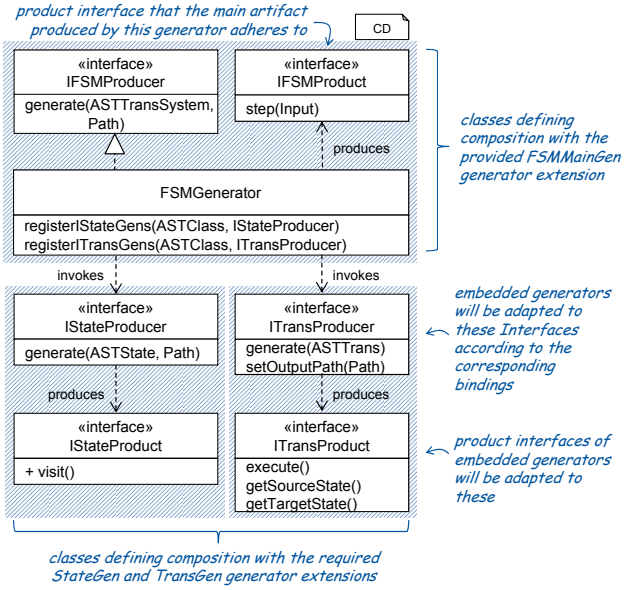


Figure 8: The generator context for the transition system DSL component. It contains the generator interfaces and classes for the extensions of the component.

5.2 DSL Families

DSL families consist of a feature model, components that realize features of the model, and bindings between these components.

Figure 9 depicts an excerpt of the `StateMachineFamily` that describes a family of FSM languages. The family contains a textual feature model to arrange the components within the family (**R2**). It is an excerpt of the one presented in Figure 1 (ll. 3-10). After the feature model, the family defines features in terms of names, realizing component, and bindings (ll. 12-31) such that each feature of the feature model is realized through a DSL component. For instance, in the `StateMachineFamily` the feature `StateMachines` (ll. 12ff) is realized through the DSL component `TransitionSystem` (l. 13). A detailed insight into this component is given in Figure 7.

All non-root features yield bindings that connect the provided extensions of their DSL components with the required bindings of their parent feature's component–or its ancestor, if the parent feature is abstract. For instance, the `InitialAndFinalState` feature is realized through the component `InFinState` (ll. 17ff), which is illustrated in Figure 12. Afterwards (ll. 19-23), bindings for productions, generators and well-formedness rules are defined between the feature-realizing component `InFinState` and the grandparent feature's component `TransitionSystem` (*cf.* Figure 7). Through these bindings, *e.g.,* the provided grammar extension `InitialState` of `InFinState` is bound to the required grammar extension `IState` of `TransitionSystem`.

5.3 DSL Component Customization

The DSL component customization realizes open variability (**R4**). In contrast to the DSL family configuration, it enables the DSL owner to customize a DSL component by binding its required extensions to other DSL components that might not have been part of the

```
01 family StateMachineFamily {                                   LF
02
03   feature diagram StateMachines {        // root feature
04     mandatory PseudoStates {
05       mandatory InitialAndFinalState;
06       abstract History {…} or Junction or //...;
07     }
08     optional HierarchicalStates;
09     optional TimedTransitions;
10   }
11                                  // feature definition
12   feature StateMachines {
13     component ts.comp.TransitionSystem;
14     // Bindings of the StateMachines feature
15   }
16
17   feature InitialAndFinalState {
18     component ps.comp.InFinState;
19     bind production InitialState  -> IState;
20     bind production FinalState    -> IState;
21     bind generator  InitStateGen  -> StateGen;
22     bind generator  FinalStateGen -> StateGen;
23     bind wfrs       CheckStateCardinality;
24   }
25
26   feature TimedTransitions {
27     component tt.comp.TransitionsWithTiming;
28     bind production TimedTrans -> ITrans;
29     bind generator  TTGen      -> TransGen;
30     bind wfrs       TimingCorrectness;
31   }
32   // Definitions of further features
33 }
```

Figure 9: Excerpt of a textual model of the `StateMachineFamily` DSL family (*cf.* Figure 1).

DSL family. Furthermore, the customization can assign values to parameters of the identified DSL component. Figure 10 illustrates a customization by example.

The customization `RobotArmWithClock` customizes the component `RobotArmLang` (see Figure 14). Bindings in the customization have the same syntax as in the feature definition of the language family and are applied in the same fashion. The left side of the binding is the source, *i.e.,* a provided extension or well-formedness rule set, and the right side of the binding is the target, *i.e.,* a required extension or well-formedness rule set. In the customization the source of the binding is the fully qualified name of the language component and the name of the respective provided extension or well-formedness rule set. The required extension or well-formedness set which is the target of the binding, always originates from the customized component. The customization `RobotArmWithClock` binds a grammar production and a generator for a clock expression (ll. 3ff) of a DSL component `ce.comp.Clock` to the customized component `RobotArmLangComp`. Furthermore, it limits the number of initial states to one by setting the corresponding parameter (l. 6). Customization produces a new composed component that contains the bound extensions and no longer contains the set parameter.

5.4 Language Engineering Framework

For demonstration of the feasibility of our approach, we have implemented the framework for deriving languages from the family (**R3**) in four different modules that each relate to different activities and their modeling languages (*cf.* Figure 11).

The `DSL Component Processor` (top left) is responsible for parsing, processing, and validating DSL components (*cf.* Figure 4) and used by the language engineering expert. Therefore, the

```
01 import ce.comp.Clock;                                          CC
02
03 customization RobotArmWithClock for RobotArmLangComp {
04   bind production Clock.ClockExpr        -> ITimedExpr;
05   bind generator  Clock.ClockGen         -> TimerGen;
06   assign          numberOfInitialStates  =  1;
07 }
```

Figure 10: The textual model of a customization for the DSL component `RobotArmWithClock` (*cf.* Figure 14). It contains two bindings and a parameter assignment.

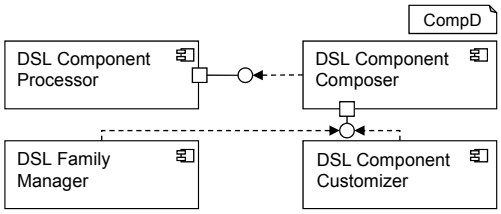


Figure 11: Our framework comprises modules for processing, composing, customizing DSL components, and managing DSL families.

module holds the corresponding language processing tools (parser, lexer, well-formedness checker) as well as an interface for calculating implications. A MontiCore-specific implication calculator implements the calculator interface to resolve and validate implications. After processing a DSL component model, it produces a DSL component that can be reused in DSL families and other contexts.

The `DSL Component Composer` (top right) takes two DSL components with a set of bindings and composes these as presented in Sec. 4.2. It relies on the DSL Component Processor to parse components, before it composes their interfaces and artifacts. For the latter, it provides an interface to integrate modules specific to the technological space operated within. The `DSL Family Manager` (bottom left) evaluates and resolves the DSL families and related feature configurations. To this end, it comprises three modules that process feature models, feature configurations, and resolve the latter. It also interacts with the DSL Component Composer for composing DSL components in the process of applying the DSL family configuration and deriving a new DSL (of component).. The `DSL Component Customizer` (bottom right) reads and applies the customization configuration. It parses and validates customization configurations and applies these.

6 APPLICATION EXAMPLE

This section provides an insight into applying our concepts on the example of deriving an FSM DSL used for describing state-based behavior of a robot arm (*cf.* Sec. 2) [39]. This includes selecting required features from the DSL family, showing the feature-realizing DSL components, their composition, and the artifact composition exemplified via the composition of grammar productions.

Consider language engineering experts that design DSLs for different concerns of Figure 1 and create DSL components accordingly. For instance, the DSL component `InFinState` (see Figure 12) contains a grammar and a generator (ll. 2-3) to provide productions

```
01 dsl component InFinState {                          LC
02   grammar mc.InitialAndFinalState;   <- contained grammar
03   gen IFG context infinstate._gen.InFinGenerators;   <- generator
04
05   provides production InitialState;
06   provides production FinalState;    <- provided grammar extensions
07
08   provides gen InGen  for InitialState with IFG;
09   provides gen FinGen for FinalState  with IFG;   <- provided generator extensions
10
11   wfrs CheckStateCardinality {
12     infinstate._cocos.InitStatesCardinality;
13   }
14                                                  <- well-formedness rule parameter
15   wfr parameter Integer numberOfInitialStates for
16                 infinstate._cocos.InitStatesCardinality;
17 }
```

```
01 grammar InitialAndFinalState {                       MCG
02   InitialState = "initial" "state" Name;
03   FinalState   = "final"   "state" Name;
04 }
```

Figure 12: The DSL component `InFinState` that provides grammar productions and generator for the language elements initial and final state (top) and its grammar (bottom).

```
01 dsl component TransitionsWithTiming {               LC
02   grammar mc.TimedTransition;
03   gen TTG context time._gen.TTGenerators;
04
05   provides production TimedTrans;
06   requires optional production ITimedExpr;
07
08   provides gen InGen    for TimedTrans with TTG;
09   requires gen TimerGen for ITimedExpr with TTG;
10
11   wfrs TimingCorrectness {
12     time._cocos.IsTimingPositive;
13   }
14 }
```

```
01 grammar TimedTransition {                            MCG
02   interface ITimedExpr;
03   TimedTrans = Name "-" timer:ITimedExpr ">" Name;
04   IntegerTimer implements ITimedExpr = IntLiteral "sec";
05 }
```

Figure 13: The DSL component `TransitionsWithTiming` (top) and its grammar (bottom).

and generators for initial and final state definitions (ll. 5-9). In addition, it contains a well-formedness rule that limits the number of initial states. The number itself is configurable via the parameter `numberOfInitialStates` (ll. 15ff). Figure 12 depicts the referenced grammar with the productions provided by the DSL component. The `TransitionsWithTiming` (see Figure 13) provides a production and a generator for timed transitions with a counter that decreases over time and is extensible with additional timing expressions. A DSL family architect then models a family for FSMs. Therefore, she arranges several DSL components resulting in the DSL family `StateMachineFamily` (*cf.* Figure 9).

Based on this family, a DSL owner then can derive an FSM DSL for describing the state-based behavior of robot arms. For this, she selects the features `StateMachines`, the abstract feature `PseudoStates`, `InitialAndFinalState`, and `TimedTransitions` in a feature configuration `RobotArmLang`. The feature `StateMachines` is realized by the `TransitionSystem` component, the feature `InitialAndFinalState` is realized by the component `InFinState` (see Figure 12), and the feature `TimedTransitions` is realized by the `TransitionsWithTiming` component. Based on the feature selection, the DSL components `InFinState` and `TransitionsWithTiming` are composed with the DSL component `TransitionSystem`. This results in a composed DSL component `RobotArmLangComp` (see Figure 14). Since bound provided extensions are embedded into the component that is the target of the binding, the provided extensions of the embedded components are no longer available in the composed component. However, to preserve further extension, the required extensions of the embedded components remain available in the composed component. Thus, the composed component adopts the provided extensions of the component `TransitionSystem` (ll. 7, 12). Furthermore, the required extensions of the embedded component `TransitionsWithTiming` are available (ll. 10, 15). Through binding the well-formedness rule set in the feature `TimedTransitions`, the set `TimingCorrectness` is present in the composed component (ll. 25ff) as well as the well-formedness rule sets of the component `TransitionSystem` (ll. 17-20 and ll. 21-24). As bindings of the feature `InitialAndFinalState` include the well-formedness rule set `CheckStateCardinality` (ll. 28ff) of component `InFinState`, the parameter `numberOfInitialStates` is available in the composed component (ll. 31ff).

The composition of the language artifacts according to the bindings defined by the selected features results in a composed grammar `RobotArmLangGrammar` (l. 2) named after the DSL family configuration and a new generator context `RobotArmLangGenerators` (l. 5). Our approach implements the composition of grammars, generators, and well-formedness rule sets as presented in [6, 7]. Hence, the generator context contains the abstract adapter classes between the producer and product interfaces of the extended and embedded generators that are necessary to compose the implementations of the bound generators.

Figure 15 depicts the composed grammar `RobotArmLangGrammar`. The grammar results from applying the bindings defined in the selected features of the DSL family (see Figure 9). The feature `InitialAndFinalState` defines grammar bindings that bind the provided extensions `InitialState` and `FinalState` of the component `InFinState` to the required extension `IState` of component `TransitionSystem` (ll. 20ff). Identifying the grammar of the provided production requires insights into the DSL component. The component model of `InFinState` (*cf.* Figure 12) references the grammar `InitialAndFinalState` depicted in Figure 12. It contains the two productions referenced by the provided extensions of the DSL component. The required extension of the component `TransitionSystem` references the interface production `IState` of the grammar `TS` (*cf.* Figure 7). From this, the tooling generates the composed grammar depicted in Figure 15. For the grammar bindings of the feature `InitialAndFinalState`, the composed grammar extends the grammars `TS` and `InitialAndFinalState` referenced by the bound components (l. 2). Also, the grammar defines two productions `InitialState2IState` and `FinalState2IState` adapting the bound productions to another. Processing the grammar bindings of the feature `TimedTransitions` is similar to the ones of the feature `InitialAndFinalState`. Here, the composed grammar introduces a new

LC
```
dsl component RobotArmLangComp {
  grammar mc.RobotArmLangGrammar;
  gen FSMG context fsm._gen.TSGenerators;
  gen TTG context time._gen.TTGenerators;
  gen RAG context ra._.RobotArmLangGenerators;

  provides          production TransSystem;
  requires optional production IState;
  requires optional production ITrans;
  requires optional production ITimedExpr;

  provides          gen TSMainGen for TransSystem with FSMG;
  requires optional gen StateGen  for IState      with FSMG;
  requires optional gen TransGen  for ITrans      with FSMG;
  requires optional gen TimerGen  for ITimedExpr  with TTG;

  wfrs TransitionsCorrect {
    fsm._cocos.TransitionSourceStateExists;
    fsm._cocos.TransitionTargetStateExists;
  }
  wfrs TSCorrect {
    fsm._cocos.AllStatesReachable;
    fsm._cocos.NamesAreUpperCase;
  }
  wfrs TimingCorrectness {
    time._cocos.IsTimingPositive;
  }
  wfrs CheckStateCardinality {
    infinstate._cocos.InitStatesCardinality;
  }
  wfr parameter Integer numberOfInitialStates for
                infinstate._cocos.InitStatesCardinality;
}
```

Figure 14: The component resulting from the feature configuration contains the DSL components added by the bindings defined in the selected features.

production implementing the interface of the required extension and extends the production of the provided extension.

The DSL owner needs a timed expression to define a specific trigger time. As this is not available in the DSL family, she customizes the derived DSL component (see Figure 14). With the customization depicted in Figure 10, she binds a production and generator realizing the expression that enables her to define a condition based on a certain time. Here, she limits the number of initial states to one state by setting the parameter `numberOfInitialStates`. The composition produces a new DSL component containing the language features added via the customization. A modeler then can use an FSM DSL tailored specifically to her needs.

MCG
```
grammar RobotArmLangGrammar
        extends FSM, TimedTransition, InitialAndFinalState {
  start StateMachine;

  InitialState2IState extends InitialState implements IState
                     = "initial" "state" Name;
  FinalState2IState extends FinalState implements IState
                     = "final" "state" Name;
 TimedTrans2ITrans extends TimedTrans implements ITrans
                     = Name "-" timer:ITimedExpr ">" Name;
}
```

Figure 15: The composed grammar after applying the feature configuration. It adapts the provided grammar productions to the productions of the required extensions.

7 DISCUSSION AND RELATED WORK

Encapsulating constituents relating to a language concern in an explicit DSL component eases their reuse as it mitigates the challenge of identifying how the usually only loosely coupled language constituents can be reused without becoming an expert in their implementation (**R1**). Through arranging DSL components in families, their systematic reuse can be guided, which eases composing these components accordingly (**R2**). This separation of concerns along the different roles also can liberate domain experts enacting as DSL owners from needing in-depth language engineering expertise (**R3**). Customization enables open variability of DSL components with capabilities not foreseen in the DSL family (**R4**). However, our approach entails additional efforts in defining language components and empirically measuring their impact demands further research.

Our approach to DSL engineering is limited to textual, external, and translations DSL and has comprehensive requirements for compatible technological spaces. Based on these assumptions, it uses specific composition operations, namely embedding (grammars), merging (context conditions), and adapted embedding (code generators). Removing parts of a language by selecting features, thus, is not possible. While the set of valid models can be restricted through adding new features (that contain suitable context conditions), the non-terminals, context conditions, and code generators selected by other features remain part of the language (family). Moreover, we currently use the technological space of MontiCore for realizing our concepts as well as for engineering language families. This might introduce biases towards MontiCore in our concepts, we are currently experimenting with the language workbenches Neverlang [44] and Xtext [18].

Several language engineering tools such as MPS [46], Spoofax [47], and Melange [16] provide means for language composition and customization, but do not provide methods for systematic reuse through DSL families. Other approaches for systematically reusing language parts do not make their interfaces explicit, which hampers reusing these modules [3, 33, 44, 45], or do not support all three component dimensions [16, 36].

Overall, our approach builds upon ideas formulated as concern-oriented language development [13, 31], which proposes to engineer languages based on components (called "concerns") with three kinds of interfaces representing their variability, customization, and use. In this vision, concerns comprise artifacts linked with each other that conform to meta-languages which are typed by "perspectives" contained in libraries. With respect to this vision, our approach addresses the componentization of languages and their systematic reuse only. However, we are unaware of any other similar comprehensive realizations of this part of the vision.

8 CONCLUSION

We have presented concepts for reusing 3D DSL components through closed variability of DSL families (product lines) and open customization. These concepts are intended to be used in a systematic fashion by different stakeholders involved in language engineering, who are supported by a collection of integrated modeling languages to model DSL families and their constituents. While our concepts and their application method are currently limited to textual, external, and translational DSLs, they greatly facilitate DSL reuse and, hence, foster the adoption of modeling languages by domain experts. In the future, we plan to relax our method's assumptions (**A1**-**A5**) and integrate further language definition dimensions.

REFERENCES

[1] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*. Springer, 7–20.

[2] Olaf Berndt, Uwe Freiherr von Lukas, and Arjan Kuijper. 2015. Functional Modelling And Simulation Of Overall System Ship-Virtual Methods For Engineering And Commissioning In Shipbuilding.. In *ECMS*. 347–353.

[3] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[4] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. 2004. Variability management with feature models. *Science of Computer Programming* 53, 3 (2004), 333–352.

[5] Jonathan Bohren and Steve Cousins. 2010. The SMACH High-Level Executive. *IEEE Robotics & Automation Magazine* 17, 4 (2010), 18–20.

[6] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. 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)*. ACM, 75–82.

[7] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM.

[8] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2019. Systematic Composition of Independent Language Features. *Journal of Systems and Software* 152 (2019), 50–69.

[9] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. 2015. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*.

[10] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2–Componentised Language Development for the JVM. In *International Conference on Software Composition*. Springer, 17–32.

[11] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09) (LNCS 5795)*. Springer, 670–684.

[12] Tony Clark, Mark G. J. van den Brand, Benoit Combemale, and Bernhard Rumpe. 2015. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*. Springer, 7–20.

[13] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. 2018. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures* 54 (2018), 139 – 155. http://www.se-rwth.de/publications/Concern-Oriented-Language-Development-COLD-Fostering-Reuse-in-Language-Engineering.pdf

[14] Krzysztof Czarnecki and Ulrich W Eisenecker. 2000. Generative Programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen* (2000), 15.

[15] Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, and Andreas Wortmann. 2019. Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext. In *International Workshop on Domain-Specific Modeling (DSM'19)*, Jeff Gray, Matti Rossi, Jonathan Sprinkle, and Juha-Pekka Tolvanen (Eds.). ACM, 40–49.

[16] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States, 25–36.

[17] Jiwang Du, Qichang He, and Xiumin Fan. 2013. Automating generation of the assembly line models in aircraft manufacturing simulation. In *2013 IEEE International Symposium on Assembly and Manufacturing (ISAM)*. IEEE, 155–159.

[18] Moritz Eysholdt and Heiko Behrens. 2010. Xtext - Implement your Language Faster than the Quick and Dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 307–309.

[19] Stefan Feldmann, Sebastian JI Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan JJ Paredis, and Birgit Vogel-Heuser. 2015. Towards effective management of inconsistencies in model-based engineering of automated production systems. *IFAC-PapersOnLine* 48, 3 (2015), 916–923.

[20] Charles Forsythe. 2013. *Instant FreeMarker Starter*. Packt Publishing.

[21] Ricardo Bedin Franca, Jean-Paul Bodeveix, Mamoun Filali, Jean-Francois Rolland, David Chemouil, and Dave Thomas. 2007. The AADL behaviour annex–experiments and roadmap. In *null*. IEEE, 377–382.

[22] Sanford Friedenthal, Alan Moore, and Rick Steiner. 2014. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.

[23] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. 2010. Using state machines for a model driven development of web service-based sensor network applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. 2–7.

[24] 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. 2015. 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)*. SciTePress, 74–85.

[25] Object Management Group. 2010. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03).

[26] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer* 37, 10 (October 2004), 64–72.

[27] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. 2010. Integrating OCL and textual modelling languages. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 349–363.

[28] Katrin Hölldobler and Bernhard Rumpe. 2017. *MontiCore 5 Language Workbench Edition 2017*. Shaker Verlag. http://www.se-rwth.de/phdtheses/MontiCore-5-Language-Workbench-Edition-2017.pdf

[29] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures* 54 (2018), 386–405.

[30] Botond Kádár, Walter Terkaj, and Marco Sacco. 2013. Semantic Virtual Factory supporting interoperable modelling and evaluation of production systems. *CIRP Annals* 62, 1 (2013), 443–446.

[31] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. 2016. VCU: the three dimensions of reuse. In *International Conference on Software Reuse*. Springer, 122–137.

[32] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley.

[33] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and Picky: Configuration of Language Product Lines. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, 71–80.

[34] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. 2002. Technological Spaces: an Initial Appraisal. *CoopIS, DOA* 2002 (2002).

[35] Aung Sithu Kyaw. 2013. *Unity 4. x Game AI Programming*. Packt Publishing Ltd.

[36] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 11.

[37] Michael Lütjen and Daniel Rippel. 2015. GRAMOSA framework for graphical modelling and simulation-based analysis of complex production processes. *The International Journal of Advanced Manufacturing Technology* 81, 1-4 (2015), 171–181.

[38] David F. Méndez Acuña. 2016. *Leveraging Software Product Lines Engineering in the Construction of Domain Specific Languages*. Ph.D. Dissertation. INRIA Rennes.

[39] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2014. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag.

[40] Chantal Steimer, Jan Fischer, and Jan C Aurich. 2017. Model-based design process for the early phases of manufacturing system planning using SysML. *Procedia CIRP* 60 (2017), 163–168.

[41] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional.

[42] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract features in feature modeling. In *2011 15th International Software Product Line Conference*. IEEE, 191–200.

[43] Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 819–820.

[44] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[45] Markus Völter and Konstantin Solomatov. 2010. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE* 16 (2010), 3.

[46] Markus Völter and Eelco Visser. 2010. Language Extension and Composition with Language Workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 301–304.

[47] Guido H Wachsmuth, Gabriël DP Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE software* 31, 5 (2014), 35–43.

[48] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. 2009. Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE software* 26, 4 (2009), 47–53.

[49] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. 2019. Modeling Languages in Industry 4.0: an Extended Systematic

Mapping Study. *Software and Systems Modeling* 19, 1 (January 2019), 67–94. http://www.se-rwth.de/publications/Modeling-languages-in-Industry-4-0-an-extended-systematic-mapping-study.pdf

[50] Steffen Zschaler, Dimitrios S Kolovos, Nikolaos Drivalos, Richard F Paige, and Awais Rashid. 2009. Domain-specific metamodelling languages for software language engineering. In *International Conference on Software Language Engineering*. Springer, 334–353.

Systems Modeling and Evolution

This section reproduces the publications summarized in Chapter 4.

Paper 7	I. Drave, B. Rumpe, A. Wortmann, J. Berroth, G. Höpfner, G. Jacobs, K. Spütz, T, Zerwas, C. Guist, J. 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, 2020. Reference: [DRW+20]
Paper 8	O. Kautz, B. Rumpe, and A. Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures, In: Automated Software Engineering, 27, pages 119–151, Springer, 2020. Reference: [KRW20]
Paper 9	A. Butting, O. Kautz, B. Rumpe, A. Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In: International Conference on Software Architecture (ICSA'17), Gothenburg, pages 145-154. IEEE, 2017. Reference: [BKRW17b]
Paper 10	A. Butting, O. Kautz, B. Rumpe, and A. Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution, Patrizio Pelliccione, Jan Bosch, Mikic Marija, editors, In: Journal of Systems and Software, 149, pages 437-461, Elsevier, 2019. Reference: [BKRW19]

Modeling Mechanical Functional Architectures in SysML

Imke Drave
Bernhard Rumpe
Andreas Wortmann
Software Engineering
RWTH Aachen University
http://www.se-rwth.de

Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas
Institute for Machine Elements and Systems Engineering
RWTH Aachen University
http://imse.rwth-aachen.de

Christian Guist
Jens Kohl
BMW Group
http://www.bmw.de

ABSTRACT

Innovations in Cyber-Physical System (CPS) are driven by functionalities and features. Mechanical Engineering, on the other hand, is mainly concerned with the physical product architecture, *i.e.,* the hierarchical arrangement of physical components and assemblies that forms the product, which is not explicitly linked to these functions. A holistic model-driven engineering approach for CPS, therefore, needs to bridge the gap between functions and the physical product architecture to enable agile development driven by automation. In the theoretical field of mechanical design methodology, functional architectures describe the functionality of the system under development as a hierarchical structure. However, in practice, these are typically not considered let alone modeled. Existing approaches utilizing mechanical functional architectures, however, do not formalize the relation between the functional architecture and the geometric design. Therefore, we conceived a meta-model that defines modeling-languages for modeling functional architectures of mechanical systems and physical solutions, *i.e.,* interconnections of physical effects and geometries, as refinements of the functional components. We have encoded the meta-model as a SysML profile and applied it within an interdisciplinary, industrial project to model an automotive coolant pump. Our contribution signposts the potential of functional structures to not only bridge the gap between function and geometry in mechanics but also to integrate the heterogeneous domains participating in CPS engineering.

CCS CONCEPTS

• **Cyber-Physical Systems**; • **Functions in Mechanical Engineering**; • **Functional Architectures**; • **SysML-Profile**;

KEYWORDS

Systems Engineering, Cyber-Physical Systems, Functional Architecture, Mechanical Design Methodology SysML, SysML-Profile, Product Development Process

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410938

ACM Reference Format:
Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. 2020. Modeling Mechanical Functional Architectures in SysML. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410938

1 INTRODUCTION

CPS are characterized by the interaction of mechanical, electronic, and software systems [1, 10, 42]. Social and technological challenges [7, 17] as well as the customer's demand for more functionalities and a shorter time-to-market, raise the complexity of engineering such systems [14, 29, 40].

In Mechanical Engineering (ME), engineers integrate the physical components of a product, into assemblies such that geometric constraints, *e.g.,* regarding design space, mounting, and maintenance are satisfied [40]. Further design requirements, such as lifetime requirements, or energy efficiency, contribute to the complexity of mechanical systems. We refer to the hierarchical arrangement of physical components and assemblies that forms the product as the *(physical) product architecture.* Mechanical systems fulfill certain functionalities through physical effects acting between components and assemblies of the physical product. Thus, there is a strong correlation between physical effects and the product architecture. Engineers control the impact of physical effects by manipulating the geometric shape of components or their material, as the effects themselves are set by laws of nature [40]. As a result, mechanical engineers tend to directly design the geometry of components based on given requirements, without explicating the functionality to implement. Therefore, the geometric and physical integration of components into mechanical products has been optimized and the physical product architecture has become the element that structures the development activities in ME [40].

This raises a gap between the functional CPS requirements [14, 29] and the physical product architecture as the physical components are not directly linked to the functions or features they implement. Therefore, reusing existing implementations in other systems is hardly possible and functional testing occurs late in the Product Development Process (PDP), *i.e.,* when changes are cost-intensive.

In Software Engineering (SE), the problem-implementation gap arises whenever the solution to a problem is described at a lower level of abstraction than the problem itself [17]. Model-Driven Engineering (MDE) aims to enable developers to focus on their respective domains by abstracting from the complexities of the

implementation platform [17]. The application of abstraction, separation of concerns, and architectural modeling [8] have shown to reduce this gap effectively [8, 17]. A software architecture describes the functions of the software system under development as a hierarchy of interacting, *i.e.,* message-exchanging, functional components [9]. Among others, formal modeling based on domain-specific adjustable Architecture Description Languages [13, 35, 36] enables verification at early design stages, reuse, and supports automation to enhance agile development in the software domain.

The problem-implementation gap present in automotive software engineering [14, 17, 29] resembles the gap between functional requirements and the product architecture in ME: Functional requirements imposed upon the product are stated by the customer, typically in natural language at a high level of abstraction [29]. Geometric models, such as *e.g.,* Computer-Aided Design (CAD) models, describe the assemblies and components at a level of detail that reaches from the engine as a whole to the screws holding it together. Thus, the product architecture which is part of the mechanical solution domain is described at a lower level of abstraction (the level of screws) than the functional requirements which belong to the mechanical problem domain.

Mechanical design theory considers functional structures, *i.e.,* a hierarchical decomposition of the required system functionality, as a means to systematize the design process in ME [28, 40]. Once formalized as models, functional structures have the potential to not only narrow the gap between functional requirements and the physical product architecture, but also to bridge the gap between the latter and the software architecture. Hence, the contributions of this paper are

(1) a meta-model for functional architectures of CPS from an ME point of view based on [28, 40];
(2) a demonstration of how to encode the meta-model as a SysML profile, that enables mechanical engineers to model these architectures in a way that fosters reuse and early exploration of innovative solutions; and
(3) an example from industry modeling an automotive coolant pump using the profile which emerged as part of an interdisciplinary project to demonstrate the profile's usage and possible benefits.

The contributions aim to signpost the potential of functional architectures not only to systematize the PDP through reuse and automation enabled by formal models but also to enhance collaboration of experts from heterogeneous domains in a holistic CPS engineering approach.

The rest of this paper is structured as follows: Section 2 introduces a running example. Section 3 provides preliminaries regarding the SysML elements extended or reused in in the proposed SysML profile. Section 4 gives insight into functional architectures in mechanical design theory from a language engineering point of view and constitutes the concepts in a meta-model. Section 5 encodes this meta-model as a SysML profile. Section 6 illustrates the results from using the profile for engineering an automotive electrical coolant pump within our project. Section 7 discusses related work and the findings before Section 8 concludes.

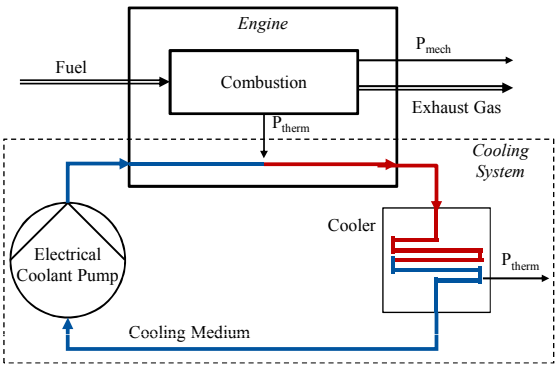


Figure 1: Diagrammatic illustration of an automotive combustion engine with a cooling system.

2 RUNNING EXAMPLE

To illustrate domain and language concepts, we use the following running example from automotive engineering throughout the paper: For propulsion, automotive systems contain a drive system. The main functionality of the drive system is to convert input energy into mechanical energy and to transfer the mechanical energy onto the road, where it causes the vehicle to move. Speaking in terms of components, the engine performs the former, while the drive train and the wheels perform the latter task. Combustion drives, for instance, convert the chemical energy held by fuel into mechanical energy. Electric drives, on the other hand, convert electrical to mechanical energy.

Figure 1 shows the principle set up of a combustion engine diagrammatically. The physical effect that makes combustion engines serve their purpose is the combustion of the fuel that is injected into the engine's cylinders. The combustion converts a portion of the chemical energy held by the fuel into thermal energy which causes the pressure in the combustion chamber to increase. The released exhaust gas holds the rest of the chemical energy. The increasing pressure acts on the surface of the engine's piston as mechanical energy (P_{mech}) causing the piston to move. However, the thermal energy (P_{therm}) is released as heat which causes the engine's temperature to increase. Once a maximum temperature is reached, the engine overheats and stops functioning. To prevent the engine from overheating, it has to be cooled, *i.e.,* the thermal energy released as heat has to be dissipated. Often, water cooling systems, as sketched at the bottom of Figure 1, take on this task. Driven by an electric motor, a cooling medium circulates between the combustion engine and a cooler. By the law of convection [50], the circulating cooling medium absorbs the combustion heat at the engine. The cooler releases the heat absorbed by the cooling medium to the surrounding air. Keeping the cooling medium circulating is required for absorbing the heat, as otherwise convection would not take place. Thus, a cooling medium pump is a necessary component of the cooling system.

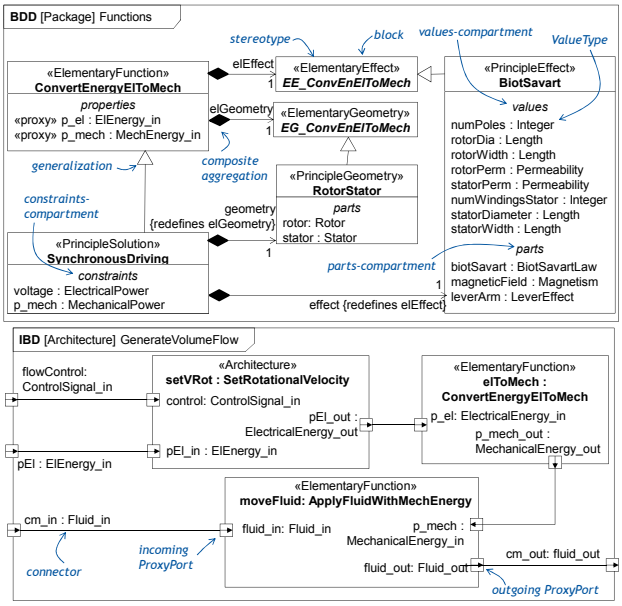


Figure 2: Top: SysML Block Definition Diagram (BDD) of the functional architecture of the running example. Bottom: Internal Block Diagram (IBD) of `GenerateVolumeFlow`. For details on stereotypes and contents see Sections 4 to 6.

3 PRELIMINARIES

The SysML profile introduced in Section 5.2 is tailored for ME and provides a language for explicating the functional structure of a mechanical system, which is understood as a reusable basis of the PDP in [40], in a model. This section briefly summarizes the SysML elements that are extended or reused in the profile.

SysML is a general-purpose modeling language family for systems engineering [33] that reuses and extends a subset of the Unified Modeling Language (UML) 2.5 [32] to represent aspects of system software and hardware in an integrated way [25]. To this end, it comprises four behavior modeling languages, four structure modeling languages, and requirement diagrams. The structure modeling languages comprise BDDs that describe the structure, interfaces, and properties of blocks. IBDs provide a means to describe the internal structure of a block. Figure 2 shows examples for a BDD and an IBD comprising most of the SysML modeling elements reused or specialized in the profile presented in Section 5:

Blocks extend UML classes and are used to model system decomposition, system interaction, and various system properties such as values [33]. The properties of blocks are organized in compartments. The *values*-compartment lists a block's *ValueProperties*, which are *ValueTypes* having composite aggregation, *e.g.,* `numPoles` of the block `BiotSavart`. *PartProperties*, listed in the *parts*-compartment, are blocks that have composite aggregation [33], *e.g.,* `leverArm` of `BiotSavart`. *ConstraintBlocks* are specific blocks used to integrate engineering analyses, *e.g.,* reliability, but also to specify physical constraints as mathematical expressions [33]. *ConstraintProperties* of a block are ConstraintBlocks having composite aggregation, *e.g.,* `voltage` of `SynchronousDriving` in Figure 2. *ConstraintParameters* are the ValueProperties of ConstraintBlocks and represent the variables of such expressions. *ProxyPorts* make features or internal parts of a block available for other components, but do not represent separate parts of the system nor exhibit behavior or comprise internal parts [33]. They are properties of a block typed by *InterfaceBlocks* and identified by the stereotype «proxy», *e.g.,* `p_el` of the block `ConvertEnergyElToMech` typed by the InterfaceBlock `ElEnergy_in`. InterfaceBlocks specify the elements that flow between a block and its environment through *FlowProperties* with direction *in, out* or, *inout* [33]. Section 5 gives more details on InterfaceBlocks and their usage. An internal structure, *i.e.,* the interconnection of a composition of blocks, is modeled by an IBD that belongs to the composed block. The bottom of Figure 2 shows an IBD that models the functional structure of the running example introduced in Section 2. The IBD shows the interaction between the PartProperties of the block `GenerateVolumeFlow` through *Connectors* between the ProxyPorts of the PartProperties. In IBDs, ProxyPorts, typed by InterfaceBlocks that have FlowProperties of only one direction which is not `inout`, hold an arrow showing this unique direction. For example, `p_el` typed by the InterfaceBlock `ElEnergy_in` in the IBD in Figure 2, has only FlowProperties of direction `in`. *Parametric diagrams* are restricted IBDs that show only the usage of ConstraintBlocks. *BindingConnectors* are connectors that specify the equality of the numeric values of the properties at both ends and hold the stereotype «equal» [33]. Being defined on UML [32], SysML offers infrastructure to create profiles by defining stereotypes as extensions of meta-classes or as sub-stereotypes [33].

4 A META-MODEL FOR FUNCTIONAL ARCHITECTURES OF CPS

Ongoing research in ME deals with narrowing the gap between functional requirements and the product architecture. Proclaimed methods differ in terms of terminology and details. Prevalently, these methods describe the product's function as a *functional structure* and use descriptions of *physical effects* as links to geometric components [40, 54]. Design catalogs [28, 44, 53] document recurring elements, such as functions, physical effects, or geometries to enable their systematic variation and rational reuse. In practice, however, mechanical engineers rarely explicate functional structures or utilize design catalogs during development. Mostly, the link between function and geometrical component is kept in the engineer's mind. Formalizing the knowledge from design catalogs and linking the information to detailed models describing physical effects and geometry provides the basis for systematic reuse in an MDE approach. A modeling language that enables to model functions, solutions, geometry and physical effects, where models of solutions comprise the latter two serves this purpose. The formal nature of the modeling language establishes systematic relationships between models of functions and solutions, as well as geometry and physical effect. Models in this language can be (re)used to, *e.g.,* investigate different solutions for products at early development stages by varying solutions to functions. This section summarizes and extends the concepts of [28, 40] and presents a formalizing meta-model [48] which provides the conceptual basis for defining such modeling languages.

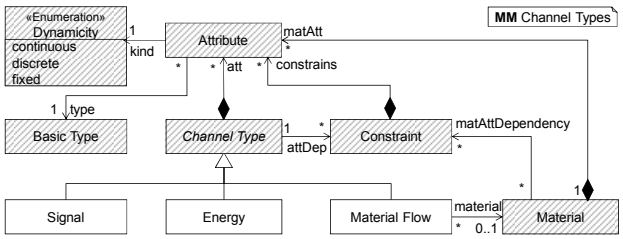


Figure 3: Meta-model that describes the types of functional flows. The shaded classes mark extensions to [28, 40].

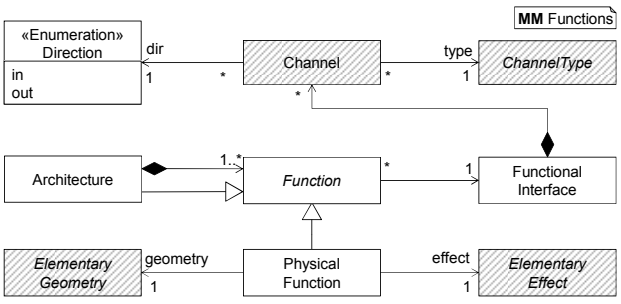


Figure 4: Meta-model of architectures. The shaded classes are extensions of [28, 40].

4.1 Functional Structures

Mechanical design theory gives a definition of function based on the concept that a system or a part of it can be delimited by a boundary, through which physical quantities can enter and leave the system as *functional flows* [40]. The *function* of the delimited system transforms the incoming functional flows to the outgoing flows. Therein, functional flows are classified as flows of *signal*, *energy* and *material* [40, 52]. In the running example (see Section 2), the main function of the pump is to "apply fluid with mechanical energy". The function applies the incoming cooling medium flow with mechanical energy (P_{mech}), such that the cooling medium leaving the system boundary is accelerated. The function of a mechanical system breaks down into several sub-functions linked by functional flows [40]. Functions are referred to as *elementary functions* if the transformation of flows they represent does not physically decompose further [28]. The following section captures and extends these concepts in the meta-models shown in Figure 3 and 4 from a language engineering point of view. The shaded classes mark extensions of [28, 40] to enable the formalization.

Functional Interfaces: To capture kinds of functional flows, our meta-model uses a concept of *channel types*. In an object-oriented fashion [47, 48], *channel type*s specify the type of a functional flow by means of *attributes* and *constraints*. The channel types *energy*, *signal*, and *material flow* [28, 40] comprise attributes representing the characteristics of physical flows of energy or material and of logical signal flows. A material flow may additionally reference a *material* representing the physical material that is flowing in detail, *e.g.,* [31]. Material engineering is out of the scope of this paper and we consider materials solely as types specified by attributes and constraints. Here, constraints are mathematical expressions and represent (physical) dependencies between the attributes of a channel type or a material.

The attributes of channel types have a *basic type* specifying the data type of the attribute. The *dynamicity* specifies how an attribute changes its value during system runtime. For example, theory on software functions often considers signals to change *discrete*ly [11]. Attributes that represent physical characteristics belong to either an energy, or material flow channel type or to a material. These attributes typically represent static, *i.e., fixed*, characteristics of the physical entity, such as *e.g.,* the specific heat capacity of a material, or they represent characteristics that change *continuous*ly at system runtime, *e.g.,* the temperature of a physical part.

Functional Architectures: Describing system functions as a decomposed hierarchy is common to ME [28, 40] and SE [8, 17]. Therein, the functional composition of the sub-functions specifies the behavior of the decomposed function [11, 40]. Each *function* has an *interface* comprising a set of typed *channels* with an unambiguous *direction*, that is either *in* or *out* (*cf.* Figure 4). The meta-model, utilizes the composite pattern [19] to capture an *architecture* as a hierarchical composition of *function*s. The meta-model defines modeling languages for ME, thus, the leaves of a functional architecture are *physical function*s, which capture the elementary functions of [28]. The abstract *elementary effect* and *elementary geometry* bridge the gap between the physical function and its principle solutions since their implementations represent effects and geometries suited to realize the physical function. As detailed in Section 4.2, specifying an interaction of a physical effect and a quantitative geometry adds a behavior to the physical function. Extending the meta-model to capture functional architectures across domains, is possible by integrating description techniques for the behavior of leaf-functions from other domains: Software and control engineering, for example, often utilize various kinds, of automata to specify functional behavior, *e.g.,* [1, 13, 42, 47].

4.2 Effect Catalogs and Principle Solutions

ME created design catalogs to rationalize the PDP by storing proven solutions for recurring design tasks [22, 40]. Various design catalogs exist in ME, *e.g.,* for machine elements [44] or mechanic connections [45]. A popular contribution focuses on physical effects to support engineering solutions that realize a functional structure within the PDP [27, 28, 40]. The catalog comprises 350 physical effects that are mapped to the elementary function they are suited to fulfill [28]. For the conversion of electrical to mechanical energy in the running example (see Section 2), the effect catalog lists, *e.g.,* the *Biot-Savart effect* (*cf.* Figure 11). *Principle solutions* characterize how a physical effect, given a qualitative geometry with certain material properties [40], fulfills a physical function, *i.e.,* the leaf of a functional structure. The principle solution of the physical function "apply fluid with mechanical energy" in the running example (see Section 2) could, for example, specify that the selected effect should be implemented with the principle geometry of a rotating wheel mounted within the cylinder through which the fluid flows, to which Section 5.3 provides further details.

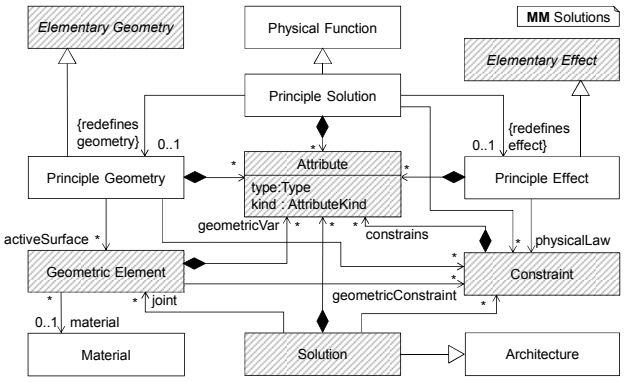


Figure 5: Meta-model of solutions. The shaded classes are extensions to [28, 40].

Meta-model of Solutions. Figure 5 shows the meta-model for principle solutions of physical functions. To enable systematic design of principle solutions to physical functions, the former must be *consistent* to the latter. That is, the specification of the function must be fulfilled by the solution. Therefore, we consider a *principle solution* to refine a *physical function* (*cf.* Figure 4) by selecting a *principle effect* and a *principle geometry*. To support the idea of [28], that elementary functions point to the physical effects suited to realize them, principle effects implement the abstract elementary effect of the physical function. Similarly, suitable principle geometries implement the elementary geometry of the respective physical function. Models of physical functions, principle effects, and principle geometries thereby become reusable. The implementation relation from the elementary geometry and elementary effect to their principle counterparts formalizes the mapping between elementary function and physical effect proposed in [28].

Physical phenomena are described as mathematical formulas over geometric and dynamic variables, captured by the constraints a principle effect is composed of in the meta-model in Figure 5. The lever effect, for example, requires a lever arm to be mounted on a pivot. Putting force on the longer end of the lever arm, by the lever effect, causes a larger force to occur at the other end. The mathematical formula relates the strength of both forces to the lengths of the lever emanating from the pivot. Principle solutions describe how such phenomena can be utilized to fulfill a physical function by relating attributes of the principle effect and the principle geometry through constraints. A physical effect often comprises an interaction of multiple physical laws. Principle effects model physical effects by specifying this interaction as a network of constraints, where each constraint represents a physical law. A principle solution relates the attributes of the principle effect to the attributes of the principle geometry and the attributes of the channel types of the incoming and outgoing channels. The principle effect, thereby, mathematically describes the transformation of the incoming flows to the outgoing flows, *i.e.,* the behavior of the physical function, dependent on the chosen principle geometry. The meta-model considers principle geometries as compositions of *geometric element*s which are types characterized by attributes and constraints between these attributes. The fixed attributes of a geometric element represent static characteristics, *e.g.,* dimensions, continuous attributes specify dynamic characteristics, *e.g.,* velocity, and constraints specify dependencies between these attributes, *e.g.,* that the volume is the product of the dimensions. ME practitioners typically rely on simulation languages to model constraints or physical effects [2, 6, 21, 41, 55], which are tailored particularly for modeling differential equations, and respective tooling often comes with powerful solvers. For geometry, CAD models are the typical choice of the domain experts. Languages encoding the meta-model may integrate such models.

Solutions implement architectures and redefine the inherited physical functions or architectures to principle solutions or solutions, respectively. A solution's components, *i.e.,* (principle) solutions, are interconnected by functional flows as inherited from the architecture. Constraints of a solution express the mathematical dependencies between the attributes of its components.

5 MODELING MECHANICAL FUNCTIONS AND SOLUTIONS IN SYSML

The previous section conceived an expressive meta-model which formalizes and extends the concepts of mechanical design theory [28, 40] from a language engineering point of view. The extension of [28, 40] includes, *e.g.,* the notion of channel types, the systematic mapping of principle solutions to elementary functions as well as the relation between functions and solutions. The latter provides the foundation to systematically define domain-specific modeling languages for the ME domain which enable to model the functional architecture of a mechanical system. In the following, we propose SysML for Functional Mechanical Architectures (SysML4FMArch), a SysML profile which gives a concrete syntax that encodes the meta-model by specifying suitable stereotypes and relations between them.

5.1 Functional Interface

Functions interact by means of signal, material or energy flows [40], which our meta-model captures as channel types. To this effect, SysML4FMArch specifies the stereotypes «Signal», «Energy», and «MaterialFlow», as well as «Material» and «Attribute» which encode the respective elements of the meta-model (*cf.* Figure 3). Attributes are specific ValueTypes with a property of the enumeration type «Dynamicity» which encodes the respective element of the meta-model and indicates how the numeric value of the modeled attribute changes during runtime. The abstract channel type of the meta-model does not have a respective stereotype in SysML4FMArch to assure that functional flows are always classified. The attributes of channel types or materials are represented as «Attribute»-typed ValueProperties. Constraints between these attributes are modeled by ConstraintBlocks and BindingConnectors [33] .

The examples in Figure 6 illustrate this in the BDD and the parametric diagram at the bottom left: A flow of fluid is represented by a «MaterialFlow»-block with three attributes. One is of the real number type `Pressure` and specifies the unit `Pascal` as well as the «Dynamicity» `cont`, which indicates, that ValueProperties typed by `Pressure` change their value continuously during system runtime.

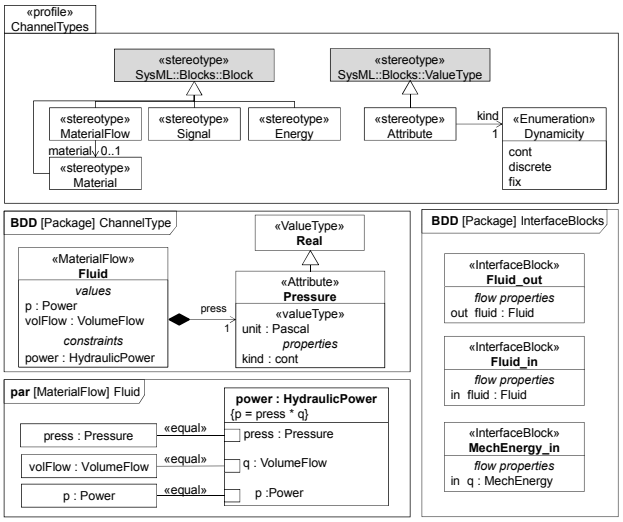


Figure 6: Top: SysML4FMArch's encoding of the meta-model in Figure 3. Bottom: Examples for channel types modeled in SysML4FMArch.

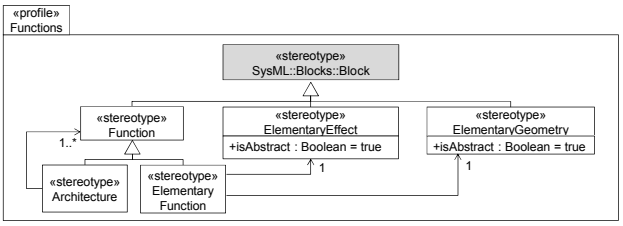


Figure 7: SysML encoding of functions as defined by the meta-model in Figure 4.

The ConstraintBlock `HydraulicPower`, shown at the bottom left, models the physical relationship between the attributes of `Fluid`.

Channels of functional interfaces are represented by ProxyPorts [33]. SysML4FMArch requires the InterfaceBlocks typing these ProxyPorts to have FlowProperties of unambiguous direction, *i.e.,* the usage of direction `inout` or specifying FlowProperties of multiple directions is not allowed. The bottom right of Figure 6 shows examples for InterfaceBlocks to be used for typing ProxyPorts. Note that `Fluid` and `MechEnergy` are models of the physical entities and not of flows of information about these entities.

5.2 Functions

Figure 7 shows how SysML4FMArch encodes the notions of architecture, function, physical function, elementary effect and elementary geometry, that were introduced in Section 4.1.

Architectures and physical functions are encoded as stereotyped blocks (see Figure 7). Since the profile is intended for users with a background in ME, physical functions are encoded by the stereotype «ElementaryFunction» to convey the terminology of [28, 40]. SysML4FMArch provides the «Function»-stereotype that encodes

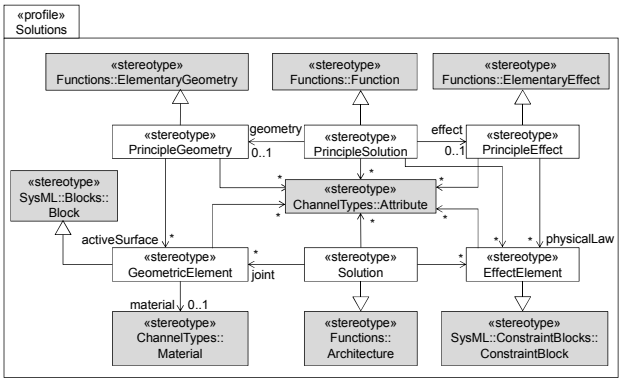


Figure 8: SysML4FMArch encoding of solutions (*cf.* Figure 5).

the notion of function which is abstract in the meta-model. This enables black-box use of functions and allows to postpone specifying a behavior, as it may not be known at early development stages whether the function has to be further decomposed.

To illustrate this, consider the example «Architecture» shown in Figure 2. The IBD at the bottom shows the internal structure of the «Architecture» `GenerateVolumeFlow` which comprises two PartProperties of «ElementaryFunction»-type, *i.e.,* `moveFluid` and `elToMech` as well as the «Architecture» `setVRot`.

The physical functions described in [28] can be digitized by storing respective «ElementaryFunction»-blocks in a SysML ModelLibrary [33]. The general specifications of [28] often need to be refined to integrate them as part of an architecture. Utilizing a specialization of an «ElementaryFunction» allows to refine the function's interface while preserving consistency.

Elementary Effects and Elementary Geometries: A physical function can be realized by selecting a physical effect from a finite list [28, 40]. The interconnection of a physical effect and geometry can be interpreted as the behavior of a physical function. Thus, physical functions comprise an elementary geometry and an elementary effect in the meta-model (*cf.* Section 4.1). SysML4FMArch encodes these elements by abstract blocks with respective stereotypes (*cf.* Figure 7). The abstract «ElementaryGeometry» and «ElementaryEffect» serve as placeholders for the «PrincipleEffect» and «PrincipleGeometry». The latter are implementations of their elementary counterparts that can be selected when creating a «PrincipleSolution» that specializes an «ElementaryFunction» (see Section 5.3). The example at the top of Figure 2 illustrates this: The «PrincipleSolution» `SynchronousDriving` specializes `ConvertEnergyElToMech` which redefines the elementary effect and elementary geometry to their implementations `BioSavart` and `RotorStator`, respectively.

5.3 Solution Architectures

To specify a principle solution for a physical function in an architecture, the engineer chooses implementations of the elementary geometry and elementary effect owned by the physical function and specifies the constraints between the values of both components. Figure 8 shows the encoding of the meta-model in Figure 5 in SysML4FMArch which is detailed in the following paragraphs.

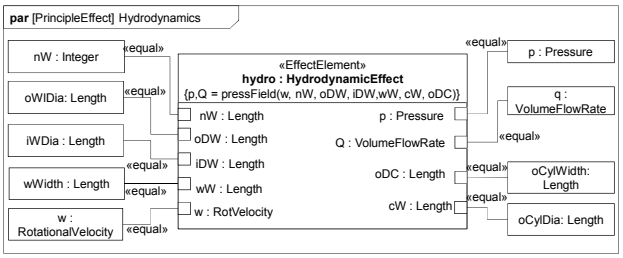


Figure 9: Principle effect representing the cause for turbulences in flowing fluids, modeled in SysML4FMArch.

Effect Elements and Geometric Elements: SysML4FMArch represents geometric elements as blocks with the stereotype «GeometricElement», comprising ValueProperties typed by «Attribute»-ValueTypes. Constraints between attributes of geometric elements are modeled, either as BindingConnectors in case of equalities or as regular ConstraintBlocks in case of more complex mathematical relationships. Effect elements, *i.e.,* physical laws or relations between attributes of principle effects, are modeled as specific SysML ConstraintBlocks with the «EffectElement»-stereotype. Their ValueProperties are typed by «Attribute»-ValueTypes and represent the attributes of the constraint (*cf.* Figure 8). An «EffectElement» may link to a simulation model, which can be realized *e.g.,* as proposed in [25]. In SysML4FMArch, the variables of the simulation are represented by the ConstraintProperties of «EffectElements». The attributes of a geometric element are modeled by the ValueProperties of a «GeometricElement».

Principle Geometry and Principle Effect: Principle geometries comprise the active surfaces between which physical effects come into action. In analogy to the meta-model, a «PrincipleGeometry» therefore comprises «GeometricElement»-PartProperties, each representing an active surface. Since a principle effect is an interaction of physical laws, a «PrincipleEffect» in SysML4FMArch comprises ConstraintProperties typed by an «EffectElement». BindingConnectors connect the «Attribute»-ValueProperties of a «PrincipleEffect» to attributes of its «EffectElement»-ConstraintProperties and, thereby represent equality of the numeric values of the ValueProperties at the connector's ends.

To illustrate this, consider the principle effect modeled in Figure 9. The equation $p \cdot Q = M \cdot \omega$ (1) describes the physical law that causes turbulences, *i.e.,* a rotational velocity, within a flowing fluid [43]. Here, p is the fluid's pressure, Q is the volume flow rate, ω is the rotational velocity, and M is a momentum imposed by a mechanical energy. Thus, the «PrincipleEffect» `Hydrodynamics` has «Attributes» of respective types which all specify the «Dynamicity» `cont` (*e.g.,* Figure 6 shows the definition of `Pressure`). The momentum strongly depends on the geometric setup, through which the fluid is flowing. In the context of the running example (*cf.* Section 2), we assume that the fluid flows through a tubular pipe with a length of `oCylWidth` and a diameter of `oCylDia`. In the pipe, the fluid flows through a paddle wheel with `nW` paddles, an outer diameter of `oWDia`, an inner diameter of `iWDia`, and a width of `wWidth`. These ValueProperties of `Hydrodynamics` represent geometric variables of `fix` kind, as these attributes are assumed to not change their value at system runtime. The «EffectElement» `HydrodynamicEffect` links to a simulation model that calculates a difference `p` between pressures of the incoming and the outgoing fluid, and a volume flow rate `q` according to Equation (1). BindingConnectors between the ConstraintParameters of the ConstraintProperty `hydro` model the physical relationships between the attributes of the principle effect as stated by the physical law modeled by the «EffectElement» `HydrodynamicEffect`.

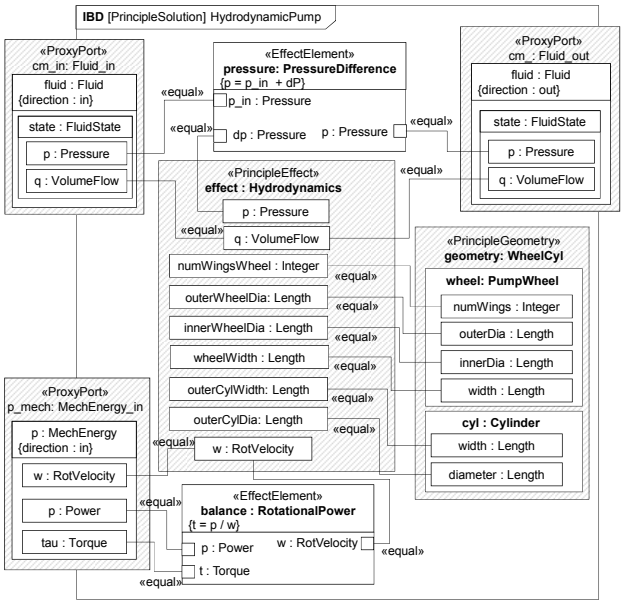


Figure 10: Model of a possible principle solution to realize the elementary function `ApplyFluidWithMechanicalEnergy`.

Solutions and Principle Solutions: A principle solution inherits the functional interface, the elementary geometry and the elementary effect from the physical function it fulfills. By redefining the latter two to concrete implementations, *i.e.,* principle effect and principle geometry, the engineer creates a solution to realize the functionality [28]. The interaction of the principle effect and the principle geometry specifies the behavior of the principle solution. Figure 8 shows how SysML4FMArch encodes this: Principle solutions are represented by blocks with the respective stereotype composed of parts typed by a «PrincipleGeometry» and a «PrincipleEffect» (*cf.* Figure 8). A «PrincipleSolution» must specialize an «ElementaryFunction» and may redefine the inherited «ElementaryGeometry» and «ElementaryEffect» to a «PrincipleGeometry» and a «PrincipleEffect», respectively. The selection of either one may be delayed, indicated by the multiplicity `0..1`. SysML4FMArch uses BindingConnectors to specify the constraints between attributes of principle geometries and principle effects as well as the function's interface. Effectively, this models the behavior of the function by specifying how the function changes the attributes in the representations of the flows that enter or leave the function through its interface. The physical interaction of principle

geometry and principle effect are represented by BindingConnectors between the «EffectElement»-ConstraintProperties and the «GeometricElement»-PartProperties of a «PrincipleSolution». The constraints between ValueProperties of a «Solution» and the contained «PrincipleSolution» components are modeled equivalently.

Figure 10 shows an IBD of HydrodynamicPump, which specializes the «ElementaryFunction» ApplyFluidWithMechEnergy and inherits its interface. Hydrodynamics specializes the «ElementaryEffect» of ApplyFuidlWithMechEnergy (both specialization relations are not shown in Figure 10). A simulation linked to the «EffectElement» modeling Equation (1) assumes the fluid to flow through a tubular pipe comprising a paddlewheel. The «PrincipleGeometry» WheelCyl specializes the «ElementaryGeometry» of ApplyFluidWithMechEnergy and has PartProperties of type PumpWheel and Cylinder. These represent a pair of active surfaces possible to enforce the represented effect, and assign the attributes of the effect to distinguishable geometric shapes. The pressure of the outgoing fluid is determined as the sum of the pressure of the incoming fluid and the pressure difference which results from the hydrodynamic effect acting on the fluid, which is modeled by the «EffectElement» PressureDifference. The rotational velocity, power, and torque imposed by the incoming mechanical energy must obey the law of energy conservation modeled by the «EffectElement» RotationalPower.

6 MODELING EXAMPLE: AUTOMOTIVE ELECTRICAL COOLANT PUMP

This section presents an extract from the results of an interdisciplinary industrial project, involving researchers from SE and ME as well as practitioners from the automotive industry. In the project, we have applied SysML4FMArch to model the cooling system for an automotive combustion engine drive train (*cf.* Section 2). This section presents and explains the SysML4FMArch models of the coolant pump, a part of the cooling system, in detail.

6.1 Architecture of the Electric Coolant Pump

The coolant pump's main functionality is to keep the cooling medium flowing which is physically necessary for it to absorb the heat from the engine's cylinders. The IBD of the «Architecture» GenerateVolumeFlow in Figure 2 shows the decomposition of this functionality. The architecture has ProxyPorts representing three incoming flows, *i.e.,* cm_in which represents an incoming cooling medium, an electrical energy pEl, and a signal flowControl, as well as cm_out which represents the outgoing flow of the cooling medium. Figure 6 shows the InterfaceBlocks for typing the ProxyPorts representing the functional flows of fluid. The other InterfaceBlocks have FlowProperties typed by a «Signal» ControlSignal which is defined similar to Fluid but specifies the unit m/s and the AttributeKind discrete, and by an«Energy» ElEnergy which represents electrical energy by means of a real number, the unit Watt, and the AttributeKind cont. The flow flowControl represents an information flow (changing its value discretely at runtime) telling how fast the outgoing fluid has to flow, in order to absorb enough heat from the engine, which enters the function SetRotationalVelocity. The latter is modeled as «Architecture» that calculates a necessary amount of electrical energy. The outgoing signal flow enters an actuator function which

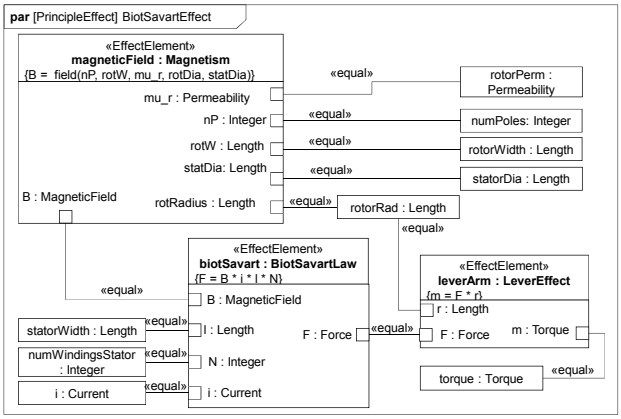


Figure 11: The Biot-Savart effect is an interaction of physical laws: Magnetism, the LeverEffect and the BiotSavartLaw. The figure shows a model of this principle effect to be used in principle solutions of ConvertEnergyElToMech .

outputs a flow of electrical energy (pEl_out). The «ElementaryFunction» ConvertEnergyElToMech represents a physical function that converts the flow of electrical energy into a flow of mechanical energy p_mech_out. The physical function represented by the «ElementaryFunction» ApplyFluidWithMechEnergy impinges this mechanical energy p_mech upon the incoming fluid cm_in and resulting in the outgoing flow fluid_out.

6.2 Solution-Models

The «Architecture» GenerateVolumeFlow comprises two «ElementaryFunctions» for which [28] lists physical effects suitable to realize their functionality. In the solution of this architecture considered here, the hydrodynamic effect provides the acceleration of the fluid specified by ApplyFluidWithMechEnergy. Figure 9 shows the SysML4FMArch-model of the principle solution using the hydrodynamic effect which was explained previously in Section 5.3. This section details a principle solution to convert electrical to mechanical energy using the *Biot-Savart-Effect* [24]. This principle solution realizes the elementary function ConvertEnergyElToMech in our running example (*cf.* Figure 2).

Principle Solution to Convert Electrical to Mechanical Energy: The BDD in Figure 12 shows the «PrincipleEffect» BiotSavart which specializes the «ElementaryEffect» EE_ConvEnElToMech. The principle effect is an interaction of multiple physical laws, therefore, BiotSavart comprises three «EffectElements», *i.e.,* Magnetism, BiotSavartLaw and LeverEffect, connected by BindingConnectors which represents the following: An electromagnetic coil (stator) is positioned within a magnetic field B. The magnetic field is created by a permanent-magnet (rotor), that is placed at a distance r to a rotation axis such that it may rotate around the stator. Once a voltage implies a current i in the conductor, the Lorentz-force starts acting on the rotor. Due to the *lever-effect*, a mechanical torque M occurs around the rotation axis, causing the rotor to rotate. The rotation reflects the existence of mechanical energy. The physical laws are (1) $B = \mu_0 \cdot \mu_r \cdot H$, (2) $M = F \cdot r$, and (3) $F = B \cdot i \cdot l \cdot N$, where

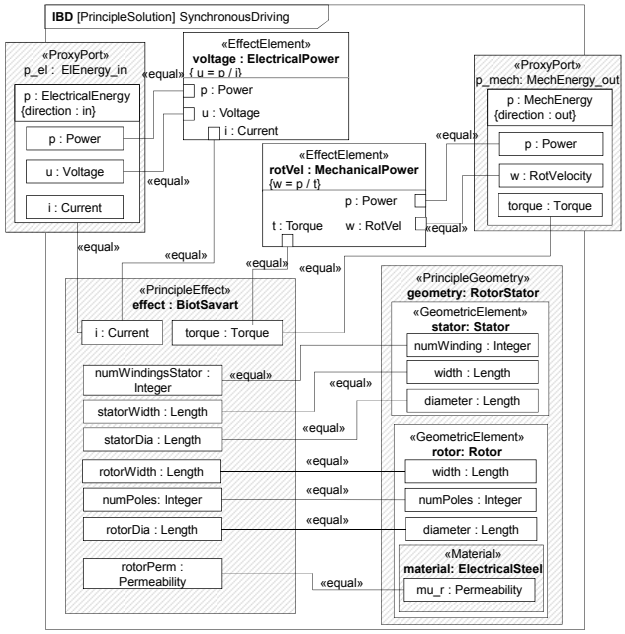


Figure 12: Principle solution of `ConvertEnergyElToMech` using the «PrincipleEffect» `BiotSavart`, which represents the electric drive of the cooling pump.

μ_0 is the vacuum permeability, μ_r is the permeability of the rotor, and H is the magnetic field strength induced by the rotor. Further, l is the length and N the number of windings of the stator. If losses are not considered, electrical input power is equal to mechanical output power (see, *e.g.,* [24] for details).

Figure 11 shows a SysML4FMArch-model of this principle effect: The magnetic field strength H depends on the number of poles `numPoles` and the diameter of the rotor as well as the diameter of the conductor. By means similar to [25], the «EffectElement» `Magnetism` links to a simulation model that calculates the magnetic field `B` from the geometric attributes of the stator, *i.e.,* the conductor and the rotor, according to Equation (1). The «EffectElement» `BiotSavartLaw` models Equation (2): The Lorentz-force `F` depends on the magnetic field `B`, the electric current `i`, the stator's `statorWidth` and the number of windings of the stator `numWindingsStator`. The «EffectElement» `leverEffect` models Equation (3): The torque that acts around the rotation axis depends on the Lorentz-force acting on the rotor and length of the lever arm, *i.e.,* the diameter of the rotor `rotorDia`.

Figure 12 shows a principle solution to apply a fluid with mechanical energy that (re-)uses the «PrincipleEffect» `BiotSavart`: `SynchronousDriving` specializes the «ElementaryFunction» `ConvertEnergyElToMech` (*cf.* Figure 2) and therefore inherits the interface. Further, `SynchronousDriving` specifies the «PrincipleEffect» `BiotSavart` explained above and the «PrincipleGeometry» `RotorStator` which models a geometry comprising a rotor and a stator, both characterized by attributes of `fix` AttributeKind. The BindingConnectors between the attributes of the modeled principle effect and of the geometric elements of the represented principle geometry as well as the attributes of the represented channel types model the equality of their numeric values. The «EffectElements» `ElectricalPower` and `MechanicalPower` represent the physical law of energy conservation.

Solution to Generate a Volume Flow. The models of the principle solutions introduced above can be used to model a solution to the «Architecture» `GenerateVolumeFlow` whose internal structure is modeled in Figure 2. A solution to `GenerateVolumeFlow` is a «Solution»-block that specializes the «Architecture» `GenerateVolumeFlow`. The «PrincipleSolution»-blocks `HydrodynamicPump` and `SynchronousDriving` specialize the «ElementaryFunction»-blocks `ApplyFluidWithMechEnergy` as well as `ConvertEnergyElToMech`, respectively. The latter type the PartProperties `moveFluid` and `elToMech` of `GenerateVolumeFlow`, as shown in Figure 2. A «Solution» to this «Architecture» inherits the interface and the PartProperties of `GenerateVolumeFlow`. By redefining `ConvertEnergyElToMech` to `SynchronousDriving` and `ApplyFluidWithMechEnergy` to `HydrodynamicPump`, this solution integrates these «PrincipleSolutions» and forms a model of the solution to the entire architecture. In this case, the solution models an electrical coolant pump.

7 RELATED WORK AND DISCUSSION

Modeling as the act of describing or prescribing properties of the system under development, is the essential foundation for systematically engineering (cyber-physical) systems. MDE employs formal modeling languages to enable frontloading of analysis and design exploration to reduce engineering costs, facilitate collaboration among domain experts, and supports the synthesis of system parts by automation. Research on this topic is scattered across the domains of CPS engineering, including ME and SE.

Related Work. Ongoing research has produced theories, and modeling languages for engineering software and electronic functions of CPS, *e.g.,* [1, 42], as well as for designing [23, 49], engineering [4], and operating [3] CPS in various domains. Most of these approaches consider modeling only through the lens of software engineering, *i.e.,* for discrete and functional systems. Where continuity and geometry are supported, the theories and languages do not support established processes or modeling concepts from other (*i.e.,* the "physical") domains, such as ME.

In the Focus theory [11], systems are composed of components that realize stream processing functions. As functions communicate via channels only, they can be (de)composed and refined systematically, where refinement considers both, structure and the behavior of components. Applying Focus's notion of refinement within a model-driven functional PDP is subject to ongoing research.

In ME, a variety of design catalogs to aid the design process regarding various aspects [18] exist in the literature, *e.g.,* [40, 46]. Approaches that digitize such catalogs, *e.g.,* [18, 34], focus on making the (extended) information from existing design catalogs accessible by providing digital textual descriptions complemented with mathematical expressions or sketches. Lacking a representation in a formal modeling language that also enables to integrate the information within a functional architecture of a mechanical system hinders to apply these approaches in a model-driven PDP. Modeling languages based on UML or SysML have

emerged or been used in the ME domain, *e.g.,* MechatronicUML [12], SysML4Modelica [5], or SysML4Mechatronics [26], in the field of production systems engineering [16], and in the context of Industry 4.0 [58], *e.g.,* UML4IoT [51]. Neither of these languages enable to relate (elementary) functions and (principle) solutions of mechanical systems as part of a systematic PDP. The FAS-method [30, 56], extended for ME by FAS4M [39] promotes modeling functional architectures for system design and both define respective SysML profiles. As introduced in [37, 38] the latter uses trace links to underly SysML elements with informal sketches of geometric components. The focus of these contributions lies on the connection between requirements and function. In contrast to our approach, principle solutions, here, are described by informal sketches that neither distinguish between principle geometry and principle effect nor enable automatic processing. This prevents utilizing the information from design catalogs such as, *e.g.,* [28], and to compose the physical product architecture of geometric elements related to physical effects by a principle solution. This holds similarly for the techniques proposed in [15, 20, 57, 59]. In particular, the approaches in [15, 20] do not consider functional structures in the sense of [28, 40] (see Section 4.1) and do not systematically establish consistency between function and principle geometry in a model-driven approach. Currently, precise modeling languages tailored to support the PDP based on the foundations of functional architectures established in [27, 28] do not exist. Explicit modeling techniques for the PDP in ME do not support the time-honored paradigms that paved the way for the success of software engineering, such as abstraction, automation, composition, refinement, and separation of concerns.

Discussion. The meta-model and its encoding in SysML emerged during an interdisciplinary project comprising researchers from SE and ME as well as practitioners from the automotive industry. So far, the meta-model captures and extends the notion of functional architectures prevalent in mechanical design theory [28, 40]. Therein, components interact through functional flows, and physical functions are implemented by principle solutions, *i.e.,* an interaction of a physical effect and a geometry. Integrating description techniques for functional behavior prevalent, *e.g.,* in the software [11, 13] and control engineering domains [1, 42], is subject to ongoing research. As SysML is fairly in known in the automotive domain [14, 29] and since there exist modeling tools with integrated model-processing, we encoded the meta-model as a SysML profile. To test the approach in the ME domain, we engineered an automotive coolant pump and modeled its functional architecture as well as the solutions to each function in SysML4FMArch. The model comprises multiple SysML4FMArch-diagrams which were presented exemplary throughout the paper. The systematic relation between functions and solutions in the SysML4FMArch-models enabled to use the tooling effectively for automation during the mechanical design process. For example, we tested the suitability of the chosen principle solutions (*cf.* Figure 10 and Figure 12) by linking virtual simulations to the effect elements in the SysML4FMArch model and for virtual dimensioning of the pump wheel, *i.e.,* the automatic manipulation of values of its geometric attributes, (*cf.* Figure 10), which was enabled by utilizing the automatic model execution functionality of existing SysML tools. Towards the end of the project, the pump wheel was 3D-printed to obtain a prototype of a part of the physical product. Further automation for functional testing and dimensioning as well as digitizing a design catalog such as [28] are subject to ongoing research. However, SysML has its drawbacks, *e.g.,* regarding modeling efficiency, and intuitiveness. SysML's nature as a general purpose language and the inherited UML concepts decrease understanding and ease of use for ME practitioners, as, for them, these concepts are not as intuitive as for SE practitioners. Further, the graphical nature of SysML may hinder manageability of SysML4FMArch models with many attributes. The lack of formal semantics for SysML hinder the implementation of product-specific automatic model processing and tailored model analyses based on mathematical theories.

8 CONCLUSION

This paper formalized and extended the concepts of [28, 40] in a meta-model that defines modeling languages for the ME domain. Further, we encoded the meta-model in the SysML profile SysML4FMArch and employed the language within in an interdisciplinary, industrial project to engineer an automotive coolant pump. As a result, the models could be used for automatic, virtual dimensioning and testing, which holds out the prospect of an agile model-driven PDP supported by automation. While SysML has its drawbacks regarding formality and intuitiveness, the results of the project signpost the potential of utilizing modeling languages for explicating functional architectures of a technical system and making the knowledge of design catalogs assimilable for a holistic MDE approach that narrows the gap between the functional and the product architecture by means of abstraction.

REFERENCES

[1] Rajeev Alur. 2015. *Principles of cyber-physical systems.*

[2] Faysal Andary, Joerg Berroth, and Georg Jacobs. 2019. An Energy-Based Load Distribution Approach for the Application of Gear Mesh Stiffness on Elastic Bodies. *Journal of Mechanical Design* 141, 9 (2019).

[3] Patrick Bareiss, Daniel Schütz, Rafael Priego, Marga Marcos, and Birgit Vogel-Heuser. 2016. A model-based failure recovery approach for automated production systems combining sysml and industrial standards. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA).* IEEE, 1–7.

[4] Luca Berardinelli, Stefan Biffl, Arndt Lüder, Emanuel Mätzler, Tanja Mayerhofer, Manuel Wimmer, and Sabine Wolny. 2016. Cross-disciplinary engineering with AutomationML and SysML. *at-Automatisierungstechnik* 64, 4 (2016), 253–269.

[5] Olaf Berndt, Uwe Freiherr von Lukas, and Arjan Kuijper. 2015. Functional Modelling And Simulation Of Overall System Ship-Virtual Methods For Engineering And Commissioning In Shipbuilding.. In *ECMS.* 347–353.

[6] Joerg Berroth, Georg Jacobs, Tobias Kroll, and Ralf Schelenz. 2016. Investigation on pitch system loads by means of an integral multi body simulation approach. *Journal of Physics: Conference Series* 753 (2016).

[7] Manfred Broy. 2006. Challenges in Automotive Software Engineering. In *Proceeding of the 28th international conference on Software engineering - ICSE '06.*

[8] Manfred Broy. 2007. Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *Innovations Syst Softw Eng* 3 (2007).

[9] Manfred Broy. 2010. A Logical Basis for Component-Oriented Software and Systems Engineering. *Comput. J.* 53, 10 (2010).

[10] Manfred Broy, · Heinrich Daembkes, and Janos Sztipanovits. 2019. Editorial to the theme section on model-based design of cyber-physical systems. *Software & Systems Modeling* 18 (2019).

[11] M Broy and Ketil Stølen. 2001. *Specification and development of interactive systems.*

[12] Sven Burmester, Holger Giese, and Matthias Tichy. 2004. Model-driven development of reconfigurable mechatronic systems with mechatronic UML. In *Model Driven Architecture.* Springer, 47–61.

[13] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference*

on Modelling Foundations and Applications (ECMFA'17) (Marburg) *(LNCS 10376)*. Springer, 53–70.
[14] Imke Drave, Steffen Hillemacher, Timo Greifenberg, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing.
[15] Martin Eigner, Torsten Gilz, and Radoslav Zafirov. 2012. Proposal for functional product description as part of a PLM solution in interdisciplinary product development.
[16] Stefan Feldmann, Sebastian JI Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan JJ Paredis, and Birgit Vogel-Heuser. 2015. Towards effective management of inconsistencies in model-based engineering of automated production systems. *IFAC-PapersOnLine* 48, 3 (2015), 916–923.
[17] Robert France and Bernhard Rumpe. 2007. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE*.
[18] H.-J Franke, S Löffler, and M Deimel. 2004. Increasing the Efficiency of Design Catalogues By Using Modern Data Processing Techniques. In *DS 32: Proceedings of DESIGN 2004, the 8th International Design Conference, Dubrovnik, Croatia*.
[19] Erich. Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley.
[20] Jürgen Gausemeier, Rafal Dorociak, Sebastian Pook, Alexander Nyßen, and Axel Terfloth. 2010. Computer-Aided Cross-Domain Modeling of Mechatronic Systems, Vol. 2.
[21] Reza Golafshan, Georg Jacobs, Matthias Wegerhoff, Pascal Drichel, and Joerg Berroth. 2018. Investigation on the Effects of Structural Dynamics on Rolling Bearing Fault Diagnosis by Means of Multibody Simulation. *International Journal of Rotating Machinery* 2018 (2018), 1–18.
[22] Karl-Heinrich Grote and Erik K. Antonsson. 2009. *Springer Handbook of Mechanical Engineering*. Springer, Berlin.
[23] IEEE Architecture Working Group. 2000. *IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems*. Technical Report. IEEE.
[24] Austin Hughes and Bill Drury. 2019. *Electric motors and drives: fundamentals, types and applications*. Newnes.
[25] Thomas Johnson, Aleksandr Kerzhner, Christiaan J.J. Paredis, and Roger Burkhart. 2012. Integrating models and simulations of continuous dynamics into SysML. *Journal of Computing and Information Science in Engineering* 12, 1 (2012).
[26] K. Kernschmidt and B. Vogel-Heuser. 2013. An interdisciplinary SysML based modeling approach for analyzing change influences in production plants to support the engineering. In *2013 IEEE International Conference on Automation Science and Engineering (CASE)*.
[27] Rudolf Koller. 2014. *Konstruktionslehre für den Maschinenbau - Grundlagen zur Neu- und Weiterentwicklung technischer Produkte mit Beispielen* (4 ed.). Springer, Berlin.
[28] Rudolf Koller and Norbert Kastrup. 1998. *Prinziplösungen zur Konstruktion technischer Produkte*. Springer Berlin.
[29] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. 2018. Improving model-based testing in automotive software engineering. In *Proceedings - International Conference on Software Engineering*.
[30] Jesko G. Lamm and Tim Weilkiens. 2014. Method for Deriving Functional Architectures from Use Cases. 17 (2014).
[31] Donald Leo. 2008. *Engineering Analysis of Smart Material Systems*.
[32] Object Management Group. 2015. OMG Unified Modeling Language (OMG UML) Version 2.5.
[33] Object Management Group. 2019. OMG Systems Modeling Language (OMG SysML) Version 1.6.
[34] Johannes Mathias, Tobias Eifler, Roland Engelhardt, Hermann Kloberdanz, Herbert Birkhofer, and Andrea Bohn. 2011. Selection of Physical Effects Based on Disturbances and Robustness Rations in The Early Phases of Robust Design. In *International Conference on Engineering Design*. 11–15.
[35] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. 2007. Moving architectural description from under the technology lamppost. *Information and Software Technology* 49, 1 (2007), 12–31.
[36] Nenad Medvidovic and Richard N. Taylor. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering* 26 (2000).
[37] Georg Moeser. 2015. Example on "Usage of Free Sketches in MBSE". (04 2015). https://doi.org/10.5445/IR/1000047231
[38] G. Moeser, A. Albers, and S. Kümpel. 2015. Usage of free sketches in MBSE raising the applicability of Model-Based Systems Engineering for mechanical engineers. In *2015 IEEE International Symposium on Systems Engineering (ISSE)*. 50–55.
[39] Georg Moeser, Christoph Kramer, Martin Grundel, Michael Neubert, Stephan Kümpel, Axel Scheithauer, Sven Kleiner, and Albert Albers. 2015. Fortschrittsbericht zur modellbasierten Unterstützung der Konstrukteurstätigkeit durch FAS4M. In *Tag des Systems Engineering*. Carl Hanser Verlag GmbH & Co. KG, 69–78.
[40] Gerhard Pahl, W. Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. 2007. *Engineering Design - A Systematic Approach* (3 ed.). Springer, London.
[41] Gerwin Pasch, Georg Jacobs, Gregor Höpfner, and Joerg Karl Berroth. 2019. Multi-Domain Simulation for the Assessment of the NVH Behaviour of a Tractor with Hydrostatic-Mechanical Power Split Transmission. *Land.Technik AgEng 2019 : Hannover* 77th International Conference on Agricultural Engineering / VDI-Wissensforum ; Supporters: VDI Max-Eyth Society for Agricultural Engineering (2019).
[42] Claudius Ptolemaeus. 2014. System Design, Modeling, and Simulation using Ptolemy II.
[43] Sulzer Pumps. 2010. *Centrifugal Pump Handbook*. Elsevier.
[44] Karlheinz Roth. 1994. *Konstruieren mit Konstruktionskatalogen - Band I: Konstruktionslehre* (2 ed.). Springer, Berlin.
[45] Karlheinz Roth. 1996. *Konstruieren mit Konstruktionskatalogen - Band III: Verbindungen und Verschlüsse - Lösungsfindung* (2 ed.). Springer, Berlin.
[46] Karlheinz Roth. 2011. Selection of Physical Effects Based on Disturbances and Robustness Rations in The Early Phases of Robust Design. In *International Conference on Engineering Design*.
[47] Bernhard Rumpe. 2016. *Modeling with UML: Language, Concepts, Methods*. Springer International.
[48] Bernhard Rumpe. 2017. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International.
[49] Chantal Steimer, Jan Fischer, and Jan C Aurich. 2017. Model-based design process for the early phases of manufacturing system planning using SysML. *Procedia CIRP* 60 (2017), 163–168.
[50] K. Stephan. 1994. Thermodynamics. In *Dubbel Handbook of Mechanical Engineering*. Springer London.
[51] Kleanthis Thramboulidis and Foivos Christoulakis. 2016. UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry* 82 (2016), 259–272.
[52] Karl Ulrich and Steven Eppinger. 2003. *Product Design and Development* (3 ed.). McGraw-Hill, New York.
[53] VDI. 1982. *VDI 2222 Blatt 2 - Konstruktionsmethodik - Erstellung und Anwendung von Konstruktionskatalogen*. Beuth Verlag, Berlin.
[54] VDI. 1997. *VDI 2222 Blatt 1 - Konstruktionsmethodik - Methodisches Entwickeln von Lösungsprinzipien*. Beuth Verlag, Berlin.
[55] Matthias Wegerhoff, Georg Jacobs, and Pascal Drichel. 2018. Noise, vibration and harshness validation methodology for complex elastic multibody simulation models: With application to an electrified drive train. *Journal of Vibration and Control* 25, 2 (2018).
[56] Tim. Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker. 2015. *Model-based system architecture*.
[57] Stefan Wölkl and Kristina Shea. 2009. A Computational Product Model for Conceptual Design Using SysML.
[58] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. 2020. Modeling Languages in Industry 4.0: an Extended Systematic Mapping Study. *Software and Systems Modeling* 19, 1 (January 2020), 67–94.
[59] Christian Zingel, Albert Albers, Sven Matthiesen, and Michael Maletz. 2012. Experiences and Advancements from One Year of Explorative Application of an Integrated Model-Based Development Technique Using C&C²-A in SysML. *International Journal of Computer Science* 34-39 (2012).

[KRW20] O. Kautz, B. Rumpe, A. Wortmann:
Automated semantics-preserving parallel decomposition of finite component and connector architectures.
In: Automated Software Engineering, Springer, April 2020.
www.se-rwth.de/publications/

Automated semantics-preserving parallel decomposition of finite component and connector architectures

Oliver Kautz[1] · Bernhard Rumpe[1] · Andreas Wortmann[1]

Received: 6 February 2019 / Accepted: 20 March 2020

Abstract

For the systematic development of logical, message-driven architectures, automating parallel decomposition of software components is important to achieve efficient modular and parallel system development. During development, monolithic components that realize multiple independent concerns need to be decomposed to obtain a higher quality architecture of cohesively encapsulated, better comprehensive components. Previous work did not address automated parallel decomposition of finite message-driven and logically timed components with respect to the influence of messages received via input channels on the messages sent via output channels. This, however, is a necessary prerequisite to enable the analysis of event chains across logically distributed architectures. To address this, we present a concept of influence between channels of components that supports automated semantics-preserving parallel decomposition of finite deterministic component implementations into independent, more comprehensible components that are better accessible for analysis and development. Therefore, we extend the Focus theory of time-synchronous components with the concept of influence, present a decomposition procedure leveraging this, and prove that the resulting system is semantically equivalent. This enables automatically decomposing monolithic software components (e.g., for stepwise refinement or refactoring) into smaller components of better cohesion and comprehensibility and thus facilitates automated software engineering.

Keywords Automated modeling · Architecture decomposition · Refinement · Refactoring

✉ Oliver Kautz
kautz@se-rwth.de

Bernhard Rumpe
rumpe@se-rwth.de

Andreas Wortmann
wortmann@se-rwth.de

[1] Software Engineering, RWTH Aachen University, Aachen, Germany

Published online: 16 April 2020

1 Introduction

Component-based software engineering (Naur et al. 1968) promises efficient software development through reuse of independently developed and validated components. Usually, these components are realized in general-purpose programming languages (GPLs) and are hence subject to the conceptual gap between the problem domains and the software development, which arises from addressing problem domain challenges through programming complexities (France and Rumpe 2007).

Model-driven development (MDD) (Völter et al. 2013) reduces this gap by lifting domain-specific, abstract models to primary development artifacts. These models are specified in terms of domain-specific vocabulary to be better comprehensible, more abstract, and, hence, better suited towards analysis and transformation than the programs of GPLs.

Architecture description languages (ADLs) (Medvidovic and Taylor 2000) leverage the potential of MDD for the description of software architectures. Research has produced over 120 ADLs (Malavolta et al. 2013 for different domains, such as automotive (Debruyne et al. 2005), avionics (Feiler and Gluch 2012), consumer electronics (Van Ommering et al. 2000), or robotics (Schlegel et al. 2011). In domains, where ADLs are popular, explicating the precise semantics of architecture models is crucial, e.g., due to safety concerns. Nonetheless, many ADLs provide translational semantics, i.e., ground the meaning of architectures through their transformation into better-understood formalisms (e.g., GPL code), only. And even where the ADL's semantics are explicitly available, the MDD related processes rarely exploit these to facilitate modeling.

Where the semantics of an ADL is made explicit, semantically sound system analyses and automated refactorings and refinements become possible. FOCUS (Broy and Stølen 2001; Broy 2010; Ringert and Rumpe 2011), for instance, is a framework and semantic foundation that captures logical component and connector software architectures as stream-processing functions. Stream processing functions describe the histories of messages communicated over communication channels established by connectors between the components' interfaces. Architecture modeling formalisms explicating component semantics, such as FOCUS, communicating sequential processes (CSPs) (Hoare 1978), or the π-calculus (Milner 1999) enable systematic *stepwise refinement* (Broy 2010), a software engineering methodology for continuous architecture modeling based on controlled evolution and progressive improvement of components: each subsequent version of a component model must adhere to properties already proven for its predecessors. Ideally, this process starts with several underspecified components which are iteratively refined according to their requirements. FOCUS is one of the rare frameworks, where refinement and decomposition are compatible, i.e., refining a single component of an architecture always refines the complete architecture. A *component refactoring* is a special refinement step where the resulting component's semantics is equal to the semantics of the original. With this, from

an external observer's viewpoint, the behaviors of the original and the resulting components are indistinguishable.

Manual refinement and refactoring without tool support, however, is tedious and error-prone. To facilitate this, we present a method for *automated refactoring* via parallel component decomposition based on the notion of *influence* between channels of components. This method uses time-synchronous port automata to represent the essence (i.e., reduced abstract syntax) of common ADLs, such as AADL (Feiler and Gluch 2012), AutoFocus (Hölzl and Feilkas 2010), EAST-ADL (Debruyne et al. 2005), MontiArc (Butting et al. 2017a), SysML's blocks (Friedenthal et al. 2011), and similar languages. Given a component implementation, we propose to automatically decompose it into subcomponents according to their influence relation. To this effect, we assume the availability of a model that describes the external interface of the component (e.g., an ADL model) and a description of the implementation of the component. It is irrelevant whether the description of the component implementation is available in source code that can be transformed to a time-synchronous port automaton or whether the implementation is directly described by a time-synchronous port automaton. To this end, our contributions are:

- A notion of *influence* between channels of a logical software architecture that is grounded in the Focus theory.
- A method to *automatically refactor* components with finite state spaces via parallel decompositions according to the influence relation.

The resulting architecture can be evolved more efficiently by different stakeholders, yet is guaranteed to be semantically equivalent to the previous architecture. Hence, all original properties still hold, despite being less complicated and better to evolve and maintain.

In the following, Sect. 2 sketches the idea of automated decomposition based on the influence relations between channels. Section 3 presents the system model that underlies the approach and has been introduced in previous work. Section 4 presents the notion of *influence* formalized in the Focus terminology and the process of decomposition based on it. Section 4.3 shows that the influence relation is decidable in the context of finite-state components. Afterwards, Sect. 5 presents its application on the example of the elevator control system presented and evaluated in Butting et al. (2017b). Section 6 discusses observations. Section 7 highlights related work, before Sect. 8 concludes.

2 Example

Modern software systems comprise hundreds or thousands of components. Starting development with the correct and final software architecture structure is phantasmal. Consequently, agile methods call for methods to iteratively evolve and complete software architectures. Stepwise refinement is such a method for agile software architecture modeling. With stepwise refinement, properties proven for

a specific version of a component hold for all its refined successors. Hence, even early versions of architectures can be used to prove properties relevant to the customers without the burden of proving these for each new version again as long as refinement is respected.

Consider, for example, developing the software architecture for a cyber-physical system in terms of its components through stepwise refinement, such as the elevator control system (ECS) presented in Strobl et al. (1999). At some point, the team developed an initial `ECS` architecture that consists of a single, monolithic component managing elevator requests, lights on the floors, cabin movement, as well as opening and closing the elevator cabin's door based on messages received from its environment. This component yields a single state-based behavior implementation realizing parts of the customers' requirements, i.e., is potentially shippable. Figure 1 illustrates the `ECS` component, which receives environmental messages through its input channels and outputs messages via its output channels. Engineering the (initial) software of such a system monolithically is valid with respect to stepwise refinement, but raises two challenges:

1. *Analysis challenge:* Proving architectural properties, for example, that the elevator control system eventually serves each floor for which the request button was pressed, already for initial architectures enables fixating properties relevant to customers early. However, model checking the complete `ECS` architecture might be challenging to impossible due to its complex implementation intertwining the different concerns unrelated to the property under consideration (here, e.g., management of floor lights).
2. *Implementation challenge:* Evolving functionalities implemented by such a monolith usually is overly complicated: in the worst case, parts of the implementation are scattered over different places and are hardly documented. This makes evolution error-prone and costly.

Addressing both challenges can be facilitated by properly decomposing the monolithic software architecture prior to analyzing or evolving it. For instance, decomposing the `ECS` architecture into subcomponents focusing on the *influence* between channels related to the property under consideration (such as `btn1` and `at1`) can facilitate model checking and implementation evolution. However, this

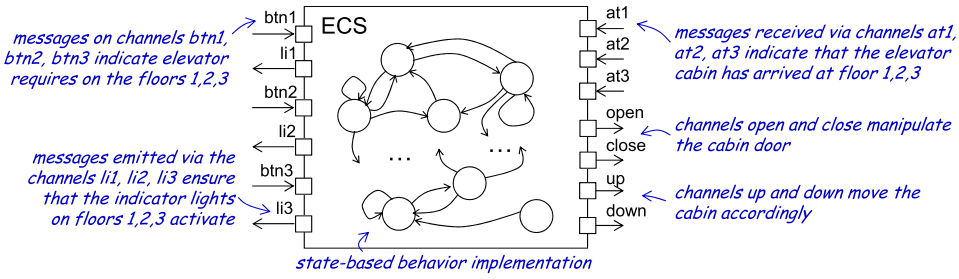


Fig. 1 Initial software architecture of the elevator control system

raises challenges in properly decomposing the architecture at hand, such that the resulting decomposition into interconnected subcomponents is actually a refactoring of the original.

An automated procedure for decomposing component and connector architectures that supports both challenges must ensure that resulting subcomponent configurations are a valid refactoring of the input architecture, and support selecting input channels and output channels that should be considered as bundles to capture the developers' knowledge about channel semantics and, hence, ultimately lead to useful subcomponents. The following sections present a procedure that supports both.

3 Preliminaries

This section presents a system model for time-synchronous systems. The system model has been introduced in previous work (Butting et al. 2017b). Architectures are networks of autonomous components that interact with each other via messages on typed channels. A time-synchronous (Broy and Stølen 2001; Broy 2010; Ringert and Rumpe 2011; Butting et al. 2017b; Grosu and Rumpe 1995) architecture is interpretable as a system where execution is divided into time-units. Time units are an abstract modeling concept. In implementations, components may be unaware of time, have local times, or even synchronize mimicking a global clock. In each time unit, each component receives finitely many input messages, performs finitely many internal computations, and then eventually outputs finitely many messages to its environment.

Notation We denote by $[X \to Y]$ the set of all functions from a set X to a set Y. For a function $f \in [X \to Y]$ and a set $Z \subseteq X$, we denote by $f|_Z \in [Z \to Y]$ the function that satisfies $f|_Z(x) = f(x)$ for all $x \in Z$, called the restriction of f to Z.

3.1 Streams

The history of messages received or emitted by a component is formally described by a stream (sequence/word) of messages. Let M be an arbitrary non-empty set. Similar to (Broy and Stølen 2001; Butting et al. 2017b), M^* denotes the set of all finite streams over M. M^∞ denotes the set of all infinite streams over M and $M^\omega = M^* \cup M^\infty$ denotes the set of all finite and infinite streams over M. We denote the empty stream by $\varepsilon \in M^*$. The concatenation of two streams $s, t \in M^\omega$ is denoted by $s \cdot t$. If $s \in M^\infty$, then $s \cdot t = s$ for all $t \in M^\omega$. The prefix relation $\sqsubseteq$ over streams is defined as usual: $s \sqsubseteq t \Leftrightarrow \exists u \in M^\omega : s \cdot u = t$. For $t \in \mathbb{N}$, the $(t+1)$-th element of a stream s is denoted by $s.t$. Similarly, $s \downarrow_t$ denotes the prefix of s of length t. For example,

- $p = 3, 1, 4 \in \mathbb{N}^*$ is a finite stream over the natural numbers where $p.0 = 3$, $p.1 = 1$ and $p.2 = 4$.
- The stream $s = 7, 8, 9, \cdots \in \mathbb{N}^\infty$ where $s.0 = 7$ and $s.(t+1) = 1 + s.t$ for all $t \in \mathbb{N}$ is an example for an infinite stream of natural numbers.

- It holds that $7, 8 \sqsubseteq s$, i.e., the stream 7, 8 is a prefix of the stream s.
- The concatenation $p \cdot s$ yields the infinite stream $p \cdot s = 3, 1, 4, 7, 8, 9, \cdots \in \mathbb{N}^{\infty}$.
- The prefix of length two of s is the stream $s \downarrow 2 = 7, 8$.

3.2 Messages, types

In the remainder, let M denote an arbitrary but fixed set of data elements (messages) that contains a designated element $\xi \in M$ modeling the mathematical concept of an empty pseudo-message. In a time-synchronous setting, where in each time unit at most one message is communicated via each channel, the empty message ξ can be used to model the progress of time, i.e., the message ξ is not explicitly communicated. It is important to emphasize that this communication model permits logical time while abstracting from real time. We model data types by sets of messages. Each message type contains the empty message. With this, it is possible to explicitly model the absence of a message on a communication channel in a time unit. Let *Type* denote a set of types where each type $t \in \mathit{Type}$ satisfies $t \subseteq M$ and $\xi \in t$. Types are used to restrict the set of messages that are allowed to be sent via a communication channel. As a concrete example, the type $\mathit{Nat} \in \mathit{Type}$ containing all natural numbers and the empty message ξ can be defined by $\mathit{Nat} = \{\xi\} \cup \mathbb{N}$.

3.3 Channels, histories

A channel is a communication link between components. Each channel has a unique name. In the remainder, let C denote a set of channels names. The function $\mathit{type} \in [C \to \mathit{Type}]$ maps each channel $c \in C$ to its type $\mathit{type}(c) \in \mathit{Type}$. A channel assignment is a function that maps channels to messages of the channels' types: A channel assignment for a set of channels $B \subseteq C$ is a function $a \in [B \to M]$ that satisfies $\forall b \in B : a(b) \in \mathit{type}(b)$. We denote by $B^{\rightarrow}$ the set of all channel assignments over B. A communication history for a set of channels $B \subseteq C$ is an infinite stream $h \in (B^{\rightarrow})^{\infty}$. The set of all communication histories for a set of channels $B \subseteq C$ is denoted by B^{Ω}. Thus, a communication history is a function that maps time units to channel assignments over their types. With this, each communication history models a full observation of the messages sent and received by a component. It should be noted that the set of communication histories $B^{\Omega} = (B^{\rightarrow})^{\infty}$ is isomorphic to the set $[B \to M^{\infty}]$ that satisfies $\forall b \in B : h(b) \in \mathit{type}(b)^{\infty}$, i.e., the set of all functions that map the channels in B to infinite streams of messages of their types. For a communication history $b \in B^{\Omega}$ and a time unit $t \in \mathbb{N}$, the prefix $b \downarrow_t$ thus models the communication history observed up to time t. We lift the $\downarrow$ operator to sets of communication histories in a point-wise manner: For $H \subseteq B^{\Omega}$, we define $H \downarrow_t = \bigcup_{h \in H} h \downarrow_t$. The restriction of a communication history $h \in B^{\Omega}$ to the channels in $R \subseteq B$ is defined as the communication history $h|_R$ that satisfies $(h|_R).t = (h.t)|_R$ for all $t \in \mathbb{N}$, i.e., each channel assignment in h is restricted to the channels in R.

As concrete examples,

- If $a, b \in C$ are channels typed with the natural numbers, then $type(a) = type(b) = Nat$.
- A channel assignment for the set of channels $\{a, b\}$ is given by $\alpha = \{a \mapsto 7, b \mapsto 8\} \in \{a, b\}^{\rightarrow}$. This assignment maps the channel a to the message 7 and the channel b to the message 8.
- The infinite stream $h = \alpha^{\infty} \in \{a, b\}^{\Omega}$ is a communication history for the set of channels $\{a, b\}$. In each time unit, this communication history maps the channel a to the message 7 and the channel b to the message 8.
- The prefix $h \downarrow 2 = \alpha, \alpha$ models the part of the communication history h observed in the first two time units.
- The restriction of the communication history h to the set of channels $\{a\}$ is given by $h|_{\{a\}} = \alpha|_{\{a\}}, \alpha|_{\{a\}}, \ldots = \{a \mapsto 7\}, \{a \mapsto 7\}, \cdots \in \{a\}^{\Omega}$.

3.4 Finite time-synchronous port automata

A finite time-synchronous port automaton (TSPA) specifies (an excerpt of) an interactive system architecture (Butting et al. 2017b; Grosu and Rumpe 1995). We assume a white-box view on components where each component implementation is represented by a finite TSPA. Complex system architectures are modeled via component composition, i.e., via the composition of the TSPAs representing the individual components' implementations.

A finite TSPA is a tuple $A = (I, O, S, \iota, \delta)$ where

- $I, O \subseteq C$ with $I \cap O = \emptyset$ are finite and disjoint sets of the TSPA's input and output channels,
- the type $type(c)$ of each channel $c \in I \cup O$ is finite,
- S is a finite set of states,
- $\iota \in S$ is the initial state, and
- $\delta \subseteq S \times (I \cup O)^{\rightarrow} \times S$ is the transition relation.

In the following, we simply refer to a finite TSPA by TSPA. We sometimes reference the syntactic elements of A as follows: $I_A = I$, $O_A = O$, $C_A = C(A) = I_A \cup O_A$, $S_A = S$, $\iota_A = \iota$, and $\delta_A = \delta$. A TSPA may fire a transition $(s, \theta, t) \in \delta$ if it receives $\theta|_I$ while residing in state s. When firing the transition, the automaton changes its internal state to t and outputs $\theta|_O$.

A TSPA A is called *reactive* iff $\forall s \in S_A : \forall i \in I_A^{\rightarrow} : \exists (u, a, v) \in \delta_A : u = s \wedge a|_I = i$. Reactive TSPAs are adequate models for interactive components as they define a possible reaction to every possible input and every possible state. If a TSPA is not reactive, then it may be in a state in which it receives an input for which no reaction in terms of a transition is defined. This behavior is erroneous as components are required to be able to react to every possible input in every time unit. A TSPA A is called *deterministic* iff $\forall s \in S_A : \forall i \in I_A^{\rightarrow} : |\{t \in S_A \mid \exists \theta \in C_A^{\rightarrow} : (s, \theta, t) \in \delta_A \wedge \theta|_{I_A} = i\}| = 1$, i.e., it defines exactly one transition for each possible input it may receive for each of its states. A nondeterministic TSPA resembles underspecification in

a component that can be resolved by subsequent refinement steps and/or left open to a nondeterministic implementation. An execution σ of A is an infinite, alternating sequence of states and channel assignments starting with the initial state ι: $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ such that $s_0 = \iota$ and $\forall i \in \mathbb{N} : (s_i, \theta_i, s_{i+1}) \in \delta$. We denote by $execs(A)$ the set of all executions of A. The behavior of an execution $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ of A is defined as the infinite sequence $beh(\sigma) = \theta_0, \theta_1, \ldots$ containing only the channel assignments in σ. An execution comprises a TSPA's internal behavior, which is invisible to its environment, whereas a behavior represents an execution from a black-box viewpoint. We denote by $behs(A)$ the set of all behaviors of A.

As concrete examples, Fig. 2 depicts two TSPAs. As usual, circles represent states and edges between states represent transitions. Initial states are marked with an arrow that originates from a back dot. The transitions are labeled with their channel valuations. The TSPA A has a single input channel i and a single output channel o. The TSPA A is not deterministic and thus highly underspecified. In fact, it models all possible behaviors over the channels $i, o \in C$ where $type(i) = type(o) = \{\xi, 1\}$. The other TSPA *Switch* can be interpreted to model a simple light control switch.

Initially, the TSPA is in state `off`, which models that the light is turned off. In case, the switch is not pressed, the TSPA does not receive a message via its input channel i, represented by the empty message ξ. If the switch is pressed, the TSPA receives the message 1 via its input channel i. If the TSPA is in state `off` and the switch is not pressed, the TSPA outputs the empty pseudo-message ξ via its output channel o and remains in state `off`. This represents that the light remains turned off. In case the TSPA is in state `off` and the switch is pressed, the TSPA outputs the message 1 via its output channel o and switches its state to `on`. This represents that the light is turned on. Vice versa, the TSPA remains in state `on` and the light remains turned on as long as the switch is not pressed. As soon as the switch is pressed while the TSPA is in state `on`, the TSPA switches to state `off` and the light is turned off. A possible execution of the TSPA *Switch* is the infinite alternating sequence of states and transitions $e = (off, \{i \mapsto 1, o \mapsto 1\}, on, \{i \mapsto 1, o \mapsto \xi\})^\infty$. In the execution e, the light is frequently turned on and off. The behavior $beh(e)$ of the execution e is given

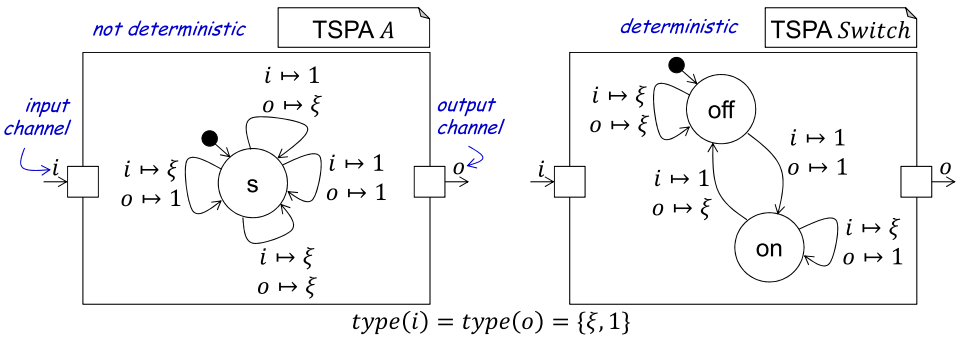


Fig. 2 An underspecified TSPA A and a deterministic TSPA *Switch*

by $beh(e) = (\{i \mapsto 1, o \mapsto 1\}, \{i \mapsto 1, o \mapsto \xi\})^\infty$, which is the sequence of channel assignments that represents the externally visible behavior of the execution.

3.5 TSPA composition

The composition of two TSPAs is a TSPA that captures the behavior of the architecture resulting from synchronously executing the TSPAs simultaneously where communication is carried out via the TSPAs' channels (Butting et al. 2017b; Grosu and Rumpe 1995). Multiple TSPAs may receive messages via the same channels, whereas at most one TSPA is permitted to send messages via a channel: Two TSPAs A, B are called compatible iff $O_A \cap O_B = \emptyset$.

The composition of two compatible TSPAs A and B is defined as $A \otimes B = (I, O, S_A \times S_B, (\iota_A, \iota_B), \delta)$ where

- $O = O_A \cup O_B$,
- $I = (I_A \cup I_B) \backslash O$, and
- $\delta = \{((s_1, s_2), \theta, (t_1, t_2)) \mid (s_1, \theta|_{C(A)}, t_1) \in \delta_A \wedge (s_2, \theta|_{C(B)}, t_2) \in \delta_B\}$.

Figure 3 illustrates the composition of two TSPAs. If the two TSPAs A and B represent the implementations of two components, then the composed TSPA $A \otimes B$ represents the implementation of the system obtained from running the components in parallel.

The composition of two compatible, reactive TSPAs does not always yield a reactive TSPA (Butting et al. 2017b; Grosu and Rumpe 1995). Thus, composing two components is not always meaningful as the composition of two components represented by two TSPAs may not be well-defined. This is because of causality problems (Broy and Stølen 2001; Broy 2010; Butting et al. 2017b; Grosu and Rumpe 1995) that can only exist if each of the TSPAs has an output channel that is an input channel of the respective other TSPA. The causality problem is guaranteed to be avoided if one of the TSPAs is strongly-causal (Butting et al. 2017b; Grosu and Rumpe 1995) with respect to its connected channels. However, if two reactive TSPAs are composed in parallel (without feedback), i.e., neither of

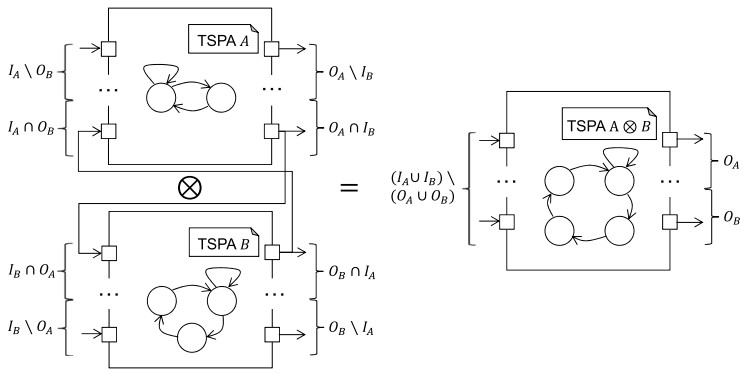


Fig. 3 General TSPA composition with feedback

the TSPAs has an output channel that is an input channel of the respective other TSPA, then the composition always yields a well-defined reactive TSPA (Butting et al. 2017b; Grosu and Rumpe 1995). As this paper is solely concerned with parallel decomposition and thus, vice versa, only with parallel composition, we refer to related work (Broy and Stølen 2001; Broy 2010; Butting et al. 2017b) for a discussion about causality complications.

3.6 TSPA restriction

Hiding is an important concept to achieve modularity (Broy and Stølen 2001; Broy 2010; Grosu and Rumpe 1995). Hiding an output channel facilitates concealing unimportant information to an environment. Similarly, it is possible to hide an input channel. Hiding an input channel does not affect the set of output histories. It relaxes the transition relation in the sense that messages on the hidden channel do not constrain the TSPA's behavior anymore. Thus, hiding an input channel effectively leads to more underspecification. Any transition is enabled independent of the messages received via the hidden channel, assumed that the messages received via the other input channels are part of the transition's channel valuation.

Let A be a TSPA and let $B \subseteq C_A$ be a set of channels. The restriction of A to the channels in B is defined as $A \restriction B = (I_A \cap B, O_A \cap B, S_A, \iota_A, \delta)$ where

$$\delta = \{(s, \theta, t) \mid \exists \kappa \in C_A^{\rightarrow} : \theta = \kappa|_B \wedge (s, \kappa, t) \in \delta_A\}.$$

As concrete examples, Fig. 4 depicts the TSPAs resulting from restricting the TSPA *Switch* (cf. Fig. 2) to the set of channels $\{i\}$ and from restricting the TSPA *Switch* to the set of channels $\{o\}$. Restricting the TSPA *Switch* to its input channel yields a TSPA that is still deterministic. However, restricting the TSPA *Switch* to its output channel yields an underspecified TSPA that is not deterministic.

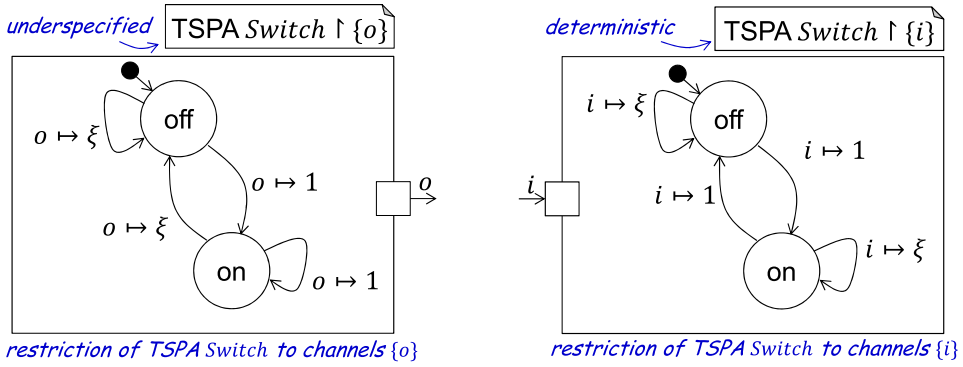


Fig. 4 The restriction of the TSPA *Switch* to the set of channels $\{o\}$ and the restriction of the TSPA *Switch* to the set of channels $\{i\}$

4 Semantics preserving parallel decomposition respecting influences between channels

This paper contributes to the parallel decomposition of deterministic TSPAs. Figure 5 overviews the key idea of the approach: The decomposition method takes a deterministic TSPA representing a component as input. Based on the influence relation between the TSPA's input and output channels, the method decomposes the component into multiple subcomponents (further TSPAs). The parallel composition of the resulting TSPAs yields a TSPA that has the same behaviors as the input TSPA. For example, Fig. 5 indicates that the output channel p is influenced by the input channels i and j. In contrast, the output channel o is solely influenced by the input channel i. The method can be fully automated. Therefore, we obtain an automatic method for refactoring monolithic components into multiple subcomponents such that the behaviors of the composition of the subcomponents are equal to the behaviors of the monolithic component.

The method may produce TSPAs that are not deterministic but *unambiguously specified* as intermediate decomposition results. Intuitively, a TSPA is unambiguously specified if it defines exactly one (infinite) output for every (infinite) input. Every unambiguously specified TSPA can be transformed to a deterministic TSPA having the same behaviors (cf. Sect. 4.1). Thus, the transformation enables the definition of a decomposition procedure for deterministic TSPAs that again yields an architecture of deterministic TSPAs.

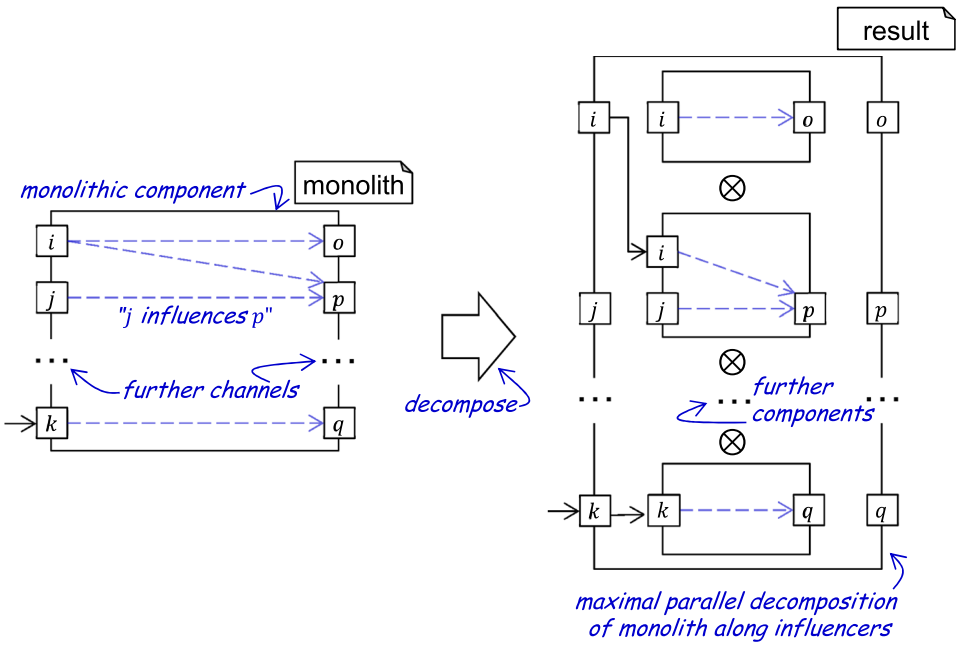


Fig. 5 Schematic representation of a monolithic component that is maximally decomposed along the influences between channels

Section 4.1 formally defines the notion unambiguously specified for TSPAs and presents properties of unambiguously specified TSPAs that are relevant to show the decomposition method's correctness. Afterwards, Sect. 4.2 defines the influence relation between channels of a TSPA. Then, Sect. 4.3 presents a decision procedure for determining whether an input channel of a TSPA influences an output channel of the same TSPA. Subsequently, Sect. 4.4 presents the fully automatic decomposition method based on the channel influence relation.

4.1 Unambiguously specified TSPAs

Hiding an input channel in a deterministic TSPA might result in a TSPA that is by definition not deterministic, but behaves as if it was deterministic from a black-box viewpoint. For example, this is because the TSPA's reachable part is deterministic and there exists a non-reachable part that is not deterministic. Figure 6 depicts a concrete example: The TSPA D is deterministic, whereas restricting it to its output channel yields a TSPA that exhibits the single behavior $\{o \mapsto \xi\}^\infty$, thus behaves deterministically from a black-box viewpoint. However, the restricted TSPA is internally non-deterministic, because of the non-reachable state b containing underspecification regarding the message sent via the output channel.

A TSPA might also be not deterministic and have multiple executions for the same inputs that produce the same outputs. In such a case, the TSPA is also not deterministic but behaves as if it was deterministic from a black-box viewpoint. Figure 7 depicts a concrete example: The TSPA U is not deterministic and exhibits the single behavior $\{i \mapsto \xi, o \mapsto \xi\}^\infty$. Therefore, it behaves deterministically from a black-box viewpoint. TSPAs that behave deterministically from a black-box viewpoint are *unambiguously specified*:

Definition 1 A TSPA A is unambiguously specified iff

$$\forall i \in I_A^\Omega : |\{\alpha \in behs(A) \mid \alpha|_I = i\}| = 1.$$

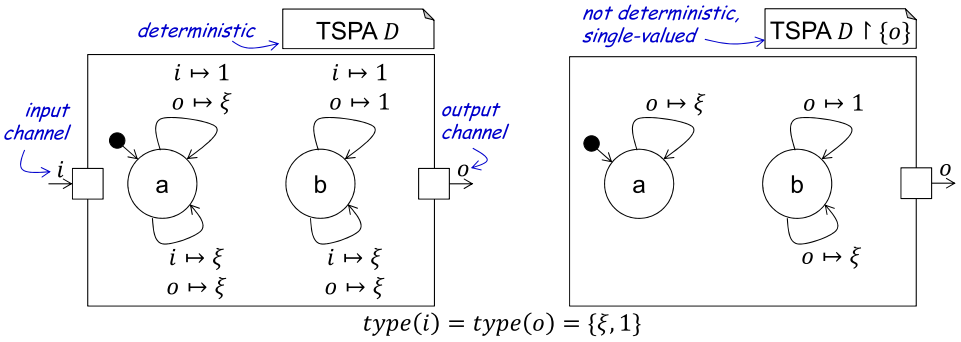


Fig. 6 Deterministic TSPA D and underspecified and unambiguously specified TSPA $D \restriction \{o\}$ resulting from hiding the input channel i in D

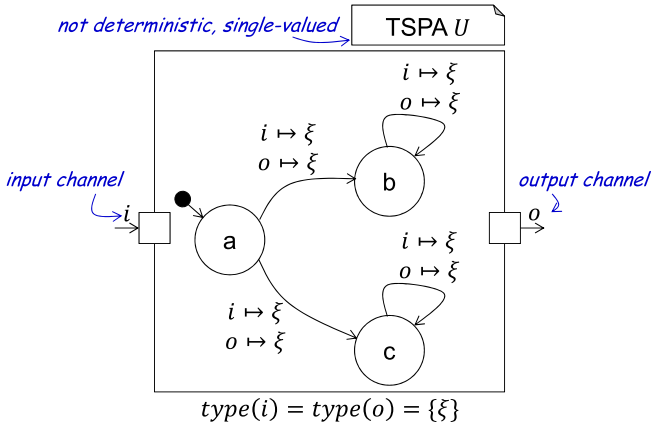


Fig. 7 Underspecified TSPA that behaves deterministically from a black-box viewpoint

The notion *unambiguously specified* for TSPAs and infinite behaviors is related to the notion *single-valued* for finite transductions of transducers (Weber and Klemm 1995; Weber 1998; Béal and Carton 2002). According to Weber and Klemm (1995), Weber (1998), and Béal and Carton (2002), a transducer is single-valued if it maps each input sequence to at most one output sequence. In contrast, we require that each input is mapped to exactly one output. Further, in each computation step, a transducer may map a single input symbol to a sequence of output symbols, whereas a TSPA maps one input channel valuation to exactly one output channel valuation.

Our approach aims at decomposing deterministic TSPAs. It is easy to see that every deterministic TSPA is also unambiguously specified but that the opposite does not necessarily hold. However, for every unambiguously specified TSPA, it is possible to construct an equivalent deterministic TSPA, i.e., the unambiguously specified and the deterministic TSPAs have the same behaviors.

Theorem 1 *For every unambiguously specified TSPA A, there exists a deterministic TSPA D with behs*$(A) = behs(D)$.

Proof (Sketch) A TSPA is interpretable as a special transducer over infinite words where all states are final. Sufficient and necessary conditions enabling the determinization of transducers over infinite words where all states are final are studied in Béal and Carton (2000, 2002).

Specifically, a TSPA is interpretable as a transducer over infinite words where

- Each transition transduces exactly one input symbol to exactly one output symbol,
- There is exactly one initial state,
- All states are final, and
- The transducer has no cyclic path with an empty output.

It has been shown that a transducer over infinite words where all states are final, the transducer has no constant states, and the transducer has no cyclic path with an empty output can be determinized, if the transducer obtained after removing all *constant states* (Béal and Carton 2000, 2002) satisfies the *twinning property* (Béal and Carton 2000, 2002). When transferring these notions to TSPAs, the TSPA A obtained from removing the constant states from an unambiguously specified TSPA is a TSPA that satisfies $\forall i \in I_A^\Omega : |\{\alpha \in behs(A) \mid \alpha|_{I_A} = i\}| \leq 1$. If this TSPA did not have the twinning property, then there would exist an input $i \in I_A^\Omega$ such that $|\{\alpha \in behs(A) \mid \alpha|_{I_A} = i\}| \geq 2$. Furthermore, Béal and Carton (2000, 2002) present a construction that can be used for transforming an unambiguously specified TSPA to an equivalent deterministic TSPA. The construction is a subset construction on the TSPA obtained from removing all unreachable states. □

Thus, every unambiguously specified TSPA can be transformed to a TSPA in which the output in any time unit only depends on the current input and state.

The following introduces general properties of unambiguously specified TSPAs that are later used for proving the correctness of the decomposition method. Two unambiguously specified TSPAs are equivalent if, and only if, one of the automata is a refinement of the other automaton:

Theorem 2 *Let A and B be unambiguously specified TSPAs with $I_A = I_B$ and $O_A = O_B$. Then, $behs(A) \subseteq behs(B)$ if, and only if, $behs(A) = behs(B)$.*

Proof Let A and B be given as above.

"⇒": Assume $behs(A) \subseteq behs(B)$. Let $I = I_A$ and $O = O_A$. Suppose towards a contradiction $behs(B) \nsubseteq behs(A)$. Then, there exists a behavior $b \in behs(B)$ such that $b \notin behs(A)$. As A is unambiguously specified, $I_A = I_B$ and $O_A = O_B$, there exists a behavior $b' \in behs(A)$ with $b'|_I = b|_I$. As $b \notin behs(A)$ and $b'|_I = b|_I$, we have that $b'|_O \neq b|_O$. As $behs(A) \subseteq behs(B)$, it holds that $b' \in behs(B)$. This contradicts that B is unambiguously specified, because $b, b' \in behs(B)$, $b|_I = b'|_I$ and $b|_O \neq b'|_O$ implies for $i = b|_I$ that $|\{\alpha \in behs(B) \mid \alpha|_I = i\}| \geq 2$.

"⇐": $behs(A) = behs(B)$ implies $behs(A) \subseteq behs(B)$. □

TSPAs do not influence the behaviors of each other when executed in parallel, i.e., when neither of the TSPAs has an output channel that is an input channel of the respective other TSPA. Thus, the parallel composition of two unambiguously specified TSPAs is again an unambiguously specified TSPA:

Theorem 3 *Let A and B be two compatible unambiguously specified TSPAs such that $O_A \cap I_B = O_B \cap I_A = \emptyset$. Then, $A \otimes B$ is an unambiguously specified TSPA.*

Proof Let A and B be given as above and let $K = A \otimes B$. We need to show that $|\{\alpha \in behs(K) \mid \alpha|_{I_K} = i\}| = 1$ for all $i \in I_K^\Omega$.

(1) We first show that $|\{\alpha \in behs(K) \mid \alpha|_{I_K} = i\}| > 0$ for all $i \in I_K^\Omega$: Let $i \in I_K^\Omega$. As A and B are unambiguously specified, there exist behaviors $b \in behs(A)$ and

$b' \in behs(B)$ such that $b|_{I_A} = i|_{I_A}$ and $b'|_{I_B} = i|_{I_B}$. Let $\sigma = s_0, \theta_0, s_1, \theta_1 \ldots$ be an execution of A such that $beh(\sigma) = b$ and let $\tau = s'_0, \kappa_0, s'_1, \kappa_1 \ldots$ be an execution of B such that $beh(\tau) = b'$. As σ and τ are executions, we have that $s_0 = \iota_A$ and $s'_0 = \iota_B$ and $(s_t, \theta_t, s_{t+1}) \in \delta_A$ and $(s'_t, \kappa_t, s'_{t+1}) \in \delta_B$ for all $t \in \mathbb{N}$. As $b|_{I_A} = i|_{I_A}$ and $b'|_{I_B} = i|_{I_B}$, we have that $i|_{I_A \cap I_B} = b|_{I_A \cap I_B} = b'|_{I_A \cap I_B}$. Hence, for all $t \in \mathbb{N}$, we have that $\theta_t|_{I_A \cap I_B} = \kappa_t|_{I_A \cap I_B}$. For all $t \in \mathbb{N}$, we define $\mu_t \in C_K^{\rightarrow}$ as follows: $\mu_t(c) = \theta_t(c)$, if $c \in C_A$, and $\mu_t(c) = \kappa_t(c)$, if $c \in C_B \backslash C_A$. Then, by definition $\mu_t|_{C_A} = \theta_t$. Further, $\mu_t|_{C_B} = \kappa_t$ because $\theta_t|_{I_A \cap I_B} = \kappa_t|_{I_A \cap I_B}$ and by definition $\mu_t|_{C_B \backslash C_A} = \kappa_t|_{C_B \backslash C_A}$. Thus, by definition of TSPA composition, we have that $((s_t, s'_t), \mu_t, (s_{t+1}, s'_{t+1})) \in \delta_K$ for all $t \in \mathbb{N}$. This implies with $s_0 = \iota_A$ and $s'_0 = \iota_B$ that $e = (s_0, s'_0), \mu_0, (s_1, s'_1), \mu_1 \ldots$ is an execution of K with $beh(e)|_{I_K} = i$.

(2) We now show that $|\{\alpha \in behs(K) \mid \alpha|_{I_K} = i\}| < 2$ for all $i \in I_K^{\Omega}$:

Suppose towards a contradiction there exist $i \in I_K^{\Omega}$ and $\alpha, \beta \in behs(K)$ such that $\alpha|_{I_K} = \beta|_{I_K} = i$ and $\alpha \neq \beta$. Thus, $\alpha|_{O_K} \neq \beta|_{O_K}$. As α, β are behaviors of K, there exist executions σ and τ of K such that $\alpha = beh(\sigma)$ and $\beta = beh(\tau)$. Let $\sigma = (s_0^A, s_0^B), \alpha_0, (s_1^A, s_1^B), \alpha_1 \ldots$ be an execution of K such that $beh(\sigma) = \alpha$. Further, let $\tau = (s_0'^A, s_0'^B), \beta_0, (s_1'^A, s_1'^B), \beta_1 \ldots$ be an execution of K such that $beh(\tau) = \beta$. Using the definitions of execution and composition, we obtain that $\sigma_A = s_0^A, \alpha_0|_{C_A}, s_1^A, \alpha_1|_{C_A} \ldots$ and $\tau_A = s_0'^A, \beta_0|_{C_A}, s_1'^A, \beta_1|_{C_A} \ldots$ are execution of A. Similarly, we have that $\sigma_B = s_0^B, \alpha_0|_{C_B}, s_1^B, \alpha_1|_{C_B} \ldots$ and $\tau_B = s_0'^B, \beta_0|_{C_B}, s_1'^B, \beta_1|_{C_B} \ldots$ are executions of B. As $O_K = O_A \cup O_B$ and $\alpha|_{O_K} \neq \beta|_{O_K}$, it holds that $beh(\sigma_A) \neq beh(\tau_A)$ or $beh(\sigma_B) \neq beh(\tau_B)$. Without loss of generality, assume $beh(\sigma_A) \neq beh(\tau_A)$. Then, $\alpha|_{I_K} = \beta|_{I_K}$ implies $beh(\sigma_A)|_{I_A} = beh(\tau_A)|_{I_A}$ since $I_A \cap O_B = \emptyset$ and thus $I_A \subseteq I_K$ by definition of composition. This contradicts that A is unambiguously specified because $beh(\sigma_A), beh(\tau_A) \in behs(A)$ and $beh(\sigma_A)|_{I_A} = beh(\tau_A)|_{I_A}$ and $beh(\sigma_A) \neq beh(\tau_A)$. □

The TSPA obtained from hiding an unambiguously specified TSPA's output channel is again an unambiguously specified TSPA. Hiding an input channel does usually not preserve the unambiguously specified property.

Theorem 4 *Let A be an unambiguously specified TSPA and let $o \in O_A$ be an output channel of A. Then, $A \upharpoonright (C_A \backslash \{o\})$ is unambiguously specified.*

Proof Let A and o be given as above. Let $B = A \upharpoonright (C_A \backslash \{o\})$. Suppose B is not unambiguously specified. Then there exist executions $\sigma = s_0, \theta_0, s_1, \theta_1 \ldots$ and $\tau = s'_0, \kappa_0, s'_1, \kappa_1 \ldots$ of B such that $beh(\sigma)|_{I_B} = beh(\tau)|_{I_B}$ and $beh(\sigma) \neq beh(\tau)$. By definition of TSPA restriction, this implies there exist executions $\sigma' = s_0, \theta'_0, s_1, \theta'_1 \ldots$ and $\tau' = s'_0, \kappa'_0, s'_1, \kappa'_1 \ldots$ of A such that $\theta_i = \theta'_i|_B$ and $\kappa_i = \kappa'_i|_B$ for all $i \in \mathbb{N}$. This contradicts that A is unambiguously specified because $beh(\sigma')|_{I_A} = beh(\sigma)|_{I_B} = beh(\tau)|_{I_B} = beh(\tau')|_{I_A}$ and $beh(\sigma') \neq beh(\tau')$ since $beh(\sigma')|_{C_B} = beh(\sigma) \neq beh(\tau) = beh(\tau')|_{C_B}$. □

4.2 An influence relation between channels of components

A component's input channel influences an output channel if the messages sent via the latter depend on the messages received via the former.

Definition 2 (*Channel Influence Relation*) Let $A = (I, O, S, \iota, \delta)$ be an unambiguously specified TSPA, let $i \in I$ be an input channel of A, and let $o \in O$ be an output channel of A. The channel i influences the channel o in A (denoted $i \rightsquigarrow_A o$) iff

$$\exists \alpha, \beta \in behs(A) : \alpha|_{I\setminus\{i\}} = \beta|_{I\setminus\{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}.$$

The above definition requires that there exist two behaviors α, β with the same messages on all input channels except i such that the behaviors are different on the output channel o. As the inputs are equal on all channels except i, the values received on i are responsible for the differences regarding the possible outputs on o.

The other way around, the channel i does not influence the channel o in A iff for any two possible inputs that are equal on all channels except i, the automaton A always produces the same outputs on o when processing the inputs. More formally, negating the definition we obtain: a channel i does not influence a channel o in A (denoted $i \not\rightsquigarrow_A o$) iff $\forall \alpha, \beta \in behs(A) : \alpha|_{I\setminus\{i\}} = \beta|_{I\setminus\{i\}} \Rightarrow \alpha|_{\{o\}} = \beta|_{\{o\}}$. Hiding an input channel does not always preserve the unambiguously specified property (cf. Sect. 4.1). However, if an input channel i does not influence an output channel o in an unambiguously specified TSPA A, then hiding the input channel i and all output channels except o results again in an unambiguously specified TSPA:

Theorem 5 *Let A be an unambiguously specified TSPA, let $i \in I_A$ be an input channel of A, and let $o \in O_A$ be an output channel of A. If $i \not\rightsquigarrow_A o$, then $A \upharpoonright (\{o\} \cup I\setminus\{i\})$ is unambiguously specified.*

Proof Let A, i, and o be given as above. Let $B = A \upharpoonright (\{o\} \cup I\setminus\{i\})$. We need to show that $|\{\alpha \in behs(B) \mid \alpha|_{I_B} = h\}| = 1$ for all $h \in I_B^\Omega$.

(1) We first show that $|\{\alpha \in behs(B) \mid \alpha|_{I_B} = h\}| > 0$ for all $h \in I_B^\Omega$: Let $h \in I_B^\Omega$. As A is unambiguously specified and $I_B \subseteq I_A$, there exists a behavior $b \in behs(A)$ such that $b|_{I_B} = h$. Let $\sigma = s_0, \theta_0, s_1, \theta_1 \ldots$ be an execution of A such that $beh(\sigma) = b$. Then, by definition of execution $s_0 = \iota_A$ and $(s_j, \theta_j, s_{j+1}) \in \delta_A$ for all $j \in \mathbb{N}$. By definition of restriction, we have that $s_0 = \iota_A = \iota_B$ and $(s_j, \theta_j|_{C_B}, s_{j+1}) \in \delta_B$ for all $j \in \mathbb{N}$. Hence, $\tau = s_0, \theta_0|_{C_B}, s_1, \theta_1|_{C_B} \ldots$ is an execution of B with $\tau|_{I_B} = h$.

(2) We now show that $|\{\alpha \in behs(B) \mid \alpha|_{I_B} = h\}| < 2$ for all $h \in I_B^\Omega$: Suppose towards a contradiction there exist $h \in I^\Omega$ and $\alpha, \beta \in behs(B)$ such that $\alpha|_{I_B} = \beta|_{I_B}$ and $\alpha \neq \beta$. Thus, $\alpha|_{O_B} \neq \beta|_{O_B}$. Let $\sigma = s_0, \alpha_0, s_1, \alpha_1 \ldots$ and $\tau = s'_0, \beta_0, s'_1, \beta_1 \ldots$ be executions of B such that $beh(\sigma) = \alpha$ and $beh(\tau) = \beta$. As σ and τ are executions of B, we have $s_0 = s'_0 = \iota_B$ and $(s_j, \alpha_j, s_{j+1}), (s'_j, \beta_j, s'_{j+1}) \in \delta_B$ for all $j \in \mathbb{N}$. By definition of TSPA restriction, this implies $s_0 = s'_0 = \iota_B = \iota_A$ and for all $j \in \mathbb{N}$, there exist $\theta_j, \kappa_j \in C_A^{\rightarrow}$ such that $(s_j, \theta_j, s_{j+1}) \in \delta_A$ and $\theta_j|_{C_B} = \alpha_j$ and $(s'_j, \kappa_j, s'_{j+1}) \in \delta_A$ and $\kappa_j|_{C_B} = \beta_j$. Hence, $\sigma' = s_0, \theta_0, s_1, \theta_1 \ldots$ and $\tau' = s'_0, \kappa_0, s'_1, \kappa_1 \ldots$ are executions of A. This contradicts that $i \not\rightsquigarrow_A o$ because

$beh(\sigma')|_{I\setminus\{i\}} = beh(\sigma')|_{I_B} = beh(\sigma)|_{I_B} = \alpha|_{I_B} = \beta|_{I_B} = beh(\tau')|_{I\setminus\{i\}}$ and
$beh(\sigma')|_{\{o\}} = \alpha|_{\{o\}} \neq \beta|_{\{o\}} = beh(\tau')|_{\{o\}}$. □

If there exists a pair of an input and an output channel of an unambiguously specified component such that the input channel does not influence the output channel, it is possible to split the component into a semantically equivalent architecture of two components. This architecture models a new component that is functionally better separated as the original component. This does not only improve the architecture's design but also increases understandability of the architecture and enables independent functional testing. Further, dividing the component also facilitates compositional architecture verification: A property might be independent of the behaviors of one of a composed component's subcomponents. Thus, verifying the property is possible without considering the subcomponent not influencing the property's satisfaction.

Section 4.3 shows that the channel influence relation of every finite unambiguously specified TSPA is decidable. Subsequently, Sect. 4.4 introduces the automated decomposition procedure based on the channel influence relation.

4.3 Deciding influence in unambiguously specified TSPAs

This section shows that it is decidable whether one channel influences another channel in an unambiguously specified finite TSPA. The decision relies on the construction of finite Büchi automata (BA) accepting infinite words (Büchi 1962; Farwer 2002; Safra 1988). BAs are well-known and studied in the automata theory domain. The next section fixes our notation for BAs and recaps decidability properties of BAs used in this paper before Sect. 4.3.1 presents the decision procedure.

A Büchi automaton (BA) is a tuple $(\Sigma, Q, I, F, \delta)$ where

- Σ is a finite alphabet,
- Q is a finite set of states,
- $I \subseteq Q$ is a set of initial states,
- $F \subseteq Q$ is a set of accepting states, and
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.

Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be a BA. A run of $\mathcal{A}$ on a word $w = \sigma_1, \sigma_2 \cdots \in \Sigma^\infty$ starting in a state $q_0 \in Q$ is an infinite sequence $q_0, q_1 \ldots$ such that $(q_{j-1}, \sigma_j, q_j) \in \delta$ for all $j \in \mathbb{N}$ with $j > 0$. The run $q_0, q_1 \ldots$ is accepting if $q_0 \in I$ and $q_i \in F$ for infinitely many $i \in \mathbb{N}$. The accepted language of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Pi^\infty \mid \text{there exists an accepting run of } \mathcal{A} \text{ on } w\}$. The emptiness problem, asking whether $\mathcal{L}(\mathcal{A}) = \emptyset$ for a BA $\mathcal{A}$ is decidable (Büchi 1962; Farwer 2002). The language of BAs is further closed under intersection (Büchi 1962): For all BAs $\mathcal{A}$ and $\mathcal{B}$, there exist an algorithm for constructing a BA $\mathcal{C}$ such that $\mathcal{L}(\mathcal{C}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. The languages accepted by BAs are closed under complement: For every BA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, there is an algorithm for computing a BA $\mathcal{B}$ such that $L(\mathcal{B}) = \Sigma^\infty \setminus \mathcal{L}(\mathcal{A})$ (Safra 1988). We denote the BA accepting the complement of the language accepted by a BA $\mathcal{A}$ with $\overline{\mathcal{A}}$.

4.3.1 Deciding influence

In the remainder of this section, let A be a finite unambiguously specified TSPA, let $i \in I_A$ be an input channel of A and let $o \in O_A$ be an output channel of A. The procedure for checking whether i influences o in A relies on constructing three Büchi automata $\mathcal{A}$, $\mathcal{I}$, and $\mathcal{O}$.

- The automaton $\mathcal{A}$ encodes all tuples of behaviors of A.
- The automaton $\mathcal{I}$ models the set of all tuples of behaviors in C_A^Ω that are equal on all input channels in $I_A \backslash \{i\}$.
- The automaton $\mathcal{O}$ encodes the set of all tuples of behaviors in C_A^Ω that are equal on the output channel o.

Thus, the automaton accepting $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$ accepts all tuples of behaviors of A that are equal on the input channels in $I_A \backslash \{i\}$ and not equal on the output channel o. We show that $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) = \emptyset$ if, and only if, i does not influence o in A.

The Büchi automaton $\mathcal{A}$ that encodes all tuples of behaviors of A is constructed as follows:

$\mathcal{A} = (C_A^{\rightarrow} \times C_A^{\rightarrow}, S_A \times S_A, \{(\iota_A, \iota_A)\}, S_A \times S_A, \delta)$, where

$\delta = \{((s,u),(a,b),(t,v)) \mid (s,a,t),(u,b,v) \in \delta_A\}$

As A is finite, $C_A^{\rightarrow}$ and S_A are finite. Hence, $C_A^{\rightarrow} \times C_A^{\rightarrow}$ and $S_A \times S_A$ are finite. This implies that δ is finite. Therefore, $\mathcal{A}$ is a well-defined BA.

Theorem 6 *For all $\alpha, \beta \in C_A^\Omega$, it holds that*

$$\alpha, \beta \in behs(A) \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}).$$

Proof Let $\alpha, \beta \in C_A^\Omega$.

"⇒": Assume it holds that $\alpha, \beta \in behs(A)$. Then, there exist two executions $\sigma = s_0, \theta_0, s_1, \theta_1 \cdots \in execs(A)$ and $\tau = s'_0, \kappa_0, s'_1, \kappa_1 \cdots \in execs(A)$ such that $beh(\sigma) = \alpha$ and $beh(\tau) = \beta$. By definition of execution we have that $s_0 = s'_0 = \iota_A$ and $(s_t, \theta_t, s_{t+1}), (s'_t, \kappa_t, s'_{t+1}) \in \delta_A$ for all $t \in \mathbb{N}$. By definition of the transition relation δ of the BA $\mathcal{A}$, this implies $((s_t, s'_t), (\theta_t, \kappa_t), (s_{t+1}, s'_{t+1})) \in \delta$ for all $t \in \mathbb{N}$. Hence, $(s_0, s'_0), (s_1, s'_1) \ldots$ is a run of $\mathcal{A}$ on the word $(\theta_0, \kappa_0), (\theta_1, \kappa_1) \ldots$. As all states in $\mathcal{A}$ are accepting, all states on the run are accepting. As further $(s_0, s'_0) = (\iota_A, \iota_A)$, we have that the run is accepting. Thus, it holds that $(\theta_0, \kappa_0), (\theta_1, \kappa_1) \ldots$ is a word accepted by $\mathcal{A}$. Observing that $\theta_t = \alpha.t$ and $\kappa_t = \beta.t$ for all $t \in \mathbb{N}$, we can conclude $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A})$.

"⇐": Assume $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A})$. This implies there exists an accepting run $\sigma = (s_0, s'_0), (s_1, s'_1) \ldots$ on the word $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \ldots$ in $\mathcal{A}$. Thus, we have $(s_0, s'_0) = (\iota_A, \iota_A)$ and $((s_t, s'_t), (\alpha.t, \beta.t), (s_{t+1}, s'_{t+1})) \in \delta$ for all $t \in \mathbb{N}$ where δ is the transition relation of $\mathcal{A}$. Using the definition of the transition relation δ of $\mathcal{A}$, the above implies $(s_t, \alpha.t, s_{t+1}), (s'_t, \beta.t, s'_{t+1}) \in \delta_A$ for all $t \in \mathbb{N}$. Hence, by definition of execution $\sigma = s_0, \alpha.0, s_1, \alpha.1 \cdots \in execs(A)$ and $\tau = s'_0, \beta.0, s'_1, \beta.1 \cdots \in execs(A)$.

From observing that $beh(\sigma) = \alpha$ and $beh(\tau) = \beta$, we can conclude that $\alpha, \beta \in behs(A)$. □

The constructions of the BAs $\mathcal{I}$ and $\mathcal{O}$ are analogous to each other. We thus first present a more general construction before defining $\mathcal{I}$ and $\mathcal{O}$. Let $B \subseteq C_A$ be a set of channels of A. The BA $\mathcal{E}(B)$ encoding all pairs of behaviors in C_A^Ω that are equal on the channels in B is constructed as follows:

$$\mathcal{E}(B) = (C_A^{\rightarrow} \times C_A^{\rightarrow}, \{\top\}, \{\top\}, \{\top\}, \delta) \text{ where } \delta = \{(\top, (a_1, a_2), \top) \mid a_1|_B = a_2|_B\}.$$

As A is finite, $C_A^{\rightarrow}$ is finite. Thus, $C_A^{\rightarrow} \times C_A^{\rightarrow}$ is finite. Further, $\mathcal{E}(B)$ has exactly one state. Hence, δ is finite and $\mathcal{E}(B)$ is well-defined.

Theorem 7 *Let $B \subseteq C_A$. For all behaviors $\alpha, \beta \in C_A^\Omega$ it holds that $\alpha|_B = \beta|_B \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{E}(B))$.*

Proof Let $B \subseteq C_A$ and let $\alpha, \beta \in C_A^\Omega$.

"⇒": Assume $\alpha|_B = \beta|_B$. This implies $\alpha.t|_B = \beta.t|_B$ for all $t \in \mathbb{N}$. Thus, by definition of the transition relation of $\mathcal{E}(B)$, we have that $(\top, (\alpha.t, \beta.t), \top) \in \delta$ for all $t \in \mathbb{N}$ where δ is the transition relation of $\mathcal{E}(B)$. Using the definition of accepting run, we have that $\top, \top, \top \ldots$ is an accepting run on the word $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \ldots$ in $\mathcal{E}(B)$. Thus, $(\alpha.0, \beta.0), (\alpha.1, \beta.1), (\alpha.2, \beta.2) \cdots \in \mathcal{L}(\mathcal{E}(B))$.

"⇐": Assume $\gamma = (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{E}(B))$. Then, there exists an accepting run σ of $\mathcal{E}(B)$ on the word γ. As $\top$ is the only state of $\mathcal{E}(B)$, we have that $\sigma.t = \top$ for all $t \in \mathbb{N}$. As σ is a run of $\mathcal{E}(B)$, we have $(\sigma.t, (\alpha.t, \beta.t), \sigma.(t+1)) = (\top, (\alpha.t, \beta.t), \top) \in \delta$ for all $t \in \mathbb{N}$ where δ is the transition relation of $\mathcal{E}(B)$. By definition of the transition relation, this implies $\alpha.t|_B = \beta.t|_B$ for all $t \in \mathbb{N}$. This is equivalent to $\alpha|_B = \beta|_B$. □

The Büchi automata $\mathcal{I}$ and $\mathcal{O}$ are defined as $\mathcal{I} = \mathcal{E}(I_A \backslash \{i\})$ and $\mathcal{O} = \mathcal{E}(\{o\})$.

Theorem 8 *It holds $i \rightsquigarrow_A o$ iff $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) \neq \emptyset$.*

Proof Using Theorems 6 and 7, we have for all behaviors $\alpha, \beta \in C_A^\Omega$:

$\alpha, \beta \in behs(A) \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A})$ and
$\alpha|_{I_A \backslash \{i\}} = \beta|_{I_A \backslash \{i\}} \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{I})$ and
$\alpha|_{\{o\}} \neq \beta|_{\{o\}} \Leftrightarrow (\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\overline{\mathcal{O}})$.

Combining the three equivalences, we obtain for all behaviors $\alpha, \beta \in C_A^\Omega$:

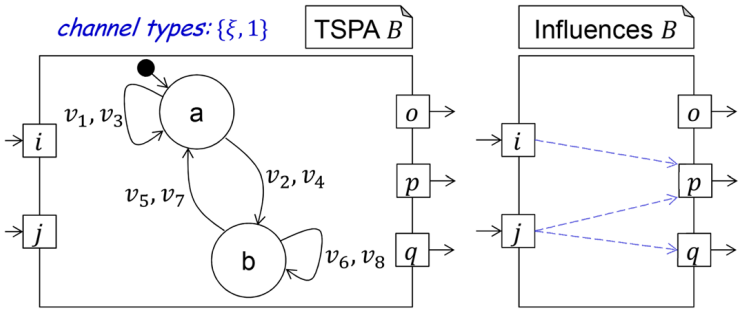
$(\alpha, \beta \in behs(A) \wedge \alpha|_{I_A \backslash \{i\}} = \beta|_{I_A \backslash \{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}) \Leftrightarrow$
$(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$.

"⇒": Assume $i \rightsquigarrow_A o$. Then, there exist behaviors $\alpha, \beta \in behs(A) : \alpha|_{I\setminus\{i\}} = \beta|_{I\setminus\{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}$. Using the above, this implies $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$. Thus, $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) \neq \emptyset$.

"⇐": Assume $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}}) \neq \emptyset$. This implies that there exists a word $(\alpha.0, \beta.0), (\alpha.1, \beta.1) \cdots \in \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{I}) \cap \mathcal{L}(\overline{\mathcal{O}})$. Let α, β be two behaviors defined by: $\alpha = \alpha.0, \alpha.1 \cdots \in C_A^\Omega$ and $\beta = \beta.0, \beta.1 \cdots \in C_A^\Omega$. Using the above, we obtain $\alpha, \beta \in behs(A) \wedge \alpha|_{I_A\setminus\{i\}} = \beta|_{I_A\setminus\{i\}} \wedge \alpha|_{\{o\}} \neq \beta|_{\{o\}}$. This implies $i \rightsquigarrow_A o$. □

For example, Fig. 8 depicts the TSPA B. The TSPA has the two input channels i and j and the three output channels o, p, and q. Each channel has the type $\{\xi, 1\}$. The graphical representation of the TSPA uses eight transition labels that are defined in the table, which is depicted at the bottom of Fig. 8. The top-right part of Fig. 8 sketches the influence relation between the input and output channels of the TSPA B. For instance, the channel i influences the channel p, but the channel i does not influence the channel q. From the graphical representation, the channel influence relation in the TSPA B is not obvious. Using the procedure presented in this section, the channel influence relation can be computed fully automatically.

The following example demonstrates the construction to show that the input channel i influences the output channel p in the TSPA B. In the following, we construct the three BAs A, I, and $\overline{O}$ for determining whether the input channel i influences the output channel p. From these BAs, we construct the BA $A \times I \times \overline{O}$ that recognizes



Label	i	j	o	p	q
v_1	ξ	ξ	1	ξ	ξ
v_2	ξ	1	1	1	ξ
v_3	1	ξ	1	1	ξ
v_4	1	1	1	ξ	ξ
v_5	ξ	ξ	1	1	1
v_6	ξ	1	1	ξ	1
v_7	1	ξ	1	ξ	1
v_8	1	1	1	1	1

Fig. 8 TSPA where one output channel is not influenced by any input channel, one output channel is influenced by one input channel, and one output channel is influenced by two input channels

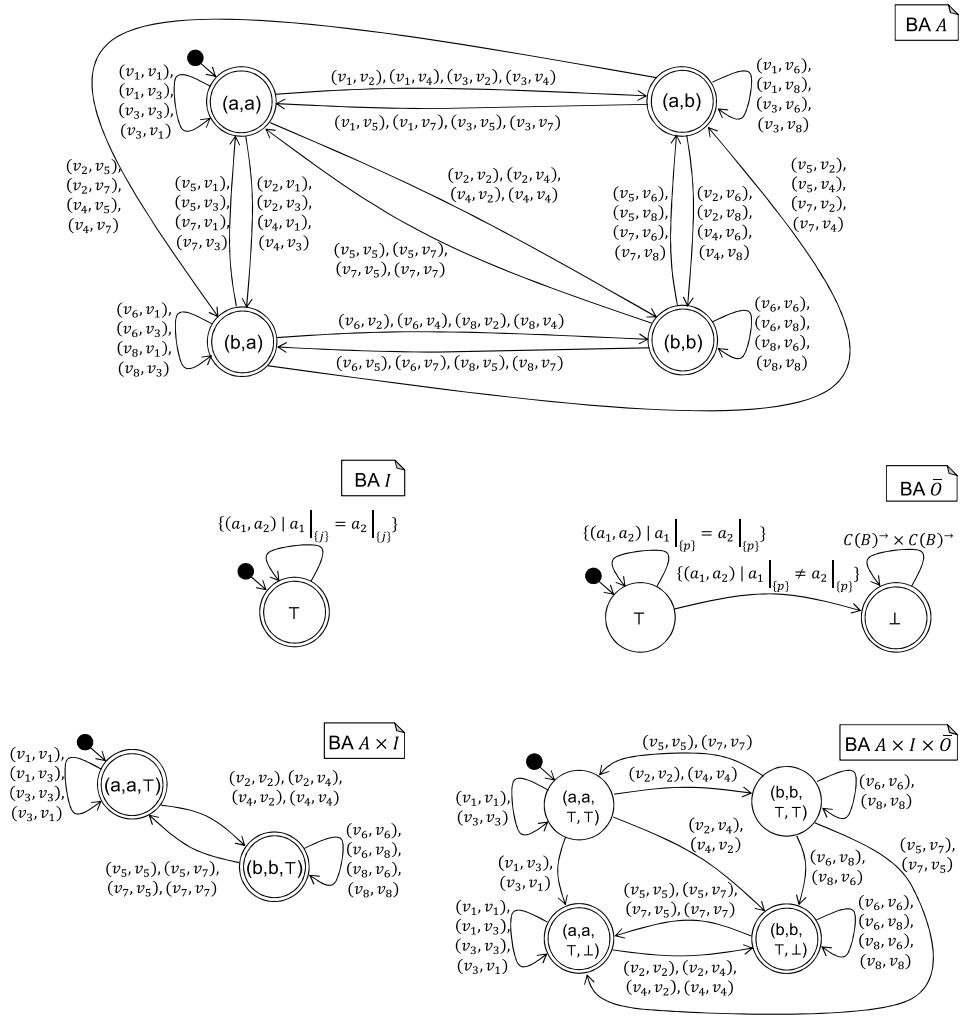


Fig. 9 The BA A models all tuples of behaviors of the TSPA B, which is depicted in Fig. 8. The BA I models the set of all tuples of behaviors in $C(B)^{\Omega}$ that are equal on the input channels in $I_B \backslash \{i\}$. The BA $\overline{O}$ models the set of all tuples of behaviors in $C(B)$ that are not equal on the output channel p. The reachable part of the BA $A \times I$ models all tuples of behaviors of A that are equal on all input channels in $I_B \backslash \{i\} = \{j\}$. The reachable part of the BA $A \times I \times \overline{O}$ models all tuples of behaviors of A that are equal on the input channel j and not equal on the output channel p

the language $\mathcal{L}(A) \cap \mathcal{L}(I) \cap \mathcal{L}(\overline{O})$. Using Theorem 8, the language recognized by $A \times I \times \overline{O}$ is not empty iff the channel i influences the channel p in the TSPA B. The BA A modeling all tuples of behaviors of the TSPA B is graphically illustrated in the top of Fig. 9. This BA uses the same transition labels as the TSPA B, which are defined in Fig. 8. The BAs I and $\overline{O}$ are depicted in the middle of Fig. 9. The BA I models the set of all tuples of behaviors in $C(B)^{\Omega}$ that are equal on all input channels in $I_B \backslash \{i\} = \{j\}$. The BA $\overline{O}$ represents the set of all behaviors in $C(B)^{\Omega}$ that are not equal on the output channel p. The bottom left of Fig. 9 depicts the reachable part of the BA $A \times I$ that accepts the intersection of the languages accepted by the BAs

A and I. Thus, the BA $A \times I$ models the set of all tuples of behaviors of A that are equal on the input channel j. The bottom right of Fig. 9 depicts the reachable part of the BA $A \times I \times \overline{O}$. This BA models all tuples of behaviors of A that are equal on the input channel j and not equal on the output channel p. As the language accepted by this BA is not empty, the channel i influences the channel p. For example, a word accepted by this BA is given by $w = (v_2, v_4) \cdot ((v_5, v_5), (v_2, v_2))^\infty$. The word w represents the behaviors $\alpha = v_2 \cdot (v_5, v_2)^\infty$ and $\beta = v_4 \cdot (v_5, v_2)^\infty$ where $\alpha|_{I_B \setminus \{i\}} = \beta|_{I_B \setminus \{i\}}$ and $\alpha|_p \neq \beta|_p$. Thus, the word w encodes a concrete proof in the form of two behaviors proving that the channel i influences the channel p.

The following example demonstrates the construction to show that the input channel i does not influence the output channel q in the TSPA B. To this effect, we first construct the BA $\overline{O'}$. This BA models all behaviors in $C(B)^\Omega$ that are not equal on the output channel q. Afterwards, we construct the BA $A \times I \times \overline{O'}$, which models all behaviors of A that are equal on all channels in $I_B \setminus \{i\} = \{j\}$ and not equal on the output channel q. The reachable part of the BA $\overline{O'}$ is depicted in the left part of Fig. 10. The right part of Fig. 10 depicts the reachable part of the BA $A \times I \times \overline{O'}$. The language of this BA is empty. Thus, with Theorem 8, the input channel i does not influence the output channel q in the TSPA B: For every input, the output on the channel q does not depend on the input on channel i.

4.4 Decomposing components along influencers

Composing the TSPAs obtained from decomposing a TSPA into two compatible TSPAs in parallel, such that the composition contains exactly the channels of the original, always results in a TSPA that generalizes the behavior of the original. This holds because hiding an input channel from a TSPA removes information that restricts the TSPA's behaviors:

Theorem 9 *Let A be a TSPA and let $D, E \subseteq C_A$ such that $D \cap E \cap O_A = \emptyset$ and $D \cup E = C_A$. Then, $behs(A) \subseteq behs(A \upharpoonright D \otimes A \upharpoonright E)$.*

Proof Let A, D, and E be given as above. Let $X = A \upharpoonright D$ and let $Y = A \upharpoonright E$. X and Y are compatible because $D \cap E \cap O_A = \emptyset$ implies $O_X \cap O_Y = \emptyset$. Let $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ be an execution of A. By definition of execution it holds that $s_0 = \iota_A$ and $(s_i, \theta_i, s_{i+1}) \in \delta_A$ for all $i \in \mathbb{N}$. Hence, using the definition of restriction we have that $(s_i, \theta_i|_{C_X}, s_{i+1}) \in \delta_X$ and $(s_i, \theta_i|_{C_Y}, s_{i+1}) \in \delta_Y$ for all $i \in \mathbb{N}$. Thus,

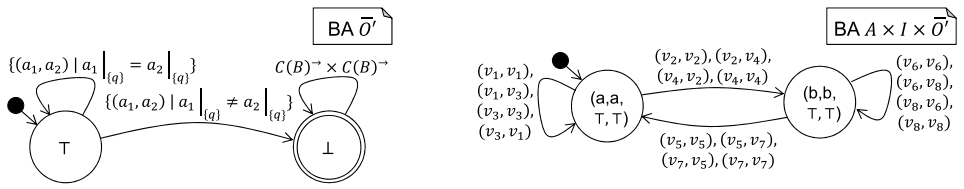


Fig. 10 The BA $\overline{O'}$ and the reachable part of the BA $A \times I \times \overline{O'}$ that models all tuples of behaviors of A that are equal on the input channel j and not equal on the output channel q

as by assumption $C_X \cup C_Y = C_A$, by definition of TSPA composition, this implies $((s_i, s_i), \theta_i, (s_{i+1}, s_{i+1})) \in \delta_{X \otimes Y}$. Observing that $(s_0, s_0) = (\iota_A, \iota_A)$ is the initial state of $X \otimes Y$, we can conclude that $\kappa = (s_0, s_0), \theta_0, (s_1, s_1), \theta_1, \ldots$ is an execution of $X \otimes Y$. Thus, $beh(\kappa) = beh(\sigma) \in behs(X \otimes Y)$. To conclude: for each execution of A, there exists an execution of $X \otimes Y$ such that the executions have the same behaviors. This implies that each behavior of A is also a behavior of $X \otimes Y$. Thus, $behs(A) \subseteq behs(X \otimes Y)$. □.

As hiding may remove information that restrict a TSPA's behaviors, the other direction does not necessarily hold. Thus, the composition of two TSPAs resulting from a decomposition may have behaviors that are not present in the original TSPA. However, if the decomposition is performed along channels that do not influence each other, then the composition of two TSPAs resulting from the decomposition has exactly the same behaviors as the original:

Theorem 10 *Let A be an unambiguously specified TSPA, let $i \in I_A$, and let $o \in O_A$. If $i \not\rightsquigarrow_A o$, then $behs(A \upharpoonright (\{o\} \cup I_A \backslash \{i\}) \otimes A \upharpoonright (C_A \backslash \{o\}) = behs(A)$.*

Proof Let A, i and o be given as above. Let $D = A \upharpoonright (\{o\} \cup I_A \backslash \{i\})$ and let $E = A \upharpoonright (C_A \backslash \{o\})$. As A is unambiguously specified, Theorem 4 guarantees that E is unambiguously specified. As $i \not\rightsquigarrow_A o$, Theorem 5 guarantees that D is unambiguously specified. As D and E are unambiguously specified and $O_D \cap I_E = \{o\} \cap I_A = \emptyset = (O_A \backslash \{o\}) \cap (I_A \backslash \{i\}) = O_E \cap I_D$, using Theorem 3, we have that $D \otimes E$ is unambiguously specified. By definition of D and E, we have $O_D \cap O_E = \{o\} \cap (O_A \backslash \{o\}) = \emptyset$ and $C_D \cup C_E = (\{o\} \cup I_A \backslash \{i\}) \cup (C_A \backslash \{o\}) = C_A$. Hence, with Theorem 9, we have $behs(A) \subseteq behs(D \otimes E)$. Therefore, as A and $D \otimes E$ are unambiguously specified and $O_A = \{o\} \cup (O_A \backslash \{o\}) = O_D \cup O_E = O_{D \otimes E}$ and $behs(A) \subseteq behs(D \otimes E)$, using Theorem 2 we can conclude $behs(A) = behs(D \otimes E)$. □

This enables decomposing components based on channel pairs that do not influence each other. Algorithm 1 is a procedure for iteratively determining a maximal decomposition with respect to the influence relation between channels in a TSPA. The basic operations are TSPA restriction and checking whether there exist channels that influence each other in a TSPA. A procedure for determining whether an input channel influences an output channel is detailed in the previous Sect. 4.3.

Algorithm 1 An algorithm for the parallel decomposition of a TSPA A based on the influences-in-A relation.

```
1: function DECOMPOSE(TSPA A)
2:     if ∃i ∈ I_A : ∃o ∈ O_A : i ↛_A o then
3:         let (i, o) ∈ I_A × O_A such that i ↛_A o
4:         D ← A↾({o} ∪ I_A \ {i})
5:         E ← A↾(C_A \ {o})
6:         return DECOMPOSE(D) ∪ DECOMPOSE(E)
7:     else
8:         return {A}
9:     end if
10: end function
```

For example, decomposing the TSPA B of Fig. 8 with Algorithm 1 yields the decomposition represented by the set $\{B \upharpoonright \{j\}, B \upharpoonright \{o\}, B \upharpoonright \{j,q\}, B \upharpoonright \{i,j,p\}\}$.

5 Elevator control system example revisited

Section 2 presented the software component for an elevator control system (ECS) as inspired by Butting et al. (2017b), Strobl et al. (1999) and Ringert et al. (2016). At some point, the engineers developed a monolithic `ECS` component as depicted in Fig. 1. The `ECS` component is a finite state system (Butting et al. 2017b; Strobl et al. 1999; Ringert et al. 2016) that can be transformed to a finite TSPA (Butting et al. 2017b). The component's implementation has already been shipped but is still available. Due to changed requirements for the elevator's successor version, the team needs to adjust the component's behavior concerning the control of the floor lights in response to the elevator's cabin position. The floor lights are controlled with messages sent via the channels `li1`, `li2`, and `li3`. The elevator's position is indicated by messages received via the channels `at1`, `at2`, and `at3`. Changing the implementation is error-prone as the architecture is monolithic, i.e., changing the implementation may change the component's behavior on channels that are not impacted by the changed requirement. For instance, as the component is not adequately decomposed, changing the component's implementation may result in a change of its behavior on the channels `up` and `down` for steering the elevator cabin, although the behavior on these channels does not need to be adjusted to satisfy the changed requirement. The engineering team is also uncertain which input channels influence which output channels, i.e., whether there are hidden influence dependencies between channels. The team thus uses our method for the automated decomposition of components.

Figure 11 depicts three `ECS` architectures that are obtained as intermediate results during the decomposition of the initial `ECS` implementation. The initial implementation is illustrated in the top-left of Fig. 11.

The decomposition procedure initially detects that the input channel `btn2` does not influence the output channel `li1` (cf. Algorithm 1, l. 2). An automatic procedure for checking whether an input channel influences an output channel is detailedly

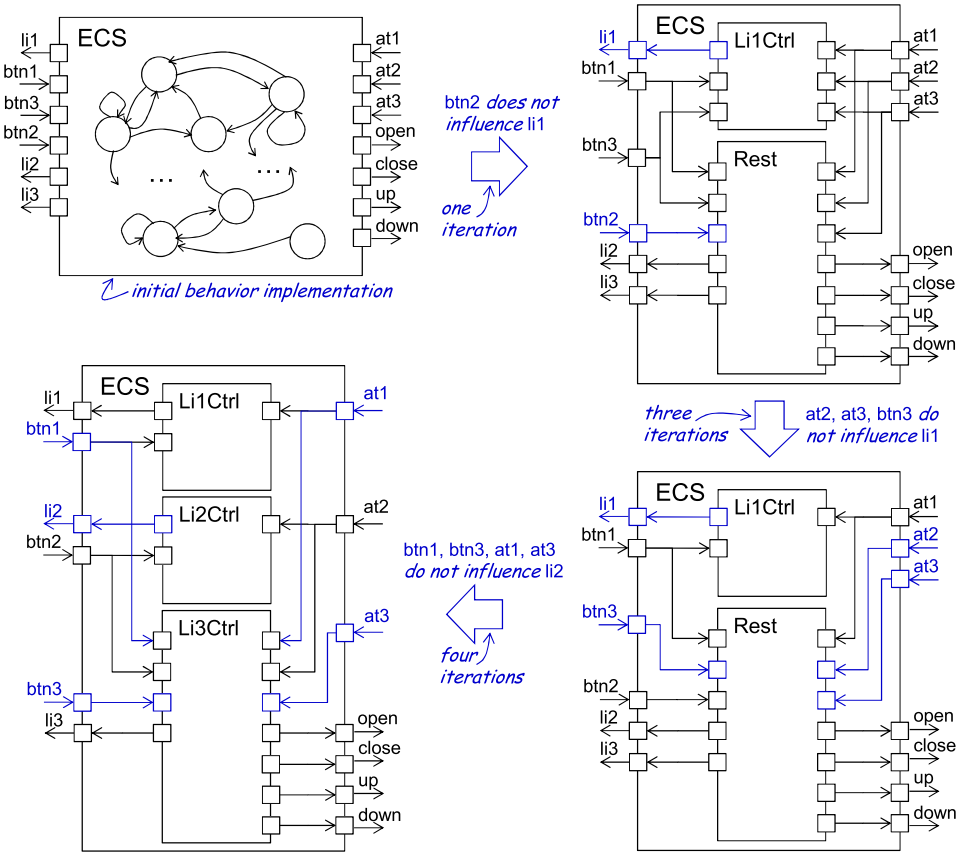


Fig. 11 Representation of different intermediate architectures obtained during the automatic decomposition. Top-left describes the initial behavior representation as presented in Fig. 1. The architectures represented clockwise describe intermediate results after various iterations

described in Sect. 4.3.1. The algorithm splits the `ECS` implementation into the two components `Li1Ctrl` and `Rest` (called *D* and *E* in Algorithm 1, ll. 4–5). The resulting architecture is depicted in the top-right of Fig. 11. The component `Li1Ctrl` has the single output channel `li1` and the five input channels `btn1`, `btn3`, `at1`, `at2`, `at3`. As the input channel `btn2` does not influence the output channel `li1` in the component `ECS`, the channel `btn2` is no input channel of the component `Li1Ctrl`. At this stage during the decomposition procedure, it is not clear whether other input channels do not influence the output channel `li1`, either. Similarly, at this stage, it has not been detected which channels do not influence the other output channels. Therefore, all input channels of the initial `ECS` component are also input channels of the component `Rest` and all output channels of the initial `ECS` component except the channel `li1` are the output channels of `Rest`.

In the next three iterations of the decomposition, Algorithm 1 detects that the channels `at2` , `at3`, and `btn3` do not influence the channel `li1` in `Li1Ctrl`, either. Therefore, Algorithm 1 decomposes the component `Li1Ctrl` accordingly: The input channels `at2`, `at3`, and `btn3` are removed from the component

`Li1Ctrl`. As byproducts from the decomposition, the algorithm produces components without output channels. As these components do not sent messages to their environments, they can be safely removed without changing the semantics of the architecture and are not depicted above. The resulting architecture after the decomposition and the removal of the components is depicted in the bottom-right of Fig. 11.

Similarly, in the next four iterations of the decomposition procedure, the algorithm detects that the input channels `btn1`, `btn3`, `at1`, and `at3` do not influence the channel `li2` in `Rest` and decomposes the component `Rest` accordingly. The resulting architecture after removing the components without output channels is depicted in the bottom-left of Fig. 11.

Analogously, the input channels `btn1`, `btn2`, `at1`, and `at2` do not influence the channel `li3` in `Rest`. Therefore, the algorithm decomposes the component `Rest` accordingly. The resulting architecture after removing all components without output channels is depicted Fig. 12. In this architecture, every input channel of every component influences every output channel of the component. Therefore, the decomposition procedure terminates.

By the decomposition procedure's properties, the decomposed component (cf. Fig. 12) is semantically equivalent to the original and clearly better separated regarding the influence relation between channels. From reviewing the new architecture, the engineers now understand that messages emitted via a channel for controlling a floor light only depend on the corresponding elevator cabin position sensor and whether the corresponding request button has been pressed. The implementation of a light controller can now be changed without the threat of accidentally changing the behavior on other channels. They also understand that all input channels influence the channels `open`, `close`, `up`, and `down`. Thus, the messages the component sends via these channels depend on the messages received via all input

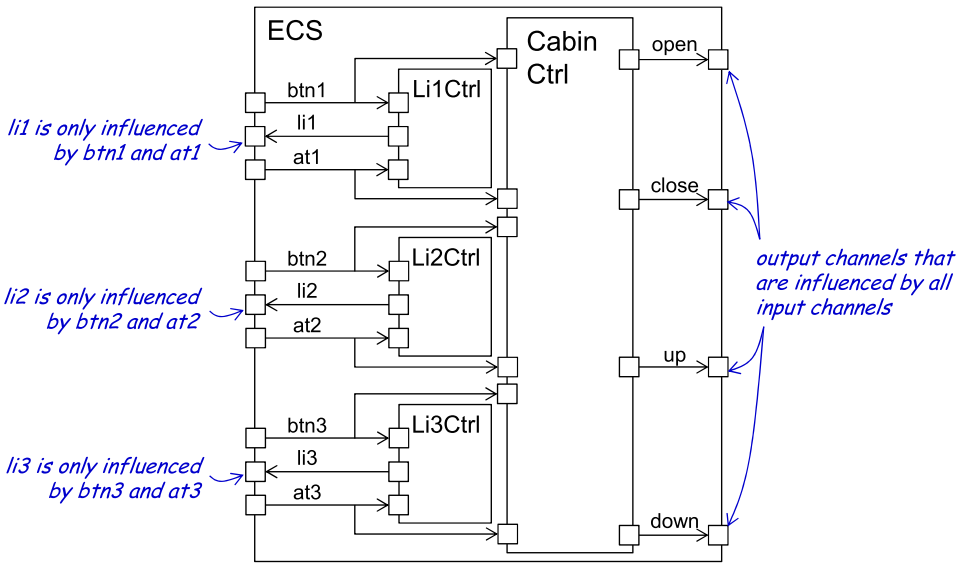


Fig. 12 Semantically equivalent decomposed variant of the ECS

channels. The behavior of the floor lights controlling components and the cabin controlling component can now be unit tested and formally verified individually. As the decomposition is a refactoring, the satisfactions of preexisting symbolic system tests and formally specified requirements for the `ECS` component are preserved.

6 Discussion

Currently, our approach applies only to unambiguously specified and deterministic component implementations. This prevents automated decomposition of component specifications, which usually are underspecified (e.g., by non-determinism). Also, our influence-based decomposition is limited to time-synchronous systems. While these are ubiquitous in embedded and cyber-physical systems, other domains, such as cloud computing, usually rely on event-based message passing. Although FOCUS supports both, non-deterministic specification and untimed communication, applying the notion of channel influencing requires additional research. We consider this as interesting future work.

As our notion of influencing channels establishes relations from input channels to output channels, the resulting decomposition always is parallel, i.e., produces subcomponents connecting a subset of the input channels to a subset of the output channels. Prescribing intermediate channels for more detailed decomposition might be additionally helpful. This also is subject to future research.

Algorithm 2 Parallel decomposition of a TSPA A based on the influences in A relation while respecting pairs of channels that must not be separated.

function DECOMPOSE(TSPA $A, I \subseteq I_A \times O_A$)
  if $\exists i \in I_A : \exists o \in O_A : (i, o) \notin I \wedge i \not\rightsquigarrow_A o$ **then**
    let $(i, o) \in I_A \times O_A$ **such that** $(i, o) \notin I \wedge i \not\rightsquigarrow_A o$
    $D \leftarrow A{\restriction}(\{o\} \cup I_A \setminus \{i\})$
    $E \leftarrow A{\restriction}(C_A \setminus \{o\})$
    return DECOMPOSE(D) $\cup$ DECOMPOSE(E)
  else
    return $\{A\}$
  end if
end function

The algorithm for the decomposition of components as presented in Sect. 4 always computes a maximal decomposition: It decomposes the input component (respectively the intermediate decomposition results) as long as there exists at least one input/output channel pair where the input channel does not influence the output channel. A user might consider an input channel to be associated with an output channel, although the input channel does not influence the output channel. This might be the case, for instance, because the channels are functionally related. In the ECS example (cf. Fig. 12), for instance, a user might consider each button-channel (`btn1`, `btn2`, `btn3`) to be associated with each light-channel (`li1`, `li2`, `li3`). This might be the case, because the channels are functionally related in the sense

that they are all used for steering different floor lights. In such cases, the user might be not interested in a maximal decomposition of the system. Instead, she might be interested in a decomposition procedure that does definitely not decompose predefined pairs of input and output channels, disregarding whether the input channel of a pair influences the output channel of the pair. Algorithm 2 is an adjusted version of Algorithm 1 for accomplishing this task. The algorithm additionally takes a set I (for inseparable) of pairs of input and output channels of the TSPA as input. The algorithm separates an input channel from an output channel iff the input channel does not influence the output channel and the input/output channel pair is no element of the set I containing the pairs of inseparable channels. Thus, the adjusted algorithm computes a maximal decomposition while respecting pairs of channels that should not be separated from each other.

FOCUS operates on component instances, i.e., the information about component types is implicit only. Consequently, our approach produces component instances also. If these, as illustrated by components `Li1Ctrl`, `Li2Ctrl`, and `Li3Ctrl` of Fig. 12, are equivalent, we could deduce type information and synthesize new component types for patterns identified through decomposition accordingly. This might facilitate component reuse. In this case, the decomposition would derive a new component type `LightCtrl` and instantiate it three times accordingly.

This paper presents the theoretical foundations of automated decomposition along pairs of channels that influence each other. The automated decomposition rests on the assumption that the systems largely comprise components that are free of side effects, i.e., "pure", FOCUS components. Where components yield side effects, checking whether system functions or capabilities have changed demands additional measures, such as sufficient test coverage or manual analysis. Another challenge in using our method for automated decomposition is its scaling-up. For instance, the `ECS` sketched in Fig. 1 and based on Butting et al. (2019) will be translated into a TSPA with a large number of transitions, which might be too large for human comprehension and reproduction in this paper. However, usually, the models that engineers start with are specified manually and, from our experience, thus, small and comprehensible.

Our approach for automated decomposition is limited to FOCUS-compatible architectures, which belong to a group of more formal modeling techniques that might not yet be state-of-practice. For modelers operating within less well-defined or incompatible technological spaces, we consider our contribution towards the automated evolution of software architecture models a relevant case in point for at least investigating the benefits of more formal modeling techniques in practice. Whether the results from the decomposition are useful for engineers needs further evaluation including real systems and engineers. We consider this interesting future work.

7 Related work

While agile architecting has been under investigation lately, e.g., driven by change impact analysis (Díaz et al. 2013), cost-and-risk analysis (Poort 2014), or for specific domains (Díaz et al. 2014), there are only a few approaches towards agile

architecting with semantically well-defined ADLs and these usually rest on Focus or the π-calculus (Milner 1999).

7.1 Automata decomposition

The decomposition of automata has been subject to research for several decades. For instance, our contribution also relates to parallel decomposition of automata (Gerace and Gestri 1967). While it also aims at a practical decomposition (Nozaki 1978), i.e., the resulting components yield fewer states than the component they were decomposed from, in contrast to more current related work (Uygur and Sattler 2013), it operates specifically on time-synchronous port automata. Similarly, while port automata generally can be decomposed into compositions consisting of FIFOs and XORs only (Koehler and Clarke 2009), this resulting granularity does not produce automata accessible for constructive systems engineering. Related decomposition approaches also exist for probabilistic automata (Carlsson and Yu 2015) or linear automata (Plotkin and Plotkin 2015), none of which consider automated decomposition in the presence of influencing channels.

There also are related approaches in the parallel decomposition of processes (Jongmans et al. 2016). Here, the decomposition leverages the underlying Reo (Razavi and Sirjani 2006) process algebraic semantics (Kokash et al. 2010). With Reo, communication is untimed in the Focus (Broy and Stølen 2001; Broy 2010) sense and the decomposition follows process actions instead of shared channels. How the parallel decomposition of Reo processes can be translated to untimed Focus systems is subject to ongoing research.

7.2 Agile architecting

Industry and research have produced over 120 ADLs (Malavolta et al. 2013). Most of these feature the composition of components into larger architectures and some of these also feature the denotational semantics necessary to support agile architecting through automated decomposition. This section discusses related ADLs and their support for automated decomposition.

AutoFocus 3 (Hölzl and Feilkas 2010) and MontiArc (Butting et al. 2017a) are ADLs featuring tool chains for developing architectures of reactive software systems that are grounded in Focus (Broy and Stølen 2001). This paper's system model describes the formal foundations of both ADLs. AutoFocus 3 supports model checking the behavior of architectures against LTL and CTL properties (Campetelli et al. 2011). MontiArc supports semantic differencing of components (Butting et al. 2017b). However, both currently lack fully automated component decomposition methods. Hence, even if employed in agile processes, the challenge of manually decomposing monolithic architectures remains. Our approach can directly be integrated into the tool chains of both ADLs.

The π-ADL supports model checking for verifying software architectures against DynBLTL properties (Cavalcante et al. 2016). Therefore, a statistical model of finite system executions is created and the probability of satisfying a property within

confidential bounds is calculated. However, we are unaware of any agile architecting methods based on the π-calculus. As our approach is based on FOCUS and not on the π-calculus, it cannot be directly integrated into the tool chains of ADLs that are based on the π-calculus. Developing an influence relation and a decomposition procedure for systems based on the π-calculus is interesting future work.

7.3 Applicability to other automata models

Other automata models, such as I/O automata (Lynch and Tuttle 1989), Interface automata (de Alfaro and Henzinger 2005), team automata (ter Beek et al. 2003), and component-interaction automata (Brim et al. 2006), do not include the notion of channel. Instead, they distinguish between input, internal, and output actions. Composition operators compose different automata according to their actions. As these automata models do not explicitly incorporate the notion of channel, it is not possible to define an influence relation between the channels of these automata. However, it could be interesting to define a notion of influence between input and output actions of the automata. The relation could be defined such that it identifies whether the receipt of a specific input action influences the output of a specific output action. Transferring this idea to the automata model used in this paper, the above corresponds to the question whether a specific message on a specific input channel influences the output of a specific message on a specific output channel. We consider the definition of such a relation and the development of automated tool support as interesting future work. This would enable a more fine-grained analysis as presented in this paper. Whether this analysis or the analysis presented in this paper is more appropriate depends on the use case and intention by the developer.

For other automata models that include the notion of channel, such as port automata (Grosu and Rumpe 1995), time-synchronous channel automata (Butting et al. 2019), and MAA_{ts} automata (Ringert 2014), it is possible to transfer the notion of influence between channels. However, some of these automata models use a different semantics as the automaton model used in this paper. The method for detecting whether one channel influences another channel needs to be adjusted depending on the semantics of the respective automaton model. Consequently, the decomposition method also needs to be adjusted depending on the composition operator of the respective automaton model.

8 Summary

We have presented a method to automatically decompose a monolithic deterministic component into an architecture consisting of multiple subcomponents that are composed in parallel. This supports agile architecting by reducing the effort for analyzing and implementing system behavior along subcomponents and facilitates refinement and refactoring of architectures. To this end, we have conceived a notion of influence between channels and formalized it in the FOCUS (Broy and Stølen 2001) theory. We have proven that this decomposition is an actual refactoring, i.e., the

resulting systems are semantically equivalent to the original systems. Hence, this decomposition can be applied to stepwise refinement and ultimately facilitates architecture modeling.

Acknowledgements This research has partly received funding from the German Federal Ministry for Education and Research under Grant No. 01IS16043P. The responsibility for the content of this publication is with the authors.

References

Béal, M., Carton, O.: Determinization of transducers over infinite words. In: ICALP, Springer, Lecture Notes in Computer Science, vol. 1853, pp. 561–570 (2000)

Béal, M.P., Carton, O.: Determinization of transducers over finite and infinite words. Theor. Comput. Sci. **289**(1), 225–251 (2002)

Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-interaction automata as a verification-oriented component-based system specification. SIGSOFT Softw. Eng. Notes **31**, 4-es (2006)

Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. **53**(10), 1758–1782 (2010)

Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement. Springer, Heidelberg (2001)

Büchi, J.R.: On a decision method in restricted second order arithmetic. In: International Congress on Logic, Methodology and Philosophy of Science, pp. 1–11 (1962)

Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A.: Systematic language extension mechanisms for the MontiArc Architecture Description Language. In: Modelling Foundations and Applications (ECMFA'17), Held as Part of STAF 2017, Springer International Publishing, pp. 53–70 (2017)

Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Semantic differencing for message-driven component & connector architectures. In: International Conference on Software Architecture (ICSA'17), IEEE, pp. 145–154 (2017)

Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. J. Syst. Softw. **149**, 437–461 (2019)

Campetelli, A., Hölzl, F., Neubeck, P.: User-friendly model checking integration in model-based development. In: International Conference on Computer Applications in Industry and Engineering (2011)

Carlsson, G., Yu, J.: A prime decomposition of probabilistic automata (2015). arXiv preprint arXiv :150301502

Cavalcante, E., Quilbeuf, J., Traonouez, L.M., Oquendo, F., Batista, T., Legay, A.: Statistical model checking of dynamic software architectures. In: European Conference on Software Architecture (2016)

de Alfaro, L., Henzinger, T.A.: Interface-based design. In: Engineering Theories of Software Intensive Systems (2005)

Debruyne, V., Simonot-Lion, F., Trinquet, Y.: EAST-ADL—an architecture description language. In: Architecture Description Languages, pp. 181–195. Springer (2005)

Díaz, J., Pérez, J., Garbajosa, J., Yagüe, A.: Change-impact driven agile architecting. In: 2013 46th Hawaii International Conference on System Sciences, pp. 4780–4789 (2013)

Díaz, J., Pérez, J., Garbajosa, J.: Agile product-line architecting in practice: a case study in smart grids. Inf. Softw. Technol. **56**(7), 727–748 (2014)

Farwer, B.: ω-Automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata Logics and Infinite Games: A Guide to Current Research, pp. 3–21. Springer, Berlin (2002)

Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Boston, MA (2012)

France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering 2007 at ICSE (2007)

Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. The MK/OMG Press. Elsevier Science, Amsterdam (2011)
Gerace, G., Gestri, G.: Decomposition of synchronous sequential machines into synchronous and asynchronous submachines. Inf. Control **11**(5), 568–591 (1967)
Grosu, R., Rumpe, B.: Concurrent timed port automata. Technical Report TUM-I9533, TU Munich (1995)
Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
Hölzl, F., Feilkas, M.: AutoFocus 3—a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) Model-Based Engineering of Embedded Real-Time Systems. Springer, Berlin (2010)
Jongmans, S.S., Clarke, D., Proença, J.: A procedure for splitting data-aware processes and its application to coordination. Sci. Comput. Program. **115**, 47–78 (2016)
Koehler, C., Clarke, D.: Decomposing port automata. In: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1369–1373. ACM (2009)
Kokash, N., Krause, C., de Vink, E.P.: Data-aware design and verification of service compositions with Reo and mCRL2. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM (2010)
Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**(3), 219–246 (1989)
Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**, 869–891 (2013)
Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)
Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, New York, NY (1999)
Naur, P., Randell, B. (eds.): Software engineering: report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct 1968, Brussels, Scientific Affairs Division, NATO (1969)
Nozaki, A.: Practical decomposition of automata. Inf. Control **36**(3), 275–291 (1978)
Plotkin, B., Plotkin, T.: Decompositions and complexity of linear automata (2015). arXiv preprint arXiv :150606017
Poort, E.R.: Driving agile architecting with cost and risk. IEEE Softw. **31**(5), 20–23 (2014)
Razavi, N., Sirjani, M.: Using Reo for formal specification and verification of system designs. In: Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-design, 2006. MEMOCODE'06. Proceedings. IEEE Computer Society, pp. 113–122 (2006)
Ringert, J.O.: Analysis and Synthesis of Interactive Component and Connector Systems. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Herzogenrath (2014)
Ringert, J.O., Rumpe, B.: A little synopsis on streams, stream processing functions, and state-based stream processing. Int. J. Softw. Inform. **5**(1–2), 29–53 (2011)
Ringert, J.O., Rumpe, B., Wortmann, A.: Model-based specification of component behavior with controlled underspecification. In: Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16) (2016)
Safra, S.: On the complexity of σ-automata. In: Proceedings of the 29th Annual Symposium on Foundations of Computer Science, pp. 319–327. IEEE Computer Society (1988)
Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Chugo, D., Yokota, S. (eds.) Introduction to Modern Robotics. iConcept Press, Hong Kong (2011)
Strobl, F., Wisspeintner, A., Marz, A.: Specification of an elevator control system. Technical report, TU Munich (1999)
ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in team automata for groupware systems. Comput. Support. Coop. Work (CSCW) **12**, 21–69 (2003)
Uygur, G., Sattler, S.M.: Parallel decomposition for safety-critical systems. In: 2013 3rd International Electric Drives Production Conference (EDPC), pp. 1–8 (2013)
Van Ommering, R., Van Der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)
Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley Software Patterns Series. Wiley, Hoboken (2013)

Weber, A.: Transforming a single-valued transducer into a mealy machine. J. Comput. Syst. Sci. **56**(1), 46–59 (1998)
Weber, A., Klemm, R.: Economy of description for single-valued transducers. Inf. Comput. **118**(2), 327–340 (1995)

Semantic Differencing for Message-Driven Component & Connector Architectures

Arvid Butting, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de

***Abstract*—Stepwise refinement is a development methodology in which software components progressively evolve under strict adherence of proven properties. This requires means to check whether a new version of a component – with potentially different interface and behavior implementation – refines the behavior of its predecessor. Where architecture description languages (ADLs) support refinement checking, the complexity of their semantic domain requires (partially) manual proving to establish refinement between component versions. We identified a subset of the FOCUS semantics for describing distributed systems as stream processing functions that is powerful enough to model complex and realistic systems, yet sufficiently powerful to support fully automated refinement checking. Leveraging this, we present a refinement checking method for ADLs yielding semantics that can be expressed as stream processing functions. This method relies on transforming architectures into composed port automata and translating these to Büchi automata prior to proving refinement using RABIT for language inclusion checking. This method enables to compare the behaviors of component versions with minimal effort, yields witnesses for non-refining component pairs, and, thus, ultimately facilitates stepwise component refinement.**

I. INTRODUCTION

Stepwise refinement [3], [4] is a development methodology for continuous architecture modeling based on controlled evolution and progressive improvement of components: each successor component version must adhere to properties already proven for its predecessors. To this effect, checking whether successor component versions *refine* their predecessors in terms of observable input/output behavior is crucial.

Architecture description languages (ADLs) [20] leverage the potential of model-driven engineering [32] for the description of software architectures. Research has produced over 120 ADLs [19] for different domains, such as automotive [9], avionics [11], consumer electronics [31], or robotics [28].

Similar to UML [21], the specific semantics of many ADL details are encoded in their infrastructures and tools only. Where fully detailed denotational or operational semantics are available, such as FOCUS [5], these are usually too complex for fully automated refinement checking and typically require to (partially) manually prove refinement between two component versions. This impedes stepwise refinement so severely that it becomes a "highly idealistic" [3] idea. However, enabling stepwise refinement for software architecture models would greatly facilitate development in domains where component adherence to certain properties is crucial.

We identified a subset of the FOCUS [5] semantics for time-synchronous, distributed, interactive systems that is powerful enough to model complex and realistic systems and yet enables fully automated refinement checking between components. Based on this, we present an approach to transform software component models into a variant of port automata [12], compose these syntactically, and translate these into Büchi automata, where their refinement can be checked via language inclusion. This approach is realized with the MontiArcAutomaton component & connector ADL [23], [25] and the RABIT [1], [2] tool for fully automated language inclusion checking between Büchi automata. It enables modeling software architectures with powerful ADLs and checking refinement on a push-button basis. To this effect, the contributions of this paper are:

- Formulation of the semantics domain of time-synchronous [5] stream processing functions (TSSPFs) inspired by the notion of stream processing function [24].
- Presentation of a time-synchronous variant of port automata (TSPA) [12] with operational semantics based on execution traces and denotational semantics based on sets of TSSPFs.
- A semantically compositional syntactic composition operator for TSPAs: The semantics of the syntactic composition of two TSPAs is equal to the composition of the semantics of the individual TSPAs.
- A transformation from finite TSPAs to Büchi automata.
- A proof showing the operational semantics of a finite TSPA and the language accepted by the Büchi automaton resulting from such a transformation coincide.
- The result that refinement checking and disproof generation in form of semantic difference witnesses for software architectures where components can be mapped to finite TSPAs can be reduced to language inclusion checking and counterexample generation for Büchi automata.
- An implementation based on MontiArcAutomaton [23], [25] and RABIT [1], [2].

In the following, Sec. II sketches the idea of stepwise refinement, before Sec. III presents the FOCUS subset used as semantics domain for components. Afterwards, Sec. IV presents semantic differencing based on this subset and Sec. V presents the implementation of our approach with MontiArcAutomaton and RABIT and evaluates its applicability. Sec. VI discusses observations and Sec. VII highlights related work. Sec. VIII concludes.

II. EXAMPLE

Consider the model-driven development of an elevator control system (ECS) as presented in [29]. The ECS depicted

[BKRW17] A. Butting, O. Kautz, B. Rumpe, A. Wortmann:
Semantic Differencing for Message-Driven Component & Connector Architectures.
In: International Conference on Software Architecture (ICSA'17), pages 145-154. IEEE, 2017.
www.se-rwth.de/publications/

in Fig. 1 comprises two hierarchically composed components representing the three floors the elevator serves (component `Floors`) and the elevator cabin (component `Elevator`) itself. Whenever a button on a floor (indicated, for example, by a message on the incoming port `btn1`) is pressed, the `ECS` should activate the light (by sending a message via outgoing port `li1`) on the corresponding floor and instruct the elevator cabin to visit that floor. The control logic of the elevator is modeled via a statechart variant embedded into the `Elevator`'s subcomponent `Control`. This component receives messages upon arriving at a specific floor (*e.g.,* via incoming port `at1`) and sends messages to `Door` and `Motor` to operate its door and to move between the floors. The latter two embed models of compact action languages to describe their respective behavior.

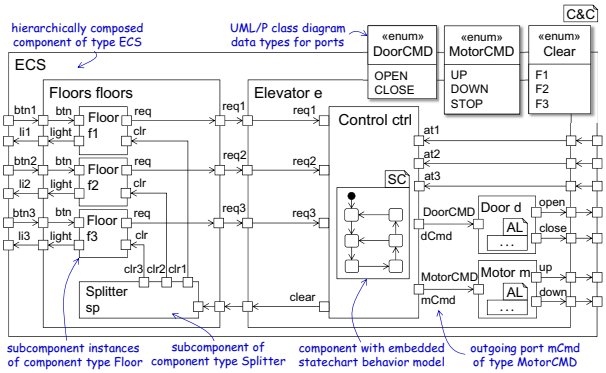


Fig. 1. The elevator control system `ECS` comprises subcomponents to manage serving elevation requests on up to three floors.

For this version of `ECS`, the company has proven that certain properties hold (*e.g.,* that it cannot produce blocking situations). Now the company aims to replace the `Elevator` component with an improved version that reacts only to elevator requests on a floor if there is no such request yet. To this effect, the company employs stepwise refinement to avoid proving the properties of `Elevator` again for its successor version `NewElevator`. Therefore, the behavior descriptions of all subcomponents are translated into port automata. For composed components, the behavior descriptions of their subcomponents are translated also and merged iteratively. This ultimately eliminates all hierarchy levels but the last. The result of this transformation is depicted in Fig. 2, where the behavior descriptions of all three subcomponents have been transformed accordingly and merged into a single port automaton. The same is performed for the improved `NewElevator` component before both are transformed into nondeterministic Büchi automata as presented in Sec. V.

Using this transformation reduces semantic component refinement to language inclusion on Büchi automata and can be solved automatically using RABIT. Hence, with this infrastructure in place, the company now can fully automated ensure whether the `NewElevator`, and its potential successors, actually refine their predecessors or require further adjusting. Where refinement is refuted, difference witnessing

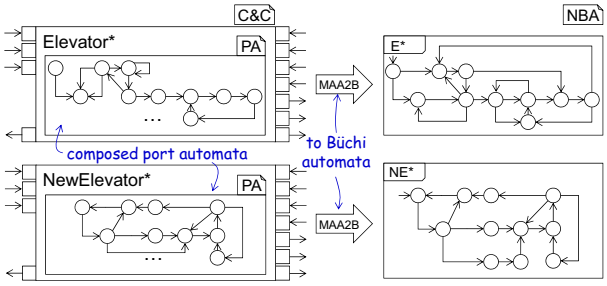


Fig. 2. The composed components `Elevator` and `NewElevator` each are transformed into flat components with a single port automaton prior to being transformed into Büchi automata and checked for language inclusion.

input/output pairs are produced. This automation of stepwise refinement can increase the pace of each refinement step and, hence, overall development efficiency.

III. A Semantics Domain for Components

This section introduces the semantics domain for components based on the FOCUS framework [3], [5], [12], [24], [27] and recaps the most important results from [12], which underlie the approach presented in this paper.

We interpret software architectures as networks of autonomously acting components communicating in a time-synchronous manner via directed, typed channels connecting the components' interfaces. A time-synchronous architecture can be interpreted as a system where component computations are performed concurrently and controlled by a global clock that splits runtime into discrete and equidistant time units. In every time unit, each component receives finitely many input messages via its interfaces and outputs finitely many messages to its environment. The computations of each component in every time unit must terminate.

In the remainder, we denote by $[X \to Y]$ the set of all functions from a set X to a set Y. For a function $f \in [X \to Y]$ and a set $Z \subseteq X$, the restriction of f to Z is the function $f|_Z \in [Z \to Y]$ that satisfies $f|_Z(x) = f(x)$ for all $x \in Z$. Given two functions $f \in [X \to A]$ and $g \in [Y \to B]$, the overriding union of f with g is the function $f + g \in [(X \cup Y) \to (A \cup B)]$ that satisfies $(f+g)(x) = g(x)$ if $x \in Y$ and $(f+g)(x) = f(x)$ if $x \in X \setminus Y$ for all $x \in X \cup Y$.

A. Streams, Messages, Types, and Communication Histories

The history of messages a component receives or sends via an interface is formally described as a stream that contains messages in order of their transmission. Let M be an arbitrary alphabet. A stream over the set M is a finite or infinite sequence of elements from M. Following [5], we denote by

- M^* the set of all finite streams over M,
- M^∞ the set of all infinite streams over M,
- $\langle\rangle$ the empty stream, which is an element of M^*,
- $s^\frown t$ the concatenation of two streams s and t such that $((M^* \cup M^\infty), ^\frown, \langle\rangle)$ is a monoid. If $s \in M^\infty$ then $s^\frown t = s$.
- $\sqsubseteq$ the prefix relation over streams, which is a partial order defined by: $\forall s, t \in (M^* \cup M^\infty) : s \sqsubseteq t \Leftrightarrow \exists u : s^\frown u = t$,

- $s.t$ the t-th element of a stream $s \in M^\infty$,
- $s{\downarrow_t}$ the prefix of a stream $s \in M^\infty$ of length $t \in \mathbb{N}$.

In the remainder, let M denote an arbitrary but fixed set of data elements, such as messages, and let $Type$ be a set of data types such that each $t \in Type$ satisfies $t \subseteq M$. Types facilitate restricting the set of messages a component may emit or receive via an interface. We assume a discrete model of time where component computation is divided into discrete time units of equal and finite duration. In each time unit each component receives at most one message via each incoming interface, may perform finitely many state changes and emits at most one message via each outgoing interface. We use the special symbol $\varepsilon \in M$ to denote the absence of a message during a time unit and require $\varepsilon \in t$ for each $t \in Type$.

A *channel* is an identifier for a communication link between interface elements of components. In the following we denote by C a set of typed channel names. The function $type \in [C \to Type]$ maps each channel in the set C to its type. Let $B \subseteq C$ be an arbitrary set of channel names. A *communication history* is an element of the set B^Ω defined as follows:

$B^\Omega \stackrel{\text{def}}{=} \{h \in [B \to M^\infty] \mid \forall b \in B : h(b) \in type(b)^\infty\}$.

A communication history $h \in B^\Omega$ is used to model the history of messages emitted via the channels in the set B.

Let $h \in B^\Omega$ be a communication history, $H \subseteq B^\Omega$ a set of communication histories, and $t \in \mathbb{N}$ a natural number. We lift the operator $\downarrow$ to communication histories and sets of communication histories in a point-wise manner, *i.e.,* $b{\downarrow_t} \in [B \to M^*]$ denotes the function that satisfies $b{\downarrow_t}(i) = b(i){\downarrow_t}$ for all $i \in B$ and $H{\downarrow_t} \stackrel{\text{def}}{=} \bigcup_{h \in H} h{\downarrow_t}$ denotes the set resulting from applying the operator to each element in H.

B. Time-Synchronous Stream Processing Functions

We model the semantics of distributed interactive systems as sets of time-synchronous stream processing functions (TSSPFs). The notion of TSSPFs is inspired by the notion of timed SPFs [5], [12], [24], [27]. The major and crucial difference between the two notions is that TSSPFs process exactly one message per channel per time unit, whereas SPFs process a stream of messages per channel per time unit. The key idea is to treat components as black-boxes having an observable behavior characterized by the interactions on channels between systems and subsystems while hiding internal implementation details. A component is mapped to a set of functions describing the component's possible behaviors. Such a function maps communication histories over the set of input channels of a component to communication histories over the set of the component's output channels. Thus, each function in the semantics of a component with input channels $I \subseteq C$ and output channels $O \subseteq C$ is of the form $f \in [I^\Omega \to O^\Omega]$. However such functions are not always realizable in the sense that they can be implemented [5], [22]. Intuitively, the characterizing properties for realizability are that a component cannot change messages received or sent in the past and cannot react to messages received in the future [5], [22], [24], [27]. Thus, the output of a behavior describing function until time t must be completely determined by its input until time t:

Definition 1 (Time-Synchronous Stream Processing Function)**.** *Let $I, O \subseteq C$ be two disjoint sets of input and output channels. A function $f \in [I^\Omega \to O^\Omega]$ is called (weakly causal) time-synchronous stream processing function iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow f(i){\downarrow_t} = f(i'){\downarrow_t}$.

We denote by $[I^\Omega \xrightarrow{wc} O^\Omega]$ the set of all TSSPFs mapping input histories in I^Ω to output histories in O^Ω. The semantics of components are modeled as closed sets of TSSPFs.

Definition 2 (Component Describing)**.** *Let $I, O \subseteq C$ be two disjoint sets of channels. A set of TSSPFs $F \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ is called component (semantics) describing iff it satisfies $\forall g \in [I^\Omega \xrightarrow{wc} O^\Omega] : ((\forall i \in I^\Omega : \exists f \in F : g(i) = f(i)) \Rightarrow g \in F)$.*

The definition above makes the semantics domain of components fully abstract [12], [13] in the sense of [15] and allows to handle unbounded nondeterminism [12]. Full abstraction is achieved by the closeness property, which requires that each TSSPF resulting from a combination of TSSPFs included in the set F is also included in F. The closeness property is also important to make component semantics as little distinguishing as possible. This is illustrated by the fact that two different arbitrary sets of TSSPFs may encode the same component behaviors. The reason for this is that one may find a TSSPF $g \notin F$ that is not included in a set of TSSPFs F, which can be interpreted as a combination of different TSSPFs contained in F. It thus does not induce a new behavior not already covered by a TSSPF in F but, for instance, induces a semantic difference between a component with semantics described by F and a component with semantics described by $F \cup \{g\}$. As a result the semantics of two components that have the exact same observable behaviors may be considered unequal. Consequently, full abstraction is not achieved. Thereby, the closeness property is necessary.

1) Composition of TSSPFs: Composition is an important concept to achieve modularity. Composing the semantics of the individual components of a system leads to the semantics of the whole system. Composing arbitrary sets of TSSPFs can lead to realizability problems in delay-free feedback loops where the output of a component in time unit t depends on its input in time unit t and vice versa. Thus, composition is only defined for TSSPFs where causality between inputs and outputs on channels connected via a feedback loop is ensured. This is the case if one of the TSSPFs participating in a composition is strongly causal with respect to its channels connected by the composition. Intuitively, a set of TSSPFs F is strongly causal with respect to (J, P), if the output of at least one TSSPF $f \in F$ on the channels in P until time unit $t+1$ is not influenced by the function's inputs received on the channels in J after time unit t.

Definition 3 (Strongly Causal Modulo)**.** *Let $f \in [I^\Omega \xrightarrow{wc} O^\Omega]$ be a TSSPF and let $J \subseteq I$ and $P \subseteq O$ be two subsets of input and output channels names. The TSSPF f is called strongly causal with respect to (J, P) iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : (i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J} \Rightarrow f(i)|_P{\downarrow_{t+1}} = f(i')|_P{\downarrow_{t+1}}$.

A set of TSSPFs F is called strongly causal with respect to (J, P) iff there exists a function $f \in F$ that is strongly causal with respect to (J, P). The causality complication is avoided, if causality between the inputs and outputs on the connected channels of a composition's participant is guaranteed:

Definition 4 (Composable)**.** *Two sets of TSSPFs $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ and $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ are called composable iff F_1 is strongly causal with respect to $(I_1 \cap O_2, I_2 \cap O_1)$ or F_2 is strongly causal with respect to $(I_2 \cap O_1, I_1 \cap O_2)$.*

Components communicate with each other via unidirected, typed channels established by connectors connecting component interfaces. Multiple components may read from the same channel, whereas only one component is permitted to write messages on a channel. This ensures that no merging of messages emitted from different components via the same channel is necessary. Thus the output channels of the functions of two sets of TSSPFs need to be disjoint to enable composition. The composition of two sets of TSSPFs yields a set of TSSPFs:

Definition 5 (Composition)**.** *Let $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ and $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ be two component describing and composable sets of TSSPFs with disjoint output channel sets $O_1 \cap O_2 = \emptyset$. Let $I = (I_1 \backslash O_2) \cup (I_2 \backslash O_1)$ and $O = O_1 \cup O_2$. The composition $F_1 \otimes F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ of F_1 and F_2 is defined by*

$$F_1 \otimes F_2 \stackrel{\text{def}}{=} \{f \mid \forall i \in I^\Omega : \exists f_1 \in F_1 : \exists f_2 \in F_2 : f(i) = o + p \quad \texttt{where } o = f_1((i+p)|_{I_1}), p = f_2((i+o)|_{I_2})\}$$

The composition operator is defined similar as in [12], [13], [27] with the difference that we consider the time-synchronous system model instead of the more general timed system model [5]. The composition is well defined and thus results in a component semantics describing set of TSSPFs.

Theorem 1. *If F_1 and F_2 are two component describing and composable sets of TSSPFs with disjoint output channel sets, then $F_1 \otimes F_2$ is also component describing.*

Proof. Analogous to proof of Thm. 9 in [12] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

C. Time-Synchronous Port Automata

A TSPA specifies the behavior (of parts) of an interactive system and represents a component semantics describing set of TSSPFs that is given by its semantics. TSPAs as introduced in this paper are strongly inspired by port automata as introduced in [12], I/O* automata as introduced in [27], [24], and MAA$_{ts}$ automata as defined in [22]. Port and I/O* automata consume and produce time slices of arbitrary but finitely many input messages in every transition step. In contrast, TSPAs and MAA$_{ts}$ automata consume and output at most one message per input channel in each time slice. Given the set of states and the channel types of an automaton are finite, MAA$_{ts}$ automata and the automata presented here are guaranteed to have finitely many transitions. This is not the case for I/O* and port automata since both have to define a transition for each state and each possible input communication history. Even if the type of a channel is finite, the number of communication histories (streams) of the channel's type is infinite. I/O* and MAA$_{ts}$ automata enforce causality between input and output histories by requiring initial outputs on all channels. In contrast, TSPAs do not require initial outputs. While the syntax of MAA$_{ts}$ automata treat variables explicitly, variables have to be represented implicitly in the state space of TSPAs. TSPAs can be treated as a special case of port automata as presented in [12]. Thereby the proofs of many theorems presented in the following are analog to proofs, which have already been carried out in [12]. In case we are stating an analogous theorem we refer to the appropriate corresponding proof in [12].

A TSPA consists of a set of states, an interface given by input and output channels, and transitions defining the TSPA's behavior. The interface is encoded by a port signature.

Definition 6 (Port Signature)**.** *Let $I, O \subseteq C$ be two disjoint sets of channel names (ports). A port signature is a tuple $\Sigma = (I, O)$. We denote by $C(\Sigma) \stackrel{\text{def}}{=} I \cup O$ the set of all ports in Σ. A port signature Σ is called finite iff $C(\Sigma)$ and $type(c)$ for all $c \in C(\Sigma)$ are finite.*

Let $B \subseteq C$. A *port assignment* is an element of the set $B^\rightarrow$ defined as $B^\rightarrow \stackrel{\text{def}}{=} \{a \in [B \to M] \mid \forall b \in B : a(b) \in type(b)\}$.

TSPAs must not block their environments and must be able to react to any possible well-typed input in any time unit. Therefore, a TSPA must define a reaction to every possible input for each of its states. The reactions of a TSPA are defined by its transitions. In each time unit, a TSPA performs exactly one state change by executing one transition enabled by its input and outputs exactly one message on each output channel.

Definition 7 (Time-Synchronous Port Automaton)**.** *A time-synchronous port automaton is a tuple $A = (\Sigma, S, \iota, \delta)$ where:*

- *$\Sigma = (I, O)$ is a port signature,*
- *S is a set of states,*
- *$\iota \in S$ is the initial state,*
- *$\delta \subseteq S \times C(\Sigma)^\rightarrow \times S$ is the transition relation, which is required to be reactive, i.e., $\forall s \in S : \forall i \in I^\rightarrow : \exists t \in S : \exists \theta \in C(\Sigma)^\rightarrow : (s, \theta, t) \in \delta \wedge \theta|_I = i$.*

A is called finite iff Σ and S are finite.

For convenience we sometimes write $s \xrightarrow{\theta}_\delta t$ instead of $(s, \theta, t) \in \delta$ and simply $s \xrightarrow{\theta} t$ if δ is clear from the context.

1) Execution and Behavior Semantics of TSPAs: This section formalizes the intuitive descriptions of a TSPA's behavior.

Definition 8 (Execution)**.** *Let $A = (\Sigma, S, \iota, \delta)$ be a TSPA. An execution σ of A is an infinite, alternating sequence of states and port assignments starting with the initial state ι:*

$\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ s.t. $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i} s_{i+1}$. The set of all executions of A is denoted by $execs(A)$.

Executions comprise the state changes and interactions performed by a TPSA. Abstracting from state changes allows to treat TSPAs as black boxes with hidden internal details.

Definition 9 (Behavior)**.** *Let $A = (\Sigma, S, \iota, \delta)$ be a TSPA with port signature $\Sigma = (I, O)$. The behavior of an execution $\sigma = s_0, \theta_0, s_1, \theta_1, \ldots$ of A is defined as the sequence $beh(\sigma) \stackrel{\text{def}}{=}$*

$\theta_0, \theta_1, \ldots$ *containing only port assignments. We denote by* $behs(A) \stackrel{\text{def}}{=} \bigcup_{\sigma \in execs(A)} beh(\sigma)$ *the set of all behaviors of all executions of* A*. The named communication history* h_α *induced by a behavior* $\alpha \in behs(A)$ *with* $\alpha = e_0, e_1, \ldots$ *is defined as the function* $h_\alpha \in (I \cup O)^\Omega$ *that satisfies* $h_\alpha(x).t = e_t(x)$ *for all* $x \in I \cup O$ *and* $t \in \mathbb{N}$.

Given a TSPA $A = (\Sigma, S, \iota, \delta)$ with $\Sigma = (I, O)$ and an input history $i \in I^\Omega$, we denote the set of communication histories induced by a behavior of A with input i by

$A[i] \stackrel{\text{def}}{=} \{o \in O^\Omega \mid \exists \alpha \in behs(A) : o = h_\alpha|_O \wedge h_\alpha|_I = i\}$.

2) Composition of TSPAs: As for TSSPFs, causality expresses the dependency between the inputs and outputs of a TSPA. A TSPA's output in time t must be completely determined by its input until time t. Thus it cannot change messages sent in the past and cannot predict messages it receives in the future (*cf.* pulse drivenness in [12]):

Definition 10 (Weakly Causal TSPA)**.** *A TSPA* $A = (\Sigma, S, \iota, \delta)$ *with* $\Sigma = (I, O)$ *is called weakly causal iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow A[i]{\downarrow_t} = A[i']{\downarrow_t}$.

Weak causality states that for every two inputs i, i' having a common prefix of length t and for every behavior $\alpha \in A[i]$ there is a behavior $\beta \in A[i']$ having a common prefix of length t with α. Similar as for TSSPFs, weak causality can lead to composition complications, which are avoidable analogously.

Definition 11 (Strongly Causal Modulo)**.** *Let* $A = (\Sigma, S, \iota, \delta)$ *be a TSPA with port signature* $\Sigma = (I, O)$ *and let* $J \subseteq I$ *and* $P \subseteq O$ *be two sets of input and output ports of* A*. The TSPA* A *is called strongly causal with respect to* (J, P) *iff*

$\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} : (i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J} \Rightarrow (A[i]|_P){\downarrow_{t+1}} = (A[i']|_P){\downarrow_{t+1}}$.

Intuitively, a TSPA is strongly causal with respect to (J, P), if its outputs on the channels in P until time $t + 1$ are not influenced by its inputs on the channels in J after time t.

TSPAs communicate with each other via their input and output ports. Multiple automata may read from the same channel, whereas only one automata is permitted to write messages on a channel. This ensures no merging of messages on channels emitted by different automata is necessary.

Definition 12 (Compatible Port Signatures)**.** *Two port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$ *are called compatible iff* $O_1 \cap O_2 = \emptyset$.

By composing two TSPAs, the output ports of one automaton are connected to the input ports with the same name of the other automaton. The connected input channels are hidden implicitly. The set of output channels of the new automaton is the union of the sets of the output channels of the two original TSPAs. The input channels of the new automaton are the input channels of the two automata that do not share a common name with the output channels of the other automaton.

Definition 13 (Composition of Signatures)**.** *The composition of two compatible port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$ *is defined as* $\Sigma_1 \otimes \Sigma_2 \stackrel{\text{def}}{=} (I, O)$ *where* $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ *and* $O = (O_1 \cup O_2)$.

The following defines the composition operator for TSPAs.

Definition 14 (Composition of TSPA)**.** *Let* $A_1 = (\Sigma_1, S_1, \iota_1, \delta_1)$ *and* $A_2 = (\Sigma_2, S_2, \iota_2, \delta_2)$ *be two TSPAs with compatible port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$*. The composition of* A_1 *and* A_2 *is defined as* $A_1 \otimes A_2 \stackrel{\text{def}}{=} (\Sigma_1 \otimes \Sigma_2, S_1 \times S_2, (\iota_1, \iota_2), \delta)$ *where the transition relation* δ *is defined by the following rule:*

$$\frac{s_1 \xrightarrow{\theta|_{C(\Sigma_1)}}_{\delta_1} t_1 \ \wedge \ s_2 \xrightarrow{\theta|_{C(\Sigma_2)}}_{\delta_2} t_2}{(s_1, s_2) \xrightarrow{\theta}_{\delta} (t_1, t_2)}$$

TPSAs can block each other if they simultaneously require an input emitted by another TSPA to produce the next output. Composing such TSPAs results in a structure with an empty transition relation, which is no TSPA since the requirement for reactiveness in Def. 7 implies that the transition relation of a TSPA is not empty. However, there is a sufficient condition ensuring the resulting transition relation is reactive.

Definition 15 (Composability of TSPAs)**.** *Two TSPAs* $A_1 = (\Sigma_1, S_1, \iota_1, \delta_1)$ *and* $A_2 = (\Sigma_2, S_2, \iota_2, \delta_2)$ *with port signatures* $\Sigma_1 = (I_1, O_1)$ *and* $\Sigma_2 = (I_2, O_2)$ *are called composable iff* A_1 *is strongly causal with respect to* $(I_1 \cap O_2, I_2 \cap O_1)$ *or* A_2 *is strongly causal with respect to* $(I_2 \cap O_1, I_1 \cap O_2)$.

The following theorem states that composing two composable TSPAs always results in a well-formed TSPA.

Theorem 2. *If* A_1 *and* A_2 *are composable TSPAs with compatible port signatures, then* $A_1 \otimes A_2$ *is a TSPA.*

Proof. Analogous to proof of Thm. 3 in [12] by replacing the set the function i is chosen from with $I^\rightarrow$. □

3) TSSPF semantics of TSPAs: This section defines the semantics of TSPAs by sets of TSSPFs and reveals an important relation between the composition operators: The semantics of the syntactic composition of two TSPAs A and B is equal to the composition of the semantics of the individual automata.

Definition 16 (TSSPF Semantics of a TSPA)**.** *The TSSPF semantics* $[\![A]\!]$ *of a TSPA* $A = (\Sigma, S, \iota, \delta)$ *with port signature* $\Sigma = (I, O)$ *is defined as follows:*

$[\![A]\!] \stackrel{\text{def}}{=} \{f \in [I^\Omega \xrightarrow{wc} O^\Omega] \mid \forall i \in I^\Omega : \exists \alpha \in behs(A) : i = h_\alpha|_I \wedge f(i) = h_\alpha|_O\}$

For each behavior, the semantics contain a function that maps inputs to outputs as encoded by the history induced by the behavior, *i.e.,* no behavior is lost in the semantic mapping.

Theorem 3. *Let* A *be a TSPA. For each* $\alpha \in behs(A)$ *there is a function* $f \in [\![A]\!]$ *such that* $f(h_\alpha|_I) = h_\alpha|_O$.

Proof. Analogous to proof of Thm. 11 in [12] by replacing the definition of maximality with $\forall i \in I^\Omega : i \in S|_I$. □

The semantics of TSPAs are well formed, *i.e.,* TSPAs can be used to specify component behavior because the semantics of every TSPA is component semantics describing.

Theorem 4. *The semantics $[\![A]\!]$ of a TSPA A is component semantics describing.*

Proof. Analogous to proof of Thm. 12 in [12] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

The semantics of the composition of two TSPAs is equal to the composition of their individual semantics:

Theorem 5. *For two composable TSPAs A and B with compatible signatures the following holds: $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$.*

Proof. Analogous to proof of Thm. 13 in [12] by replacing the applications of $[\![\cdot]\!]$ for PAs and $\otimes$ for SPFs by applications of the corresponding definitions for TSPAs and TSSPFs. □

An important implication of the theorem is that we can first syntactically compose the individual automata of an architecture and then perform analysis on the semantics of the automaton encoding the behavior of the whole system. This gives another basis for analysis that does not necessarily require to compose the semantics of the individual components of a system as, for example, done in [26].

IV. Semantic Differencing of Component Behavior: From TSPAs to BAs

After introducing the notations for Büchi Automata (BAs) used in this paper, this section presents a theorem stating that there is a nondeterministic BA for each finite TSPA that accepts exactly the behaviors of the TSPA. Afterwards, it is shown that refinement checking and semantic difference witness generation for TSPAs can be reduced to language inclusion checking and counterexample generation for BAs.

A. Büchi Automata

Büchi automata [2] are a variant of finite automata that are acceptors for infinite words and thus induce languages consisting of infinite words. They are well known and much used in the model checking domain. Infinite words over an alphabet Π are infinite sequences of symbols in Π. The set of all infinite words over an alphabet Π is denoted by Π^ω.

Definition 17 (Büchi Automaton)**.** *A BA is a tuple (Π, Q, I, F, δ) where Π is a finite alphabet, Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of accepting states, and $\delta \subseteq Q \times \Pi \times Q$ is the transition relation.*

Let $\mathcal{B} = (\Pi, Q, I, F, \delta)$ be a BA. A run of $\mathcal{B}$ on a word $w = \sigma_1, \sigma_2 ... \in \Pi^\omega$ starting in a state $q_0 \in Q$ is an infinite sequence $q_0, q_1, ...$ such that $q_{j-1} \xrightarrow{\sigma_j}_\delta q_j$ for all $j > 0$. A run $q_0, q_1, ...$ is accepting if $q_0 \in I$ and $q_i \in F$ for infinitely many $i > 0$. The accepted language of $\mathcal{B}$ is defined as $\mathcal{L}(\mathcal{B}) \stackrel{\text{def}}{=} \{w \in \Pi^\omega \mid \text{there exists an accepting run for } w \text{ in } \mathcal{B}\}$.

Checking language inclusion between two Büchi automata is PSPACE-complete [18], though decidable. Although the computational complexity is large, several approaches for checking language inclusion and counterexample (diff witness) generation have been implemented and produce promising results in practice [2]. In the next section, we present a translation from finite TSPAs to BAs and thereby reduce semantic differencing and refinement checking for finite TSPAs to the language inclusion problem for Büchi automata.

B. From TSPAs to BAs

We consider semantic differencing and refinement checking for architectures where the individual components have a finite state space, communicate over finitely many communication channels, and where the types of messages emitted via component interfaces are finite. There exists a nondeterministic BA for each finite TSPA that accepts exactly the TSPA's behaviors.

Theorem 6. *For any finite TSPA A there exists a BA $\mathcal{B}$ such that $behs(A) = \mathcal{L}(\mathcal{B})$.*

Proof. Let $A = (\Sigma, S, \iota, \delta)$ with $\Sigma = (I, O)$ be a finite TSPA. Let $\mathcal{B} = (\Pi, S, \{\iota\}, S, \Delta)$ where

- $\Pi = [(I \cup O) \to \bigcup_{c \in I \cup O} type(c)]$ and
- $\Delta = \{(s, l, t) \mid \exists \theta \in C(\Sigma)^\to : (s, \theta, t) \in \delta \wedge \theta = l\}$.

The TSPA A is finite. Thus, S, I, O, and $\bigcup_{c \in I \cup O} type(c)$ are finite. As therefore Π and Δ are finite, $\mathcal{B}$ is a well-defined BA. It remains to show that $behs(A) = \mathcal{L}(\mathcal{B})$.

$\subseteq$: Let $s_0, \theta_1, s_1, \theta_2, s_2, ... \in execs(A)$ be an execution of A. By definition of execution $s_{j-1} \xrightarrow{\theta_j} s_j$ for all $j > 0$ and $s_0 = \iota$. By definition of $\mathcal{B}$ we have that $(s_{j-1}, \theta_j, s_j) \in \Delta$ for all $j > 0$. Thus, $s_0, s_1, s_2, ...$ is a run of $\mathcal{B}$ on the word $\theta_1, \theta_2, ...$ Since all states $s \in S$ are accepting, the run is accepting. Thus, $beh(s_0, \theta_1, s_1, \theta_2, s_2, ...) = \theta_1, \theta_2, ... \in \mathcal{L}(\mathcal{B})$.

$\supseteq$: Assume that $\sigma = \sigma_1, \sigma_2, \sigma_3, ... \in \mathcal{L}(\mathcal{B})$ and let $q_0, q_1, q_2, ...$ be an accepting run of $\mathcal{B}$ on σ. By definition of run we have $q_{j-1} \xrightarrow{\sigma_j} q_j$ for all $j > 0$. Thus, by definition of Δ we have that there are $\theta_j \in C(\Sigma)^\to$ with $(q_{i-1}, \theta_j, q_j) \in \delta$ and $\theta_j = \sigma_j$ for each $j > 0$. Thus $\tau = q_0, \theta_1, q_1, \theta_2, ...$ is an execution of A. Therefore, by definition of behavior we have that $beh(\tau) = \sigma_1, \sigma_2, ... \in behs(A)$ is a behavior of A. □

C. Semantic Differencing for Component Behavior

The semantics of components are defined as sets of TSSPFs. We denote the semantics of a component c by $[\![c]\!]$. Each function $f \in [\![c]\!] \setminus [\![c']\!]$ in the semantics of one component c that is no member of the semantics of another component c' is a representative for the difference between the components' semantics. However, such a representative defines an output for each possible component input, even if the semantic difference is only given by a single input/output pair. Thus, such a TSSPF does not effectively reveal the differences between the component semantics. In contrary, the exact input/output pairs for which there is a function in the semantics of one component that maps the input to the output and for which there is no function in the semantics of the other component mapping the input to the output clearly reveals a difference. If two components have different interfaces, *i.e.,* they read and write from and to different channels, each input/output pair of

the first component indicates a difference to the semantics of the other component. However, if the components have channels of the same types one can easily avoid this problem by channel renaming and hiding [3]. Thus, we define the semantic difference for components having the same interfaces, only.

Definition 18 (Diff Witness)**.** *Let $F_1, F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ be two sets of TSSPFs. A diff witness distinguishing F_1 from F_2 is a communication history $w \in (I \cup O)^\Omega$ satisfying*

$\exists f_1 \in F_1 : f_1(w|_I) = w|_O \wedge \forall f_2 \in F_2 : f_2(w|_I) \neq w|_O$.

We denote by $\Delta(F_1, F_2)$ the set of all diff witnesses distinguishing F_1 from F_2.

A set of diff witnesses may be finite but is typically infinite. The following theorem reveals the relation between the differences of the behaviors and of the semantics of TSPAs.

Theorem 7. *Let $A_1 = (\Sigma, S_1, \iota_1, \delta_1)$ and $A_2 = (\Sigma, S_2, \iota_2, \delta_2)$ with $\Sigma = (I, O)$ be two TSPAs and let $w \in (I \cup O)^\Omega$ be a communication history. The following holds:*

$w \in \Delta([\![A_1]\!], [\![A_2]\!]) \Leftrightarrow$
$\exists \alpha \in behs(A_1) : w = h_\alpha \wedge \alpha \notin behs(A_2)$.

Proof. Let A_1, A_2, and w be given as above.

$\Rightarrow$: Assume $w \in \Delta([\![A_1]\!], [\![A_2]\!])$ is a diff witness. By definition of Δ, we have that there is a function $f_1 \in [\![A_1]\!]$ such that $f_1(w|_I) = w|_O$ and $f(w|_I) \neq w|_O$ for all $f \in [\![A_2]\!]$. In the following let f_1 be such a function that satisfies the above. By definition of $[\![\cdot]\!]$ we have that $\forall i \in I^\Omega : \exists \alpha \in behs(A_1) : i = h_\alpha|_I \wedge f_1(i) = h_\alpha|_O$. When substituting $w|_I$ for i, we get $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge f_1(w|_I) = h_\alpha|_O$. Since $f_1(w|_I) = w|_O$ we can substitute $w|_O$ for $f_1(w|_I)$ and obtain $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge w|_O = h_\alpha|_O$, which is equivalent to $\exists \alpha \in behs(A_1) : w = h_\alpha$.

In the following, let such an α with $w = h_\alpha$ be given. It remains to show $\alpha \notin behs(A_2)$. Towards a contradiction we assume $\alpha \in behs(A_2)$. By Thm. 3 we get there is a function $g \in [\![A_2]\!]$ such that $g(h_\alpha|_I) = h_\alpha|_O$. By definition of α we have $w = h_\alpha$ and thus $g(w|_I) = w|_O$. But since $w \in \Delta([\![A_1]\!], [\![A_2]\!])$, it holds that $\forall f \in [\![A_2]\!] : f(w|_I) \neq w|_O$. Substituting g for f yields a contradiction.

$\Leftarrow$: Assume there is an $\alpha \in behs(A_1)$ such that $w = h_\alpha$ and $\alpha \notin behs(A_2)$. Using Thm. 3 we get there is a function $f \in [\![A_1]\!]$ such that $f(h_\alpha|_I) = h_\alpha|_O$. By definition of w we have that $w = h_\alpha$ and thus obtain by substitution that $f(w|_I) = w|_O$. Thus there is a function $f \in [\![A_1]\!]$ such that $f(w|_I) = w|_O$. It remains to show that $g(w|_I) \neq w|_O$ for all $g \in [\![A_2]\!]$. Towards a contradiction we assume that there is a function $g \in [\![A_2]\!]$ such that $g(w|_I) = w|_O$. By definition of $[\![\cdot]\!]$ we get that $\forall i \in I^\Omega : \exists \beta \in behs(A_2) : i = h_\beta|_I \wedge g(i) = h_\beta|_O$. Substituting $w|_I$ for i we obtain $\exists \beta \in behs(A_2) : w|_I = h_\beta|_I \wedge g(w|_I) = h_\beta|_O$. Since by assumption $w|_I = h_\alpha|_I$ and $g(w|_I) = w|_O$ by definition of g, this is equivalent to $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge w|_O = h_\beta|_O$. By assumption we have $w = h_\alpha$ and thus obtain via substitution $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge h_\alpha|_O = h_\beta|_O$, which is equivalent to $\exists \beta \in behs(A_2) : h_\alpha = h_\beta$. Using the definitions of h_α and h_β, this is equivalent to $\exists \beta \in behs(A_2) : \alpha = \beta$, which is equivalent to $\alpha \in behs(A_2)$ and contradicts the assumption. □

In the previous section, we presented a translation from finite TSPAs to BAs. Each word accepted by a BA resulting from such a translation corresponds to a behavior of the input TSPA. Existing algorithms for checking language inclusion and counterexample generation for BAs can thus be used for refinement checking and diff witness generation of architectures as defined above: Given two TSPAs A_1 and A_2 we use the translation defined in proof of Thm. 6 to obtain two Büchi automata $\mathcal{B}_1$ and $\mathcal{B}_2$ such that $\mathcal{L}(\mathcal{B}_1) = behs(A_1)$ and $\mathcal{L}(\mathcal{B}_2) = behs(A_2)$. Using Thm. 7 and Thm. 6 we can transform a word accepted by $\mathcal{B}_1$ that is not accepted by $\mathcal{B}_2$ to a corresponding diff witness that semantically distinguishes the automata A_1 and A_2. By definition, if $\mathcal{L}(\mathcal{B}_1) = \mathcal{L}(\mathcal{B}_2)$ then the two TSPAs A_1 and A_2 are equivalent and if $\mathcal{L}(\mathcal{B}_1) \subseteq \mathcal{L}(\mathcal{B}_2)$ then the automaton A_1 refines the automaton A_2.

V. Implementation and Evaluation

This section recapitulates the MontiArcAutomaton ADL [23], [25], presents the application of refinement checking to its models and evaluates our approach.

A. The MontiArcAutomaton ADL

The MontiArcAutomaton ADL [23], [25] comprises the modeling elements common to many popular component & connector ADLs [20], *i.e.,* hierarchical components with interfaces of typed, directed ports and unidirectional connectors (typed FIFO channels) exchanging messages between these ports. The components are black-boxes and either atomic or composed: atomic components yield behavior descriptions in form of embedded automata (following the I/O$^\omega$ [27] paradigm) or in form of Java implementations. The behavior of composed components solely emerges from the interaction of their subcomponents. Components are scheduled by a global clock and perform cycles of 1.) read all messages on incoming ports; 2.) compute behavior (which might entail invoking subcomponents); 3.) produce a single message to each outgoing port. Each computation consumes a time slice, *i.e.,* the output for messages received at the global clock's i-th tick is produced at its i+1-th tick earliest. The MontiArcAutomaton ADL also distinguishes between component types and their instances, supports component type inheritance, generic type parameters for components (*e.g.,* to be used with generic port types), and constructor-like configuration of these instances.

The MontiArcAutomaton ADL is a textual modeling language implemented with the MontiCore [17] language workbench. The textual representation of the composed component type `Elevator` is illustrated in Listing 1. It begins with the keyword "component", followed by the component type's name and a body delimited by curly brackets (l. 1). The body contains an interface of typed ports (ll. 2-5), declares three subcomponents (ll. 7-9), and multiple connectors (ll. 11-13). The subcomponent declarations reference component types imported from artifacts (such as `Control`).

MontiArcAutomaton

```
component Elevator {
  port in Bool req1, in Bool at1,
       // ... further ports ...
       out Bool open, out Bool close,
       out Clear clear;

  component Control ctrl; // named
  component Motor m;      // subcomponent
  component Door d;       // instances

  connect req1 -> control.req1;
  // ... further connectors ...
  connect control.clear -> clear;
}
```

Listing 1. Textual representation of the component `Elevator`.

B. Semantic Differencing of MAA Components

The implementation comprises a translation from MontiArcAutomaton architectures to semantically equivalent TSPAs. TSPAs are only handled internally as representatives for sets of TSSPFs modeling component semantics and are not explicitly modeled by component developers. Each atomic component directly translates to a TSPA. The TSPA of a composed component is computed by composing the TSPAs of its subcomponents according to the architectural configuration defined by the composed component's connectors. The implementation further consists of a translation from TSPAs to BAs and generators that produce models in the "BA format", which is the input format of the tool RABIT [2]. In case a BA does not refine another BA, RABIT provides a counterexample serving as a concrete disproof for refinement. The counterexamples can be translated back to diff witnesses. Using the tool chain described above enables automated refinement checking and diff witness generation for MontiArcAutomaton architectures.

C. Evaluation

We evaluated the approach to semantic differencing with six MontiArcAutomaton architectures previously used for evaluation in [26]. We specifically chose these architectures for evaluation since the approach presented in [26] failed for some specifications, which we considered to be challenging, and to enable comparability. The architectures were slightly modified for this evaluation to resolve technical MontiArcAutomaton version compatibility issues. We reused the completion strategies [26] for completing the automata implementations of the architectures' atomic components.

The first architecture is given by an implementation of an elevator control system (ECS) (cf. Sec. II). It comprises 3 composed and 5 atomic components. The second example consists of four variants of a mobile robot. We only report on the evaluation of the most challenging variant. This variant comprises 4 components in total whereof 3 components are atomic. The last architecture implements a pump station consisting of 3 composed and 10 atomic components.

In [26], for each of the architectures three specification checks are executed: it is checked whether the semantics of a component is equal to itself, whether a component refines a component with the same interfaces that implements arbitrary behavior, *i.e.,* all possible behaviors, and whether the semantics of a component are equal to the semantics of a component implementing arbitrary behavior. We performed the same checks on a computer with 3.0 GHz Intel Core i7 CPU, 16 GB Ram, Windows 10, and RABIT 2.4 using our translation from MontiArcAutomaton architectures to BAs and the language inclusion checking tool RABIT [2] (cf Sec. V-B).

TABLE I
TIME FOR REFINEMENT CHECKING AND DIFF WITNESS CALCULATION.

	$\Delta([\![\cdot]\!],[\![\cdot]\!])$	$\Delta([\![\cdot]\!],Chaos)$	$\Delta(Chaos,[\![\cdot]\!])$
Floors	62ms	526ms	909ms
Elevator	75ms	2510ms	6064ms
ECS	463ms	7166ms	16537ms
SensorReading	94ms	764ms	1558ms
Controller	15ms	17ms	43ms
Pumpstation	119ms	334ms	486ms
MobileRobot	52ms	75ms	106ms

Table I summarizes the computation times of RABIT given the BAs resulting from the transformation as input. For component ECS, for instance, checking whether it refines itself took 463ms, checking refinement with arbitrary behavior took 7166ms, and calculating a diff witness distinguishing the component from arbitrary behavior took 16537ms. Table II depicts the sizes of the automata resulting from the translations. For component ECS, for instance, the TSPA and the BA resulting from the transformation have 746 states and 98496 transitions. RABIT reported the tool has reduced the BA to 8 states and 1728 transitions after internal preprocessing. For every component we modeled arbitrary behavior (Chaos) with a TSPA consisting of one state and a transition for every possible component input/output combination. The TSPA and the BA modeling arbitrary behavior for component ECS, for instance, comprise 472392 transitions (*cf.* Table II). In contrast to the translation from MontiArcAutomaton architectures to the model checker Mona [26], our implementation succeeded for all example architectures. The longest computation time of our evaluation (16537ms, *cf.* Table I) resulted from semantic differencing arbitrary behavior with the ECS component. We conclude that our translation provides promising results. Nevertheless the evaluation was only performed on a few specific architectures. Thus the results are not generalizable to all possible architectures: the time needed by our tool may vary strongly from system to system.

VI. DISCUSSION

If the semantics domain of an ADL is overly general, undecidability of the underlying mathematical problems renders automated formal verification impossible. Then, architecture properties have to be proven manually, which is too expensive to be carried out in continuous architecture modeling and thus hinders employing agile development in architecture modeling projects: little changes to requirements or implementations can

TABLE II
THE NUMBERS OF STATES AND TRANSITIONS OF THE TSPAS TRANSLATED FROM THE ARCHITECTURES AND OF THE GENERATED BAS.

	TSPA/BA		BA AP		Chaos
	#states	#trans.	#states	#trans.	#trans.
Floors	32	1024	32	1024	23328
Elevator	34	10206	1	729	236196
ECS	746	98496	8	1728	472392
SensorReading	2	1296	2	1296	69984
Controller	1	9	1	9	108
Pumpstation	6	3888	4	2592	17496
MobileRobot	150	2700	12	216	1152

entail changing many manually performed proofs. In contrast, where automated formal verification is possible, sound and complete proofs can be generated automatically, supporting agile implementation evolution.

FOCUS is a comprehensive framework that supports specifying the observable input/output behavior of interactive systems. Its complexity requires carrying out proofs for system behavior verification manually. FOCUS provides various constructs for describing the semantics of distributed systems [24]. Examples are relations, set-based functions, sets of functions, assumption/guarantee predicates, or state-based representations. As identified in [24], the most fine-grained domain for describing the semantics of distributed systems using FOCUS are sets of SPFs. Independent of the style, specifications can describe timed or untimed behavior. Untimed behavior only considers the causality regarding the order of inputs and outputs. Timed specifications additionally concern causality regarding the passage of time. Many requirements are not only concerned with the order of messages but also state requirements with respect to passage of time. Thus, we employ a variant of the timed subset of FOCUS and thereby use sets of TSSPFs as semantics domain [24], [27].

Our approach is limited to systems where the data types' domains are finite and is restricted to the time-synchronous model of computation. However, our system model fits well into the kinds of systems developed for embedded systems such as automotive or robotics applications. Thus, our results enable fully automated tool support for many systems in such domains. Emphasizing that our approach cannot be generalized to the timed model of FOCUS as, for example, used in [12], is important: Timed SPFs (*cf.* [12], [24], [27]), for instance, are too general to be applicable to our approach. A timed SPF processes infinite sequences of finite sequences (of arbitrary lengths) of messages. Each of the finite sequences represents a finite stream of messages received or sent by a component in a single time unit. In contrast, TSSPFs only process single messages per time unit. The set of finite streams of messages over a non-empty finite data type is already infinite. Thus, for each time unit, a timed SPF needs to define a possible behavior for infinitely many tuples of input streams, whereas a TSSPF needs to define a reaction for all possible tuples of input messages, which are finitely many if the messages' data types are finite. From a practical viewpoint it is rarely required to specify the reaction in a time unit in response to the receipt of an arbitrary number of messages. Usually it either requires to handle single messages (TSSPFs) or sequences of messages where the length of the sequence is bound by an arbitrary but fixed natural number. The latter can be reduced to the former by introducing lists of fixed length as message types.

The underlying theoretical problem for semantic differencing used in our approach is language inclusion checking between Büchi automata. Its complexity can be considered as another limitation of our approach. However, our main focus is not verifying a system's properties (*e.g.,* refinement or semantic differencing) within seconds, which is most often already rendered impossible due to the complex nature of the safety critical system under development. We believe that nonetheless the possibility to apply formal fully automated verification (*e.g.,* over night) greatly facilitates continuous architecture modeling.

VII. RELATED WORK

Studies on the verification techniques of ADLs have been conducted, *e.g.,* in [30] and [33]. The study in [33] surveys verification techniques supported by ADLs with formal semantics, the translation of architectures to inputs for model checkers, and tool support as well as usability, scalability, and expressiveness. As supported by our approach, the study states that architecture verification for practical applications requires tool-support and automation. The study in [30] compares different verification tools and applies them to various ADLs. All architectures are transformed into intermediate labeled transition systems before the verification tools are applied, hampering the direct comparison with our approach.

The following surveys concrete approaches for formally analyzing hierarchical architecture descriptions. AutoFOCUS 3 [14] is a tool for the development of reactive embedded systems that also bases its semantics on FOCUS [5]. Although AutoFOCUS 3 supports model checking architectures against LTL and CTL formulas that specify properties concerning component behavior [6], we are not aware of a fully automated refinement checking method for AutoFOCUS 3. The π-ADL supports statistical model checking for verifying dynamic software architectures against DynBLTL properties [7]. To this effect, a statistical model of finite system executions is built and the probability of satisfying a property within a confidential bound is calculated. This approach is particularly tailored to dynamic architectures and is only concerned with finite traces. In contrast, our approach deals with infinite traces, static architectures, and full certainty. Refinement of architectures specified with timed I/O is described in [16]. Similar to behaviors of TSPAs, the semantics of a timed automaton is given by a set of traces. Refinement between timed I/O automata is defined similar as in our approach by trace inclusion. However, timed I/O automata are only marked with one message per transition and composition is defined differently. Further, the timing concept of I/O automata is more powerful and complicated than the one

of our approach [12]. A game-based extension of the timed I/O automaton model enabling tool supported refinement checking has been proposed in [8]. Another approach to automated refinement checking based on the time synchronous frame of FOCUS is described in [22], [26]. This approach is based on a relational semantics domain where the semantics of a component is given as a relation between the component's possible inputs and outputs. In contrast, our approach uses a more fine grained [24] semantics domain consisting of sets of functions. Refinement checking in [22], [26] relies on translating component semantics into WS1S and is implemented using the model checker Mona [10]. The approach suffers from the tool's high computational complexity, which is grounded in the non-elementary complexity of solving W1S1 problems. In contrast, we define a translation to Büchi automata and thereby obtain a PSPACE-complete complexity for refinement checking. While the relational approach is based on analyzing the result from composing the semantics of the individual components of a system, our approach first syntactically composes the individual components and bases analysis on the semantics of the compound.

VIII. Conclusion

We have presented an implementation of stepwise refinement for ADLs using a subset of the FOCUS semantics for distributed systems. This subset consists of time-synchronous stream processing functions, and hence the corresponding software architecture models, can be translated to a variant of port automata. Via a transformation from port automata to Büchi automata, we can reduce component refinement to language inclusion. As the evaluation has shown, the fully automated implementation supports checking refinement for MontiArcAutomaton architecture models in reasonable time. While this might be improved further, we believe our approach facilitates continuous architecture modeling.

References

[1] RABIT Tool Homepage, 2016. http://http://www.languageinclusion.org/ [accessed 2016-12-31].

[2] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. Advanced Ramsey-Based Büchi Automata Inclusion Testing. In *International Conference on Concurrency Theory, CONCUR 2011*, 2011.

[3] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 2010.

[4] Manfred Broy and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TU Munich, 1992.

[5] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[6] Alarico Campetelli, Florian Hölzl, and Philipp Neubeck. User-friendly Model Checking Integration in Model-based Development. In *International Conference on Computer Applications in Industry and Engineering*, 2011.

[7] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flávio Oquendo, Thaís Batista, and Axel Legay. Statistical Model Checking of Dynamic Software Architectures. In *European Conference on Software Architecture*, 2016.

[8] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O Automata: A Complete Specification Theory for Real-time Systems. In *ACM International Conference on Hybrid Systems: Computation and Control*, 2010.

[9] Vincent Debruyne, Françoise Simonot-Lion, and Yvon Trinquet. EAST-ADL - An architecture description language. In *Architecture Description Languages*. Springer, 2005.

[10] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In *Computer-Aided Verification*, 1998.

[11] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[12] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical report, TU Munich, 1995.

[13] Radu Grosu, Ketil Stølen, and Manfred Broy. A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing, 1997.

[14] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, 2007.

[15] Bengt Jonsson. A Fully Abstract Trace Model for Dataflow and Asynchronous Networks. *Distributed Computing*, 1994.

[16] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS 2003)*, 2003.

[17] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008.

[18] Orna Kupferman and Moshe Y. Vardi. Verification of Fair Transition Systems. In *International Conference on Computer Aided Verification*, 1996.

[19] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 2013.

[20] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[21] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/ [accessed 2017-01-13].

[22] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Shaker Verlag, 2014.

[23] 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)*, 2015.

[24] 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.

[25] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014.

[26] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*, 2016.

[27] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, TU Munich, 1996.

[28] Christian Schlegel, Andreas Steck, and Alex Lotz. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. In *Introduction to Modern Robotics*. iConcept Press, 2011.

[29] Frank Strobl and Alexander Wisspeintner. Specification of an Elevator Control System. Technical report, TU Munich, 1999.

[30] Jeffrey J.P. Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 2000.

[31] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 2000.

[32] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2013.

[33] Pengcheng Zhang, Henry Muccini, and Bixin Li. A Classification and Comparison of Model Checking Software Architecture Techniques. *Journal of Systems and Software*, 2010.

[BKRW18a] A. Butting, O. Kautz, B. Rumpe, A. Wortmann:
Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution.
In: Journal of Systems and Software, 2018.
www.se-rwth.de/publications/

Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution

Arvid Butting[a], Oliver Kautz[a,*], Bernhard Rumpe[a], Andreas Wortmann[a]

[a]*Software Engineering, RWTH Aachen University, Aachen, Germany, www.se-rwth.de*

Abstract

Understanding the semantic differences of continuously evolving system architectures by semantic analyses facilitates engineers during evolution analysis in understanding the impact of the syntactical changes between two architecture versions. To enable effective semantic differencing usable in practice, this requires means to fully automatically check whether one version of a system admits behaviors that are not possible in another version. Previous work produced very general system models for message-driven time-synchronous (MDTS) systems that impede fully automated semantic differencing but very adequately describe such systems from a black-box viewpoint abstracting from hidden internal component behavior. This paper presents a system model for MDTS systems from a white-box viewpoint (assuming component implementation availability) and presents a sound and complete method for semantic differencing of finite MDTS system architectures. This method relies on representing (sub-)architectures as channel automata and a reduction from the semantic differencing problem for such automata to the language inclusion problem for Büchi automata. The system model perfectly captures the logical basics of MDTS systems from a white-box viewpoint and the method enables to fully automatically calculate semantic differences between two finite MDTS systems on push-button basis, yields witnesses, and ultimately facilitates semantic evolution analysis of such systems.

Keywords: Component Software Engineering, Semantics, Automata, Refinement, Semantic Differencing, Evolution Analysis

1. Introduction

Component-based software engineering [30] promises improving software development through reuse of independently developed and validated off-the-shelf building blocks with stable interfaces. These building blocks usually are implemented in general-purpose programming languages (GPLs), Hence, they are subject to the conceptual gap between the problem domains and solution domains of discourse, which arises from addressing problem domain challenges with programming language complexities [14].

Model-driven development (MDD) [44] aims at reducing this gap by lifting domain-specific, abstract, models to primary development artifacts. Such models can leverage domain-specific vocabulary to be better comprehensible as well as more abstract and hence are better suited towards analysis and transformation than GPL programs. Software engineering also applies MDD to itself to facilitate addressing its challenges. Consequently, modeling languages for various challenges in software engineering, such as database manipulation languages, build process description languages, and architecture description languages have been developed.

Architecture description languages (ADLs) [29] leverage the potential of model-driven development [44] for the description of software architectures. In many domains, knowing the precise semantics of models is crucial due to safety concerns, but current architecture modeling processes, such as MDA [31] do not take these into account. Stepwise refinement [5, 6] is a software engineering methodology for continuous architecture modeling based on controlled evolution and progressive improvement of components: each subsequent version of a component model must adhere to properties already proven for its predecessors. To this effect, checking whether successor component versions *refine* their predecessors in terms of observable input/output behavior is crucial.

Similar to UML [32], the specific semantics of many ADL details are encoded in their infrastructures and tools only. Where fully detailed denotational or operational semantics are available, such as with Focus [7], these are usually too complex for fully automated refinement checking and typically require to (partially) manually prove refinement between two component versions. This impedes stepwise refinement so severely that it becomes a "highly idealistic" [5] idea. However, enabling automatic stepwise refinement for software architecture models would greatly facilitate development in domains where component adherence to certain properties is crucial. With automated methods, manual proofs become redundant. This enables users who are no experts in formal methods to prove or disproof refinement between architecture versions. As programmers are rarely experts in formal methods, this opens the possibility to apply stepwise refinement methodologies to a broader user range. In case an architecture is no refinement of another,

*Corresponding author
Email addresses: butting@se-rwth.de (Arvid Butting), kautz@se-rwth.de (Oliver Kautz), rumpe@se-rwth.de (Bernhard Rumpe), wortmann@se-rwth.de (Andreas Wortmann)

the method presented in this paper fully automatically calculates a behavior that is possible in the one architecture but not in the other. This behavior serves as witness and is a concrete disproof for refinement. Software engineers can use the witness as evidence for efficiently identifying the syntactic elements in the architecture's implementation that cause non-refinement.

In [9], we identified a subset of the Focus [7] semantics for time-synchronous, distributed, interactive systems that is powerful enough to model complex and realistic systems and is adaptable to enable fully automated refinement checking between components. Based on this, [9] describes an approach to transform software component models into a variant of port automata [16], compose these syntactically, and translate the results into Büchi automata, where their refinement can be checked through language inclusion [23]. This approach is realized with the MontiArcAutomaton component & connector ADL [35, 37] and the RABIT [2, 3] tool for fully automated language inclusion checking between Büchi automata. It enables modeling software architectures with powerful ADLs and checking refinement on a push-button basis. To this effect, the contributions of [9] are:

- A formulation of the semantics domain of time-synchronous [7] stream processing functions (TSSPFs) inspired by the notion of stream processing function [36].
- A variant of port automata: time-synchronous port automata (TSPA) [16] with operational semantics based on execution traces and denotational semantics based on sets of TSSPFs.
- A semantically compositional syntactic composition operator for TSPAs: The semantics of the syntactic composition of two TSPAs is equal to the composition of the semantics of the individual TSPAs.
- A transformation from finite TSPAs to Büchi automata.
- A proof showing the operational semantics of a finite TSPA and the language accepted by the Büchi automaton resulting from such a transformation coincide.
- The result that refinement checking and disproof generation in form of semantic difference witnesses for software architectures where components can be mapped to finite TSPAs can be reduced to language inclusion checking and counterexample generation for Büchi automata.
- An implementation based on the MontiArcAutomaton component & connector ADL [35, 37] and RABIT [2, 3].

In this paper, we enhance and extend the previous approach to achieve practical efficiency improvements and technical enhancements of the underlying formal system model. To this effect, this paper's additional contributions are:

- Time-synchronous channel automata (TSCAs): an improved variant of TSPAs that enables defining an associative and commutative syntactic composition operator, while retaining previous results regarding the relation between the system models and compositionality.
- The previous composition operator for TSPAs (cf. [9]) is neither associative nor commutative. Using the commutativity and associativity of the TSCA composition operator enables to define an intuitive notion of system architecture, which is not possible with the TSPA composition operator.
- A method for trimming finite TSCAs to reduce complexity of analyses.
- A method for composing finite TSCAs such that the compound does not contain any unproductive states to mitigate state explosions.
- The identification of a subclass of finite non-deterministic TSCAs, which is a proper superset of deterministic TSPAs, where semantic differencing is possible in polynomial time.
- The insight that the Büchi automata resulting from transforming TSCAs are always "weak" and therefore enable the application of efficient algorithms enabling, for instance, easy complementation or minimization.
- A notion of system architecture based on a white-box viewpoint on message-driven time-synchronous (MDTS) systems and the previously developed theory. The associativity and commutativity of the composition operator for TSCAs is important for the notion of system architecture to be well defined. The system architecture definition as introduced in this paper is not possible with TSPAs as introduced in [9] because TSPAs do not have a commutative and associative composition operator.
- A method for mitigating the state explosion problem during semantic differencing of finite system architectures, which is especially useful during continuous architecting when it comes to understanding the semantic differences between two successor versions. The method not only relies on trimming but also on iteratively applying refinement checking to smaller sub-architectures.
- An extended evaluation including an additional example and an improved composition method that combines composition with trimming.

This paper further contains many additional examples that increase comprehensibility and illustrate this paper's approach. The resulting fully automatic analysis technique for comparisons of TSCAs greatly supports continuously evolving projects where the overall architecture changes frequently. It also greatly facilitates analyzing the semantic differences between products of a product line architecture where the individual products are syntactically only slightly different.

1.1. Paper Structure and Overview

Section 2 sketches the idea of stepwise refinement. To this effect, it presents two architecture models, the elevator control system presented and evaluated in [9] as well as a more compact architecture serving as running example throughout this paper.

Subsequently, Section 3 presents the Focus subset used as semantics domain from a black-box viewpoint (as functions). This paper's approach is applicable to finite systems where it is possible to describe the system's semantics with the system model described in this section. It is argued that the system model is adequate for describing architectures while abstracting from hidden internal details, but hiding internal details hampers automated analyses.

This motivates Section 4, which describes a new system model that represents components from a white-box perspective (as automata). The automata model is compatible to the function model of the previous section and explicitly captures internal component details.

Afterwards, Section 5 presents automated semantic differencing based on the latter system model (automata). We obtain a more efficient semantic differencing method as described in previous work. The compatibility of the system models ensures the results equally apply to both semantic domains. However, this paper's approach is only applicable if component implementations are available and can be transformed to the automata introduced in Section 4.

Section 6 presents the implementation of our approach with MontiArcAutomaton and RABIT and evaluates its applicability. Section 7 discusses observations and Section 8 highlights related work before Section 9 concludes. The appendix describes examples used throughout the paper in more detail.

2. Examples

This section presents two example architectures for stepwise refinement. The first example illustrates the benefits of our approach on an elevator control system (Section 2.1) as presented in [9]. The second example describes a distributed Modulo-8-Counter (Section 2.2), which is used as running example throughout the remainder of this paper. While the former is suited to comprehending the benefits of stepwise refinement intuitively, the latter is compact enough to be discussed in details in the remainder.

2.1. An Elevator Control System

Consider the model-driven development of an elevator control system (ECS) as presented in [42]. The `ECS` depicted in Figure 1 comprises two hierarchically composed components representing the three floors the elevator serves (component `Floors`) and the elevator cabin (component `Elevator`). Whenever a button on a floor (indicated, for example, by a message on the incoming port `btn1`) is pressed, the `ECS` should activate the light (by sending a message via outgoing port `led1`) on the corresponding floor and instruct the elevator cabin to visit that floor. The control logic of the elevator is modeled via a statechart variant embedded into the `Elevator`'s subcomponent `Control`. This component receives messages upon arriving at a specific floor (*e.g.,* via incoming port `at1`) and sends messages to `Door` and `Motor` to operate its door and to move between the floors. The latter two embed models of compact action languages to describe their respective behavior.

For this version of the `ECS`, the software architects have proven that certain properties hold (*e.g.,* that it cannot produce blocking situations). Now they aim to replace the `Elevator` component with a smarter version that reacts only to elevator requests on a floor if there is no such request yet. To this effect, the company employs stepwise refinement to avoid proving the properties of `Elevator` again for its successor version `SmartElevator`. Therefore, the behavior descriptions of all subcomponents are translated into TSCAs. For composed components, the behavior descriptions of their subcomponents are translated also and merged iteratively. This ultimately eliminates all hierarchy levels but the last. The result of this transformation is depicted in Figure 2, where the behavior descriptions of all three subcomponents have been transformed accordingly and merged into a single TSCA. The same is performed for the improved `SmartElevator` component before both are transformed into weak non-deterministic Büchi automata as presented in Section 6.

Using this transformation reduces semantic component refinement to language inclusion on Büchi automata and can be solved automatically, for instance, by using the tool RABIT. Hence, with this infrastructure in place, the company now can fully automated ensure whether the `SmartElevator`, and its potential successors, actually refine their predecessors or require further adjustment. Where refinement is refuted, difference witnessing input/output pairs are produced. This automation of stepwise refinement can increase the pace of each refinement step and, hence, overall development efficiency.

2.2. A Modulo-8 Counter

This example presents a modulo-8 counter inspired by the model presented in [15] as demonstration of stepwise refinement along the depth of composition layers. The modulo-8 counter outputs the binary representation of a number n between 0 and 7, which can be incremented ($(n + 1)$ % 8) or reseted ($n = 0$). The initial value of n is 0. The modulo-8 counter is modeled as the MontiArcAutomaton component `Mod8Counter` depicted in Figure 3 (a). The component has two incoming ports and three outgoing ports of the data type Boolean. In the initial definition, only the behavior of the outermost component `Mod8Counter` is specified. The valuations of the outgoing ports $\mathtt{x}_2$, $\mathtt{x}_1$, and $\mathtt{x}_0$ are equal to the Boolean representations of the variables in the binary representation of n (*i.e.,* $n = x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0$). Upon receiving *true* via the incoming port `inc`, the value of n is increased if the value on port `res` is not equal to *true*, and on receipt of *true* via the port `res`, the value of n is set to 0, regardless of the value received on port `inc`.

To decouple parts of the functionality of the modulo-8 counter, *e.g.,* for individual testing, the behavior of the `Mod8Counter` is structurally refined by introducing the two subcomponents `Controller` and `Counter`, as depicted in Figure 3 (b). The controller component is responsible to delegate a reset of the counted value to the counter. This reset is triggered either after receiving a message *true* on its incoming port `rIn` or if the current counted value is 7 and the value should be further increased. The counter component realizes the counting

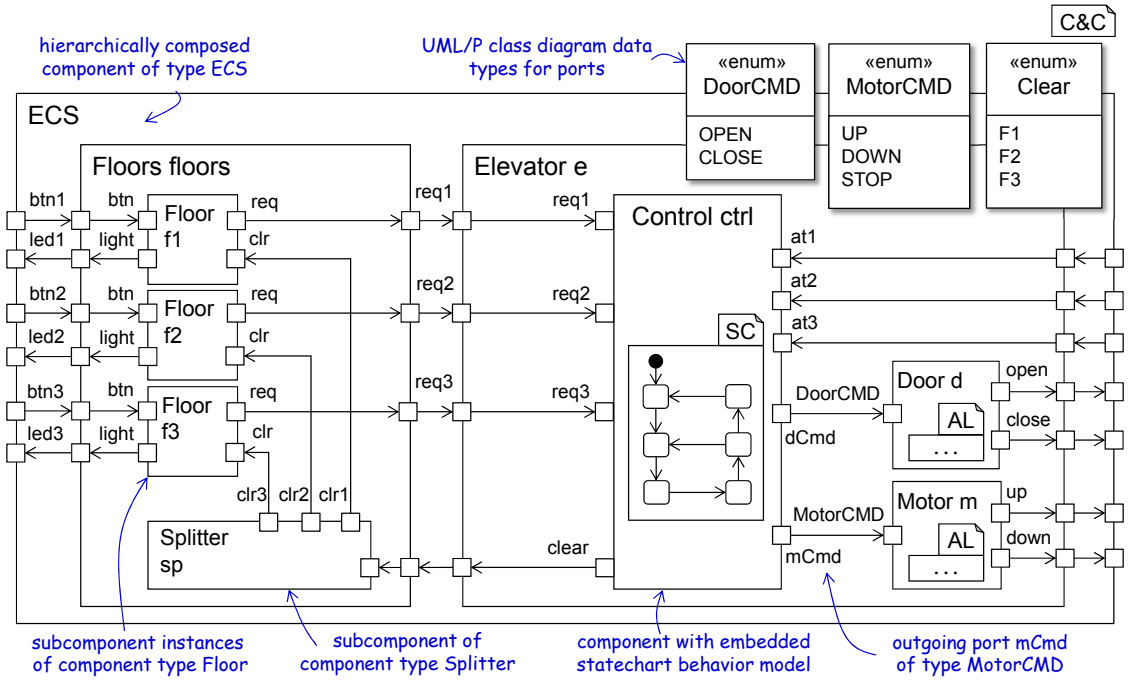


Figure 1: The elevator control system ECS comprises subcomponents to manage serving elevation requests on up to three floors.

functionality, but is unable to reset a counted value from 7 to 0 after increasing. Using the method for refinement checking presented in this paper, it is possible to fully automatically check whether the original version (atomic `Mod8Counter`) is equivalent to its successor version (composed `Mod8Counter`).

Later, the behavior of the counter is refined in a further structural refinement step (*cf.* Figure 3 (c)) by introducing subcomponents to the component `Counter`. The company reuses these subcomponents from a different project. The behavior of the component `Counter` is then defined by three counter bit components `pos0`, `pos1`, and `pos2`, which all have the same component behavior - denoted in MontiArcAutomaton by the fact that they are of the same component type `CBC`. Each of these can count a single bit component only. The MontiArcAutomaton component `CBC` with an embedded automaton realizing the component behavior is depicted in Figure 4. The bit value can be increased (modulo 2) via a message *true* on the incoming port `i` and reseted to *false* via a message *true* on the incoming port `r`. The current value of the bit is output via the outgoing port `v`, and the value of `q` is *true* iff, after increasing, the bit value changes from *true* to *false*. Otherwise, it emits *false*. Using our method, checking whether the new architecture is semantically equivalent to any of the other two architectures is possible within milliseconds.

At this point, another modeling expert notices that the design of the mod-8 counter is too complex and can be simplified, as the behavior of each `CBC` components already realizes the overflow of the modulo. Therefore, the expert proposes to model the behavior as depicted in Figure 5. As it is not obvious if the behaviors of Figure 3 (c) and Figure 5 are equivalent, the refinement check presented in this paper is employed and yields sound and complete results within milliseconds.

3. A Semantics Domain for Components

This section introduces the semantics domain for components based on the Focus framework [5, 7, 16, 36, 39] and recapitulates the most important results from [9, 16], which underlie the approach presented in this paper.

We interpret software architectures as networks of autonomously acting components communicating in a time-synchronous manner via directed, typed channels connecting the components' interfaces. A time-synchronous architecture can be interpreted as a system where component computations are performed concurrently and controlled by a global clock that splits runtime into discrete and equidistant time units. In every time unit, each component receives finitely many input messages via its interfaces and outputs finitely many messages to its environment. The computations of each component in every time unit must terminate. To this end, components partition time slices into sequences of operations (*e.g.,* assessing the guard of an embedded automaton's transition or assigning values according to its actions). Although these sequences of operations are untimed in the Focus sense, they are causally related. The semantics of component behavior thus happens logically in superdense time [28], which, following [26], distinguishes between the discrete "time continuum" (global Focus time) and "untimed causally-related actions" (a component behavior's actions within the component's time slice).

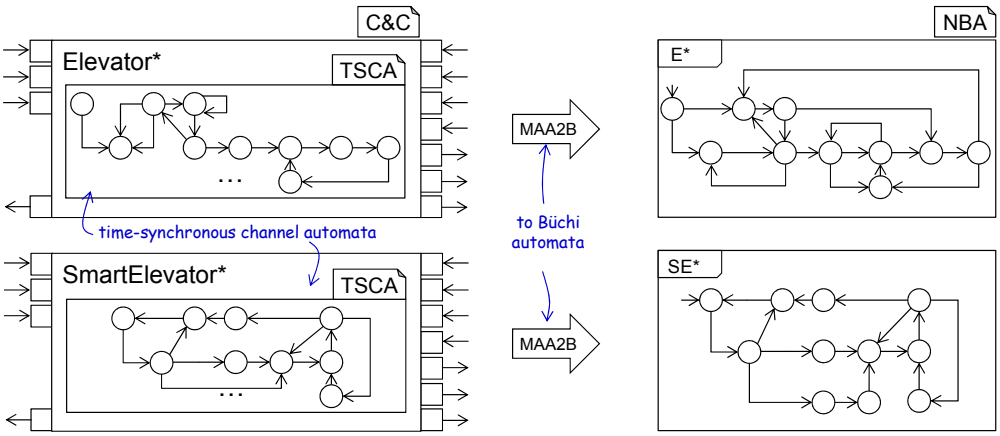


Figure 2: The composed components `Elevator` and `SmartElevator` each are transformed into flat components with a single port automaton prior to being transformed into Büchi automata and checked for language inclusion.

In the remainder, we denote by $[X \to Y]$ the set of all functions from a set X to a set Y. For a function $f \in [X \to Y]$ and a set $Z \subseteq X$, the restriction of f to Z is the function $f|_Z \in [Z \to Y]$ that satisfies $f|_Z(x) = f(x)$ for all $x \in Z$. Given two functions $f \in [X \to A]$ and $g \in [Y \to B]$, the overriding union of f with g is the function $f + g \in [(X \cup Y) \to (A \cup B)]$ that satisfies $(f + g)(x) = g(x)$ if $x \in Y$ and $(f + g)(x) = f(x)$ if $x \in X \setminus Y$ for all $x \in X \cup Y$.

3.1. Streams, Messages, Types, and Communication Histories

The history of messages a component receives or sends via an interface (*e.g.,* channel) is formally described as a stream that contains messages in order of their transmission. Let M be an arbitrary alphabet. A stream over the set M is a finite or infinite sequence of elements from M. Following [7, 39], we denote by

- M^* the set of all finite streams over M,
- M^∞ the set of all infinite streams over M,
- $\langle\rangle$ the empty stream, which is an element of M^*,
- $s^\frown t$ the concatenation of two streams s and t such that $((M^* \cup M^\infty), ^\frown, \langle\rangle)$ is a monoid. If $s \in M^\infty$ then $s^\frown t = s$.
- $\sqsubseteq$ the prefix relation over streams, which is a partial order defined by: $\forall s, t \in (M^* \cup M^\infty) : s \sqsubseteq t \Leftrightarrow \exists u : s^\frown u = t$,
- $s.t$ the $(t + 1)$-st element of a stream $s \in (M^* \cup M^\infty)$,
- $s{\downarrow_t}$ the prefix of a stream $s \in M^\infty$ of length $t \in \mathbb{N}$.

Example 1. *The finite sequence* $fib_7 = 0, 1, 1, 2, 3, 5, 8 \in \mathbb{N}^*$ *is a finite stream of natural numbers. It contains the first seven Fibonacci numbers. The infinite stream of all Fibonacci numbers* $fib \in \mathbb{N}^\infty$ *is defined by* $fib.0 = 0 \wedge fib.1 = 1 \wedge \forall t \in \mathbb{N} : t \geq 2 \Rightarrow fib.t = fib.(t-2) + fib.(t-1)$. *By definition, we have* $fib^\frown fib_7 = fib$. *Further,* $fib_7 \sqsubseteq fib$ *because the prefix of length 7 of* fib *is equal to* fib_7, *i.e.,* $fib{\downarrow_7} = fib_7$. *Thus, the first seven elements of* fib_7 *and* fib *are equal, e.g.,* $fib_7.0 = fib.0 = 0$ *and* $fib_7.3 = fib.3 = 2$.

In the remainder, let M denote an arbitrary but fixed set of data elements, such as messages, and let $Type$ be a set of data types such that each $t \in Type$ satisfies $t \subseteq M$. Types facilitate restricting the set of messages a component may emit or receive via an interface. We assume a discrete model of time where component computation is divided into discrete time units of equal and finite duration. In each time unit each component receives at most one message via each incoming interface, may perform finitely many state changes and emits at most one message via each outgoing interface. We use the special symbol $\varepsilon \in M$ to denote the absence of a message during a time unit and require $\varepsilon \in t$ for each $t \in Type$.

A *channel* is an identifier for a communication link between interface elements of components. In the following, we denote by C a set of typed channel names. The function $type \in [C \to Type]$ maps each channel in the set C to its type. Let $B \subseteq C$ be an arbitrary set of channel names. We model the history of messages emitted via the channels in the set B as a *communication history* $h \in B^\Omega$, which is an element of the set B^Ω defined as follows: $B^\Omega \stackrel{\text{def}}{=} \{h \in [B \to M^\infty] \mid \forall b \in B : h(b) \in type(b)^\infty\}$. Let $h \in B^\Omega$ be a communication history, $H \subseteq B^\Omega$ a set of communication histories, and $t \in \mathbb{N}$ a natural number. We lift the operator $\downarrow$ to communication histories and sets of communication histories in a point-wise manner, *i.e.,* $b{\downarrow_t} \in [B \to M^*]$ denotes the function that satisfies $b{\downarrow_t}(i) = b(i){\downarrow_t}$ for all $i \in B$ and $H{\downarrow_t} \stackrel{\text{def}}{=} \bigcup_{h \in H} h{\downarrow_t}$ denotes the set resulting from applying the operator to each element in H.

Example 2. *Let* $c \in C$ *be a channel of natural numbers. Then, in each time unit, the channel* c *can be either assigned a natural number or the empty message. Thus,* $type(c) = \mathbb{N} \cup \{\varepsilon\} \in Type \subseteq M$. *The communication history that assigns the channel* c *the sequence of Fibonacci numbers is given by* $h = \{c \mapsto fib\} \in c^\Omega$ *where* fib *is defined as in Example 1. The stream containing all negative integers* neg *defined by* $\forall t \in \mathbb{N} : neg.t = -t$ *is no valid assignment to channel* c *because there exists a time unit* $t \in \mathbb{N}$ *such that* $neg.t \notin type(c) = \mathbb{N} \cup \{\varepsilon\}$, *e.g., we have* $neg{\downarrow_2} = -1, -2$.

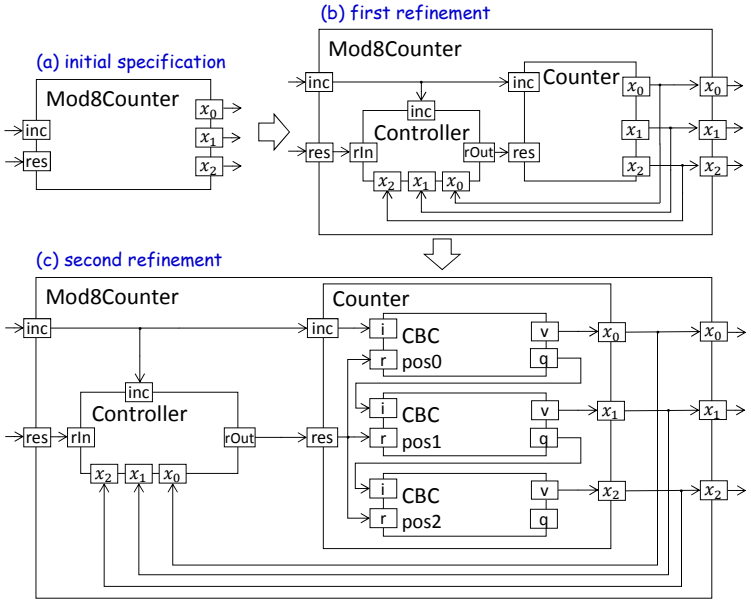


Figure 3: Graphical representation of the component `Mod8Counter` in MontiArcAutomaton syntax (a) in its initial specification and (b) after a first and (c) a second structural refinement step. All ports are of data type Boolean.

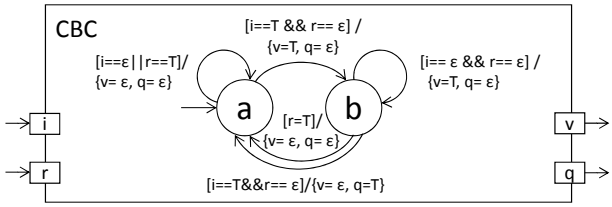


Figure 4: Automaton model realizing the component behavior of `CBC`.

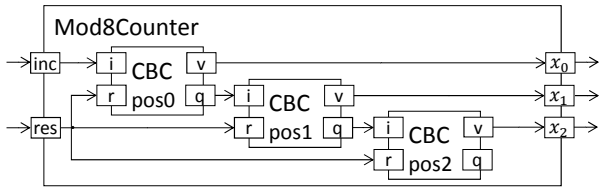


Figure 5: Alternative model of the mod-8 counter, with the behavioral equivalence to the model in Figure 3 (c) in question.

Thus, $\{c \mapsto neg\} \notin a^{\Omega}$ *is no communication history. The function mapping the channel c to its first* 7 *elements is given by* $h{\downarrow_7} = \{c \mapsto fib_7\}$ *where* fib_7 *is defined as in Example 1. Let* $empty \in \{c\}^{\Omega}$ *be defined by* $\forall t \in \mathbb{N} : empty(c).t = \varepsilon$ *denote the communication history that always assigns the channel c to the empty message. Then,* $\{h, empty\}{\downarrow_7} = \{h{\downarrow_7}, empty{\downarrow_7}\} = \{\{c \mapsto fib_7\}, \{c \mapsto \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon\}\}$.

3.2. Time-Synchronous Stream Processing Functions

We model the semantics of distributed interactive systems as sets of time-synchronous stream processing functions (TSSPFs) [9]. The notion of TSSPFs is inspired by the notion of timed SPFs [7, 16, 36, 39]. The major and crucial difference between the two notions is that TSSPFs process exactly one message per channel per time unit, whereas SPFs process a stream of messages per channel per time unit. The key idea is to treat components as black-boxes having an observable behavior characterized by the interactions on channels between systems and subsystems while hiding internal implementation details. A component is mapped to a set of functions describing the component's possible behaviors. Such a function maps communication histories over the set of input channels of a component to communication histories over the set of the component's output channels. Thus, each function in the semantics of a component with input channels $I \subseteq C$ and output channels $O \subseteq C$ is of the form $f \in [I^{\Omega} \to O^{\Omega}]$. However, such functions are not always realizable in the sense that they can be implemented [7, 34]. Intuitively, the characterizing properties for realizability are captured by the notion of weak-causality: a component cannot change messages it received or sent in the past and cannot react to messages it receives in the future [7, 34, 36, 39]. Thus, the output of a behavior describing function until time t must be completely determined by its input until time t:

Definition 1 (Time-Synchronous Stream Processing Function)**.** *Let* $I, O \subseteq C$ *be two disjoint sets of input and output channels. A function* $f \in [I^{\Omega} \to O^{\Omega}]$ *is called (weakly causal) time-synchronous stream processing function iff*

$\forall i, i' \in I^{\Omega} : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow f(i){\downarrow_t} = f(i'){\downarrow_t}$.

We denote by $[I^{\Omega} \xrightarrow{wc} O^{\Omega}]$ the set of all (weakly causal) TSSPFs mapping input histories in I^{Ω} to output histories in O^{Ω}.

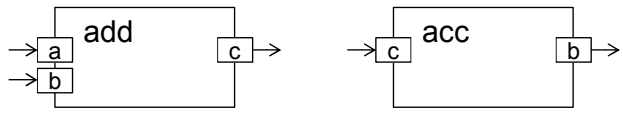


Figure 6: Graphical representation of the TSSPFs *add* and *acc*.

Example 3. *This example defines the stream processing function add that specifies the behavior of a component for adding natural numbers. The interface of the TSSPF is graphically illustrated on the left hand side of Figure 6. The input channels are* $I = \{a, b\}$ *and the set of output channels is* $O = \{c\}$*. The type of all channels is the type of natural numbers, i.e.,* $type(a) = type(b) = type(c) = \mathbb{N} \cup \{\varepsilon\} \in Type \subseteq M$*. If the function add receives natural numbers on both channels a and b in a time unit t, then the function outputs the sum of the received messages via the channel c in time unit t. Otherwise, if the function receives the empty message* ε *on any of the input channels in time unit t, then the function outputs* ε *in time unit t. The function* $add \in [I^\Omega \to O^\Omega]$ *is formally defined by* $\forall i \in I^\Omega : \forall t \in \mathbb{N} : (add(i))(c).t = . \begin{cases} i(a).t + i(b).t, & \text{if } i(a).t, i(b).t \in \mathbb{N} \\ \varepsilon, & \text{otherwise} \end{cases}$

The function add is weakly causal because its output in each time unit is fully specified by its inputs in the same time unit, i.e., in each time unit, the function's output does not depend on future input and the function does not change previously processed messages. This is verifiable with a short proof by induction over the lengths of prefixes of communication histories.

The following example illustrates that the weak causality requirement on TSSPFs is necessary.

Example 4. *This example defines the function u (unrealizable) over communication histories that is not weakly causal. We define the function over Boolean messages. The function's input channel set is given by* $I = \{in\}$ *and its output channel set is given by* $O = \{out\}$*. The types of in and out are* $type(in) = type(out) = \{\top, \bot, \varepsilon\} \in Type \subseteq M$ *where* $\top$ *represents the value true and* $\bot$ *represents the value false. In each time unit t, the function* $u \in [I^\Omega \to O^\Omega]$ *outputs the value it receives in time unit* $t + 1$*. It is formally defined by* $\forall i \in I^\Omega : \forall t \in \mathbb{N} : u(out).t = i(in).(t + 1)$*. Obviously, the functionality described by the function u cannot be implemented by a component: A component implementing the function would be able to predict its future input to determine its present output. This is formally captured by weak-causality. To proof that the function u is not weakly causal, we need to find two input channel histories* $i, i' \in I^\Omega$ *and a time unit* $t \in \mathbb{N}$ *such that* $i{\downarrow_t} = i'{\downarrow_t} \wedge u(i){\downarrow_t} \neq u(i'){\downarrow_t}$*. We choose i and i' such that* $\forall t \in \mathbb{N} : i(in).t = \bot$ *and* $i'(in).0 = \bot \wedge \forall t \in \mathbb{N} : t \geq 1 \Rightarrow i'(in).t = \top$*. Then,* $i{\downarrow_1} = \{in \mapsto \bot\} = i'{\downarrow_1}$ *and* $u(i){\downarrow_1} = \{out \mapsto \bot\}$ *and* $u(i'){\downarrow_1} = \{out \mapsto \top\}$*. Thus,* $i{\downarrow_1} = i'{\downarrow_1} \wedge u(i){\downarrow_1} \neq u(i'){\downarrow_1}$*.*

A single TSSPF is well-suited to model the semantics of a deterministic component. However, as a TSSPF maps each input to exactly one output, TSSPFs are not general enough to model the semantics of underspecified components where the exact output to an input is not fully specified. We thus model the semantics components as sets of TSSPFs:

Definition 2 (Component Semantics Describing)**.** *Let* $I, O \subseteq C$ *be two disjoint sets of channels. A set of TSSPFs* $F \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ *is called component semantics describing iff it satisfies* $\forall g \in [I^\Omega \xrightarrow{wc} O^\Omega] : ((\forall i \in I^\Omega : \exists f \in F : g(i) = f(i)) \Rightarrow g \in F)$*.*

The definition above makes the semantics domain of components fully abstract [16, 17] in the sense of [19] and allows to handle unbounded non-determinism [16]. Full abstraction is achieved by the closeness property, which requires that each TSSPF resulting from a combination of TSSPFs included in the set F is also included in F. The closeness property is also important to make component semantics as little distinguishing as possible. This is illustrated by the fact that two different arbitrary sets of TSSPFs may encode the same component behaviors. The reason for this is that one may find a TSSPF $g \notin F$ that is not included in a set of TSSPFs F, which can be interpreted as a combination of different TSSPFs contained in F. It thus does not induce a new behavior not already covered by a TSSPF in F but, for instance, induces a semantic difference between a component with semantics described by F and a component with semantics described by $F \cup \{g\}$. As a result the semantics of two components that have the exact same observable behaviors may be considered unequal. Consequently, full abstraction is not achieved. Thereby, the closeness property is necessary. However, each arbitrary set of TSSPFs $F \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ can be lifted to a component semantics describing set of TSSPFs $\overline{F} \stackrel{\text{def}}{=} \{g \in [I^\Omega \xrightarrow{wc} O^\Omega] \mid \forall i \in I^\Omega : \exists f \in F : g(i) = f(i)\}$ that satisfies $F \subseteq \overline{F}$ and $\overline{\overline{F}} = \overline{F}$.

3.2.1. Composition of TSSPFs

Composition is an important concept to achieve modularity. Composing the semantics of the individual components of a system leads to the semantics of the whole system. Composing arbitrary sets of TSSPFs can lead to realizability problems in delay-free feedback loops where the input of a component in time unit t depends on another component's output in time unit t and vice versa. Thus, composition is only defined for TSSPFs where causality between inputs and outputs on channels connected via a feedback loop is ensured. This is the case if one of the TSSPFs participating in a composition is strongly causal with respect to its channels connected by the composition. Intuitively, a TSSPFs f is strongly causal modulo the input channels J and output channels P, if the function's outputs on the channels in P until time unit $t + 1$ is not influenced by the function's inputs received on the channels in J after time unit t. Other than with weak causality, this especially includes that the outputs do not rely on the inputs received in the *same* time unit.

Definition 3 (Strongly Causal Modulo)**.** *Let* $f \in [I^\Omega \xrightarrow{wc} O^\Omega]$ *be a TSSPF and let* $J \subseteq I$ *and* $P \subseteq O$ *be two subsets of input and output channels names. The TSSPF f is called strongly causal modulo* (J, P) *iff* $\forall i, i' \in I^\Omega : \forall t \in \mathbb{N} :$

$((i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J}) \Rightarrow f(i)|_P{\downarrow_{t+1}} = f(i')|_P{\downarrow_{t+1}}$.

The following example illustrates that there are weakly causal TSSPFs that are not strongly causal.

Example 5. *The function add* $\in [I^\Omega \xrightarrow{wc} O^\Omega]$ *as defined in Example 3 and depicted in Figure 6 is not strongly causal modulo* (I, O)*. This holds because the function's output in a time unit always depends on its present input. To formally show that add is not strongly causal modulo* (I, O)*, we need to find two inputs* $i, i' \in I^\Omega$ *and a time unit* $t \in \mathbb{N}$ *such that* $i|_I\downarrow_t = i'|_I\downarrow_t$ *and* $add(i)|_O\downarrow_{t+1} \neq add(i')|_O\downarrow_{t+1}$*. We chose* i *and* i' *such that* $\forall t \in \mathbb{N} : i(a).t = i(b).t = 1$ *and* $\forall t \in \mathbb{N} : i'(a).t = 2 \wedge i'(b).t = 1$*. Then,* $i|_I\downarrow_0 = \{c \mapsto \langle\rangle\} = i'|_I\downarrow_0$ *and* $add(i)|_O\downarrow_1 = \{c \mapsto 2\}$ *and* $add(i')|_O\downarrow_1 = \{c \mapsto 3\}$*. Thus,* $i|_I\downarrow_t = i'|_I\downarrow_t$ *and* $add(i)|_O\downarrow_{t+1} \neq add(i')|_O\downarrow_{t+1}$*, which we needed to show. Using the same counterexample, it is possible to show that add is not strongly causal with respect to* $(\{a\}, O)$*, either. Analogously, it can be shown that add is not strongly causal modulo* $(\{b\}, O)$*.*

The following example describes a strongly causal TSSPF:

Example 6 (Strongly Causal TSSPF)**.** *Consider the strongly causal TSSPF acc* $\in [I^\Omega \xrightarrow{wc} O^\Omega]$ *where* $I = \{c\}$ *and* $O = \{b\}$ *and* $type(c) = type(b) = \mathbb{N} \cup \{\varepsilon\}$*. The interface of the TSSPF is graphically illustrated on the right hand side of Figure 6. The TSSPF acc specifies the behavior of an accumulator component. In each time unit, the component outputs the sum of the values it received in the past. The component initially outputs the message* 0*, which reflects that it has not received positive integers, yet. When the component receives a positive integer in a time unit, it outputs the sum of the received integer and the value emitted in the current time unit in the next time unit. When the accumulator receives the empty message* ε*, the accumulated value remains unchanged. Thus, in the next time unit, the component again emits the value that it emits in the current time unit. Thus, the output of the function acc at time unit* $t + 1$ *only depends on its input up to and including time unit* t*. We formally define the TSSPF acc by the following equation:*

$$\forall i \in I^\Omega : \forall t \in \mathbb{N} : acc(i)(b).t = \begin{cases} 0 & \text{if } t = 0 \\ acc(i)(b).(t-1) + i(c).(t-1) & \text{if } t \geq 1 \wedge i(c).(t-1) \in \mathbb{N} \\ acc(i)(b).(t-1) & \text{if } t \geq 1 \wedge i(c).(t-1) = \varepsilon \end{cases}$$

The function acc is strongly causal modulo (I, O) *by definition. This can be formally proven by induction over the length of prefixes of input communication histories:*

$t = 0$: The property is satisfied because the TSSPF add always outputs the same initial value in time unit $t = 0$, independent of its inputs in time unit $t = 0$.

Let $n \in \mathbb{N}$. Assume for all $t \leq n$ and all $i, i' \in I^\Omega$, it holds that $i|_I\downarrow_t = i'|_I\downarrow_t \Rightarrow acc(i)|_O\downarrow_{t+1} = acc(i')|_O\downarrow_{t+1}$.

Let $t = n + 1$.

Let $i, i' \in I^\Omega$ such that $i|_I\downarrow_t = i'|_I\downarrow_t$.

We need to show $acc(i)|_O\downarrow_{t+1} = acc(i')|_O\downarrow_{t+1}$.

Using the induction hypothesis: $acc(i)|_O\downarrow_t = acc(i')|_O\downarrow_t$.

Therefore, especially $acc(i)(b).(t-1) = acc(i')(b).(t-1)$.

By assumption $i|_I\downarrow_t = i'|_I\downarrow_t$ and thus $i(c).(t-1) = i'(c).(t-1)$.

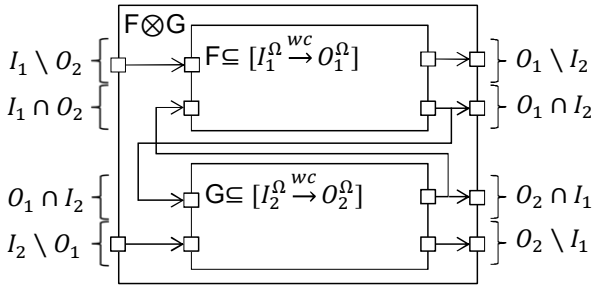


Figure 7: Graphical representation of the composition of two sets of TSSPFs.

As $t = n + 1$, we have that $t \geq 1$.
If $i(c).(t-1) = i'(c).(t-1) \in \mathbb{N}$, then the above implies $acc(i)(b).t = acc(i)(b).(t-1) + i(c).(t-1) = acc(i')(b).(t-1) + i'(c).(t-1)\} = acc(i')(b).t$
Similarly, if $i(c).(t-1) = i'(c).(t-1) = \varepsilon$, then $acc(i)(b).t = acc(i)(b).(t-1) = acc(i')(b).(t-1) = acc(i')(b).t$
We can conclude that $acc(i)|_O\downarrow_{t+1} = acc(i')|_O\downarrow_{t+1}$.

A set of TSSPFs F is called strongly causal with respect to (J, P) iff there exists a function $f \in F$ that is strongly causal with respect to (J, P). With this, the set of TSSPFs F is required to exhibit at least one realization that is strongly causal with respect to (J, P). The causality complication is avoided, if causality between the inputs and outputs on the connected channels of at least one composition participant is guaranteed:

Definition 4 (Composable)**.** *Two sets of TSSPFs* $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ *and* $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ *are called composable iff* F_1 *is strongly causal with respect to* $(I_1 \cap O_2, I_2 \cap O_1)$ *or* F_2 *is strongly causal modulo* $(I_2 \cap O_1, I_1 \cap O_2)$*.*

Example 7 (Composability)**.** *The TSSPFs add and acc are described and formally defined in Example 3 and Example 6. The interfaces of the TSSPFs are graphically presented in Figure 6. Let* $Add = \{add\}$ *and* $Acc = \{acc\}$ *denote the singleton sets containing the TSSPFs add and acc. The two sets of TSSPFs are composable because, as shown in Example 6, the TSSPF* $acc \in Acc$ *is strongly causal modulo* $(\{c\}, \{b\}) = (I_{acc} \cap O_{add}, O_{acc} \cap I_{add})$*.*

Components communicate with each other via unidirected, typed channels established by connectors connecting component interfaces. Multiple components may read from the same channel, whereas only one component is permitted to write messages on a channel. This ensures that no merging of messages emitted from different components via the same channel is necessary. Thus the output channels of the functions of two sets of TSSPFs need to be disjoint to enable composition. The output channels of the composition result are the output channels of both composition's participants. The compound's input channels are exactly the input channels of both components that are no output channels of any of the two components.

The composition of two sets of TSSPFs F and G is graphically illustrated in Figure 7. The input channels of $F \otimes G$ are

the input channels $I_1 \setminus O_2$ of F that are no output channels of G and the input channels $I_2 \setminus O_1$ of G that are no output channels of F. The output channels of $F \otimes G$ are all output channels of F and G. The composition of two sets of TSSPFs yields a set of TSSPFs:

Definition 5 (Composition)**.** *Let $F_1 \subseteq [I_1^\Omega \xrightarrow{wc} O_1^\Omega]$ and $F_2 \subseteq [I_2^\Omega \xrightarrow{wc} O_2^\Omega]$ be two component semantics describing and composable sets of TSSPFs with disjoint output channel sets $O_1 \cap O_2 = \emptyset$. Let $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ and $O = O_1 \cup O_2$. The composition $F_1 \otimes F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ of F_1 and F_2 is defined by*

$$F_1 \otimes F_2 \stackrel{\text{def}}{=} \{f \mid \forall i \in I^\Omega : \exists f_1 \in F_1 : \exists f_2 \in F_2 : f(i) = o + p, \text{ where } o = f_1((i+p)|_{I_1}), p = f_2((i+o)|_{I_2})\}$$

The composition operator is defined similar as in [16, 17, 39] with the difference that we consider the time-synchronous system model instead of the more general timed system model [7].

Example 8. *The following demonstrates the composition of sets of TSSPFs by example. Let $Add = \{add\}$ and $Acc = \{acc\}$ be sets of TSSPFs as defined in Example 7. The sets Add and Acc are composable (cf. Example 7). As both sets contain a single TSSPF, the sets are component semantics describing (cf. Definition 2). Further, the sets of output channels of the sets' TSSPFs are disjoint. Thus, the composition operator $\otimes$ is applicable. Figure 8 graphically illustrates the result from composing the two sets Add and Acc. The set of TSSPFs $Add \otimes Acc$ contains the single TSSPF $f \in [\{a\}^\Omega \xrightarrow{wc} \{b,c\}^\Omega]$ that satisfies $\forall i \in \{a\}^\Omega : f(i) = o + p$ where $o = add((i+p)|_{I_{add}})$ and $p = acc((i+o)|_{I_{acc}})$. Given an input $i \in \{a\}^\Omega$, iteratively computing the values of o, p, c, and b is possible because the TSSPF acc is strongly casual. For instance, let $i = \{a \mapsto 1, 1, ...\} \in \{a\}^\Omega$ denote the communication history that always assigns channel a to 1, i.e., $\forall t \in \mathbb{N} : i(a).t = 1$. The first output of acc via channel b is by definition always 0 (cf. Example 6). With this, we can compute that add outputs $1 = 0 + 1$ via channel c in time unit 0. This determines the output 1 of acc at time unit 1. This again enables to determine that add outputs $2 = 1 + 1$ via channel c in time unit 1. This determines that acc outputs value 3 via channel b in time unit 2. Thus, add outputs 4 via channel c in time unit 2. We can approximate the value of the TSSPF f up to every fixed time unit $t \in \mathbb{N}$ for every fixed input $i \in \{a\}^\Omega$ by using the method sketched above.*

The composition is well defined and results in a component semantics describing set of TSSPFs. This is a consequence of the requirement that one set of TSSPFs must be strongly causal modulo its connected channels.

Theorem 1. *If F_1 and F_2 are two component semantics describing and composable sets of TSSPFs with disjoint output channel sets, then $F_1 \otimes F_2$ is also component semantics describing.*

Proof. Analogous to proof of Theorem 9 in [16] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

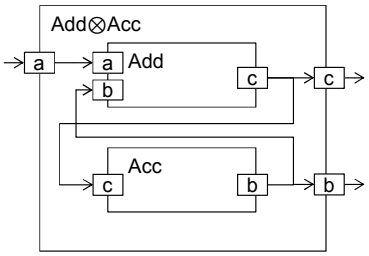


Figure 8: Graphical representation of the composition of *Add* and *Acc*.

4. Time-Synchronous Channel Automata

A TSCA specifies the behavior (of parts) of an interactive system and represents a component semantics describing set of TSSPFs that is given by its semantics. We later use TSCAs to model components. TSCAs as introduced in this paper are based on our previous work on TSPAs [9] and are strongly inspired by port automata [16], I/O* automata [36, 39], and MAA_{ts} automata [34]. Port automata and I/O* automata consume and produce time slices of arbitrary but finitely many input messages in every transition step. In contrast, TSCAs, TSPAs, and MAA_{ts} automata consume and produce at most one message per input channel in each time slice. Given the set of states and the channel types of an automaton are finite, MAA_{ts} automata, TSPAs, and TSCAs are guaranteed to have finitely many transitions. This is not the case for I/O* and port automata since both have to define a transition for each state and each possible input communication history. Even if the type of a channel is finite, the number of communication histories (streams) of the channel's type is infinite. I/O* automata and MAA_{ts} automata enforce causality between input and output histories by requiring initial outputs on all channels. In contrast, TSPAs and TSCAs do not require initial outputs. While the syntax of MAA_{ts} and TSCAs models variables explicitly, in TSPAs [9] variables have to be represented implicitly in the state space. While MAA_{ts} automata distinguish between data and control states (*i.e.,* variables and (control) states), TSCAs consist of data states (variables) only. This eliminates unnecessary complexity and notational clutter as control states can be easily represented as data states. Some proofs of some theorems presented in the following are analog to proofs that have already been carried out in [9, 16]. In case we are stating an analogous theorem, we refer to the appropriate corresponding proof in [9, 16].

A TSCA consists of a set of states, an interface of input and output channels, and transitions defining the TSCA's behavior. The interface is encoded by a channel signature.

Definition 6 (Channel Signature)**.** *Let $I, O \subseteq C$ be two disjoint sets of channel names. A channel signature is a tuple $\Sigma = (I, O)$. We denote by $C(\Sigma) \stackrel{\text{def}}{=} I \cup O$ the set of all channels in Σ. A channel signature Σ is called finite iff $C(\Sigma)$ and $type(c)$ for all $c \in C(\Sigma)$ are finite.*

A channel assignment maps channels to messages of the

channels' types. Let $B \subseteq C$. A *channel assignment* is an element of the set $B^{\rightarrow}$ defined as $B^{\rightarrow} \stackrel{\text{def}}{=} \{a \in [B \to M] \mid \forall b \in B : a(b) \in type(b)\}$. Channel assignments are used as TSCA transition labels.

Definition 7 (TSCA)**.** *A time-synchronous channel automaton is a tuple $A = (\Sigma, X, S, \iota, \delta)$ where:*

- $\Sigma = (I, O)$ *is a channel signature,*
- $X \subseteq C$ *is a set of variable symbols (internal channels),*
- $S \subseteq X^{\rightarrow}$ *is a set of states,*
- $\iota \in S$ *is the initial state,*
- $\delta \subseteq S \times C(\Sigma)^{\rightarrow} \times S$ *is the transition relation.*

For convenience, we sometimes write $s \stackrel{\theta}{\rightarrow}_{\delta} t$ instead of $(s, \theta, t) \in \delta$ and simply $s \stackrel{\theta}{\rightarrow} t$ if δ is clear from the context. To avoid notational clutter, we often denote the components of a TSCA $A = (\Sigma, X, S, \iota, \delta)$ with $\Sigma = (I, O)$ by $\Sigma_A \stackrel{\text{def}}{=} \Sigma$, $X_A \stackrel{\text{def}}{=} X$, $S_A \stackrel{\text{def}}{=} S$, $\iota_A \stackrel{\text{def}}{=} \iota$, $\delta_A \stackrel{\text{def}}{=} \delta$, $I_A \stackrel{\text{def}}{=} I$, and $O_A \stackrel{\text{def}}{=} O$. We also omit the subscripts if they are clear from the context.

Example 9 (TSCA of the component `CBC`)**.** *The TSCA of the component `CBC` is similar to the behavior automaton of the `CBC` component, which is graphically depicted in Figure 4. The channel signature comprises input and output channels. States and transitions reflect states and transitions in the behavior automaton, and the internal channel state reflects the current state of the behavior automaton. We interpret the absence of a message ("ε") equal to the Boolean value "$false$" and, again, denote "$\top$" as the Boolean value "true". The TSCA of the component `CBC` then can be formulated as $TSCA_{CBC} = (\Sigma_{CBC}, X_{CBC}, S_{CBC}, \iota_{CBC}, \delta_{CBC})$ with*

- *channel signature $\Sigma_{CBC} = (I_{CBC}, O_{CBC}) = (\{i, r\}, \{v, q\})$,*
- *channel data types: $type(i) = type(r) = type(v) = type(q) = \{\top, \varepsilon\}$,*
- *internal channel $X_{CBC} = \{state\}$ with $type(state) = \{0, 1\}$,*
- *states $S_{CBC} = X^{\rightarrow}_{CBC} = \{a, b\}$ with $a = \{state \mapsto 0\}$ and $b = \{state \mapsto 1\}$,*
- *initial state $\iota_{CBC} = a$,*
- *and transition relation $\delta_{CBC} = \{$*
  $\{(a, \theta, a) \mid (\theta(i) = \varepsilon \vee \theta(r) = \top) \wedge \theta(v) = \varepsilon \wedge \theta(q) = \varepsilon\}$
  $\cup \{(a, \theta, b) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \top \wedge \theta(q) = \varepsilon\}$
  $\cup \{(b, \theta, b) \mid \theta(i) = \varepsilon \wedge \theta(r) = \varepsilon \wedge \theta(v) = \top \wedge \theta(q) = \varepsilon\}$
  $\cup \{(b, \theta, a) \mid \theta(r) = \top \wedge \theta(v) = \varepsilon \wedge \theta(q) = \varepsilon\}$
  $\cup \{(b, \theta, a) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \varepsilon \wedge \theta(q) = \top\}\}$.

The reactions of a TSCA are defined by its transitions. In each time unit, a TSCA performs one state change by executing one transition enabled by its input and outputs one message on each output channel. Let A be a TSCA. A is said to be *reactive* iff $\forall s \in S : \forall i \in I^{\rightarrow} : \exists t \in S : \exists \theta \in C(\Sigma)^{\rightarrow} : (s, \theta, t) \in \delta \wedge \theta|_I = i$. A reactive TSCA is called *component*. Components must not block their environments and must be able to react to any possible well-typed input in any time unit. Therefore, a component must define a reaction to every possible input for each of its states. A is called *finite* iff Σ and S are finite. The *size* of A, denoted $|A|$, is defined as the sum of the number of its states and transitions, *i.e.,* $|A| = |S| + |\delta|$. A is called *deterministic* iff $\forall s \in S : \forall i \in I^{\rightarrow} : |\{t \in S \mid \exists \theta \in C(\Sigma)^{\rightarrow} : \theta|_I = i \wedge (s, \theta, t) \in \delta\}| = 1$. A is called *I/O-deterministic* iff $\forall s \in S : \forall \theta \in C(\Sigma)^{\rightarrow} : |\{t \in S \mid (s, \theta, t) \in \delta\}| \leq 1$. Reactive TSCAs are adequate models for components as they specify at least one output for each input. The size of TSCAs is used for measuring algorithmic complexities. Intuitively, if A is deterministic, then for each state and each input, A only has at most one choice for switching the state when processing the input. It thus acts as a system part in a deterministic implementation. If A is reactive and deterministic, then it has exactly one choice for switching its state. In contrast, if A is I/O-deterministic, for each state, the state A switches to when it reads an input and produces an output can be uniquely identified by the input/output pair. As shown in Section 5, semantic differencing of I/O-deterministic TSCAs is possible in polynomial time in the sizes of the automata.

Example 10 ($TSCA_{CBC}$ is reactive and deterministic)**.** *$TSCA_{CBC}$ (cf. Example 9) is reactive because in both of its states, there is an outgoing transition with a channel assignment that has all input channels in its domain. In other words, it defines an output for each possible state/input combination and therefore it describes a component. $TSCA_{CBC}$ is finite, because $|S|$ and Σ are finite: The set of states S is finite since $|S| = 2$. The channel signature Σ is finite because $|C(\Sigma)| = 4$ and $\forall c \in C(\Sigma) : |type(c)| = |\{\top, \varepsilon\}| = 2$. $TSCA_{CBC}$ is reactive and deterministic because in both states and for each possible input, there is exactly one state that the TSCA may change to.*

The following theorem shows that determinism implies I/O-determinism. The other direction, however, does not hold.

Theorem 2. *Any deterministic TSCA is I/O-deterministic.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ with $\Sigma = (I, O)$ be a deterministic TSCA. Suppose towards a contradiction there exists a state s and a channel valuation $\theta \in C(\Sigma)^{\rightarrow}$ such that $|\{t \in S \mid s \stackrel{\theta}{\rightarrow} t\}| > 1$. Thus, there exist $u, v \in S$ such that $u \neq v$ and $s \stackrel{\theta}{\rightarrow} u$ and $s \stackrel{\theta}{\rightarrow} v$. Let $i = \theta|_I$. Then, as $u \neq v$ and $s \stackrel{\theta}{\rightarrow} u$, it holds that $u, v \in \{t \in S \mid \exists \theta \in C(\Sigma)^{\rightarrow} : \theta|_I = i \wedge s \stackrel{\theta}{\rightarrow} t\}$. Thus, $|\{t \in S \mid \exists \theta \in C(\Sigma)^{\rightarrow} : \theta|_I = i \wedge s \stackrel{\theta}{\rightarrow} t\}| \geq 2$, which contradicts that A is deterministic. □

With this, problems that are efficiently solvable for I/O-deterministic TSCAs are at least as efficiently solvable for deterministic TSCAs.

4.1. Execution and Behavior Semantics of TSCAs

This section formalizes the intuitive descriptions of a TSCA's behavior. Further analyses on TSCAs will be based on the operational semantics introduced in this section.

Definition 8 (Execution)**.** *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA. An execution σ of A is an infinite, alternating sequence of states and channel assignments starting with the initial state ι:*

$\sigma = s_0, \theta_0, s_1, \theta_1, ...$ such that $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i} s_{i+1}$. The set of all executions of A is denoted by $execs(A)$.

Executions comprise the state changes and interactions performed by a TPSA. Abstracting from state changes allows to treat TSCAs as black boxes with hidden internal details. This requires explicating the *behavior* of a TSCA.

Definition 9 (Behavior)**.** *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA with channel signature $\Sigma = (I, O)$. The behavior of an execution $\sigma = s_0, \theta_0, s_1, \theta_1, ...$ of A is defined as the sequence $beh(\sigma) \stackrel{\text{def}}{=} \theta_0, \theta_1, ...$ containing only channel assignments. For $P \subseteq C(\Sigma)$, the restriction of $beh(\sigma)$ to P is defined as $beh(\sigma)|_P \stackrel{\text{def}}{=} \theta_0|_P, \theta_1|_P, ...$ We denote by $behs(A) \stackrel{\text{def}}{=} \bigcup_{\sigma \in execs(A)} beh(\sigma)$ the set of all behaviors of all executions of A. The named communication history h_α induced by a behavior $\alpha \in behs(A)$ with $\alpha = e_0, e_1, ...$ is defined as the function $h_\alpha \in (I \cup O)^\Omega$ that satisfies $h_\alpha(x).t = e_t(x)$ for all $x \in I \cup O$ and $t \in \mathbb{N}$.*

Given a TSCA A with $\Sigma_A = (I, O)$ and an input history $i \in I^\Omega$, we denote the set of communication histories induced A with input i by $A[i] \stackrel{\text{def}}{=} \{o \in O^\Omega \mid \exists \alpha \in behs(A) : o = h_\alpha|_O \wedge h_\alpha|_I = i\}$.

Example 11 (Execution and behavior of $TSCA_{CBC}$)**.** *An execution of a TSCA is an infinite sequence in general. Let a, b, θ_{ab}, $\theta_{ba}, \theta_{res}$, and θ_{nop} be given as follows:*

- $a = \{state \mapsto 0\}$, $b = \{state \mapsto 1\}$,
- $\theta_{ab} = \{i \to \top, r \mapsto \varepsilon, v \mapsto \top, q \mapsto \varepsilon\}$,
- $\theta_{ba} = \{i \mapsto \top, r \mapsto \varepsilon, v \mapsto \top, q \mapsto \top\}$,
- $\theta_{res} = \{i \mapsto \top, r \mapsto \top, v \mapsto \varepsilon, q \mapsto \varepsilon\}$, *and*
- $\theta_{nop} = \{i \mapsto \varepsilon, r \mapsto \varepsilon, v \mapsto \varepsilon, q \mapsto \varepsilon\}$.

An execution of the $TSCA_{CBC}$, for instance, is $e = a, \theta_{ab}, b, \theta_{ba}, a, \theta_{ab}, b, \theta_{res}, a(, \theta_{nop}, a)^\infty$. Accordingly, the behavior of this execution is given by $beh(e) = \theta_{ab}, \theta_{ba}, \theta_{ab}, \theta_{res}(, \theta_{nop})^\infty$. This behavior can be restricted to include only a subset of the involved channels, which is done by restricting the individual channel assignments. For instance, the restriction $e|_{\{q\}}$ of e to $\{q\}$ is given by $e|_{\{q\}} = \theta_{ab}|_{\{q\}}, \theta_{ba}|_{\{q\}}, \theta_{ab}|_{\{q\}}, \theta_{res}|_{\{q\}}(, \theta_{nop}|_{\{q\}})^\infty = \{q \mapsto \varepsilon\}, \{q \mapsto \top\}, \{q \mapsto \varepsilon\}, \{q \mapsto \varepsilon\}, \{q \mapsto \varepsilon\}^\infty$. The communication history h_e induced by the behavior e maps the channel q, for instance, to the stream $h_e(q) = \varepsilon, \top, \varepsilon, \varepsilon, \varepsilon^\infty$.

If a state is not visited by any of the TSCA's executions, then it is not productive in the sense that it does not influence any behavior. Thus, when analyzing the set of behaviors of a TSCA it suffices to analyze only the TSCA's reachable part that only consists of states visited by at least one execution. A state is reachable in a TSCA if there is an execution that visits it.

Definition 10 (Reachable)**.** *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA with channel signature $\Sigma = (I, O)$. A state $s \in S$ is called reachable in A if there exists a finite alternating sequence of states $s_0, \theta_1, s_1, \theta_2, ..., \theta_n, s_n$ starting in the initial state $s_0 = \iota$ and ending in state $s = s_n$ such that $s_i \xrightarrow{\theta_{i+1}} s_{i+1}$ for all $0 \leq i < n$. The set of all reachable states in A is denoted by $reach(A)$.*

Non-reachable states are redundant in the sense that they do not affect a TSCA's behavior.

Example 12 (Reachable states in $TSCA_{CBC}$)**.** *In $TSCA_{CBC}$, both states are reachable because $reach(TSCA_{CBC}) = \{a, b\}$. The execution e depicted in Example 11 reaches both states of the TSCA. To this effect, any prefix of e ending in state a and any prefix of e ending in state b are valid finite alternating sequences of states and channel assignments. This shows that both states are reachable.*

Removing the unreachable states from a TSCA results in a TSCA with exactly the same behaviors.

Theorem 3. *Let $A = (\Sigma, X, S, \iota, \delta)$ be a TSCA with channel signature $\Sigma = (I, O)$ and let $R = reach(A)$ denote the reachable states of A. Then, $B \stackrel{\text{def}}{=} (\Sigma, R, \iota, \delta \cap R \times C(\Sigma)^\to \times R)$ is a TSCA that satisfies $behs(A) = behs(B)$.*

Proof. Let A and B be given as above and let $\Delta = \delta \cap R \times C(\Sigma)^\to \times R$ denote the transitions of B.

$behs(A) \subseteq behs(B)$: Let $\sigma = s_0, \theta_1, s_1, \theta_2, s_2...$ be an execution of A. Then, it holds that $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$. Now, let $j \in \mathbb{N}$. As $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$ is satisfied, it especially holds that $s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$ for each $0 \leq i < j$. Thus, the finite sequence $s_0, \theta_1, s_1, \theta_2, s_2, ..., \theta_j, s_j$ satisfies $s_i \xrightarrow{\theta_{i+1}}_\delta s_{i+1}$ for all $0 \leq i < j$. From this, we can conclude that each state s_j where $j \in \mathbb{N}$ is reachable in A. As $\forall i \in \mathbb{N} : s_i \in R$, we have that $\forall i \in \mathbb{N} : (s_i, \theta_{i+1}, s_{i+1}) \in R \times C(\Sigma)^\to \times R$. From this and $\forall i \in (s_i \theta_{i+1}, s_{i+1}) \in \delta$, we can conclude $(s_i \theta_{i+1}, s_{i+1}) \in \Delta = \delta \cap R \times C(\Sigma)^\to \times R$, *i.e.*, $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_{i+1}} s_{i+1}$. From the above we can conclude $\sigma \in execs(B)$. All in all, we obtain $execs(A) \subseteq execs(B)$ and therefore $behs(A) \subseteq behs(B)$.

$behs(B) \subseteq behs(A)$: Let $\sigma = s_0, \theta_0, s_1, \theta_1, ...$ be an execution of A. Then, it holds that $s_0 = \iota$ and $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i} s_{i+1}$. As $R \subseteq S$ and $\Delta \subseteq \delta$, we obtain $\forall s, t \in R : \forall \theta \in C(\Sigma)^\to : s \overset{\theta}{\to}_\Delta \Rightarrow s \overset{\theta}{\to}_\delta t$. Therefore, $\forall i \in \mathbb{N} : s_i \xrightarrow{\theta_i}_\Delta s_{i+1}$ implies $\forall i \in \mathbb{N} : s_i \overset{\theta_i}{\to}_\delta s_{i+1}$. Thus, it holds that $\sigma \in execs(A)$. We can conclude $execs(B) \subseteq execs(A)$ and therefore $behs(B) \subseteq behs(A)$. □

Algorithm 1 shows a procedure for removing the unreachable states from any finite TSCA. The algorithm performs a depth-first traversal on the input TSCA to only retain the input TSCA's states that are reachable from its initial state. While traversing the automaton, the algorithm also adds the transitions originating from any reachable state to the resulting automaton. As any state that is the target of any transition with a reachable source state is also reachable, the transitions added in Algorithm 1 always connect reachable states. The operations *push*, *pop*, and

top denote the standard stack operations and the symbol $\bot$ denotes the empty stack. The algorithm terminates because the input TSCA is required to be finite and every state is visited at most once.

Algorithm 1 Trimming a finite TSCA.

Input: Finite TSCA $A = (\Sigma, X, S, \iota, \delta)$
Output: TSCA containing only reachable parts of A

```
define R ← {ι} as set            /* reachable, visited states */
define U ← ⊥ as empty stack      /* states to visit */
define δ' ← ∅ as set
push(ι, U)
while U ≠ ⊥ do
  s ← top(U)
  δ' ← δ' ∪ {t ∈ δ | ∃θ ∈ C(Σ)^→ : ∃r ∈ S : t = (s, θ, r)}
  if {r ∈ S | ∃θ ∈ C(Σ)^→ : (s, θ, r) ∈ δ} ⊆ R then
    pop(U)
  else
    let s' ∈ {r ∈ S | ∃θ ∈ C(Σ)^→ : (s, θ, r) ∈ δ} \ R be
    arbitrary
    push(s', U)
    R ← R ∪ {s'}
  end if
end while
return (Σ, R, ι, δ')
```

Removing the unreachable states from a component again results in a component. Thus, the reactivity property is not lost by removing unreachable states.

Theorem 4. *Let $A = (\Sigma, X, S, \iota, \delta)$ be a component with channel signature $\Sigma = (I, O)$ and let $R = reach(A)$ denote the reachable states of A. Then, $B \stackrel{\text{def}}{=} (\Sigma, R, \iota, \delta \cap R \times C(\Sigma)^{\rightarrow} \times R)$ is a component.*

Proof. Let A and B be given as above and let $\Delta = \delta \cap R \times C(\Sigma)^{\rightarrow} \times R$ denote the transitions of B.

We need to show that B is reactive: Let $r \in R$ be a state of B. As $r \in R$ is a reachable state in A, it clearly holds that each target state of any transition in A with source state r is also an element of R, *i.e.,* $\forall u \in S : (\exists \theta \in C(\Sigma)^{\rightarrow} : s \xrightarrow{\theta} u) \Rightarrow u \in R$. Thus, we have that $\{(s, \theta, t) \in \delta \mid s = r\} \subseteq R \times C(\Sigma)^{\rightarrow} \times R$. As further $\{(s, \theta, t) \in \delta \mid s = r\} \subseteq \delta$, it holds that $\{(s, \theta, t) \in \delta \mid s = r\} \subseteq \delta \cap R \times C(\Sigma)^{\rightarrow} \times R$. This is equivalent to $\forall t \in S : r \xrightarrow{\theta}_{\delta} t \Rightarrow r \xrightarrow{\theta}_{\Delta} t$. As A is reactive and each transition of A starting from a reachable state $r \in R$ is also a transition of B, we obtain that B is also reactive. □

Therefore, the resulting from trimming a component is again an equivalent component that uses less space than the original. This eases analyses of the original component's behaviors.

4.2. Composition of TSCAs

As for TSSPFs, causality expresses the dependency between the inputs and outputs of a TSCA. A TSCA's output in time t must be completely determined by its input until time t. Thus it cannot change messages sent in the past and cannot predict messages it receives in the future (*cf.* pulse-drivenness in [16]):

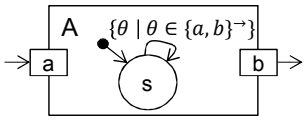


Figure 9: A strongly causal TSCA A that permits every possible output in reaction to every possible input.

Definition 11 (Weakly Causal TSCA)**.** *A TSCA A with $\Sigma_A = (I, O)$ is called weakly causal iff*

$\forall i, i' \in I^{\Omega} : \forall t \in \mathbb{N} : i{\downarrow_t} = i'{\downarrow_t} \Rightarrow A[i]{\downarrow_t} = A[i']{\downarrow_t}$.

Weak causality states that for every two inputs i, i' having a common prefix of length t and for every behavior $\alpha \in A[i]$ there is a behavior $\beta \in A[i']$ having a common prefix of length t with α. Similar as for TSSPFs, weak causality can lead to composition complications, which are avoidable analogously.

Definition 12 (Strongly Causal Modulo)**.** *Let A be a TSCA with channel signature $\Sigma_A = (I, O)$ and let $J \subseteq I$ and $P \subseteq O$ be two sets of input and output channels of A. The TSCA A is called strongly causal modulo (J, P) iff*

$\forall i, i' \in I^{\Omega} : \forall t \in \mathbb{N} :$

$((i|_J){\downarrow_t} = (i'|_J){\downarrow_t} \wedge i|_{I \setminus J} = i'|_{I \setminus J}) \Rightarrow (A[i]|_P){\downarrow_{t+1}} = (A[i']|_P){\downarrow_{t+1}}$.

Intuitively, a TSCA is strongly causal with respect to (J, P), if its outputs on the channels in P until time $t+1$ are not influenced by its inputs on the channels in J after time t.

Example 13 (Strongly Causal Modulo: $TSCA_{CBC}$)**.** *The TSCA $TSCA_{CBC}$, for instance, is not strongly causal with respect to $(\{r\}, \{v\})$. This is simple to show by contradiction: Let $in = \{r \mapsto \top^{\infty}, i \mapsto \top^{\infty}\} \in I^{\Omega}_{CBC}$ and $in' = \{r \mapsto \varepsilon^{\infty}, i \mapsto \top^{\infty}\} \in I^{\Omega}_{CBC}$ be two input histories. As $(in|_{\{r\}}){\downarrow_0} = (in'|_{\{r\}}){\downarrow_0} = \{r \mapsto \langle\rangle\}$ and $in|_{\{i\}} = in'|_{\{i\}} = \{i \mapsto \top^{\infty}\}$, the premises of the implication in Definition 12 hold for the chosen input histories and time $t = 0$. But as $(TSCA_{CBC}[in]|_{\{v\}}){\downarrow_1} = \langle\{v \mapsto \varepsilon\}\rangle$ and $(TSCA_{CBC}[in']|_{\{v\}}){\downarrow_1} = \langle\{v \mapsto \top\}\rangle$, the conclusion is not satisfied. Thus, the property stated in Definition 12 does not hold and $TSCA_{CBC}$ is not strongly causal modulo $(\{r\}, \{v\})$.*

At first sight, it might seem that a TSCA is strongly causal if, and only if, it always delays it's outputs. However, delaying of outputs is only a sufficient, not a necessary condition for a TSCA to be strongly causal. This holds because a TSCA A might simultaneously model a realization that is not strongly causal and another realization that is strongly causal, *i.e.,* a deterministic strongly causal component that only exhibits behaviors that are also possible in A. An example TSCA modeling arbitrary behavior illustrates this situation:

Example 14 (Arbitrary Behavior is Strongly Causal)**.** *Let $a, b \in C$ be two channels over Boolean values, i.e., $type(a) = type(b) = \{\varepsilon, \bot, \top\}$. Further, let $e \in C$ be a channel that only permits the empty message, i.e., $type(e) = \{\varepsilon\}$. We define the reactive TSCA A as illustrated in Figure 9 that is able to react with every possible output to every possible input as follows: $\Sigma_A = (\{a\}, \{b\})$, $X_A = e$, $S_A = \{s\}$, $\iota_A = s$,*

$\delta_A = \{(s, \theta, s) \mid \theta \in \{a, b\}^{\rightarrow}\}$ *where* $s = \{e \mapsto \varepsilon\}$. *It is easy to proof by induction that A is strongly causal modulo* (I_A, O_A) *because A permits every possible output in reaction to every possible input. Intuitively, this holds because when interpreting A as specification, we can find a strongly causal implementation I (a reactive deterministic component) that implements A, i.e.,* $behs(I) \subseteq behs(A)$. *An example for I is a TSCA that always outputs* ε *via channel b, independent of the input on channel a.*

TSCAs communicate with each other via their input and output channels. Multiple automata may read from the same channel, whereas only one automaton is permitted to write messages on a channel. Thus, no merging of messages on channels emitted by different automata is necessary.

Definition 13 (Compatible Channel Signatures)**.** *Two channel signatures* $\Sigma_A = (I_A, O_A)$ *and* $\Sigma_B = (I_B, O_B)$ *are called compatible iff* $O_A \cap O_B = \emptyset$.

By composing two TSCAs, the output channels of one automaton are connected to the input channels with the same name of the other automaton. The connected input channels are hidden implicitly. The set of output channels of the new automaton is the union of the sets of the output channels of the two original TSCAs. The input channels of the new automaton are the input channels of the two automata that do not share a common name with the output channels of the other automaton.

Definition 14 (Composition of Signatures)**.** *The composition of two channel signatures* $\Sigma_A = (I_A, O_A)$ *and* $\Sigma_B = (I_B, O_B)$ *is defined as* $\Sigma_A \otimes \Sigma_B \stackrel{\text{def}}{=} (I, O)$ *where* $I = (I_A \setminus O_B) \cup (I_B \setminus O_A)$ *and* $O = (O_A \cup O_B)$.

The composition of two TSCAs should be a TSCA that represents the behaviors of the TSCAs when they run in parallel. Therefore, we require the TSCAs participating in a composition must not share any internal variables (states).

Definition 15 (Compatible TSCAs)**.** *Two TSCAs A and B are called compatible iff* Σ_A *and* Σ_B *are compatible and* $X_A \cap X_B = \emptyset$.

Figure 10 illustrates the composition of two TSCAs A and B. The input channels of the compound $A \otimes B$ is the union of the input channels of A and B minus the union of the output channels of both TSCAs. The output channels of $A \otimes B$ are exactly the output channels of A and B. The composition of the TSCAs' states and transitions reflect the parallel execution of both TSPAs. The following formally defines the composition operator for TSCAs.

Definition 16 (Composition of TSCA)**.** *Let A and B be two compatible TSCAs. The composition of A and B is defined as the TSCA* $A \otimes B \stackrel{\text{def}}{=} (\Sigma, X, S, \iota, \delta)$ *where*

- $\Sigma = \Sigma_A \otimes \Sigma_B$,
- $X = X_A \cup X_B$,
- $S = \{s_A \cup s_B \mid s_A \in S_A \wedge s_B \in S_B\}$
- $\iota = \iota_A \cup \iota_B$

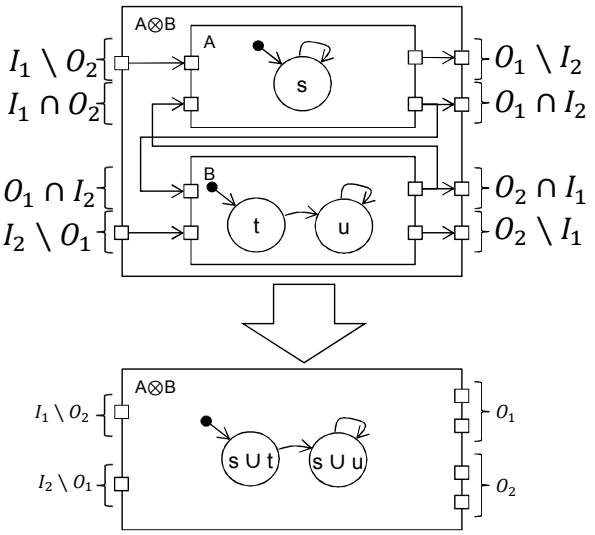


Figure 10: Composition of two compatible TSCAs.

- $\delta = \{(s, \theta, t) \in S \times C(\Sigma)^{\rightarrow} \times S \mid$
  $(s|_{S_A}, \theta|_{C(\Sigma_A)}, t|_{S_A}) \in \delta_A \wedge (s|_{S_B}, \theta|_{C(\Sigma_B)}, t|_{S_B}) \in \delta_B\}$

The union of the functions of S_A and S_B used in the definition of S (cf. Definition 16) is well defined since the the functions' domains X_A and X_B are disjoint.

Example 15 (Composition of two instances of $TSCA_{CBC}$)**.** *This example describes the composition of the TSCAs of the components* `pos0` *and* `pos1` *as depicted in Figure 3 (c). In MontiArcAutomaton, port names of different components may be equal and connectors establish channels between connected ports. In contrast, TSCAs communicate via shared channels. With this, a connector between two MontiArcAutomaton components describes a channel in the TSCA that formally describes the composed component's behaviors. Thus, the port names of the MontiArcAutomaton components have to be adjusted to achieve compatibility on TSCA level. We denote the TSCA of* `pos0` *by* CBC_0 *and the TSCA of* `pos1` *by* CBC_1. *The two TSCAs as well as their compound are depicted in Figure 11.*

They are defined by $CBC_0 = ((I_0, O_0), S_0, X_0, \iota_0, \delta_0)$ *and* $CBC_1 = ((I_1, O_1), S_1, X_1, \iota_1, \delta_1)$ *with*

- *input channels* $I_0 = \{i, r\}$ *and* $I_1 = \{q_0, r\}$ *where* $type(c) = \{\varepsilon, \top\}$ *for all* $c \in I_0 \cup I_1$,
- *output channels* $O_0 = \{x_0, q_0\}$ *and* $O_1 = \{q_1, x_1\}$ *where* $type(c) = \{\varepsilon, \top\}$ *for all* $c \in O_0 \cup O_1$,
- *internal channels* $X_0 = \{state_0\}$ *and* $X_1 = \{state_1\}$ *where* $type(state_0) = type(state_1) = \{0, 1\}$,
- *states* $S_0 = \{s_0, s_1\}$ *and* $S_1 = \{t_0, t_1\}$ *where* $s_i = \{state_0 \mapsto i\}$ *and* $t_i = \{state_1 \mapsto i\}$ *for all* $i \in \{0, 1\}$,
- *initial states* $\iota_0 = s_0$ *and* $\iota_1 = t_0$,

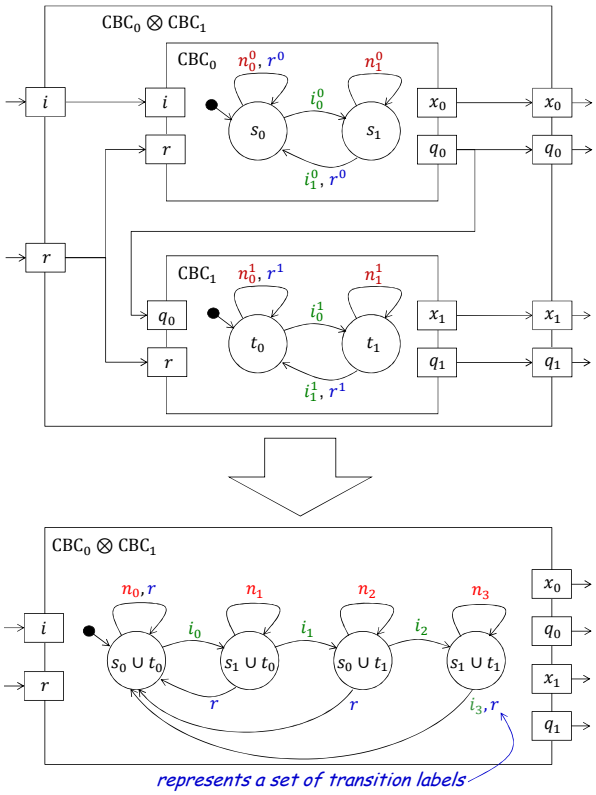


Figure 11: Composition of two CBC instances.

- *transition relations as depicted in the top part of Figure 11 where the transition labels of CBC_0 are defined as:*

  $n_0^0 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \varepsilon\}$,

  $n_1^0 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon\}$,

  $i_0^0 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon\}$,

  $i_1^0 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \top\}$,

  $r^0 = \{\theta \in (I_0 \cup O_0)^{\rightarrow} \mid \theta(r) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(q_0) = \varepsilon\}$,

  and the transition labels of CBC_1 are defined as:

  $n_0^1 = \{q_0 \mapsto \varepsilon, r \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $n_1^1 = \{q_0 \mapsto \varepsilon, r \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_0^1 = \{q_0 \mapsto \top, r \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_1^1 = \{q_0 \mapsto \top, r \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \top\}$,

  $r^1 = \{\theta \in (I_0 \cup O_0)^{\rightarrow} \mid \theta(r) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(q_1) = \varepsilon\}$.

The TSCAs CBC_0 and CBC_1 are compatible because the channel signatures are compatible ($O_0 \cap O_1 = \emptyset$) and the internal channels are pairwise disjoint $X_0 \cap X_1 = \{state_0\} \cap \{state_1\} = \emptyset$.

The composed TSCA $CBC_0 \otimes CBC_1$ is depicted in the lower part of Figure 11 and is formally given by $CBC_0 \otimes CBC_1 = (\Sigma, X, S, \iota, \delta)$ with

- *the channel signature $\Sigma = \Sigma_0 \otimes \Sigma_1 = (\{i, r\}, \{q_0, x_0, q_1, x_1\})$,*
- *internal channels $X = \{\{state_0\}, \{state_1\}\}$,*
- *states $S = \{s_{00}, s_{01}, s_{10}, s_{11}\}$, where*
  $s_{00} = \{\{state_0 \mapsto 0\}, \{state_1 \mapsto 0\}\}$,
  $s_{01} = \{\{state_0 \mapsto 0\}, \{state_1 \mapsto 1\}\}$,
  $s_{10} = \{\{state_0 \mapsto 1\}, \{state_1 \mapsto 0\}\}$, *and*
  $s_{11} = \{\{state_0 \mapsto 1\}, \{state_1 \mapsto 1\}\}$
- *the initial state $\iota = s_{00}$, and*
- *the transition relation as depicted in the bottom part of Figure 11 where the transition labels of $CBC_0 \otimes CBC_1$ are defined as:*

  $i_0 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $i_1 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \top, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_2 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $i_3 = \{i \mapsto \top, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \top, x_1 \mapsto \varepsilon, q_1 \mapsto \top\}$,

  $n_0 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $n_1 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \varepsilon, q_1 \mapsto \varepsilon\}$,

  $n_2 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \varepsilon, q_0 \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $n_3 = \{i \mapsto \varepsilon, r \mapsto \varepsilon, x_0 \mapsto \top, q_0 \mapsto \varepsilon, x_1 \mapsto \top, q_1 \mapsto \varepsilon\}$,

  $r = \{\theta \mid \theta(r) = \top \wedge \theta(q_0) = \theta(q_1) = \theta(x_0) = \theta(x_1) = \varepsilon\}$.

The result of this composition of two CBC components, i.e., two mod-2 counters, is a mod-4 counter. In this composed TSCA, all four states are reachable.

Components can block each other if they simultaneously require an input emitted by the other component to produce the next output. Composing such components results in a TSCA that is not reactive and therefore no component. However, there is a sufficient condition ensuring the resulting transition relation is reactive and the compound is a component.

Definition 17 (Composability of TSCAs)**.** *Two components A and B are called composable iff*

- *A and B are compatible and*
- *A is strongly causal with respect to $(I_A \cap O_B, I_B \cap O_A)$ or B is strongly causal with respect to $(I_B \cap O_A, I_A \cap O_B)$.*

Example 16 (Composability of $TSCA_{CBC}$)**.** *As shown in Example 15, the $TSCA_0$ of `pos0` and the $TSCA_1$ of `pos1` are compatible. To show composability between these, it is to show that $TSCA_0$ is strongly causal modulo $(I_0 \cap O_1, I_1 \cap O_0)$. This holds because $I_0 \cap O_1 = \emptyset$ and $I_1 \cap O_0 = \{q_0\}$: It is not possible that the messages emitted via an output channel of CBC_1 influence the behavior of CBC_0 because no output channel of CBC_1 is an input channel of CBC_0.*

The following theorem states that composing two composable components always results in a well-formed component.

Theorem 5. *If A and B are composable components, then the reachable part of $A \otimes B$ is a component.*

Proof. Analogous to proof of Theorem 3 in [16] by replacing the set the function i is chosen from with $I^{\rightarrow}$. □

Example 17 (The composition of two $TSCA_{CBC}$ is a component)**.** *Example 16 shows that the TSCAs CBC_0 of `pos0` and the CBC_1 of `pos1` are composable. Further, Example 10 proves that CBC_0 and CBC_1 are reactive, i.e., describe components. With Theorem 5, the composition of $TSCA_0$ and $TSCA_1$ is a component as it can be seen in Example 15.*

The composition operator further is commutative and associative. This guarantees the component resulting from composing several components is independent of the order in which the components are composed. Section 5.4 defines a notion of system architecture, which is well-defined because of the associativity and commutativity of the TSCA composition operator.

Theorem 6. *If A, B, and C are three pairwise compatible TSCAs, then the following holds:*

1. *$A \otimes B$ and C are compatible,*
2. *$A \otimes B = B \otimes A$, and*
3. *$(A \otimes B) \otimes C = A \otimes (B \otimes C)$.*

Proof. Let A, B, and C be three pairwise compatible TPSAs.

$A \otimes B$ and C are compatible: As A, B, and C are pairwise compatible, it holds that $X_A \cap X_B = X_A \cap X_C = X_B \cap X_C = \emptyset$. Thus, $X_{A\otimes B} \cap X_C = (X_A \cup X_B) \cap X_C = (X_A \cap X_C) \cup (X_B \cap X_C) = \emptyset$. As A, B, and C are pairwise compatible, it holds that Σ_A, Σ_B, and Σ_C are pairwise compatible and therefore $O_A \cap O_B = O_A \cap O_C = O_B \cap O_C = \emptyset$. Thus, it holds that $O_{A\otimes B} \cap O_C = (O_A \cup O_B) \cap O_C = (O_A \cap O_C) \cup (O_B \cap O_C) = \emptyset$. As $X_{A\otimes B} \cap X_C = O_{A\otimes B} \cap O_C = \emptyset$, $A \otimes B$ and C are compatible.

$A \otimes B = B \otimes A$: The set operations used in the definitions are all commutative. Commutativity for each part of the tuple follows directly by applying the sets' definitions.

$(A \otimes B) \otimes C = A \otimes (B \otimes C)$: Let $D = (A \otimes B) \otimes C$ and let $E = A \otimes (B \otimes C)$. As A, B, and C are all pairwise compatible, it holds by (1.) that $A \otimes B$ and C as well as A and $B \otimes C$ are compatible. The composition operator is therefore applicable for constructing D and E. Applying the operator, we obtain:

- $\Sigma_D = \Sigma_{A\otimes B} \otimes \Sigma_C$
  $= ((I_A \setminus O_B) \cup (I_B \setminus O_A), O_A \cup_B) \otimes \Sigma_C$
  $= (((I_A \setminus O_B) \cup (I_B \setminus O_A)) \setminus O_C \cup I_C \setminus (O_A \cup O_B), O_A \cup O_B \cup O_C)$
  $= (I_A \setminus (O_B \cup O_C) \cup I_B \setminus (O_A \cup O_C) \cup I_C \setminus (O_A \cup O_B), O_A \cup O_B \cup O_C)$
  $= (I_A \setminus (O_B \cup O_C) \cup (I_B \setminus O_C \cup I_C \setminus O_B) \setminus O_A, O_A \cup O_B \cup O_C)$
  $= \Sigma_A \otimes ((I_B \setminus O_C) \cup (I_C \setminus O_B), O_B \cup O_C) = \Sigma_A \otimes (\Sigma_B \otimes \Sigma_C)$
  $= \Sigma_E$,
- $X_D = X_A \cup X_B \cup X_C = X_E$,
- $S_D = \{s_A \cup s_B \cup s_C \mid s_A \in S_A \wedge s_B \in S_B \wedge s_C \in S_C\} = S_E$,
- $\iota_D = \iota_A \cup \iota_B \cup \iota_C = \iota_E$,
- $\delta_D = \{(s, \theta, t) \mid (s|_{S_{A\otimes B}}, \theta|_{C(\Sigma_{A\otimes B})}, t|_{S_{A\otimes B}}) \in \delta_{A\otimes B} \wedge (s|_{S_C}, \theta|_{C(\Sigma_C)}, t_{S_C}) \in \delta_C\}$
  $= \{(s, \theta, t) \mid (s|_{S_A}, \theta|_{C(\Sigma_A)}, t_{S_A}) \in \delta_A \wedge (s|_{S_B}, \theta|_{C(\Sigma_B)}, t_{S_B}) \in \delta_B \wedge (s|_{S_C}, \theta|_{C(\Sigma_C)}, t_{S_C}) \in \delta_C\}$
  $= \{(s, \theta, t) \mid (s|_{S_{B\otimes C}}, \theta|_{C(\Sigma_{B\otimes C})}, t|_{S_{B\otimes C}}) \in \delta_{B\otimes C} \wedge (s|_{S_A}, \theta|_{C(\Sigma_A)}, t_{S_A}) \in \delta_A\} = \delta_E$.

□

There exists a "neutral element" with respect to the composition operator. We will use this TSCA to lift the composition operator to arbitrary finite sets of TSCAs.

Definition 18 (Neutral TSCA)**.** *The neutral TSCA is defined as the TSCA N where $\Sigma_N = (\emptyset, \emptyset)$, $X_N = \emptyset$, $S_N = \{\emptyset\}$, $\iota_N = \emptyset$, and $\delta_N = \{(\emptyset, \emptyset, \emptyset)\}$. The neutral TSCA has no channels and no variables. Its sole and initial state is the empty channel valuation $v \in \emptyset^{\rightarrow} = [\emptyset \to M] = \{\emptyset\}$. It consists of one transition looping from the initial state to itself with the empty channel valuation.*

It is possible to compose the neutral TSCA with any other TSCA. It is the neutral element with respect to composition.

Theorem 7. *Let A be an arbitrary TSCA. Then, the TSCA A and the neutral TSCA N are compatible and $A \otimes N = A = N \otimes A$.*

Proof. Let A be an arbitrary TSCA. It holds that $O_A \cap O_N = O_A \cap \emptyset = \emptyset$. Thus Σ_A and Σ_N are compatible. As further $X_A \cap X_N = X_A \cap \emptyset = \emptyset$, we can conclude that A and N are compatible. The composition of A and N is $A \otimes N = (\Sigma, X, S, \iota, \delta)$ where $\Sigma = ((I_A \setminus \emptyset) \cup (\emptyset \setminus O_A), O_A \cup \emptyset) = (I_A, O_A)$, $X = X_A \cup \emptyset = X_A$, $S = \{s_A \cup s_B \mid s_A \in S_A \wedge s_B \in \{\emptyset\}\} = S_A$, $\iota = \iota_A \cup \emptyset = \iota_A$, $\delta = \{(s, \theta, t) \mid s|_{S_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t|_{S_A} \wedge s|_{\{\emptyset\}} \xrightarrow{\theta|_{\emptyset}}_{\delta_N} t|_{\{\emptyset\}}\}$. As $s|_{\{\emptyset\}} \xrightarrow{\theta|_{\emptyset}}_{\delta_N} t|_{\{\emptyset\}}$ holds for each $\theta \in C(\Sigma)^{\rightarrow}$, the above is equal to $\{(s, \theta, t) \mid s|_{S_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t|_{S_A}\} = \delta_A$. Hence, $A \otimes N = A$. By commutativity of $\otimes$ (cf. Theorem 6), we obtain $A = N \otimes A$. □

Theorem 6 guarantees that the TSCA resulting from composing several pairwise compatible TSCAs is independent of the order in which the TSCAs are composed. Theorem 7 shows that the neutral TSCA is a neutral element with respect to TSCA composition. We therefore lift the TSCA composition operator to the unique function $\bigotimes$ that takes a finite set of pairwise compatible TSCAs as input and outputs their composition under the operator $\otimes$ as usual, *i.e.,* $\bigotimes$ satisfies $\bigotimes \emptyset = N$ and $\bigotimes\{c\} = c$ for all TSCAs c and $\bigotimes(A \cup B) = (\bigotimes A) \otimes (\bigotimes B)$ for all all finite sets of TSCAs A and B such that $A \cap B = \emptyset$ and the TSCAs in $A \cup B$ are pairwise compatible. The operator is well-defined because of the properties stated in Theorem 6 and Theorem 7.

Naively applying the construction given in Definition 16 may cause the compound to consist of many unreachable states. Theorem 3 revealed that unreachable states can be safely removed from a TSCA without changing its behaviors. Unreachable states thus do not contribute to a TSCA's behavior. To defer a state explosion that occurs when composing several TSCAs with each other, adding unreachable states to TSCAs during a composition procedure should be avoided. Algorithm 2 depicts

an algorithm that takes two finite and composable TSCAs as input and outputs the trimmed TSCAs' compound. The algorithm performs a breadth-first search starting in the initial state of the compound. For each state determined as reachable, the algorithm calculates all transitions possible in the compound originating from the reachable state and checks whether the transitions' target has not been visited. In case the latter is true, the algorithm adds the state not yet visited to the set of states that are still to visit and proceeds as above.

Algorithm 2 Joined composition and trimming of finite TSCAs

Input: Two Finite and compatible TSCAs A and B
Output: Trimmed composition of A and B

```
define ι = ι_A ∪ ι_B as tuple                    /* initial state */
define δ ← ∅ as set                              /* transitions */
define R ← ∅ as set                              /* visited states */
define U ← ⊥ as empty stack                      /* states to visit */
push(ι, U)
while U ≠ ⊥ do
  s ← top(U)
  pop(U)
  R ← R ∪ {s}
  for all (u1, θ1, v1) ∈ {t ∈ δ_A | ∃u : ∃θ : (s|_A, θ, u) = t} do
    for all (u2, θ2, v2) ∈ {t ∈ δ_B | ∃u : ∃θ : (s|_B, θ, u) = t} do
      if θ1|_{C(Σ1)∩C(Σ2)} = θ2|_{C(Σ1)∩C(Σ2)} then
        define θ ← θ1 + θ2 as channel valuation
        δ ← δ ∪ {(s, θ, v1 ∪ v2)}
        if (v1 ∪ v2) ∉ R then
          push(v1 ∪ v2, U)
        end if
      end if
    end for
  end for
end while
return (Σ1 ⊗ Σ2, R, ι, δ)
```

Composition preserves I/O-determinism. This fact is important, because the size of the compound from composing several TSCAs is exponential in the number of the composed TSCAs. Thus, using the fact greatly reduces the complexity of determining whether a compound is I/O-deterministic if all the composition's participants are already I/O-deterministic. Section 5 describes the importance of I/O-determinism in detail: I/O-deterministic TSCAs induce a special structure when transforming them to Büchi automata, *i.e.,* the Büchi automata are always deterministic and weak, which enables to apply a simple complementation procedure.

Theorem 8. *If A and B are two I/O-deterministic and compatible TSCAs, then $A \otimes B$ is an I/O-deterministic TSCA.*

Proof. Let A and B be two I/O-deterministic and composable TSCAs. Let $A \otimes B = (\Sigma, X, S, \iota, \delta)$ denote the composition of A and B where $\Sigma = \Sigma_A \otimes \Sigma_B = (I, O)$. We need to show that $A \otimes B$ is I/O-deterministic. Suppose towards a contradiction that $A \otimes B$ is not I/O-deterministic. Then there exists a state $s \in S \subseteq X^{\rightarrow} = (X_A \cup X_B)^{\rightarrow}$ and a channel valuation $\theta \in C(\Sigma)^{\rightarrow}$ such that $|\{t \in S \mid s \xrightarrow{\theta}_{\delta} t\}| > 1$. This guarantees there exist $t, t' \in S$ with $t|_{X_A} \in S_A$ and $t|_{X_B} \in S_B$ and $t'|_{X_A} \in S_A$ and $t'|_{X_B} \in S_B$ such that $t \neq t'$ and $s \xrightarrow{\theta}_{\delta} t$ and $s \xrightarrow{\theta}_{\delta} t'$. By definition of composition for TSCAs we have that the following holds:

$s|_{X_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t|_{X_A}$ and $s|_{X_A} \xrightarrow{\theta|_{C(\Sigma_A)}}_{\delta_A} t'|_{X_A}$ and $s|_{X_B} \xrightarrow{\theta|_{C(\Sigma_B)}}_{\delta_B} t|_{X_B}$ and $s|_{X_B} \xrightarrow{\theta|_{C(\Sigma_B)}}_{\delta_B} t'|_{X_B}$. Since $t \neq t'$, it holds that $t|_{X_A} \neq t'|_{X_A}$ or $t|_{X_B} \neq t'|_{X_B}$. The case $t|_{X_A} \neq t'|_{X_A}$ stands in contradiction to the assumption that A is I/O-deterministic, as this would imply $|\{t \in S_A \mid s|_{X_A} \xrightarrow{\theta|_{C(\Sigma_A)}} t\}| \geq 2$. Similarly, the case $t|_{X_B} \neq t'|_{X_B}$ stands in contradiction to the assumption that B is I/O-deterministic. □

Example 18 (The composition of two $TSCA_{CBC}$ instances is I/O-deterministic)**.** *Theorem 8 guarantees that the composition of CBC_0 and CBC_1 as depicted in Example 15 is I/O-deterministic, because CBC_0 and CBC_1 are I/O-deterministic and compatible. We will now reconsider this according to the proof of Theorem 8. If $CBC_0 \otimes CBC_1$ was not I/O-deterministic, the composition would have to have two transitions with the same channel valuation from a single state s to at least two other states t and t' (with $t \neq t'$). The fact that t and t' are different implies that the restrictions of t and t' to the internal variables of CBC_0 are different or the restrictions to the internal variables of CBC_1 are different. Therefore, in CBC_0 or CBC_1 there must be a transition from one source state to at least two different target states that have the same channel valuation. This is a contraction to the assumption that both CBC_0 and CBC_1 are I/O-deterministic.*

The behaviors of a compound $A \otimes B$ are all behaviors over $C(\Sigma_{A\otimes B})$ that are possible in A when restricted to the channels of A and possible in B when restricted to the channels of B. Section 5.4 later uses this fact in Theorem 18 to show that refinement of TSCAs is compatible with composition. This is an important property, which enables independent development of different system parts. The following formalizes this property.

Theorem 9. *Let A and B be two compatible TSCAs and let $C \stackrel{\text{def}}{=} A \otimes B$. It holds that $behs(C) = \{\alpha \in C(\Sigma_C)^{\infty} \mid \alpha|_{C(\Sigma_A)} \in behs(A) \wedge \alpha|_{C(\Sigma_B)} \in behs(B)\}$.*

Proof. Let A, B, and C be given as above.

$\subseteq$: Let $\alpha \in behs(C)$ and let $\sigma = s_0, \theta_1, s_1, \theta_2, s_2, ...$ be an execution of C such that $beh(\sigma) = \alpha$. By definition of execution its holds that $s_{j-1} \xrightarrow{\theta_j}_{\delta_C} s_j$ for all $j > 0$ and $s_0 = \iota_C$. By definition of composition it holds that $s_{j-1}|_{X_A} \xrightarrow{\theta_j|_{C(\Sigma_A)}}_{\delta_A} s_j|_{C(\Sigma_A)}$ and $s_{j-1}|_{X_B} \xrightarrow{\theta_j|_{C(\Sigma_B)}}_{\delta_B} s_j|_{C(\Sigma_B)}$ for all $j > 0$.

Further it holds that $s_0|_{X_A} = \iota_C|_{X_A} = (\iota_A \cup \iota_B)|_{X_A} = \iota_A$ and $s_0|_{X_B} = \iota_C|_{X_B} = (\iota_A \cup \iota_B)|_{X_B} = \iota_B$ because ι_A and ι_B are disjoint. We can conclude $\sigma_A \stackrel{\text{def}}{=} s_0|_{X_A}, \theta_1|_{C(\Sigma_A)}, s_1|_{X_A}, \theta_2|_{C(\Sigma_A)}, s_0|_{X_A}, ... \in execs(A)$ is an execution of A and $\sigma_B \stackrel{\text{def}}{=} s_0|_{X_B}, \theta_1|_{C(\Sigma_B)}, s_1|_{X_B}, \theta_2|_{C(\Sigma_B)}, s_0|_{X_B}, ... \in execs(B)$ is an execution of B. This implies $beh(\sigma_A) = \theta_1|_{C(\Sigma_A)}, \theta_2|_{C(\Sigma_A)}, ... \in behs(A)$ is a behavior of A and $beh(\sigma_B) = \theta_1|_{C(\Sigma_B)}, \theta_2|_{C(\Sigma_B)}, ... \in behs(B)$ is a behavior of B. We can observe that $beh(\sigma_A) = beh(\sigma)|_{C(\Sigma_A)} = \alpha|_{C(\Sigma_A)}$ and $beh(\sigma_B) =$

$beh(\sigma)|_{C(\Sigma_B)} = \alpha|_{C(\Sigma_B)}$. Thus, $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $\alpha|_{C(\Sigma_B)} \in behs(B)$.

$\supseteq$: Let $\alpha = \theta_1, \theta_2, ... \in C(\Sigma_C)^\infty$ such that $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $\alpha|_{C(\Sigma_B)} \in behs(B)$. Let $\sigma_A = s_0^A, \theta_1^A, s_1^A, \theta_2^A, s_2^A, ...$ be an execution of A such that $beh(\sigma_A) = \alpha|_{C(\Sigma_A)}$ and let $\sigma_B = s_0^B, \theta_1^B, s_1^B, \theta_2^B, s_2^B, ...$ be an execution of B such that $beh(\sigma_B) = \alpha|_{C(\Sigma_B)}$. By definition of execution it holds that $s_{j-1}^A \xrightarrow{\theta_j^A} s_j^A$ and $s_{j-1}^B \xrightarrow{\theta_j^B} s_j^B$ for all $j > 0$. As $\theta_i^A = \theta_i|_{C(\Sigma_A)}$ and $\theta_i^B = \theta_i|_{C(\Sigma_B)}$ for all $j > 0$, it holds that $s_{j-1}^A \xrightarrow{\theta_j|_{C(\Sigma_A)}} s_j^A$ and $s_{j-1}^B \xrightarrow{\theta_j|_{C(\Sigma_B)}} s_j^B$ for all $j > 0$. Using the definition of TSCA composition, we obtain $((s_{j-1}^A \cup s_{j-1}^B), \theta_j, (s_j^A \cup s_j^B)) \in \delta_C$ for all $j > 0$. As additionally $\iota_C = \iota_A \cup \iota_B = s_0^A \cup s_0^B$, it holds that $\sigma \stackrel{\text{def}}{=} (s_0^A \cup s_0^B), \theta_1, (s_1^A \cup s_1^B), \theta_2, (s_2^A \cup s_2^B), ... \in execs(C)$ is an execution of C. Observing that $beh(\sigma) = \alpha$, we can conclude $\alpha \in behs(C)$. □

Hiding is an important concept to achieve modularity. The channels present in the compound resulting from the composition of several other TSCAs is always the union of the output channels of the composed TSCAs. For specifying software architectures, it is often necessary to hide several output channels to the environment. This is, for example, useful to hide unnecessary information not relevant to the architecture's environment or to explicitly hide "secret" information. Hidden channels become internal channels of the compound. For example, the bottom architecture depicted in Figure 3 illustrates this: the output channel `q` of component `pos2` is not part of the interface of component `Mod8Counter`. It is hidden from the environment, *i.e.,* the TSCA representing the `Mod8Counter` is restricted to the output channels $\mathtt{x_0}$, $\mathtt{x_1}$, and $\mathtt{x_2}$.

Definition 19 (TSCA Channel Restriction)**.** *Let A be a TSCA and let $O \subseteq O_A$ be a set of output channels of A. The restriction of A to the channels in O is defined as the TSCA $A{\restriction}O = (\Sigma, X_A, S_A, \iota_A, \delta)$ where $\Sigma = (I_A, O)$ and $\delta = \{(s, \theta, t) \in S_A \times C(\Sigma)^\rightarrow \times S_A \mid \exists (u, \theta', v) \in \delta_A : u = s \wedge v = t \wedge \theta'|_{C(\Sigma)} = \theta\}$.*

The set of output channels in $A{\restriction}O$ is restricted to the channels in O. $A{\restriction}O$ has the same input channels, internal variables, and states as A. The TSCA $A{\restriction}O$ contains a transition for each transition of A where the transition's channel valuation is restricted to the channels present in $A{\restriction}O$.

Example 19 (Restriction of CBC_0)**.** *Example 15 describes the TSCA $CBC_0 = ((I_0, O_0), S_0, X_0, \iota_0, \delta_0)$. The restriction $CBC_0{\restriction}\{x_0\}$ of CBC_0 to the set of its output channels $\{o\}$ is depicted in Figure 12. It is defined as $CBC_0{\restriction}\{x_0\} = (\Sigma, S_0, X_0, \iota_0, \delta)$ where $\Sigma = (\{i, r\}, \{x_0\})$ with transition relation δ as depicted in Figure 12. Each individual transition label is restricted to the channels of $I_0 \cup \{x_0\} = \{i, r\} \cup \{x_0\}$.*

4.3. TSSPF semantics of TSCAs

This section defines the semantics of TSCAs by sets of TSSPFs and reveals an important relation between the composition operators: the semantics of the syntactic composition of two TSCAs A and B is equal to the composition of the semantics of the individual automata.

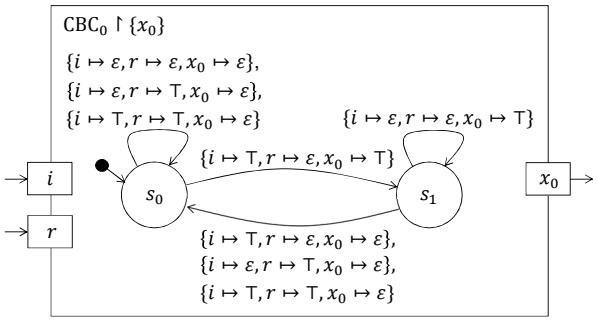


Figure 12: Graphical representation of the TSCA $CBC_0{\restriction}\{x_0\}$.

Definition 20 (TSSPF Semantics of a TSCA)**.** *The TSSPF semantics $[\![A]\!]$ of a TSCA $A = (\Sigma, X, S, \iota, \delta)$ with channel signature $\Sigma = (I, O)$ is defined as follows:*

$$[\![A]\!] \stackrel{\text{def}}{=} \{f \in [I^\Omega \xrightarrow{wc} O^\Omega] \mid \forall i \in I^\Omega : \exists \alpha \in behs(A) : i = h_\alpha|_I \wedge f(i) = h_\alpha|_O\}$$

Example 20 (TSSPF Semantics of CBC_0)**.** *The TSSPF semantics $[\![CBC_0]\!]$ of the TSCA $CBC_0 = ((I_0, O_0), S_0, X_0, \iota_0, \delta_0)$ (cf. Example 15) contains a single function f because CBC_0 is a deterministic component. For example, the function f maps the input communication history $h_I \in I_0^\Omega$ that satisfies $h(i).t = h(r).t = \varepsilon$ for all $t \in \mathbb{N}$ to the output channel history $h_O \in O_0^\Omega$ that satisfies $h_O(x_0).t = h_O(q_0).t = \varepsilon$ for all $i \in \mathbb{N}$. This holds because there exists a behavior $\alpha \in behs(CBC_0)$ (with execution looping in the initial state forever), which satisfies $\alpha.t(i) = \alpha.t(r) = \alpha.t(x_0) = \alpha.t(q_0) = \varepsilon$ for all $t \in \mathbb{N}$.*

For each behavior of a component, the semantics contain a function that maps inputs to outputs as encoded by the history induced by the behavior. Thus, no behavior is lost in the semantic mapping.

Theorem 10. *Let A be a component. For each $\alpha \in behs(A)$ there is a function $f \in [\![A]\!]$ such that $f(h_\alpha|_I) = h_\alpha|_O$.*

Proof. Analogous to proof of Theorem 11 in [16] by replacing the definition of maximality with $\forall i \in I^\Omega : i \in S|_I$. □

The semantics of components are well-formed, *i.e.,* components specify component semantics describing sets of TSSPFs.

Theorem 11. *The semantics $[\![A]\!]$ of a component A is component semantics describing.*

Proof. Analogous to proof of Theorem 12 in [16] by replacing the set the function f is chosen from with $[I^\Omega \xrightarrow{wc} O^\Omega]$. □

The semantics of the composition of two components is equal to the composition of their individual semantics:

Theorem 12. *For two composable components A and B with compatible signatures the following holds: $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$.*

Proof. Analogous to proof of Theorem 13 in [16] by replacing the applications of $[\![\cdot]\!]$ for PAs and $\otimes$ for SPFs by applications of the corresponding definitions for TSCAs and TSSPFs. □

An important implication of the theorem is that we can first syntactically compose the individual automata of an architecture and then perform analysis on the semantics of the automaton encoding the behavior of the whole system. This gives another basis for analysis that does not necessarily require to compose the semantics of the individual components of a system as, for example, done in [38]. The next sections introduce a method for semantic differencing of TSCAs and additionally shows that semantic differencing for finite I/O-deterministic TSCAs is possible in polynomial time. This paper further defines a notion of system architecture based on TSCAs. Afterwards, we introduce a method for mitigating the state explosion problem during semantic differencing of finite system architectures. In our previous work [9], we only considered semantic differencing for TSPAs in general and we did not introduce the notion of I/O-determinism. It is straightforward to transfer the results to TSCAs. The definition of system architecture as introduced in this paper is not possible with TSPAs as introduced in [9] because TSPAs do not have a commutative and associative composition operator.

5. Semantic Differencing of Component Behavior: From TSCAs to BAs

After introducing the notations for Büchi Automata (BAs) used in this paper, this section presents a theorem stating that there is a non-deterministic BA for each finite TSCA that accepts exactly the behaviors of the TSCA. Afterwards, we show that refinement checking and semantic difference witness generation for finite TSCAs can be reduced to language inclusion checking and counterexample generation for non-deterministic BAs. For finite I/O-deterministic TSCAs, semantic differencing can even be reduced to language inclusion checking for deterministic BAs, which is possible in polynomial time in the sizes of the automata.

5.1. Büchi Automata

Büchi Automata [3, 8] are a variant of finite automata that are acceptors for infinite words and thus induce languages consisting of infinite words. They are well known and much used in model checking. Infinite words over an alphabet Π are infinite sequences of symbols in Π.

Definition 21 (Büchi Automaton)**.** *A BA is a tuple* (Π, Q, I, F, δ) *where* Π *is a finite alphabet,* Q *is a finite set of states,* $I \subseteq Q$ *is a set of initial states,* $F \subseteq Q$ *is a set of accepting states, and* $\delta \subseteq Q \times \Pi \times Q$ *is the transition relation.*

For convenience we again sometimes write $s \xrightarrow{\sigma}_\delta t$ instead of $t \in \delta(s, \sigma)$ and simply $s \xrightarrow{\sigma} t$ if δ is clear from the context. Let $\mathcal{B} = (\Pi, Q, I, F, \delta)$ be a BA. The size of $\mathcal{B}$, denoted $|\mathcal{B}|$ is defined as the number of states and transitions in B, *i.e.,* $|\mathcal{B}| = |Q| + |\delta|$. A run of $\mathcal{B}$ on a word $w = \sigma_1, \sigma_2 ... \in \Pi^\infty$ starting in a state $q_0 \in Q$ is an infinite sequence $q_0, q_1, ...$ such that $q_{j-1} \xrightarrow{\sigma_j}_\delta q_j$ for all $j > 0$. A run $q_0, q_1, ...$ is accepting if $q_0 \in I$ and $q_i \in F$ for infinitely many $i > 0$. The accepted language of $\mathcal{B}$ is defined as $\mathcal{L}(\mathcal{B}) \stackrel{\text{def}}{=} \{w \in \Pi^\infty \mid \text{there exists an accepting run for } w \text{ in } \mathcal{B}\}$. The BA $\mathcal{B}$ is called *deterministic* iff $|I| \leq 1$ and $\forall q \in Q : \forall \sigma \in \Pi : |\{t \in S \mid s \xrightarrow{\sigma} t\}| \leq 1$. $\mathcal{B}$ is called *total* iff $|I| = 1$ and $\forall q \in Q : \forall \sigma \in \Pi : |\delta(q, \sigma)| = 1$. A BA $\mathcal{B} = (\Pi, Q, I, F, \delta)$ is called *weak* iff for all pairs of states $p, q \in Q$ belonging to the same strongly connected component it holds that p is accepting iff q is accepting. Deterministic weak BAs can be minimized in polynomial time [27]. This enables to efficiently minimize intermediate BA representations of an architecture to mitigate a state explosion during composition. In the general case, the minimization problem is PSPACE-complete for non-deterministic BAs [4, 21] and NP-complete for deterministic BAs [41]. Checking language inclusion between two arbitrary non-deterministic Büchi automata is PSPACE-complete [23], though decidable, in general. Although the computational complexity is large, several approaches for checking language inclusion and counterexample (diff witness) generation have been implemented and produce promising results in practice [3]. Checking language inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ is typically done in three steps by proving that there are no words in $\mathcal{L}(A)$, which are not included in $\mathcal{L}(B)$:

1. Construct a complementary automaton $\overline{B}$ of B that accepts exactly the words not accepted by B, *i.e.,* $\mathcal{L}(\overline{B}) = \Pi^\infty \setminus \mathcal{B}$.
2. Construct a Büchi automaton C that accepts exactly the words accepted by A and $\overline{B}$, *i.e.,* $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(\overline{B})$.
3. Check whether $\mathcal{L}(C) = \emptyset$, which is possible by examining whether C contains a reachable final state that is part of a cycle.

The computational hardness of checking language inclusion arises from constructing the BA $\overline{B}$ that might be exponentially larger than the BA B in the general case [24, 40]. However, in case B is deterministic, the BA $\overline{B}$ can be constructed in polynomial time in the size of B [25].

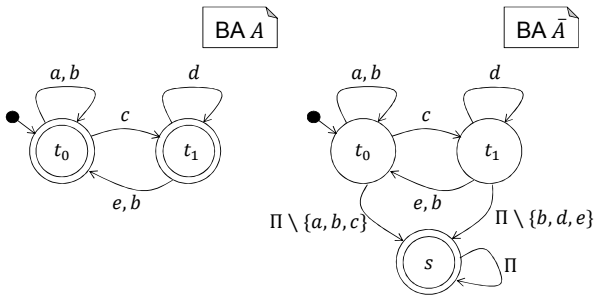


Figure 13: Two Büchi automata A and $\overline{A}$. The automaton $\overline{A}$ accepts the complementary language of A.

Example 21 (Büchi Automata)**.** *Figure 13 depicts two BAs A and* $\overline{A}$*. The BA A is formally defined by* $A = (\Pi, Q, I, F, \delta)$ *where*

- $\Pi = \{a, b, c, d, e\}$,
- $Q = \{t_0, t_1, s\}$,
- $I = \{t_0\}$,
- $F = \{t_0, t_1\}$, *and*
- $\delta = \{(t_0, a, t_0), (t_0, b, t_0), (t_0, c, t_1),$
  $(t_1, d, t_1), (t_1, e, t_0), (t_1, b, t_0)\}$.

The automaton $\overline{A}$ is defined analogously. The BA $\overline{A}$ accepts exactly the complementary language of A, i.e., it holds that $\mathcal{L}(\overline{A}) = \Pi^\infty \setminus \mathcal{L}(A)$. Both automata are deterministic and weak.

In the next section, we present a translation from finite TSCAs to BAs and thereby reduce semantic differencing and refinement checking for finite TSCAs to the language inclusion problem for Büchi automata. We show that the translation transforms a rather large subclass of TSCAs to BAs that can be complemented in polynomial time in the sizes of the resulting BAs. The subclass contains all finite I/O-deterministic TSCAs.

5.2. From TSCAs to BAs

In model-driven development, models are the primary engineering artifacts, *i.e.,* engineers (manually) create finite models to describe parts of the system under development. Hence, we consider semantic differencing and refinement checking for architectures where the individual components have a finite state space, communicate over finitely many communication channels, and where the types of messages emitted via component interfaces are finite. There exists a non-deterministic BA for each finite TSCA that accepts exactly the TSCA's behaviors.

The BA associated to a finite TSCA $A = (\Sigma, X, S, \iota, \delta)$ with $\Sigma = (I, O)$ is defined as $ba(A) \stackrel{\text{def}}{=} (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$. As the TSCA A is finite, the sets S, I, O, and δ are finite. As therefore $C(\Sigma)^{\rightarrow}$ is finite, $ba(A)$ is a well-defined BA. The size of $ba(A)$ is equal to the size of A. The following theorem shows that the language accepted by $ba(A)$ and the behaviors of A coincide.

Theorem 13. *For any finite TSCA A, it holds that $behs(A) = \mathcal{L}(ba(A))$.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ be a finite TSCA with channel signature $\Sigma = (I, O)$. Further let $ba(A) = (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$ be the BA associated to A.

$\subseteq$: Let $s_0, \theta_1, s_1, \theta_2, s_2, ... \in execs(A)$ be an execution of A. By definition of execution $s_{j-1} \xrightarrow{\theta_j} s_j$ for all $j > 0$ and $s_0 = \iota$. Thus, $s_0, s_1, s_2, ...$ is a run of $\mathcal{B}$ on the word $\theta_1, \theta_2, ...$ Since all states $s \in S$ are accepting, the run is accepting. Thus, $beh(s_0, \theta_1, s_1, \theta_2, s_2, ...) = \theta_1, \theta_2, ... \in \mathcal{L}(\mathcal{B})$.

$\supseteq$: Assume that $\sigma = \sigma_1, \sigma_2, \sigma_3, ... \in \mathcal{L}(\mathcal{B})$ and let $q_0, q_1, q_2, ...$ be an accepting run of $\mathcal{B}$ on σ. By definition of run we have $q_{j-1} \xrightarrow{\sigma_j} q_j$ for all $j > 0$. Thus $\tau = q_0, \theta_1, q_1, \theta_2, ...$ is an execution of A. Therefore, by definition of behavior we have that $beh(\tau) = \sigma_1, \sigma_2, ... \in behs(A)$ is a behavior of A. □

Example 22. *The BA $ba(CBC_0)$ associated to the finite TSCA CBC_0 (cf. Example 15) is equal to the BA A depicted in Figure 13 when assuming $a = n_0^0$, $b = r^0$, $c = i_0^0$, $d = n_1^0$, $e = i_1^0$.*

The following reveals a sufficient condition that guarantees the translation of a TSCA to its associated BA yields a deterministic BA. As language inclusion checking for deterministic BAs is possible in polynomial time [25], we obtain a method for efficiently determining if the set of behaviors of a TSCA is a subset of the behaviors of another I/O-deterministic TSCA.

Theorem 14. *The associated BA $ba(A)$ of each finite and I/O-deterministic TSCA A is deterministic.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ be a finite and I/O-deterministic TSCA and let $ba(A) = (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$ be the BA associated to A. The BA $ba(A)$ has a unique initial state. As the TSCA A is I/O-deterministic, it holds that $\forall s \in S : \forall \theta \in C(\Sigma)^{\rightarrow} : |\{t \in S \mid (s, \theta, t)\}| \leq 1$. This implies that $ba(A)$ is deterministic. □

Example 23 (The $ba(TSCA_{CBC})$ is deterministic)**.** *Example 10 shows that $TSCA_{CBC}$ is finite. Thus, the TSPAs state space S is also finite. $TSCA_{CBC}$ is I/O-deterministic, i.e., there is at most one state that the TSCA can change to, from a given source state and a given channel assignment. This follows from the fact that the TSCA is deterministic, which has been shown in Example 10 and the application of Theorem 2. According to the definition of BAs, a BA is deterministic if it has at most one initial state and for each state and for each input word, there is at most one state that the BA can change to. The BA $ba(TSCA_{CBC})$ has a single initial state and for each input, i.e., each channel assignment, there is only one transition from each state, because $TSCA_{CBC}$ is deterministic. With this (and the proof of Theorem 14), the constructed BA $ba(TSCA_{CBC})$ is deterministic.*

There exist non-deterministic BAs for which no deterministic BAs exist that accepts the same language. On the other hand, for each non-deterministic weak BA, there exists a deterministic weak BA that accepts the same language [27]. The translation from TSCAs to BAs always yields weak BAs, which can be determinized and minimized. Further, each deterministic and complete weak BA can be complemented in polynomial time by exchanging the automaton's sets of accepting and non-accepting states.

Theorem 15. *The associated BA $ba(A)$ of each finite TSCA A is weak.*

Proof. Let $A = (\Sigma, X, S, \iota, \delta)$ be a finite and I/O-deterministic TSCA and let $ba(A) = (C(\Sigma)^{\rightarrow}, S, \{\iota\}, S, \delta)$ be the BA associated to A. As every state in $ba(A)$ is accepting, it especially holds that each strongly connected component in $ba(A)$ solely contains accepting states. This implies that $ba(A)$ is weak. □

5.3. Semantic Differencing for Component Behavior

The semantics of components are defined as sets of TSSPFs. Each function $f \in [\![c]\!] \setminus [\![c']\!]$ in the semantics of one component c that is no member of the semantics of another component c' is a representative for the difference between the components' semantics. However, such a representative defines an output for each possible component input, even if the semantic difference is only given by a single input/output pair. Thus, such a TSSPF does not effectively reveal the differences between the

component semantics. In contrast, the exact input/output pairs for which there is a function in the semantics of one component that maps the input to the output and for which there is no function in the semantics of the other component mapping the input to the output clearly reveals a difference. If two components have different interfaces, *i.e.,* they read and write from and to different channels, each input/output pair of the first component indicates a difference to the semantics of the other component. However, if the components have channels of the same types one can easily avoid this problem by channel renaming and hiding [5]. Thus, we define the semantic difference for components having the same interfaces, only.

Definition 22 (Diff Witness)**.** *Let $F_1, F_2 \subseteq [I^\Omega \xrightarrow{wc} O^\Omega]$ be two sets of TSSPFs. A diff witness distinguishing F_1 from F_2 is a communication history $w \in (I \cup O)^\Omega$ satisfying $\exists f_1 \in F_1 : f_1(w|_I) = w|_O \wedge \forall f_2 \in F_2 : f_2(w|_I) \neq w|_O$.*

We denote by $\Delta(F_1, F_2)$ the set of all diff witnesses distinguishing F_1 from F_2.

A set of diff witnesses may be finite but is typically infinite and can thus not be completely enumerated.

Example 24 (Diff Witness)**.** *This example presents a diff witness between the $TSCA_{CBC} = (\Sigma_{CBC}, X_{CBC}, S_{CBC}, \iota_{CBC}, \delta_{CBC})$ and a modified version of it. The modified version $TSCA_{mod} = (\Sigma_{CBC}, X_{CBC}, S_{CBC}, \iota_{CBC}, \delta_{mod})$ has the same interface as $TSCA_{CBC}$ and a similar behavior – the only difference is that it does not emit $\top$ on the outgoing channel q if the state changes from b to a after an increase of the counted value. More technically, $\delta_{mod} = (\delta_{CBC} \setminus \delta_{ba}) \cup \delta_{ba'}$, where*
$\delta_{ba} = \{(b, \theta, a) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \varepsilon \wedge \theta(q) = \top\}$ and
$\delta_{ba'} = \{(b, \theta, a) \mid \theta(i) = \top \wedge \theta(r) = \varepsilon \wedge \theta(v) = \varepsilon \wedge \theta(q) = \varepsilon\}$

Let $in = \{r \mapsto \langle \varepsilon^\infty \rangle, i \mapsto \langle \top, \top, \varepsilon^\infty \rangle\} \in I^\Omega$ be an input history over the common interface of $TSCA_{CBC}$ and $TSCA_{mod}$. The input history describes two increase steps that change the state of the TSCA from a to b, back to a, and then remains in state a. For all $h \in TSCA_{CBC}[in|_{\{q\}}]$ and $h' \in TSCA_{mod}[in|_{\{q\}}]$, it holds that $h.1 = \top$, whereas $h'.1 = \varepsilon$. Therefore, for the given input history, the TSCAs produce different output histories.

We consider architectures where the whole system behavior can be mapped to a TSCA. The following theorem reveals the relation between the differences of the behaviors and of the semantics of TSCAs.

Theorem 16. *Let $A_1 = (\Sigma, S_1, \iota_1, \delta_1)$ and $A_2 = (\Sigma, S_2, \iota_2, \delta_2)$ with $\Sigma = (I, O)$ be two TSCAs and let $w \in (I \cup O)^\Omega$ be a communication history. The following holds: $w \in \Delta([\![A_1]\!], [\![A_2]\!]) \Leftrightarrow \exists \alpha \in behs(A_1) : w = h_\alpha \wedge \alpha \notin behs(A_2)$.*

Proof. Let A_1, A_2, and w be given as above.

$\Rightarrow$: Assume $w \in \Delta([\![A_1]\!], [\![A_2]\!])$ is a diff witness. By definition of Δ, we have that there is a function $f_1 \in [\![A_1]\!]$ such that $f_1(w|_I) = w|_O$ and $f(w|_I) \neq w|_O$ for all $f \in [\![A_2]\!]$. In the following let f_1 be such a function that satisfies the above. By definition of $[\![\cdot]\!]$ we have that $\forall i \in I^\Omega : \exists \alpha \in behs(A_1) : i = h_\alpha|_I \wedge f_1(i) = h_\alpha|_O$. When substituting $w|_I$ for i, we get that $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge f_1(w|_I) = h_\alpha|_O$. Since $f_1(w|_I) = w|_O$ we can substitute $w|_O$ for $f_1(w|_I)$ and obtain $\exists \alpha \in behs(A_1) : w|_I = h_\alpha|_I \wedge w|_O = h_\alpha|_O$, which is equivalent to $\exists \alpha \in behs(A_1) : w = h_\alpha$. In the following, let such an α with $w = h_\alpha$ be given. It remains to show $\alpha \notin behs(A_2)$. Towards a contradiction we assume $\alpha \in behs(A_2)$. By Theorem 10 we get there is a function $g \in [\![A_2]\!]$ such that $g(h_\alpha|_I) = h_\alpha|_O$. By definition of α we have $w = h_\alpha$ and thus $g(w|_I) = w|_O$. But since $w \in \Delta([\![A_1]\!], [\![A_2]\!])$, it holds that $\forall f \in [\![A_2]\!] : f(w|_I) \neq w|_O$. Substituting g for f yields a contradiction.

$\Leftarrow$: Assume there is an $\alpha \in behs(A_1)$ such that $w = h_\alpha$ and $\alpha \notin behs(A_2)$. Using Theorem 10 we get there is a function $f \in [\![A_1]\!]$ such that $f(h_\alpha|_I) = h_\alpha|_O$. By definition of w we have that $w = h_\alpha$ and thus obtain by substitution that $f(w|_I) = w|_O$. Thus there is a function $f \in [\![A_1]\!]$ such that $f(w|_I) = w|_O$. It remains to show that $g(w|_I) \neq w|_O$ for all $g \in [\![A_2]\!]$. Towards a contradiction we assume that there is a function $g \in [\![A_2]\!]$ such that $g(w|_I) = w|_O$. By definition of $[\![\cdot]\!]$ we get that $\forall i \in I^\Omega : \exists \beta \in behs(A_2) : i = h_\beta|_I \wedge g(i) = h_\beta|_O$. Substituting $w|_I$ for i we obtain $\exists \beta \in behs(A_2) : w|_I = h_\beta|_I \wedge g(w|_I) = h_\beta|_O$. Since by assumption $w|_I = h_\alpha|_I$ and $g(w|_I) = w|_O$ by definition of g, this is equivalent to $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge w|_O = h_\beta|_O$. By assumption we have $w = h_\alpha$ and thus obtain via substitution $\exists \beta \in behs(A_2) : h_\alpha|_I = h_\beta|_I \wedge h_\alpha|_O = h_\beta|_O$, which is equivalent to $\exists \beta \in behs(A_2) : h_\alpha = h_\beta$. Using the definitions of h_α and h_β, this is equivalent to $\exists \beta \in behs(A_2) : \alpha = \beta$, which is equivalent to $\alpha \in behs(A_2)$ and contradicts the assumption. □

In the previous section, we presented a translation from finite TSCAs to BAs. Each word accepted by a BA resulting from such a translation corresponds to a behavior of the input TSCA. Existing algorithms for checking language inclusion and counterexample generation for BAs can thus be used for refinement checking and diff witness generation of architectures as defined above: Given two TSCAs A_1 and A_2 we use the translation defined in Section 5.2 to obtain two Büchi automata $ba(A_1)$ and $ba(A_2)$ such that $\mathcal{L}(ba(A_1)) = behs(A_1)$ and $\mathcal{L}(ba(A_2)) = behs(A_2)$. Using Theorem 16 and Theorem 13 we can transform a word accepted by $ba(A_1)$ that is not accepted by $ba(A_2)$ to a corresponding diff witness that semantically distinguishes the automata A_1 and A_2. If A_2 is I/O-deterministic, the BA $ba(A_2)$ is deterministic and weak and can thus be easily complemented in polynomial time in the size of $\mathcal{B}_2$, which is equal to the size of A_2. Then, inclusion checking is possible in polynomial time in the sizes of $ba(A_1)$ and $ba(A_2)$.

5.4. Mitigating the State Explosion Problem When Applying Semantic Differencing to System Architectures

This section summarizes practical performance improvements to mitigate a state explosion during semantic differencing of system architectures consisting of multiple TSCAs. We first define an abstract notion of *system architecture* (SA) inspired by [33]. While [33] considers a black-box view on SAs, in this paper we assume a white-box view where component implementations are available. A SA consists of an interface observable by the system's environment given by a channel signature

and of finitely many components represented by TSCAs that are connected via their channels.

Definition 23 (System Architecture)**.** *A system architecture is a tuple $S = (\Sigma, C)$ where:*

- *$\Sigma = (I, O)$ is a channel signature,*
- *C is a finite non-empty set of pairwise compatible components,*
- *the channels of S exist in the composition of the TSCAs' channel signatures, i.e., $I = J$ and $O \subseteq P$ where $(J, P) = \bigotimes_{c \in C} \Sigma_c$ denotes the composition of the channel signatures of all TSCAs in C, and*
- *$(\bigotimes C)\restriction O$ is a component.*

S is called finite iff Σ is finite and each $c \in C$ is finite.

The channel signature Σ defines the SA's external interface. The set C consists of the SA's components. The channels encoded by the channel signature Σ are required to exist in the compound resulting from composing the SA's components. The last condition stating that $(\bigotimes C)\restriction O$ must be a component is the most abstract well-formedness rule guaranteeing the result from composing the architecture's components is a component itself. More restricting well-formedness rules implying that $(\bigotimes C)\restriction O$ is a component are also possible to describe more restricted SA subclasses. One example is to require each component $c \in C$ to be strongly causal with respect to all its channels. Another, more relaxed, example is to require each component $c \in C$ to be composable with each possible intermediate composition result $\bigotimes D$ for each $D \subseteq C \setminus \{c\}$. We omit the proofs showing that these two examples imply that $(\bigotimes C)\restriction O$ is a component. Each individual TSCA participating in a SA is interpreted as an atomic component, *i.e.,* is not considered to have any subcomponents. As the TSCAs' channel signatures must be pairwise compatible, multiple components may read from the same channel whereas only one component is permitted to write on a channel. The input channels of a SA are equal to the input channels of the TSCA resulting from the subcomponents' composition. The set of output channels must be a subset of the output of the TSCA resulting from the composition. With this, output channels not specified by the architecture are hidden to the environment.

Example 25 (System architecture of the `Mod8Counter`)**.** *This example presents the system architecture of the alternative representation of the `Mod8Counter`, depicted in Figure 5, as composition of the TSCAs of its subcomponents `pos0`, `pos1`, and `pos2`. The system architecture is $S_{Mod8b} = (\Sigma, C)$ with*

- *the channel signature $\Sigma_{Mod8} = (\{inc, res\}, \{x_0, x_1, x_2\})$ and*
- *the set of components $C = \{TSCA_{pos0}, TSCA_{pos1}, TSCA_{pos2}\}$.*

The input channel set of S_{Mod8b} is equal to the input channel set of the composition of the three TSCAs. The output channel set of S_{Mod8b} is a subset of the output channel set of the composition of the TSCAs in C. The output channel set of the composition of the TSCAs in C is $\{x_0, q_0, x_1, q_1, x_2, q_2\}$. Channels included in the set of output channels of the composition that are no elements of the set of output channels of the system architecture S_{Mod8b} are hidden. The composition $\bigotimes C$ is a component, as shown in Example 17. Intuitively, the restriction of this composition to the output channels O is also a component, because the restriction of output channels does not influence the TSCA's reactiveness. The system architecture is finite, because all $c \in C$ are finite (cf. Example 10) and Σ_{Mod8} is finite.

A system architecture's TSCA semantics is the result from restricting the channels of the compound resulting from composing the SA's components to the channels specified by the SA's interface. The behavior and TSSPF semantics are given by the behavior and TSSPF semantics of the TSCA semantics.

Definition 24 (TSCA, Behavior, and TSSPF Semantics of SAs)**.** *Let $S = (\Sigma, C)$ with $\Sigma = (I, O)$ be a SA. The TSCA semantics of S is defined as $tspa(S) = (\bigotimes C)\restriction O$. The behavior semantics of S is defined as $behs(S) \stackrel{\text{def}}{=} behs(tspa(S))$. The TSSPF semantics of S is defined as $[\![tspa(S)]\!]$.*

Composing SAs with each other is also possible as the TSCA semantics of a SA can be interpreted as a component, again.

In continuous architecting and especially in combination with agile software development methodologies, requirements typically change during system development. In case additional requirements are added or existing requirements are strengthened, underspecification in component behavior models typically needs to be restricted to adapt the current specification or implementation to match the additional requirements. The behavior of the system under development is said to be refined.

Definition 25 (Refinement)**.** *A TSCA A is called (behavior) refinement of a TSCA B, denoted $A \preceq B$, iff $\Sigma_A = \Sigma_B$ and $behs(A) \subseteq behs(B)$.*

Refinement is lifted to SAs: A SA S is called *refinement* of a SA S', denoted $S \preceq S'$, iff $tspa(S) \preceq tspa(S')$. As a refinement exhibits less behaviors as the original system, there cannot exist a diff witness distinguishing the refined system from the original one.

Theorem 17. *Let A and B be two TSCAs. If $A \preceq B$, then $\Delta([\![A]\!], [\![B]\!]) = \emptyset$.*

Proof. Let A and B be two TSCAs such that $A \preceq B$. Thus, it holds that $behs(A) \subseteq behs(B)$. Suppose towards a contradiction there exists a diff witness $w \in \Delta([\![A]\!], [\![B]\!]) \neq \emptyset$. Using Theorem 16, this implies there exists $\alpha \in behs(A)$ such that $w = h_\alpha$ and $\alpha \notin behs(B)$. This contradicts $behs(A) \subseteq behs(B)$. □

Example 26 (Refinement of the `Mod8Counter` system architecture)**.** *Consider the system architectures of the `Mod8Counter` as depicted in Figure 3 (a) with the TSCA specified in Appendix B and the system architecture as depicted in Figure 5. In the following, we will refer to the first as the system architecture S and to the latter as the system architecture S'. First, we will investigate if $S' \preceq S$ by showing that*

$tspa(S') \preceq tspa(S)$. Therefore, it must hold that $\Sigma_{S'} = \Sigma_S$ and $behs(tspa(S')) \subseteq behs(tspa(S))$. The first is satisfied, because both system architectures have the same channel signature $\Sigma_{S'} = \Sigma_S = (\{inc, res\}, \{x_0, x_1, x_2\})$. Further, it holds that $tspa(S') = (\bigotimes C_{S'})\restriction O_{S'} = (TSCA_{pos0} \otimes TSCA_{pos1} \otimes TSCA_{pos2})\restriction O_{S'}$ and $tspa(S) = (\bigotimes C_S)\restriction O_S = TSCA_{Mod8a}$. The result of $TSCA_{pos0} \otimes TSCA_{pos1} \otimes TSCA_{pos1}$ has been explained in Example 25. Due to the channel restriction, we have $tspa(S') = tspa(S)$ and therefore, $behs(tspa(S')) = behs(tspa(S))$ holds.

Behavior refinement is reflexive and transitive. More importantly, it is compatible with composition:

Theorem 18. *Let A, B, and C be TSCAs such that A and C are compatible and B and C are compatible. If $A \preceq B$, then $A \otimes C \preceq B \otimes C$.*

Proof. Let A, B, and C be given as above such that $A \preceq B$. Let $\alpha \in behs(A \otimes C)$. Using Theorem 9, this implies $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $\alpha|_{C(\Sigma_C)} \in behs(C)$. As $A \preceq B$, it holds that $behs(A) \subseteq behs(B)$. Thus, as $\alpha|_{C(\Sigma_A)} \in behs(A)$ and $behs(A) \subseteq behs(B)$, we obtain $\alpha|_{C(\Sigma_A)} \in behs(B)$. In summary, it holds that $\alpha|_{C(\Sigma_A)} \in behs(B)$ and $\alpha|_{C(\Sigma_C)} \in behs(C)$. Using Theorem 9, this implies $\alpha \in behs(B \otimes C)$. □

Refinement is also preserved by TSCA restriction.

Theorem 19. *Let A and B be TSCAs and let $O \subseteq O_B$. If $A \preceq B$, then $A\restriction O \preceq B\restriction O$.*

Proof. Let A and B be TSCAs and let $O \subseteq \Sigma_B$. Assume $A \preceq B$. By definition $A \preceq B$ it holds that $\Sigma_A = \Sigma_B$. Let $I \stackrel{\text{def}}{=} I_A = I_B$.

Let $A' \stackrel{\text{def}}{=} A\restriction O$ denote the restriction of A and let $B' \stackrel{\text{def}}{=} B\restriction O$ denote the restriction of B. As $\Sigma_A = \Sigma_B$, it especially holds that $\Sigma_{A'} = \Sigma_{B'}$. Let $\sigma = s_0, \theta_1, s_1, \theta_2, s_2, ... \in execs(A')$ be an execution of A. By definition of execution, it holds that $s_{j-1} \xrightarrow{\theta_j}_{\delta_{A'}} s_j$ for all $j > 0$. By definition of TSCA restriction, we have that $s_{j-1} \xrightarrow{\theta_j}_{\delta_{A'}} s_j$ is equivalent to $\exists (s^A_{j-1}, \theta^A_j, s^A_j) \in \delta_A : s^A_{j-1} = s_{j-1} \wedge s^A_j = s_j \wedge \theta^A_j|_{I\cup O} = \theta_j$ for each $j > 0$. Let such θ^A_j with $\theta^A_j|_{(I\cup O)} = \theta_j$ be given for each $j > 0$. As $s_{j-1} \xrightarrow{\theta^A_j}_{\delta_A} s_j$ for each $j > 0$, it holds by definition of execution that $\sigma_A \stackrel{\text{def}}{=} s_0, \theta^A_1, s_1, \theta^A_2, s_2, ... \in execs(A)$ is an execution of A. As $A \preceq B$, it holds that $beh(\sigma_A) \in behs(B)$. Therefore, there exists an execution $\sigma_B \in execs(B)$ of B such that $beh(\sigma_B) = beh(\sigma_A)$. Hence, there exist $s^B_0, s^B_1, s^B_2 ... \in S_B$ such that $\sigma_B = s^B_0, \theta^A_1, s^B_1, \theta^A_2, s^B_2, ... \in execs(B)$. This is by definition of execution equivalent to $(s^B_{j-1}, \theta^A_j, s^B_j) \in \delta_B$ for each $j > 0$. Using the TSCA restriction definition, this implies $(s^B_{j-1}, \theta^A_j|_{(I\cup O)}, s^B_j) \in \delta_{B'}$ for each $j > 0$. Thus, $\tau \stackrel{\text{def}}{=} s^B_0, \theta^A_1|_{(I\cup O)}, s^B_1, \theta^A_2|_{(I\cup O)}, s^B_2, ... \in execs(B')$ is an execution of B'. As by definition $\theta^A_j|_{(I\cup O)} = \theta_j$ for each $j > 0$, we obtain $beh(\tau) = \theta_0, \theta_1, \theta_2, ... \in behs(B')$. Observing that $\tau = \sigma$ and $beh(\tau) \in behs(B')$, we obtain $beh(\sigma) \in behs(B')$. We can conclude that for each execution $\sigma \in execs(A')$ there exists an execution $\tau \in execs(B')$ such that $beh(\sigma) = beh(\tau)$. Hence by definition of behaviors, $behs(A') \subseteq behs(B')$. □

Changing a SA to a successor version for adapting to evolved requirements often only requires to adapt the implementations of a proper subset of the SA's components without changing the architecture's topology, *i.e.,* the SA's interface is left unchanged and components neither need to be added nor removed but some component implementations are changed. In this case, it is often not strictly necessary to check whether the TSCA corresponding to the new SA is a refinement of the TSCA corresponding to the original architecture. It suffices to show that the composition of the evolved sub-architecture with any common subsystem of the original and the evolved SA is a refinement of the composition of the original sub-architecture with the same common subsystem:

Theorem 20. *Let $S_A = (\Sigma, C_A)$ and $S_B = (\Sigma, C_B)$ be two SAs having the same channel signature Σ. If there exists a set of components $Sub \subseteq (C_A \cap C_B)$ such that $\bigotimes((C_A \setminus C_B) \cup Sub) \preceq \bigotimes((C_B \setminus C_A) \cup Sub)$, then $S_A \preceq S_B$.*

Proof. Let $S_A = (\Sigma, C_A)$ and $S_B = (\Sigma, C_B)$ be two syntactically conform SAs with channel signature $\Sigma = (I, O)$. Suppose there exists a set of components $Sub \subseteq C_A \cap C_B$ such that $\bigotimes((C_A \setminus C_B) \cup Sub) \preceq ((C_B \setminus C_A) \cup Sub)$. Let $C = ((C_A \setminus C_B) \cup Sub)$ and let $C' = ((C_B \setminus C_A) \cup Sub)$.

In the following we show that $(\bigotimes C)$ and $(\bigotimes C_A \setminus C)$ as well as $\bigotimes C'$ and $\bigotimes C_B \setminus C'$ are compatible, which shows that the corresponding compositions are well-defined: As S_A is a SA, the components in C_A are all pairwise compatible. Thus, the components in $C \subseteq C_A$ and the components in $C_A \setminus C \subseteq C_A$ are also pairwise compatible. Therefore, $(\bigotimes C)$ and $(\bigotimes C_A \setminus C)$ are well-defined. As it holds that $C_A = C \cup (C_A \setminus C)$ and $C \cap (C_A \setminus C) = \emptyset$, applying the first part of Theorem 6 at most $|C|$ times, we obtain that $(\bigotimes C)$ and c are compatible for each $c \in C_A \setminus C$. As all components in C_A are pairwise compatible and each component $c \in C_A$ is compatible to $(\bigotimes C)$, applying the first part of Theorem 6 at most $|C_A \setminus C|$ times, we obtain that $(\bigotimes C)$ and $(\bigotimes C_A \setminus C)$ are compatible. A similar argument shows that $\bigotimes C'$ and $\bigotimes C_B \setminus C'$ are compatible.

In the following we show that $C_A \setminus C = C_B \setminus C'$, which enables to apply Theorem 18: It holds that $C_A \setminus C = C_A \setminus ((C_A \setminus C_B) \cup Sub) = (C_A \setminus (C_A \setminus C_B)) \setminus Sub = ((C_A \setminus C_A) \cup (C_A \cap C_B)) \setminus Sub = (C_A \cap C_B) \setminus Sub$. Using a similar argument, we obtain $C_B \setminus C' = C_B \setminus ((C_B \setminus C_A) \cup Sub) = (C_B \setminus (C_B \setminus C_A)) \setminus Sub = ((C_B \setminus C_B) \cup (C_B \cap C_A)) \setminus Sub = (C_B \cap C_A) \setminus Sub$. We can conclude $C_A \setminus C = C_B \setminus C'$.

Having shown the compatibility and $C_A \setminus C = C_B \setminus C'$ and since by assumption $\bigotimes C \preceq \bigotimes C'$, Theorem 18 guarantees $(\bigotimes C) \otimes (\bigotimes C_A \setminus C) \preceq (\bigotimes C') \otimes (\bigotimes C_B \setminus C')$. It holds that $C \cap (C_A \setminus C) = \emptyset = C' \cap (C_B \setminus C')$ and that all components in $C \cup (C_A \setminus C) = C_A$ and in $C' \cup (C_B \setminus C') = C_B$ are pairwise compatible. Thus, by definition of $\bigotimes$, the above is equivalent to $\bigotimes C_A \preceq \bigotimes C_B$. Since Theorem 19 guarantees that hiding preserves refinement, this implies $(\bigotimes C_A)|_O \preceq (\bigotimes C_B)|_O$. This is by definition of refinement equivalent to $S_A \preceq S_B$. □

Nevertheless, it might be the case that no such subsystem as described in Theorem 20 exists. Thus, in the worst case, the complete TSCAs for both architectures have to be considered.

However, we believe in practice this rarely occurs. The above leads to the following algorithm for mitigating the state explosion problem during semantic differencing of finite system architectures:

Algorithm 3 Mitigating the state explosion problem during refinement checking of system architectures.

Input: Two finite SAs $S_A = (\Sigma_A, C_A)$ and $S_B = (\Sigma_B, C_B)$.
Output: Yes, if $S_A \preceq S_B$, and $w \in \Delta([\![S_A]\!], [\![S_B]\!])$, otherwise.

```
define C = ⨂(C_A \ C_B) as TSCA
define C' = ⨂(C_B \ C_A) as TSCA
for all S ⊆ C_A ∩ C_B in increasing size do
  if behs(S ⊗ C) ⊆ behs(S ⊗ C') then
    return Yes              /* Composition without hiding */
  end if
end for
if behs(S_A) ⊆ behs(S_B) then
  return Yes                /* Composition with hiding */
else
  return w ∈ Δ([[S_A]], [[S_B]])
end if
```

In case the if-condition in the for-loop is satisfied, Theorem 20 guarantees the refinement relation holds. In case the condition is not satisfied for any $S \subseteq C_A \cap C_B$, it has to be checked whether the complete SA S_A refines the SA S_B. The difference between comparing $\bigotimes C_A$ with $\bigotimes C_B$ and $tspa(S_A)$ with $tspa(S_B)$ is that the former comparison does not consider hiding of internal channels, while the latter does. For the behavior inclusion checks and diff witness generation, existing algorithms for language inclusion checking between BAs may be used (cf. Section 5.1 and Section 5.3).

Example 27 (Application of Algorithm 3)**.** *Consider the system architectures of the* `Mod8Counter` *as depicted in Figure 3 (c) and the system architecture as depicted in Figure 5. We denote to the first one as S_A and to the second one as S_B. The goal is to determine whether $S_B \preceq S_A$ holds. Applying semantic differencing checking to these two system architectures reveals they refine each other. Both also refine the initial specification for the* `Mod8Counter` *as explained in Appendix B. More details on the evaluation regarding refinement checking between the three architectures are given in Section 6.3.*

6. Implementation and Evaluation

This section recapitulates the MontiArcAutomaton ADL [35, 37], presents the application of refinement checking to its models and evaluates our approach.

6.1. The MontiArcAutomaton ADL

The MontiArcAutomaton ADL [35, 37] comprises the modeling elements common to many popular component & connector ADLs [29], *i.e.,* hierarchical components with interfaces of typed, directed ports and unidirectional connectors (typed FIFO channels) exchanging messages between these ports. The components are black-boxes and either atomic or composed: atomic components yield behavior descriptions in form of embedded automata (following the I/O$^{\omega}$ [39] paradigm) or in form of Java implementations. Such automata and Java implementations are transformable to TSCAs for semantic differencing. The behavior of composed components solely emerges from the interaction of their subcomponents. Composing the TSCAs belonging to a composed component's subcomponent implementations results in a TSCA modeling the composed component's behavior. With this, semantic differencing of composed components is possible. Components are scheduled by a global clock and perform cycles of

- reading all messages on incoming ports;
- computing behavior (which might entail invoking subcomponents)
- producing a single message to each outgoing port.

Each computation consumes a time slice, *i.e.,* the output for messages received at the global clock's i-th tick is processed at its i+1-th tick earliest. All MontiArcAutomaton components are thereby strongly causal. The MontiArcAutomaton ADL also distinguishes between component types and their instances, supports component type inheritance, generic type parameters for components (*e.g.,* to be used with generic port types), and constructor-like configuration of these instances.

```
01 component Elevator {
02   port in Bool req1, in Bool at1,
03     // ... further ports ...
04     out Bool open, out Bool close,
05     out Clear clear;
06
07   component Control ctrl; // named
08   component Motor m;      // subcomponent
09   component Door d;       // instances
10
11   connect req1 -> control.req1;
12   // ... further connectors ...
13   connect control.clear -> clear;
14 }
```

Figure 14: Textual representation of the component `Elevator`.

The MontiArcAutomaton ADL is a textual modeling language implemented with the MontiCore [22] language workbench. The textual representation of the composed component type `Elevator` is illustrated in Figure 14. It begins with the keyword "component", followed by the component type's name and a body delimited by curly brackets (l. 1). The body contains an interface of typed ports (ll. 2-5), declares three subcomponents (ll. 7-9), and multiple connectors (ll. 11-13). The subcomponent declarations reference component types imported from artifacts (such as `Control`).

6.2. Semantic Differencing of MontiArcAutomaton Components

The implementation comprises a translation from MontiArcAutomaton architectures to semantically equivalent TSCAs.

TSCAs are only handled internally as representatives for sets of TSSPFs modeling component semantics and are not explicitly modeled by component developers. Each atomic component directly translates to a TSCA. The TSCA of a composed component is computed by composing the TSCAs of its subcomponents according to the architectural configuration defined by the composed component's connectors. A composed component's TSCA is either constructed using the composition operator's definition (cf Definition 16) or using Algorithm 2 to directly compute the trimmed TSCA of the compound. The implementation further consists of a translation from TSCAs to BAs and generators that produce models in the "BA format", which is the input format of the tool RABIT [3]. In case a BA does not refine another BA, RABIT provides a counterexample serving as a concrete disproof for refinement. The counterexamples are translated back to diff witnesses, which technically are finite prefixes of behaviors of one component that are no behaviors of another component. An engineer can use the witness to either manually inspect the component implementation for the syntactic reasons causing the semantic difference, or create a unit test where the component is provided the input encoded by the witness. When executing the unit test, the engineer may employ the usual debugging techniques provided by all common integrated development environments to identify the component implementation's elements causing the diff witness. Using the tool chain described above enables automated refinement checking and diff witness generation for MontiArcAutomaton architectures and ultimately supports engineers in detected the semantic differences between component implementations.

6.3. Semantic Differencing Evaluation

We evaluated the approach to semantic differencing with six MontiArcAutomaton architectures previously used for evaluation in [9, 38] and the modulo-8 counter architectures used as running example throughout this paper. We specifically chose the first six architectures for evaluation since the approach presented in [38] failed for some specifications, which we considered to be challenging, and to enable comparability. The architectures were slightly modified for this evaluation to resolve technical MontiArcAutomaton version compatibility issues. The example models as well as the BAs resulting from the translations are available online [1]. This paper extends the previous evaluation of [9] with the modulo-8 counter architecture that is used as running example. Further, the previous evaluation [9] always naively composes TSCAs using the definition of the composition operator (cf. Definition 16). This paper extends this evaluation by further applying the advanced composition method that simultaneous trims the compound while composing the composition's participants (cf. Algorithm 2). We reused the completion strategies [38] for completing the automata implementations of the architectures' atomic components.

The first architecture is given by an implementation of an elevator control system (ECS) (cf. Section 2). It comprises 3 composed and 5 atomic components. The second example consists of four variants of a mobile robot. We only report on the evaluation of the most challenging variant. This variant comprises 4 components in total whereof 3 components are atomic. Another architecture implements a pump station consisting of 3 composed and 10 atomic components. The modulo-8 counter specification is completely defined in Figure B.16. The result from executing the refinement checks presents in this paper slightly differ from the results presented in [9] because we repeated the evaluation of the pre-existing examples to enable comparability between the two different composition method variants. We conducted the evaluations of both composition variants on the same computer at the same date.

Table 1: Time for refinement checking and diff witness calculation.

		$\Delta([\![\cdot]\!], [\![\cdot]\!])$	$\Delta([\![\cdot]\!], \mathit{Chaos})$	$\Delta(\mathit{Chaos}, [\![\cdot]\!])$
Naive	Floors	62ms	536ms	885ms
	Elevator	83ms	2510ms	5927ms
	ECS	461ms	7124ms	15339ms
	SensorReading	62ms	753ms	1401ms
	Controller	12ms	17ms	19ms
	Pumpstation	120ms	321ms	570ms
	MobileRobot	61ms	67ms	85ms
	Mod8Counter	14ms	17ms	15ms
Trim	Floors	69ms	560ms	914ms
	Elevator	39ms	2525ms	5927ms
	ECS	94ms	9263ms	15850ms
	SensorReading	57ms	787ms	1390ms
	Controller	11ms	13ms	16ms
	Pumpstation	112ms	326ms	543ms
	MobileRobot	23ms	57ms	76ms

In [38], for each of the architectures, three specification checks are executed: it is checked whether the semantics of a component is equal to itself, whether a component refines a component with the same interfaces that implements arbitrary behavior, *i.e.,* all possible behaviors, and whether the semantics of a component are equal to the semantics of a component implementing arbitrary behavior. We performed the same checks on a computer with 3.0 GHz Intel Core i7 CPU, 16 GB Ram, Windows 10, and RABIT 2.4 using our translation from MontiArcAutomaton architectures to BAs and the language inclusion checking tool RABIT [3] (cf Section 6.2).

Table 1 summarizes the computation times of RABIT given the BAs resulting from the transformation as input. For the component ECS constructed using the naive composition method, for instance, checking whether it refines itself took 461ms, checking refinement with arbitrary behavior took 7124ms, and calculating a diff witness distinguishing the component from arbitrary behavior took 15339ms. Table 2 depicts the sizes of the automata resulting from the translations and the time required to construct a TSCA from its subcomponents' TSCAs using the denoted composition method. For component ECS, for instance, it took 3465ms to construct the TSCA using the naive composition method. The TSCA and the BA resulting from the transformation have 746 states and 98496 transitions. RABIT reported the tool has reduced the BA to 8 states and 1728 transitions after internal preprocessing. For every component we modeled arbitrary behavior (Chaos) with a

Table 2: The numbers of states and transitions of the TSCAs translated from the architectures and of the generated BAs.

			TSCA/BA		BA AP		Chaos
		time	#states	#trans.	#states	#trans.	#trans.
Naive	Floors	25ms	32	1024	32	1024	23328
	Elevator	460ms	34	10206	1	729	236196
	ECS	3465ms	746	98496	8	1728	472392
	SensorReading	7ms	2	1296	2	1296	69984
	Controller	1ms	1	9	1	9	108
	Pumpstation	19ms	6	3888	4	2592	17496
	MobileRobot	4ms	150	2700	12	216	1152
	Mod8Counter	0ms	8	32	8	32	32
Trim	Floors	267ms	32	1024	32	1024	23328
	Elevator	10ms	1	729	1	729	236196
	ECS	2829ms	8	1728	8	1728	472392
	SensorReading	118ms	2	1296	2	1296	69984
	Controller	1ms	1	9	1	9	108
	Pumpstation	3482ms	6	3888	4	2592	17496
	MobileRobot	10ms	12	216	12	216	1152

TSCA consisting of one state and a transition for every possible component input/output combination. The TSCA and the BA modeling arbitrary behavior for component ECS, for instance, comprise 472392 transitions (*cf.* Table 2). In contrast to the translation from MontiArcAutomaton architectures to the model checker Mona [38], our implementation succeeded for all example architectures. The longest computation time of our evaluation (15850ms, *cf.* Table 1) resulted from semantic differencing arbitrary behavior with the ECS component. We additionally used the implementation to automatically verify semantic equivalence of the three architectures depicted in Figure 3. We checked whether the specifications are semantically equivalent by checking refinement in both directions. Proving equivalence between the initial specification and the first structural refinement took 41ms. Checking equivalence between the initial specification and the second structural refinement took 47ms. The same check between the first and the second structural refinements was possible in 46ms.

The composition method that includes trimming the compounds yields a smaller composition duration in case the compound is smaller than the compound obtained from using the naive composition method (cf. Table 2). In case both composition methods yield the same compound, the naive composition method outperforms the method that includes trimming. This is plausible because of the overhead caused by trimming the TSCA. We conclude that our translation provides promising results. Nevertheless, the evaluation was only performed on a few specific architectures. Thus, the results are not generalizable to all possible architectures: the time needed by our tool may vary strongly from system to system.

7. Discussion

If the semantics domain of an ADL is overly general, undecidability of the underlying mathematical problems renders automated formal verification impossible. Then, architecture properties have to be proven manually, which is too expensive to be carried out in continuous architecture modeling and thus hinders employing agile development in architecture modeling projects: little changes to requirements or implementations can entail changing many manually performed proofs. In contrast, where automated formal verification is possible, sound and complete proofs can be generated automatically, supporting agile implementation evolution.

Focus is a comprehensive framework that supports specifying the observable input/output behavior of interactive systems. Its complexity requires carrying out proofs for system behavior verification manually. Focus provides various constructs for describing the semantics of distributed systems [36]. Examples are relations, set-based functions, sets of functions, assumption/guarantee predicates, or state-based representations. As identified in [36], the most fine-grained domain for describing the semantics of distributed systems using Focus are sets of SPFs. Independent of the style, specifications can describe timed or untimed behavior. Untimed behavior only considers the causality regarding the order of inputs and outputs. Timed specifications additionally concern causality regarding the passage of time. Many requirements are not only concerned with the order of messages but also state requirements with respect to passage of time. Thus, we employ a variant of the timed subset of Focus and thereby use sets of TSSPFs as semantics domain [36, 39].

Our approach is limited to systems where the data types' domains are finite and is restricted to the time-synchronous model of computation. However, our system model fits well into the kinds of systems developed for embedded systems such as automotive or robotics applications. Thus, our results enable fully automated tool support for many systems in such domains. Emphasizing that our approach cannot be generalized to the timed model of Focus as, for example, used in [16], is important: Timed SPFs (*cf.* [16, 36, 39]), for instance, are too general to be applicable to our approach. A timed SPF processes infinite sequences of finite sequences (of arbitrary lengths) of messages. Each of the finite sequences represents a finite stream of messages received or sent by a component in a single time unit. In contrast, TSSPFs only process single messages per time unit. The set of finite streams of messages over a non-empty finite data type is already infinite. Thus, for each time unit, a timed SPF needs to define a possible behavior for infinitely many tuples of input streams, whereas a TSSPF needs to define a reaction for all possible tuples of input messages, which are finitely many if the messages' data types are finite. From a practical viewpoint it is rarely required to specify the reaction in a time unit in response to the receipt of an arbitrary number of messages. Usually it either requires to handle single messages (TSSPFs) or sequences of messages where the length of the sequence is bounded by an arbitrary but fixed natural number. The latter can be reduced to the former by introducing lists of fixed length as message types.

The underlying theoretical problem for semantic differencing used in our approach is language inclusion checking between Büchi automata. Its complexity can be considered as another limitation of our approach. However, our main focus is not verifying a system's properties (*e.g.,* refinement or semantic differ-

encing) within seconds, which is most often already rendered impossible due to the complex nature of the safety critical system under development. We believe that nonetheless the possibility to apply formal fully automated verification (*e.g.,* over night) greatly facilitates continuous architecture modeling.

8. Related Work

Studies on the verification techniques of ADLs have been conducted, *e.g.,* in [43] and [45]. The study in [45] surveys verification techniques supported by ADLs with formal semantics, the translation of architectures to inputs for model checkers, and tool support as well as usability, scalability, and expressiveness. As supported by our approach, the study states that architecture verification for practical applications requires tool-support and automation. The study in [43] compares different verification tools and applies them to various ADLs. All architectures are transformed into intermediate labeled transition systems before the verification tools are applied, hampering the direct comparison with our approach.

The following surveys concrete approaches for formally analyzing hierarchical architecture descriptions. AutoFOCUS 3 [18] is a tool for the development of reactive embedded systems that also bases its semantics on FOCUS [7]. Although AutoFOCUS 3 supports model checking architectures against LTL and CTL formulas that specify properties concerning component behavior [10], we are not aware of a fully automated refinement checking method for AutoFOCUS 3. The π-ADL supports statistical model checking for verifying dynamic software architectures against DynBLTL properties [11]. To this effect, a statistical model of finite system executions is built and the probability of satisfying a property within a confidential bound is calculated. This approach is particularly tailored to dynamic architectures and is only concerned with finite traces. In contrast, our approach deals with infinite traces, static architectures, and full certainty. Refinement of architectures specified with timed I/O is described in [20]. Similar to behaviors of TSCAs, the semantics of a timed automaton is given by a set of traces. Refinement between timed I/O automata is defined similar as in our approach by trace inclusion. However, timed I/O automata are only marked with one message per transition and composition is defined differently. Further, the timing concept of I/O automata is more powerful and complicated than the one of our approach [16]. A game-based extension of the timed I/O automaton model enabling tool supported refinement checking has been proposed in [12]. Another approach to automated refinement checking based on the time-synchronous frame of FOCUS is described in [34, 38]. This approach is based on a relational semantics domain where the semantics of a component is given as a relation between the component's possible inputs and outputs. In contrast, our approach uses a more fine grained [36] semantics domain consisting of sets of functions. Refinement checking in [34, 38] relies on translating component semantics into WS1S and is implemented using the model checker Mona [13]. The approach suffers from the tool's high computational complexity, which is grounded in the non-elementary complexity of solving W1S1 problems. In contrast, we define a translation to Büchi automata and thereby obtain a PSPACE-complete complexity for refinement checking. While the relational approach is based on analyzing the result from composing the semantics of the individual components of a system, our approach first syntactically composes the individual components and bases analysis on the semantics of the compound.

9. Conclusion

We have presented an implementation of stepwise refinement for C&C ADLs using a subset of the Focus semantics for time-synchronous, distributed, interactive systems that is powerful enough to model complex and realistic systems. Based on previous work [9], we describe an approach to transform component models into time-synchronous channel automata that is based on an associative, commutative, and semantically compositional, syntactic composition operator for time-synchronous channel automata. Using this operator, the automata are composed syntactically and translated into Büchi automata, where their refinement can be checked through language inclusion. To this effect, we proved that the operational semantics of a finite time-synchronous channel automaton and the language accepted by the Büchi automaton resulting from the transformation coincide. This enables fully automated refinement checking for software architecture models in reasonable time.

We extended the previous approach [9] to improve its performance through technical enhancements of the underlying formal system model and extended previous evaluations. We further defined a notion of system architecture based on a white-box view where component implementations are assumed to be available. For such system architectures, we presented an algorithm leading to practical performance improvements for refinement checking.

This form of stepwise refinement supports continuous architecting through ensuring evolved components adhere to properties already proven for their predecessors. This ultimately reduces the effort for component evolution and, hence, facilitates continuous architecting.

References

[1] MontiArcAutomaton Models. `http://www.monticore.de/robotics/verification/`, [Online; accessed 2018-05-24].

[2] RABIT Tool Homepage, 2016. `http://www.languageinclusion.org/` [accessed 2016-12-31].

[3] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. Advanced Ramsey-Based Büchi Automata Inclusion Testing. In *International Conference on Concurrency Theory, CONCUR 2011*, 2011.

[4] Stephan Barth. Deciding Monadic Second Order Logic over ω-Words by Specialized Finite Automata. In *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, 2016.

[5] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 2010.

[6] Manfred Broy and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, TU Munich, 1992.

[7] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[8] Julius Richard Büchi. On a Decision Method in Restricted Second Order Arithmetic. In *Logic, Methodology and Philosophy of Science. Proceeding of the 1960 International Congress*. Stanford University Press, 1962.

[9] 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, 2017.

[10] Alarico Campetelli, Florian Hölzl, and Philipp Neubeck. User-friendly Model Checking Integration in Model-based Development. In *International Conference on Computer Applications in Industry and Engineering*, 2011.

[11] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flávio Oquendo, Thaís Batista, and Axel Legay. Statistical Model Checking of Dynamic Software Architectures. In *European Conference on Software Architecture*, 2016.

[12] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O Automata: A Complete Specification Theory for Real-time Systems. In *ACM International Conference on Hybrid Systems: Computation and Control*, 2010.

[13] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: New techniques for WS1S and WS2S. In *Computer-Aided Verification*, 1998.

[14] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, 2007.

[15] Max Fuchs. Formal Design of a Modulo-N Counter. Technical Report TUM-I9512, Technische Univerität München, 1995.

[16] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical report, TU Munich, 1995.

[17] Radu Grosu, Ketil Stølen, and Manfred Broy. A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing, 1997.

[18] Florian Hölzl and Martin Feilkas. AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Model-Based Engineering of Embedded Real-Time Systems*, 2007.

[19] Bengt Jonsson. A Fully Abstract Trace Model for Dataflow and Asynchronous Networks. *Distributed Computing*, 1994.

[20] Dilsun K. Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS 2003)*, 2003.

[21] Dexter Kozen. Lower Bounds for Natural Proof Systems. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, 1977.

[22] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proceedings of Tools Europe*, 2008.

[23] Orna Kupferman and Moshe Y. Vardi. Verification of Fair Transition Systems. In *International Conference on Computer Aided Verification*, 1996.

[24] Orna Kupferman and Moshe Y. Vardi. Complementation Constructions for Nondeterministic Automata on Infinite Words. In *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005*, 2005.

[25] Robert P. Kurshan. Complementing Deterministic Büchi Automata in Polynomial Time. *Journal of Computer and System Sciences*, 1987.

[26] Edward A Lee. CPS Foundations. In *Proceedings of the 47th Design Automation Conference*, pages 737–742. ACM, 2010.

[27] Christof Löding. Efficient minimization of deterministic weak ω-automata. *Information Processing Letters*, 2001.

[28] Zohar Manna and Amir Pnueli. Verifying Hybrid Systems. In *Hybrid Systems*, pages 4–35. Springer, 1993.

[29] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.

[30] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[31] Object Management Group. MDA Guide Version 1.0.1, June 2003. `http://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf` [Online; accessed 2015-12-17].

[32] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/` [accessed 2017-01-13].

[33] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In *Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE Computer Society, 1997.

[34] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Shaker Verlag, 2014.

[35] 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)*, 2015.

[36] 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.

[37] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag, 2014.

[38] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Model-Based Specification of Component Behavior with Controlled Underspecification. In *Modellbasierte Entwicklung eingebetteter Systeme (MBEES'16)*, 2016.

[39] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, TU Munich, 1996.

[40] S. Safra. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, 1988.

[41] Sven Schewe. Minimisation of Deterministic Parity and Buchi Automata and Relative Minimisation of Deterministic Finite Automata. *Computing Research Repository - CORR*, 2010.

[42] Frank Strobl and Alexander Wisspeintner. Specification of an Elevator Control System. Technical report, TU Munich, 1999.

[43] Jeffrey J.P. Tsai and Kuang Xu. A comparative study of formal verification techniques for software architecture specifications. *Annals of Software Engineering*, 2000.

[44] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2013.

[45] Pengcheng Zhang, Henry Muccini, and Bixin Li. A Classification and Comparison of Model Checking Software Architecture Techniques. *Journal of Systems and Software*, 2010.

Appendix A. `Mod8Counter` component in FOCUS

In MontiArcAutomaton, there is an explicit language construct (the connector) to indicate that two ports are connected. Besides this, MontiArcAutomaton distinguishes component types and component instances. Therefore, MontiArcAutomaton obtains unique port names by the fully qualified name of component instance and the port name. On the contrary, FOCUS has no notion of component type and has no explicit construct to indicate connectors. With this, MontiArcAutomaton is better suited for praxis, whereas FOCUS abstracts from implementation details to avoid notational clutter and improve formal representation. Thus, a MontiArcAutomaton architecture is conceptually transformed to a FOCUS architecture by omitting component types and by renaming ports such that they have identical names iff they are connected. A transformed version of the component `mod8Counter` as depicted in Figure 3 is depicted in Figure A.15.

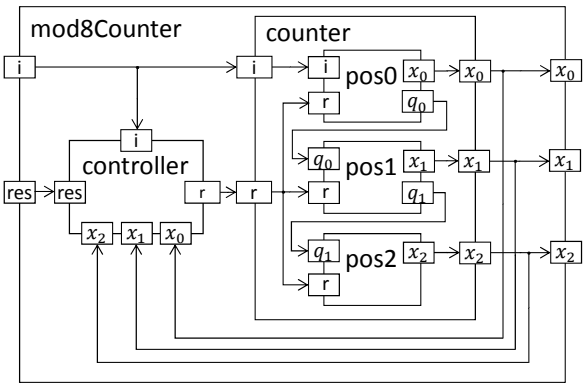


Figure A.15: FOCUS architecture of the `mod8Counter`.

Appendix B. TSCA of the `Mod8Counter` component

This section explains the TSCA of the initial specification of the `Mod8Counter` component as presented in Figure 3 (a). Figure B.16 demonstrates the TSCA in its graphical representation, where abbreviations for states and transitions are used. Transitions that increase the counted value start with the letter i, those that reset the value start with r, and those that do not alter the counted value start with an n. The textual representation of the TSCA and the abbreviations are explained in the following.

The TSCA depicted in Figure B.16 is a tuple $TSCA_{Mod8a} = (\Sigma, X, S, \iota, \delta)$, where

- $\Sigma = (\{\text{res,inc}\}, \{x_0,x_1,x_2\})$,
- the internal channels are $X = \{lv\}$ with $type(lv) = \{0, .., 7\}$,
- the set of states is defined by the set of all functions $S = X^{\rightarrow} = \{\theta \in [\{lv\} \rightarrow M] \mid \theta(lv) \in \mathbb{N} \wedge 0 \leq \theta(lv) \leq 7\}$, where for notational simplicity, we denote by $s_i = \{lv \mapsto i\}$,
- the initial state is $\iota = \{s_0\}$,
- the transition relation $\delta = I \cup R \cup N$ comprises the sets of increasing transitions $I = \bigcup_{k=0,..,8} i_k$, resetting transitions $R = \bigcup_{k=0,..,16} r_k$, and state conserving transitions $N = \bigcup_{k=0,..,8} n_k$, where
  - $i_0 = \{(s_0, \theta, s_1) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $i_1 = \{(s_1, \theta, s_2) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
  - $i_2 = \{(s_2, \theta, s_3) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
  - $i_3 = \{(s_3, \theta, s_4) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
  - $i_4 = \{(s_4, \theta, s_5) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
  - $i_5 = \{(s_5, \theta, s_6) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$
  - $i_6 = \{(s_6, \theta, s_7) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$
  - $i_7 = \{(s_7, \theta, s_0) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_0 = \{(s_0, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_1 = \{(s_1, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_2 = \{(s_2, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_3 = \{(s_3, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_4 = \{(s_4, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_5 = \{(s_5, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_6 = \{(s_6, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_7 = \{(s_7, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_8 = \{(s_0, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_9 = \{(s_1, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{10} = \{(s_2, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{11} = \{(s_3, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{12} = \{(s_4, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
  - $r_{13} = \{(s_5, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$

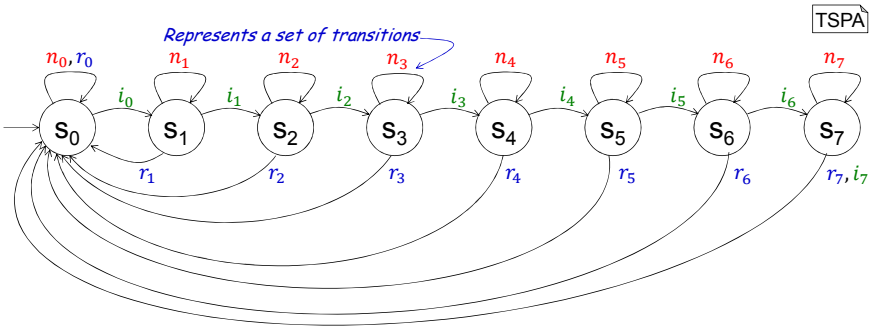


Figure B.16: TSCA of a modulo 8 counter.

- $r_{14} = \{(s_6, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $r_{15} = \{(s_7, \theta, s_0) \mid \theta(\text{res}) = \top \wedge \theta(\text{inc}) = \top \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $n_0 = \{(s_0, \theta, s_0) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $n_1 = \{(s_1, \theta, s_1) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \varepsilon\}$
- $n_2 = \{(s_2, \theta, s_2) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
- $n_3 = \{(s_3, \theta, s_3) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \varepsilon\}$
- $n_4 = \{(s_4, \theta, s_4) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
- $n_5 = \{(s_5, \theta, s_5) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \varepsilon \wedge \theta(x_2) = \top\}$
- $n_6 = \{(s_6, \theta, s_6) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \varepsilon \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$
- $n_7 = \{(s_7, \theta, s_7) \mid \theta(\text{res}) = \varepsilon \wedge \theta(\text{inc}) = \varepsilon \wedge \theta(x_0) = \top \wedge \theta(x_1) = \top \wedge \theta(x_2) = \top\}$

Operating Cyber-Physical Systems with Digital Twins

This section feature the publications summarized in Chapter 5.

Paper 11	G. Schuh, C. Häfner, C. Hopmann, B. Rumpe, M. Brockmann, A. Wortmann, J. Maibaum, M. Dalibor, P. Bibow, P. Sapel, and M. Kröger. Effizientere Produktion mit Digitalen Schatten, In: Wilhelm Bauer, Wolfram Volk, Michael Zäh, editors, In: ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb, 115, pages 105-107, Hanser, 2020. Reference: [SHH+20]
Paper 12	P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, A. Wortmann. Model-Driven Development of a Digital Twin for Injection Molding, In: Advanced Information Systems Engineering, 32nd International Conference, CAiSE 2020, Grenoble, France, June 8–12, 2020, Proceedings, pages 85-100, Springer, 2020. Reference: [BDH+20]
Paper 13	J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, A. 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, 2020. Reference: [KMR+20]

[SHH+20] G. Schuh, C. Häfner, C. Hopmann, B. Rumpe, M. Brockmann, A. Wortmann, J. Maibaum, M. Dalibor, P. Bibow, P. Sapel, M. Kröger:
Effizientere Produktion mit Digitalen Schatten.
In: ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb 115(special), Carl Hanser Verlag, Munich, April 2020.
www.se-rwth.de/publications/

Effizientere Produktion mit Digitalen Schatten

G. Schuh, C. Häfner, C. Hopmann, B. Rumpe, M. Brockmann, A. Wortmann, J. Maibaum, M. Dalibor, P. Bibow, P. Sapel und M. Kröger, Aachen

Die Digitalisierung verspricht Unternehmen, die Wandlungsfähigkeit und Produktivität bestehender Fertigungssysteme zu fördern. Durch die Komplexität cyber-physischer Produktionssysteme liegen Produktionsdaten jedoch heterogen, unstrukturiert und isoliert vor. Die für eine konkrete Aufgabe oder Fragestellung benötigten Daten werden durch Digitale Schatten zielgerichtet verknüpft, abstrahiert und aggregiert, sodass eine wissensbasierte und echtzeitfähige Entscheidungsfindung in der Produktion möglich wird.*)

Digitale Schatten bereiten heterogene Daten zielgerichtet auf

Die Digitalisierung stellt Produktionsunternehmen weltweit vor große Herausforderungen. Durch umfangreiche Datenerfassung und -analyse sollen hochflexible, wandlungsfähige Wertschöpfungssysteme entstehen, die bei reduzierten Kosten die Produktivität steigern. Aufgrund zahlreicher domänenspezifischer IT-Systeme und komplexer cyber-physischer Produktionssysteme liegen Daten sehr heterogen vor und erschweren dadurch eine zielgerichtete Optimierung [1, 2]. Durch die Komplexität des betrachteten Systems ist ein vollständiges und funktional umfassendes digitales Abbild des Produktionssystems als Digitaler Zwilling aufwändig [3, 4], kann aber bei adäquater Vorgabe der notwendigen Genauigkeit erreicht werden, da der Begriff „digital“ per se eine Abstraktion von dem physischen System beinhaltet.

Im Gegensatz zum Digitalen Zwilling umfasst der Digitale Schatten lediglich eine für eine konkrete Aufgabe oder Fragestellung benötigte Teilmenge der verfügbaren Daten. Dazu werden Daten (u. a. Mess- und Simulationsdaten) domänen- und anwendungsfallspezifisch kombiniert sowie auf inhaltlicher und zeitlicher Betrachtungsebene geeignet aggregiert. Entscheidungssituationen in der Produktion werden durch die gezielte Aufbereitung der Daten bestmöglich unterstützt und in Bezug auf konkrete Optimierungsmodelle teilautomatisiert auflösbar. Zudem bieten sie die Möglichkeit, bestehende produktionstechnische Modelle durch datenbasierte Ansätze weiterzuentwickeln und in der Praxis zu erproben.

Bild 1 verdeutlicht den potenziellen Nutzen Digitaler Schatten am Beispiel von Google Maps. Während das physische Straßennetz zur allgemeinen Orientierung oder zur Abschätzung von Entfernungen bereits als starres, Digitales Modell genutzt werden kann, werden zur Abschätzung der realen Fahrzeit und zur Identifikation der fahrzeitoptimierten Route dynamische Daten zum Verkehrsverhalten benötigt. Durch Anreichern des geographischen Modells um Daten zur aktuellen oder prognostizierten Verkehrslage als Digitaler Schatten lässt sich die tatsächliche Fahrzeit verlässlich und routenspezifisch ermitteln. Ein vollständiger Digitaler Zwilling des betrachteten Erdausschnitts ist dazu nicht notwendig.

Analog zu Google Maps lassen sich Fertigungsaufträge in der Produktion ebenfalls anhand ihrer starren Merkmale (notwendigen Arbeitsschritte, durchschnittlichen Durchlaufzeiten) in den Produktionsablauf einlasten, doch wird der Fertigstellungstermin erst durch Einbeziehen der aktuellen Auslastung und etwaiger Maschinenausfälle verlässlich planbar. Aufgrund der komplexen Wirkbeziehungen in der Produktionstechnik werden im Exzellenzcluster „Internet of Production“ daher Methoden entwickelt, um Digitale Schatten in der Produktion flexibel zu generieren und zur datengestützten Optimierung der Prozesse zu nutzen.

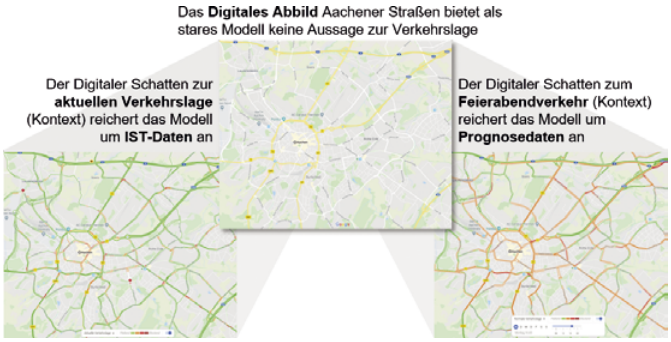


Bild 1. Beispielhafte Digitale Schatten dynamischer Daten in dem von Google Maps gebotenen Strukturmodell

*) **Förderhinweis**
Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) im Rahmen der Exzellenzstrategie des Bundes und der Länder – EXC-2023 Internet of Production.

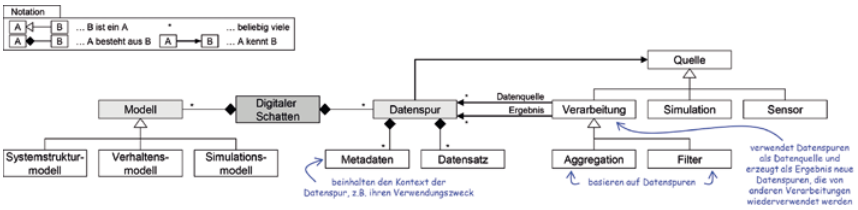


Bild 2. Metamodell des Konzepts Digitaler Schatten

Das Wesen Digitaler Schatten

Um Digitale Schatten in der Produktion erfassen und nutzen zu können, muss zunächst eine Informationsarchitektur aufgestellt werden, die es ermöglicht, die Vielzahl der in der Produktion anfallenden Daten zu erfassen, zu verarbeiten und zu verstehen. Im Software Engineering werden hierzu Modelle verwendet, die zugleich die konzeptionelle Kluft zwischen domänenspezifischen Abstraktionsebenen schließen [5]. Bestehende digitale Modelle, die zumeist statisch und unintegriert sind [6], werden dagegen zur Typisierung Digitaler Schatten und der notwendigen Datenstrukturen genutzt. Der Digitale Schatten wird hierzu gemäß des entwickelten Metamodells aus Bild 2 wie folgt definiert:

Ein Digitaler Schatten ist eine Menge von Modellen und Datenspuren, die neben dem reinen Datensatz auch kontextbeschreibende Metadaten zum jeweiligen Verwendungszweck enthalten.

Digitale Schatten bestehen demnach aus Datenspuren, die zu einem bestimmten Verwendungszweck oder Analysefokus erzeugt und ausgewertet werden. Hierzu beinhalten die Datenspuren neben den reinen Datensätzen kontextbehaftete Metadaten (z. B. Bezugsquelle, Erfassungszeitpunkt, Auflösung), die zur semantischen Verarbeitung benötigt werden. Sie können aus verschiedenen Quellen (z. B. Sensorsignale, Simulationsdaten) stammen, oder als Verarbeitung durch Aggregation oder Filterung anderer Datenspuren (z. B. Temperaturgradient aus mehreren Temperaturmessungen) entstanden sein. Darüber hinaus können Digitale Schatten Modelle enthalten, die sie um dynamische Daten anreichern oder zur Analyse der Datenspur nutzen.

Kontextspezifische Digitale Schatten entstehen daher durch zielgerichtete Selektion der benötigten Parameter, Bereinigung der Rohdaten und Kombination von Daten aus verschiedenen Quellen. Damit diese heterogenen Daten semantisch verwertet werden können, werden Modelle und Metadaten benötigt, die das Bezugssystem und die darin anfallenden Daten typisieren und strukturieren. Für Digitale Schatten der Produktion ergeben sich daher folgende Konsequenzen (K1 – K5) hinsichtlich der Modellierung, Datenerfassung und Funktionalität:

K1: Digitale Schatten müssen domänenspezifisches Wissen beinhalten
Zur zielgerichteten Analyse Digitaler Schatten und der semantischen Verknüpfung enthaltener Daten ist es notwendig, auf domänenspezifisches Wissen zurückzugreifen. Hierzu müssen die bei Planung und Betrieb eines Produktionssystems genutzten Modelle (z. B. Verhaltensmodelle, Simulationsmodelle) in geeigneter Form digitalisiert werden. In der Modellbasierten Systementwicklung haben sich dazu bereits verschiedene Modellierungssprachen (z. B. SysML, UML, CAD, Modellica) etabliert, um Wissen domänenübergreifend abzubilden. Dadurch wird es möglich, Modelle über Digitale Schatten mit dynamischen Daten anzureichern, um gleichzeitig die Modellgenauigkeit und das Bezugssystem kontinuierlich zu optimieren.

K2: Digitale Schatten sind stets kontextbezogen
Ziel des Digitalen Schattens ist es, die Entscheidungssituation einer spezifischen Aufgabe oder Fragestellung hinreichend gut zu unterstützen. Der Kontext (z. B. Wartung, Betriebspunktoptimierung) bestimmt den Analysefokus und wird in den Metadaten der Datenspur zur semantischen Verknüpfung hinterlegt. Der Digitale Schatten bildet das reale System daher nicht vollständig ab, sondern gewährt lediglich einen zweckgerichteten Blick auf das Original. Derselbe Datensatz kann dadurch in einem veränderten Kontext wiederverwendet werden und andere Erkenntnisse liefern.

K3: Digitale Schatten ermöglichen domänenübergreifende Analysen
Digitale Schatten verknüpfen Daten aus heterogenen Datenquellen. Dadurch können sie Daten beinhalten, die dem Bezugssystem unzugänglich sind (z. B. Historien-, Wartungs- oder Entwicklungsdaten) und ermöglichen multiperspektivische und systemübergreifende Analysen.

K4: Digitale Schatten benötigen angepasste IT Infrastrukturen im Fabrikbetrieb
Die aufgabenspezifische Verarbeitung (z. B. Aggregation, Filterung) der Datenspuren multiperspektivischer Digitaler Schatten stellt die derzeit verfügbaren IT-Systeme vor große Herausforderungen, da ebenso flexibel veränderbare Soft- und Hardware-Systeme notwendig werden. Datenerfassungs- und Verarbeitungssysteme im Fabrikbetrieb müssen für einen transparenten und konsistenten Datenzugriff veränderbaren Anforderungen (z. B. domänenspezifische Echtzeit, dezentraler Datenzugriff) genügen.

K5: Digitale Schatten nehmen keinen Einfluss auf das reale Systemverhalten
Digitale Schatten sind kontextbehaftete Datenspuren. Sie dienen der aufgabenspezifischen Analyse und der Anreicherung zugrundeliegender Modelle um relevante Daten. Als solche üben sie keinen aktiven Einfluss auf das reale Bezugssystem aus. Erst auf den Digitalen Schatten reagierende Systeme (z. B. Sensor-Aktor-Systeme) können voll- oder teilautomatisiert Aktionen ausführen und die Produktion beeinflussen.

Digitale Schatten in der Produktion

Die notwendige Infrastruktur zum Erzeugen und Nutzen Digitaler Schatten wird im Exzellenzcluster derzeit anhand zweier praxisnaher Anwendungsfälle erprobt. Einerseits werden die Anforderungen an Digitale Schatten bei Ultrakurzpuls (UKP)-Laseranwendungen (z. B. zur Mikrostrukturierung von Bauteilen) und der damit einhergehenden Verarbeitung hoher Datenmengen in kürzester Zeit zur präzisen Ansteuerung des Lasers untersucht. Andererseits werden phasenbezo-

gene Digitale Schatten beim Spritzgießen von Kunststoffbauteilen erzeugt und zyklisch ausgewertet, um einen selbstadaptiven Digitalen Zwilling an das aktuelle Maschinenverhalten anzupassen.

Im UKP-Prozess hat die Komplexität des Bauteils kaum einen Einfluss auf die Geschwindigkeit des Verfahrens. Vielmehr beeinflussen die Absorptionsrate des Materials und die Bewegungsgeschwindigkeit des Lasers die finale Prozess- und Bauteilqualität. Aufgrund der meist hohen Individualität und Produktionszeit der mikrostrukturierten Bauteile, kann die Qualität der Bauteile erst nach der vollständigen Bearbeitung evaluiert werden.

Zur Optimierung des Verfahrens wird mittels spezieller Sensorik und Edgedevice (K4) hochfrequent die spektrale Emission vom Verdampfen des Werkstoffs positionsgenau aufgenommen (K1). Durch eine automatisierte Auswertung dieser Datenströme entsteht ein dreidimensionaler digitaler Schatten der eingebrachten Mikrostruktur (K2), welcher für die automatische Anpassung der aktuellen Prozessparametern und der Bahnplanung auf die über das Werkstück heterogenen Materialeigenschaften genutzt werden kann. Somit ermöglicht die Nutzung dieses gezielten digitalen Schattens (K5) durch geeignete Services einen geregelten und gleichmäßigen Abtrag und verringert somit die Nachbearbeitung und den Ausschuss, schon während der Strukturierung.

Im Spritzgießen liegen dagegen in der Regel kurze Zykluszeiten zur Fertigstellung eines Bauteils und eine Vielzahl interagierender Systeme, wie z. B. Spritzgießmaschine, Temperiergeräte, Werkzeugsensorik, vor welche die Komplexität der Prozessführung und der verfügbaren Datenspur steigern. Mithilfe von Methoden der statistischen Versuchsplanung können Prozessmodelle ermittelt und robuste Arbeitspunkte für eine spezifische Werkzeug-Maschinen-Kombination identifiziert werden. Die ermittelten Prozessmodelle können jedoch nur bedingt auf andere Maschinen übertragen werden [7]. Da jede Maschine auch bei gleicher Spezifikation ein einzigartiges Betriebsverhalten aufweist (z. B. aufgrund des individuellen Verschleißzustands), müssen zur Modellierung funktionaler Digitaler Zwillinge zunächst Probleme bei der Datenerfassung heterogener Datenquellen, der Ansteuerung von Sensor-Aktor-Systemen sowie der echtzeitfähigen Datenanalyse zur Optimierung und Vorhersage der Bauteil- und Prozessqualität gelöst werden [8].

Das Konzept des Digitalen Schattens wird daher angewendet, um einen selbstadaptiven Digitalen Zwilling der Fertigungszelle zu erzeugen, der die heterogenen Datensätze der angeschlossenen Subsysteme verarbeitet (K3) und daraufhin das reale Produktionssystem ansteuert (K5). Zur Kommunikation mit den angeschlossenen Geräten liegt eine angepasste Software-Architektur vor, welche die Daten über verschiedene Schnittstellen (z. B. OPC-UA, EUROMAP, RS232) erfasst und einen konsistenten zyklusbezogenen Zugriff ermöglicht (4). Dazu wird das Domänenwissen zu relevanten Prozessparametern der Prozessphasen des Spritzgießzyklus in UML Klassen- und Aktivitätsdiagrammen abgebildet (K1). Je nach Analysefokus erzeugt der Digitale Zwilling daraus automatisiert eine Datenbank und erfasst die relevanten Prozessdaten als kontextbehaftete Datenspur (K2).

Zusammenfassung und Ausblick

Ein vollständiger und umfassend funktionaler Digitaler Zwilling ist in der Produktion kaum realisierbar, da eine Vielzahl heterogener Daten unter ständiger Konsistenzprüfung verarbeitet werden müsste [4]. Mit dem Konzept des Digitalen Schattens wird daher das Ziel verfolgt, lediglich die zu einer spezifischen Aufgabe oder Fragestellung notwendigen Daten zu erfassen, zu verarbeiten und zu analysieren. Dadurch werden Entscheidungssituationen in der Produktion bei geringen Latenzzeiten möglich. Das Konzept des Digitalen Schattens als kontextbehaftete Datenspur und sich daraus ergebende Konsequenzen für die IT-Infrastruktur innerhalb einer intelligenten Fertigung wurden hierzu erläutert.

Anhand der beschriebenen Anwendungsfälle (UKP, Spritzgießen) werden die umfassenden Rahmenbedingungen zum Erzeugen und nutzwertstiftenten Analysieren Digitaler Schatten in der Produktion erprobt. Hierzu müssen insbesondere Methoden entwickelt werden, wie gleichzeitig hohe Effizienz bei der Erstellung, aber auch Flexibilität und Konfiguration bei der Nutzung gewährleistet werden können. Durch die parallele Betrachtung zweier gegensätzlicher Anwendungsfälle sollen die Methoden eine breite Anwendbarkeit in unterschiedlichen Branchen finden.

Bibliography
DOI **10.3139/104.112339**
ZWF 115 (2020) Special; page 105 – 107

ISSN 0947 – 0085

Literatur

1. Bienzeisler B.; Schletz A.; Gahle, A.: Industrie 4.0 Ready Services Technologietrends 2020. Fraunhofer IAO, Stuttgart 2014, S. 8 – 30
2. Bauernhansl, T. et al.: WGP-Standpunkt Industrie 4.0. Wissenschaftliche Gesellschaft für Produktionstechnik WGP e. V., Darmstadt 2017, S. 6 – 30
3. Riesener, M.; Schuh, G.; Dölle, C.; Tönnes, C.: The Digital Shadow as Enabler for Data Analytics in Product Life Cycle Management. Procedia CIRP 26 (2019), S. 729 – 734 DOI: 10.1016/j.procir.2019.01.083
4. Ashtari, B.; Schlögl, W.; Weyrich, M.: Synchronisierung von digitalen Modellen. atp magazin 59 (2017) 7/8, S. 62 – 69 DOI: 10.17560/atp.v59i07-08.1902
5. France, R.; Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. IEEE (Hrsg.): Future of Software Engineering 2007 at ICSE. Minneapolis, 2007, S. 37 – 54 DOI: 10.1109/FOSE.2007.14
6. Estefan, Jeff A. et al.: Survey of Model-based Systems Engineering (MBSE) Methodologies. Incose MBSE Focus Group 25 (2007) 8, S. 1 – 12
7. Gierth, M. M.: Methoden und Hilfsmittel zur prozessnahen Qualitätssicherung beim Spritzgießen von Thermoplasten. Dissertation, RWTH Aachen, 1992
8. Liau, Y.; Lee, H.; Rhy, K.: Digital Twin Concept for Smart Injection Molding. Material Science and Engineering 324 (2018) 1, S. 1 – 5 DOI: 10.1088/1757-899X/324/1/012077

Die Autoren dieses Beitrags

Prof. Dr.-Ing. Günther Schuh ist Inhaber des Lehrstuhls für Produktionssystematik am Werkzeugmaschinenlabor WZL der RWTH Aachen.

Prof. Dr.-Ing. Christian Hopmann ist Leiter des Instituts für Kunststoffverarbeitung (IKV) in Industrie und Handwerk an der RWTH Aachen.

Prof. Dr. rer. nat. Bernhard Rumpe ist Inhaber des Lehrstuhls für Software Engineering der RWTH Aachen.

Prof. Dr. rer. nat. Constantin Häfner ist Inhaber des Lehrstuhls für Lasertechnik der RWTH Aachen und Direktor des Fraunhofer-Institut für Lasertechnik Aachen.

Dr.-Ing. Matthias Brockmann ist Geschäftsführer des Exzellenzclusters Internet of Production.

Dr. rer. nat. Andreas Wortmann ist akademischer Rat am Lehrstuhl für Software Engineering SE der RWTH Aachen.

Judith Maibaum, M. Sc., ist Wissenschaftliche Mitarbeiterin am Werkzeugmaschinenlabor WZL der RWTH Aachen.

Manuela Dalibor, M. Sc., ist Wissenschaftliche Mitarbeiterin am Lehrstuhl für Software Engineering SE der RWTH.

Pascal Bibow, M. Sc., und Patrick Sapel, M. Sc., sind Wissenschaftlicher Mitarbeiter am Institut für Kunststoffverarbeitung IKV an der RWTH Aachen.

Moritz Kröger, M. Sc., ist Wissenschaftlicher Mitarbeiter am Institut für Lasertechnik ILT an der RWTH Aachen.

Model-Driven Development of a Digital Twin for Injection Molding *

Pascal Bibow[2][0000−0002−9910−8702], Manuela Dalibor[1][0000−0002−1948−0556], Christian Hopmann[2], Ben Mainz[0000−0002−4817−446X], Bernhard Rumpe[1][0000−0002−2147−1966], David Schmalzing[1][0000−0003−2041−1475], Mauritius Schmitz[2][0000−0002−1595−4881], and Andreas Wortmann[1][0000−0003−3534−253X]

[1] Software Engineering, RWTH Aachen University, Aachen, Germany
http://www.se-rwth.de
[2] Institute for Plastics Processing in Industry and Craft at RWTH Aachen University, Aachen, Germany https://www.ikv-aachen.de

Abstract. Digital Twins (DTs) of Cyber-Physical Production Systems (CPPSs) enable the smart automation of production processes, collection of data, and thus can reduce manual efforts for supervising and controlling CPPSs. Realizing DTs is challenging and requires significant efforts for their conception and integration with the represented CPPS.
To mitigate this, we present an approach to systematically engineering DTs for injection molding that supports domain-specific customizations and automation of essential development activities based on a model-driven reference architecture. In this approach, reactive CPPS behavior is defined in terms of an event DSL and the reference architecture connects to the CPPS through a novel DSL for representing OPC-UA bindings. We have evaluated this approach with a DT of an injection molding machine that controls the machine to optimize the Design of Experiment (DoE) parameters between experiment cycles before the products are molded. Through this, our reference implementation of the DT facilitates the time-consuming setup of a DT and the subsequent injection molding activities. Overall, this facilitates to systematically engineer digital twins with reactive behavior that help to optimize machine use.

Keywords: Digital Twin · Injection Molding· Cyber-Physical Production System · Model-Driven Development · Reference Architecture

1 Introduction

DTs are an integral component of intelligent digitization [25] for smart manufacturing in Industry 4.0 [29]. Engineering DTs is time-consuming, complicated, and often not tightly integrated with the development of the system. Where

* Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2023 Internet of Production.

DTs are incapable of utilizing the knowledge about the system that exists in the form of engineering models, they cannot efficiently optimize the digitized system's behavior.

Injection molding is a manufacturing process to produce plastic parts by injecting plasticized material into a mold. Determining an ideal operation point usually requires experienced operators and extensive trials [23]. We use DTs to automate the execution of a DoE on an injection molding machine to learn about the current process characteristics and thus find ideal setting parameters. The presented architecture thereby gets evaluated in a real CPPS.

We propose a modeling method for DTs that partly automates engineering DTs to enable automated reaction to changes in the system structure and to synchronize the DT with its physical counterpart. To this end, we propose modeling the DT as a component and connector architecture with UML class diagrams specifying data objects that are exchanged between components. Furthermore, we present a Domain Specific Language (DSL) to describe events that the production system's DT reacts to. Models of this DSL are integrated into the software architecture model. From these, an integrated, reactive DT is generated that controls and optimizes injection molding behavior. The key contributions of this paper, hence, are

1. a model-driven methodology to efficiently developing DTs for CPPSs,
2. a reference architecture for DTs evaluated in injection molding,
3. a DSL connecting digital twins to their physical counterparts, and
4. modeling techniques to specify a DT's event-driven behavior.

In the following Sect. 2 introduces preliminaries, Sect. 3 presents a motivating example, and Sect. 4 explains the methodology. Subsequently, Sect. 5 describes required models and the realization, Sect. 6 describes how the DT is applied to the injection molding machine, and Sect. 7 discusses the reference architecture and methodology. Finally, Sect. 8 highlights related work, and Sect. 9 concludes.

2 Background

We realize a DT for injection molding based on our reference architecture that we implemented in MontiArc (see Sect. 2.3) [2,8]. The DT controls the molding machine via Open Platform Communication Unified Architecture (OPC-UA) [16].

2.1 Digital Shadows and Twins

The term digital twin is broadly used to describe any form of data that describes a physical system. We develop a digital twin that is partly derived from models describing the system under development. Furthermore, the twin shall provide services that allow interacting with the system or the twin itself.

Definition (Digital Twin (DT))**.** *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.*

These models may be engineering models (CAD, Simulink, *etc.*) or software models (UML, SysML, MontiArc, *etc.*) and the services may include monitoring, optimization, projection, visualization, and many more. Since the twin reflects a real system, it must also provide data that describes the system. As CPPSs produce immense amounts of data that often are too large to be fully processed by DTs, we introduce the concept of Digital Shadows (DSs).

Definition (Digital Shadow (DS))**.** *A digital shadow is a set of temporal data traces and/or their aggregation and abstraction collected concerning a system for a specific purpose with respect to the original system.*

Thus, DSs comprise the information that DTs require for fulfilling their tasks.

2.2 Injection Molding

Injection molding represents a highly automated, but to the same extent complex manufacturing process to produce, *e.g.,* plastic parts without the necessity of post-processing. Different data sources, like machinery or peripheral sensors, cavity sensors or quality control systems, enable gaining knowledge about the process. The heterogeneity in industrial machinery equipment thereby raises challenges in setting up DTs of production systems that monitor the process or execute complex machine tasks.

Due to complex interactions of production assets and setting parameters, determining settings of an ideal operation point at a specific machine is a challenging task. A well-experienced operator is capable of respecting the machine-specific characteristics in process setup as each machine differs in its respective process behavior. Differences in the process behavior exist even for machines of the same type or manufacturer due to wear of machine components or alternating control loops [14]. A DT might be able to support operators in such complex tasks by gathering current process data and providing a knowledge base for *e.g.,* process setup. The DT, therefore, needs to learn about the machine-specific behavior of the real production system it represents and adjust setting parameters according to current process data to work as a self-adaptive system.

The injection molding process consists of cyclic process phases for plasticizing the granular material, injecting it into a mold according to a specific injection flow profile and solidifying it under a set holding pressure until a molded part can be ejected. Via standardized communication protocols like OPC-UA, machine movements, and sensor data from the machine and its subordinated components are accessible. Thereby, relevant process parameters like temperatures, current volume flow or injection pressure get monitored to build up an extensive knowledge base for a DT to use.

The machine initializes an OPC-UA server during production start and notifies the server about changes in monitored items due to machine movements. For data gathering and accessing the OPC-UA server, the OPC Foundation provides standard libraries to develop connectors. The connector acts as OPC-UA client and subscribes to the server to monitor specific parameters of consideration via so-called Node-IDs. Gathered data is then passed on to a message

broker. Apache Kafka [27] is a communication platform that receives messages from a connector, acknowledges the receipt, stores the messages in a save log file, and delivers messages in case of a request. Many Kafka Brokers form a Kafka Cluster that distributes incoming data streams and messages into logical groups, so-called Kafka Topics, to keep the individual processing workload low and guarantee data access to further systems like the digital twin.

2.3 MontiArc

MontiArc is an architecture description language [17]. Its prime modeling elements are component types with interfaces of typed and directed ports. The components either are atomic, and feature a behavior model or General Purpose Language implementation, or composed. Composed components contain a topology of subcomponents that exchange messages via unidirectional connectors between the ports of their typed, directed interfaces. Their behavior emerges from the behavior of their hierarchically contained subcomponents.

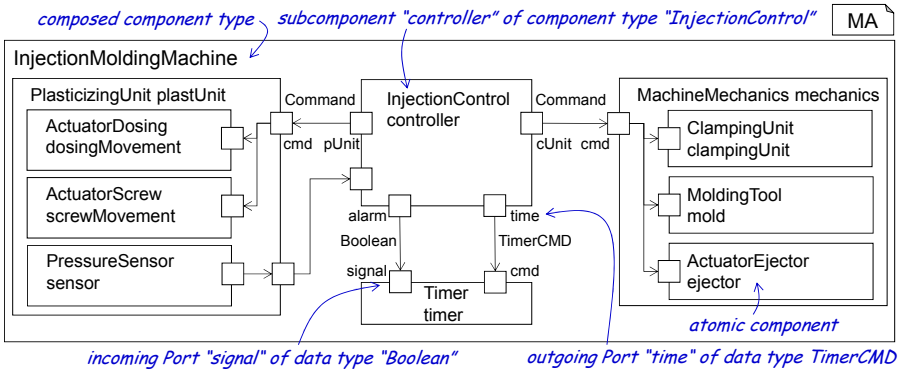


Fig. 1. MontiArc model of a simplified injection molding control flow showing injection molding machine components involved in the process

Fig. 1 illustrates the quintessential modeling elements of MontiArc by example of an injection molding machine. The component type `InjectionMoldingMachine` hierarchically contains subcomponents of types `PlasticizingUnit`, `InjectionControl`, `Timer`, and `MachineMechanics`. The subcomponent `plastUnit` of component type `PlasticizingUnit` is composed again and features three subcomponents itself. At the core of the model is the component `controller` of type `InjectionControl` that interacts with `plastUnit` and `mechanics` and manages the injection molding process.

3 Example and Challenges

Several setting parameters like the volume flow profile, the ideal switchover volume from injection phase to holding pressure phase as well as the right processing

temperatures influence the reproducibility and the profitability of the current operating point in injection molding processes. To produce plastic parts with high quality, the interdependencies of these parameters need to be respected during setup. However, a correlation of setting parameters to the final part quality is, in most cases, only possible implicitly as the settings induce a specific process behavior – represented via process models – that results in process data like a respective cavity pressure. A quality model afterward describes the correlation of process data to the final part quality [13]. To determine the ideal operating point, a well-experienced operator is necessary or an extensive DoE that uncovers correlations by statistical analysis of targeted trials.

The phases of the cyclic process require specific values that – in most cases – refer to basic estimations. The clamping force, for example, is necessary to keep the mold closed during injection and to hold against the injection pressure. Therefore, basic estimations refer *e.g.,* to a known specific clamping force (*e.g.,* 3.0 - 6.5 kN/cm for a standard polypropylene) multiplied by the projected area of the part geometry and the number of cavities inside the mold [20]. However, high values for the clamping force can lead to high energy consumption and increased wear of the mold that can be avoided by an automated adaption to the realized injection pressure during injection. Nevertheless, feedback of the machine data for automated adaption to current process behavior is rarely implemented.

The actual injection is one of the most crucial process phases as it determines crucial quality aspects like weld lines, incomplete filling or burners. Therefore an operator needs to set an injection flow profile [cm/s] in accordance with the respective part geometry. Due to differences in the wall thickness of the part and the overall part geometry, the melt front velocity tends to accelerate or decelerate if the screw induces a constant volume flow. A constant melt front velocity inside the mold, on the contrary, is beneficial to realize high quality for the molded parts. Cavity pressure sensors are capable of monitoring the characteristic volume flow as a constant melt front velocity results in a linear slope of the pressure curve during injection [22,28]. A digital twin thereby might be able to analyze the incoming digital shadow from the filling process as data trace from cavity pressure sensors and adjust the volume flow profile to realize a constant melt front velocity for high-quality parts.

4 Methodology

In the industry, there are digital twins of products, CPPSs and their services, and complete production facilities. We present a reference architecture for digital twins on the use case of injection molding that facilitates adaptivity and extensibility while reusing existing engineering artifacts that describe the system, the production process, or the workpiece. Thereby, our development method is model-driven, facilitates consistency, and reduces manual effort. The term Model-Driven Engineering is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed into concrete implementations [5].

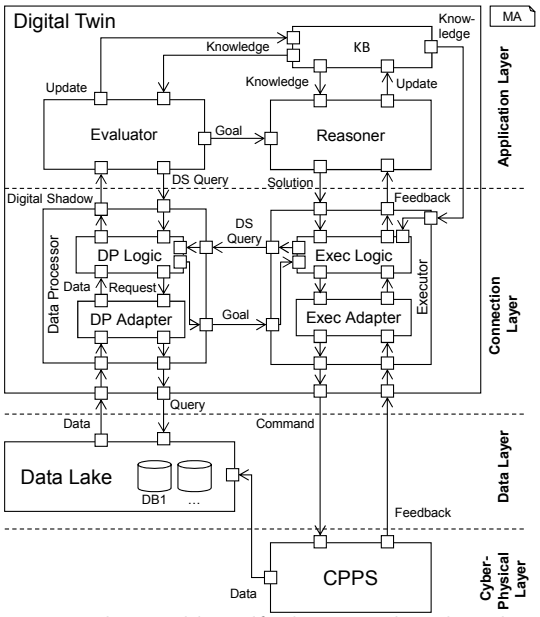


Fig. 2. Architecture that enables self-adaptation based on digital shadows.

4.1 Digital Twin Reference Architecture

We describe the reference architecture for DTs as a component and connector architecture in MontiArc. The components of the DT interact with each other via typed and directed ports. Fig. 2 depicts the reference architecture and its layers: cyber-physical layer, data layer, connection layer, and application layer.

Cyber-Physical Layer. The cyber-physical layer describes the CPPS and its components that the DT controls. The reference architecture specifies one general component that may be hierarchically composed of more specific components describing the system in detail.

Data Layer. The `Data Lake` [9] is an extensive data storage consisting of multiple databases or other data providers and is situated in the data layer. It stores data from a wide variety of sources *e.g.,* sensors inside of the CPPS in a raw format or in a preprocessed form. It can contain both unstructured and structured data. To support reusability, the data is annotated with metadata containing semantic information. Data Lakes also offer logic for data preparation and processing. This logic is implemented by the suppliers of data lakes, thus we do not model its components here.

Connection Layer. The connection layer contains a `Data Processor` and an `Executor`. The `Data Processor` links the `Data Lake` with components at the application layer. It creates digital shadows that encapsulate exactly the information that is required by components at the application layer. The `Data`

`Processor` contains two inner components. The `Data Processor Logic` receives DS queries of the application layer and transforms these into data requests. The `Data Processor Adapter` transforms data requests into queries for specific databases that are part of the `Data Lake`. The `Data Processor Adapter` returns the collected data to the `Data Processor Logic` that links this data, and creates a DS. The `Executor` interacts with the CPPS and controls its behavior. It receives DSs that describe the CPPS's operating context and current status from the `Data Processor`. Further, it receives a solution from the application layer that describes how the CPPS should behave. To realize this behavior, it potentially requires knowledge about the system and its structure that is available in the knowledge base. The executor has two inner components: the `Execution Logic` and the `Execution Adapter`. The `Execution Logic` derives a concrete plan that shall be executed at the CPPS and its surrounding systems. The `Execution Adapter` sends concrete commands to specific parts of the CPPS and thus controls the next actions. Feedback about the success of these commands is also processed and handed back to the application layer.

Application Layer. The application layer contains the actual smartness of the digital twin. The `Evaluator` analyzes DSs and reacts to events that occur within the system or its context. To decide on which events it must react, it refers to design-time models that describe the expected behavior of the system and also possibly erroneous behavior. The `Evaluator` also relies on knowledge from the knowledge base to decide when an event is considered negative and must be handled. If the `Evaluator` detects patterns that indicate an event, it can query more detailed DSs from the `Data Processor`. The `Evaluator` also learns about changes in the system, and adds new knowledge to the knowledge base, *e.g.,* that a new sensor is detected. Depending on the system's state and evaluation results, the `Evaluator` creates goals that it sends to the `Reasoner`. The `Reasoner` receives goals that specify what should be changed in the system's state. The `Reasoner` uses the knowledge contained in the knowledge base to decide how to adapt the system's behavior.

4.2 Model-Driven Development of a Digital Twin

We develop a model-driven methodology that facilitates automatic generation of DTs from models describing a CPPS. Fig. 3 describes the development and adaptation process for developing DTs that ground on our reference architecture. The reference architecture is implemented in MontiArc but leaves domain-specific decisions open. Thus, software engineers can adapt it to various domains. The first step is to create a domain model that describes the structure of data that is exchanged between components of the DT. As the twin monitors the system's state, the next step is to decide what kinds of events occur in the system and how the twin should react if they occur. To this end, we developed a domain-specific language that facilitates the specification of events and actions. An event describes a situation in the real system, *e.g.,* a monitored parameter reaching a threshold. Actions specify the twin's reaction to an event. Rules link events and

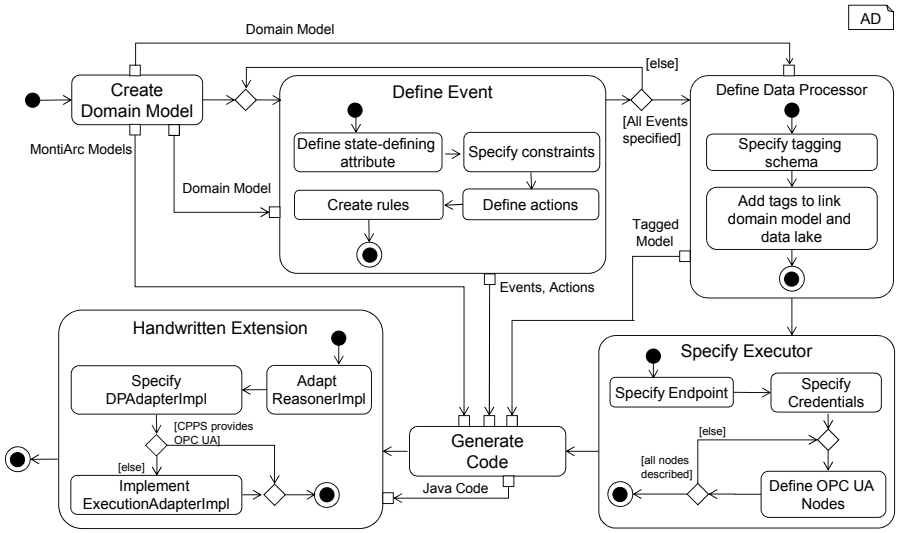


Fig. 3. Activity diagram of the development process of DSs based on our reference architecture and tooling.

actions. Thus, if an event occurs, the twin reacts with one or multiple actions. Events reference classes of the domain model that specify the structure of data that the event processes. The state-defining attribute specifies on which changes of the system's data the event should be evaluated.

The `Data Processor` receives raw data from the data lake and transforms it into DSs that the evaluator can monitor and evaluate. Tagging [7] the class diagrams allows enriching the domain model with specific data retrieval information in order to build up DSs. The `Executor` interacts with the CPPS and sends commands that adapt the CPPS's behavior. We developed the OPC-UA Description Language (OPCDL) to specify the communication interface to the CPPS. If the production system provides an OPC-UA interface, it suffices to specify the endpoint, credentials, and nodes to realize the executor. The generator parses the models describing the DT MontiArc architecture, domain, data processor, and executor and creates Java code for the DT. Finally, the developer of the DT adapts the generated code where necessary. The developer implements the `Reasoner's` behavior that describes the purpose of the DS and how it manipulates the physical system. The DT developer also implements adapter for specific databases of the data lake and in case the CPPS does not provide an OPC-UA interface another adapter for interacting with the system. As the domain model centrally specifies the parameters relevant for the process and the control and the other models reference these, only one model has to be adapted when changes occur. The generator links information from all models and derives Java artifacts for the DT. This way, we can ensure that component implementations always stay consistent and syntactically understand the exchanged data.

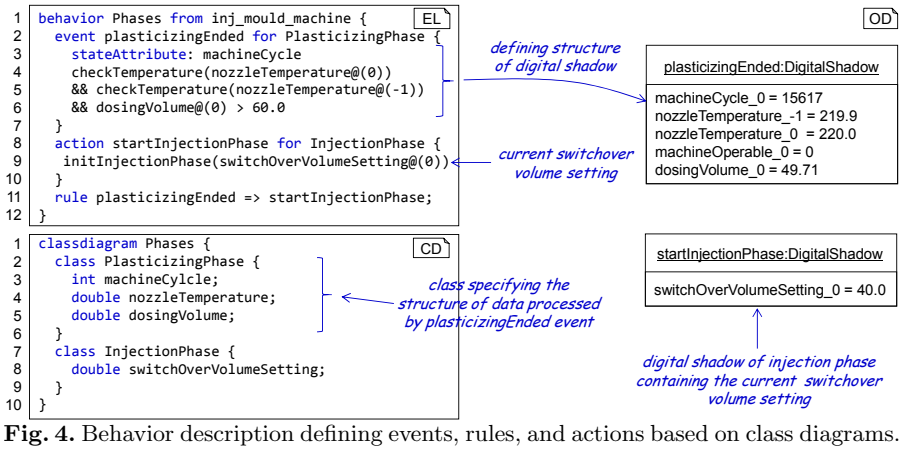


Fig. 4. Behavior description defining events, rules, and actions based on class diagrams. Additionally, showing that the structure of the DS is determined by the behavior definition.

5 Technical Realization

The DT reference architecture presented in this paper is built to be flexible by using exchangeable components implemented in MontiArc and a model-based approach for describing CPPS-specific properties. Our DT detects and reacts to patterns gathered from CPPS data. The Event Language (EL) supports the formulation of events based on attributes of class diagrams (CDs). A generator then produces code which comprises the logic for checking events and performing the related actions.

Fig. 4 shows an excerpt of the behavior definition of the phases of an injection molding machine, which contains the event `plasticizingEnd` and the action `startInjectionPhase`. The keyword `for` (l. 2) indicates the corresponding domain class whose information is used to check the event. A `stateAttribute` is an attribute whose value is stored, and the corresponding event is only triggered if the evaluated value of the state attribute has changed compared to the last event trigger. The event definition block contains expressions about the values of the DT, such as external calls (l. 5), logical expressions (&&, ||, !), and value comparisons (l. 7), . The rule (l. 13) links the event and the corresponding action. The right side of the figure shows the corresponding DS, which are used to either check the event or perform the action. Type safety is ensured by the CD. The @-notation specifies the point in time from which the value is queried. @(0) specifies the current value, whereas @(−1) specifies the previous value of a parameter. As DTs work remotely, some information about data retrieval is required. A tagging language [7] is used to add this information to the CD while at the same time keeping it clean. Hence, the tagged values are available for the `DataProcessor`. When configuring the injection molding machine for production, the optimal values of the parameters highly depend on the wear of the machine, and environmental influences. To this end, usually, a series of experiments with varying parameter values are evaluated. The DT architecture

```
1  design of experiment VarySwitchOverVolumeAndNozzleTemp {                    DoE
2    factorized = fully
3
4    // injection phase
5    param StageCountInjectionPhase = 2
6    param SwitchOverVolume         = (min = 39, intermediate = 40.5, max= 41), 20
7    param InjectionFlow            = 30.0, 31.0
8    param InjectionPressure        = 800.0, 800.0
9    param TemperatureFeedingZone   = 40
10   param CylinderHeating1         = 210.0
11   param NozzleTemperature        = (min = 215.0, max = 218.0)
12   // dosing phase
13   param StageCountDosingPhase    = 2
14   param DosingVolume             = 80.0
15   param BackPressure             = 150, 145
16   // ...
17 }
```

fully factorial design method

second stage with a fixed value of 20

first stage with variable values between 39 and 41

Fig. 5. Fully factorial design of experiment for varying switch-over volumes and nozzle temperatures.

automates the design of such experiments by providing the modeling language DoE. The language supports the fixed or variable assignment of parameter values, optionally configuring the number of adjustment and measuring cycles, and several factorial design methods, including fractional factorial designs [4]. When a DoE model is provided, the `Reasoner` manages the optimal and automated execution of the trials.

Fig. 5 shows the DoE for varying the switchover volume (l. 6 first stage) and nozzle temperature (l. 11). As the factorial design method is set to `fully` (l. 2), the plan represents $3^2 = 9$ (all combinations of three variable values for the two parameters) different parameter settings. A value can be assigned directly to a parameter or is described variably with a minimum, an intermediate value, and a maximum. The intermediate value is inferred as the average if only a minimum and maximum is specified (l. 11). Furthermore, in practice, some parameters are finely adjustable in several stages. `BackPressure` (l. 15) has a value of 150 bar in the first stage and 145 bar in the second stage. The `Reasoner` orders all resulting parameter settings such that the overall number of changes in temperature values between consecutive settings is minimized. Thus, it optimizes the resulting work piece and reduces production and waiting times. Closely related to the DoE is the configuration and accessibility of the parameters on the actual machine. The provided interfaces across different machines and domains vary, but more and more machine manufacturers implement OPC-UA or a respective specification as standard communication interface. We developed the OPCDL that provides the definition of OPC object nodes. Additionally, the model designer has the option to specify connection information, including authentication and encryption aspects.

Fig. 6 shows the parts of the OPC-UA interface of an all-electric injection molding machine of the type ARBURG ALLROUNDER 520 A 1500 that is used in the field test. Login, endpoint, and encryption information are stated to enable establishing a connection to the machine (ll. 2-8). An OPC object node is provided also (ll. 10-22). It comprises all important information about

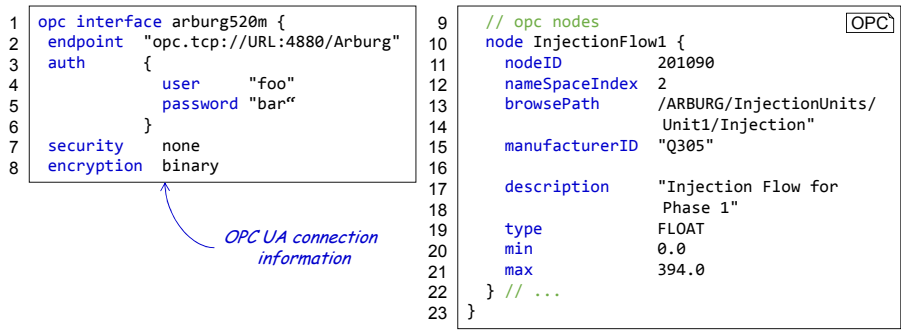


Fig. 6. OPC UA Description Language model describing OPC object nodes.

the node, such as the `nodeID` and the `type`. The properties `min` and `max` help the `Reasoner` and `Executor` to detect an invalid value before sending it to the machine. The `manufacturerID` is not required for the communication with the machine but usually known and used as term by the machine operator and mechanical engineers. The node `InjectionFlow1` (l. 10) corresponds to the first stage of the DoE in Fig. 5 (l. 7, first value). Both models, DoE and OPCDL, are automatically linked in the `Executor` based on the names of the DoE parameters and OPC nodes.

6 Case Study

Injection molding requires time-consuming experiments to determine the ideal settings to run a reproducible and high-quality production process. A central composite design for three variating parameters already takes 15 operating points, each with several process cycles to run until the injection molding machine reaches a steady state and additional process cycles and parts produced for the actual measuring of data and quality criterions. Therefore, a digital twin is necessary that is capable of generating and executing DoEs autonomously and evaluating the resulting influences.

The proposed architecture supports the desired purpose as the developed DT is generally capable of performing experiments autonomously. Based on an analysis focus for specific parameters, the DT generates a DoE and suggests appropriate upper and lower values. Additionally, the twin arranges the planned trials in such a way that changes between the parameters of each trial are minimal. The order of trials is crucial, as, *e.g.,* temperature variations require some time for balancing and, thus, should be minimal. At the current proof-of-concept status, the DT implementation accesses the control of the injection molding machine by ARBURG. Via OPC-UA, it sets the respective values for running an operating point of the DoE. Currently, an operator still has to finally start the process and accompany it, while the DT changes parameters in the ongoing process. For data gathering, the DT connects to the Kafka Broker and gathers data about, *e.g.,* the injection phase as a digital shadow.

In our case study, the DT investigates the optimal values injection phase, where the significant parameters are the injection flow, nozzle temperature, and switchover volume. The injection flow defines how fast the machine injects plasticized material in terms of volume per time. The nozzle temperature describes the temperature at the nozzle through which the machine injects material into the mold cavity. The switchover volume specifies the volume for a phase transition from injection to holding pressure to occur. The DT automatically designs experiments that test different values for these parameters. The DoE variates the injection flow from 30 cm/s to 50 cm/s, the nozzle temperature from 220 C to 260 C and the switchover volume from 10 cm to 20 cm. In the upcoming developments, the DT will analyze the machine and process data it gets from the Kafka Broker and parameterizes a static process model (*e.g.,* regression model). The first estimation for a local optimum can thereby be derived and set as an operating point with ongoing data monitoring as a continuous digital shadow. However, further CPPS components like the linear handling robot and the weight control need, therefore, to be automated.

7 Discussion

The presented methodology and reference architecture enable the automated generation of a DT for setting up and executing a DoE on an injection molding machine. The DT gathers relevant data and is able to transmit commands to the machine in order to make changes to its current settings. The DT is thus capable of detecting events and reacting to these. However, a fully-automated production run cannot be initialized as an operator is required to access physical controls. Respective signals cannot be set digitally as the machine denies write access to these values. The current technical implementation furthermore only covers a proof-of-concept state. Further integration of and interconnection with additional assets, like a tempering unit or a weight control, needs to follow, as must enhanced automation, in order to give the DT extensive control access.

However, the model-driven approach of the architecture highly supports exchangeability and flexibility. For example, if in another setup, we want to examine the machine data and adapt the injection molding process in real-time, an exchange of the `Reasoner` component (Fig. 2) is required while leaving the other parts of the architecture unaffected. The `Reasoner` itself, on the contrary, has to be developed and implemented manually for specific use cases as various applications of DTs for CPPS exist. In the case study, we realized the execution of a DoE but in other scenarios, DTs have different behaviors and goals. Another entry barrier for using our reference architecture and methodology is the use of domain-specific languages. They are tailored to support the specification of DTs but in other fields of application different notations might be common and therefore, modeling relevant data elements and behaviors might be challenging.

8 Related Work

In the field of Industry 4.0, Internet of Things and Internet of Production, there exist various application domains of DTs. In the automotive domain among others, [6] presents DT approach addressing safety, maintenance and reliability of parts or built-in systems of vehicles. Furthermore, the prediction of potential future actions of neighboring vehicles in order to increase safety is presented in [3]. [1, 25, 30] on the contrary address smart shopfloor management. Linking of human-based production tasks [18], geometry assurance in individualized production [24], and parallel controlling of smart workshops [15], and the integration of edge, fog and cloud computing in smart manufacturing [19] shows the diversity of DT in manufacturing. All DTs mentioned above represent very specific and individualized solutions to the respective problems. Contrary to this, the DT reference architecture presented in this paper is highly flexible and supports reusability for different use case scenarios. It is adaptable to all kinds of problems and domains. The model-driven development process enables automating major parts of the development process and thus reduce manual effort for adapting the DT for new CPPS.

Injection molding represents a relevant use case for realizing smart production processes. Previous work in the Cluster of Excellence at RWTH Aachen University and at the Institute for Plastics Processing have already elaborated data-driven approaches for process setup [10–12, 26]. Artificial Neural Networks, therefore, are trained with simulation data to learn about parameter correlations from engineering models. Each process point of the previously simulated DoE is conducted at the real production system. The resulting data is then fed back to the Neural Network for post-training and adjusting the estimations. The methodology has already been implemented as a closed-loop system that uses autonomously conducted DoEs for targeted data gathering and post-training [21]. However, the implementation caused high effort for a single application scenario that serves now as starting point for autonomous code generation and for developing self-adjusting DTs.

9 Conclusion

We have presented a model-driven reference architecture and DSLs to realize reactive DT for Cyber-Physical Production Systems. The reference architecture is specified in MontiArc and thus facilitates the exchangeability of components of the DT. The presented method relies on models describing the DT's situations (events) and reactions. We, therefore, introduced a DSL to specify events that occur in the CPPS and how the twin should react to these events. Furthermore, we presented a DSL for specifying the communication with the CPS via OPC-UA that facilitates connecting MontiArc models with embedded behavior models to manufacturing CPS. We evaluated the described methodology for automating experiments that determine an ideal operating point for an injection molding machine. Thus, we showed that the DT reference architecture serves as a starting point for systematically developing DTs for injection molding. In the future, we

plan to apply our reference architecture and its DSLs to different manufacturing domains to improve the usage of manufacturing equipment and resources. This, ultimately, can reduce resource consumption, manufacturing time, and cost.

References

1. Brenner, B., Hummel, V.: Digital twin as enabler for an innovative digital shopfloor management system in the ESB Logistics Learning Factory at Reutlingen-University. Procedia Manufacturing **9**, 198–205 (2017)
2. Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Architectural Programming with MontiArcAutomaton. In: In 12th International Conference on Software Engineering Advances (ICSEA 2017). pp. 213–218. IARIA XPS Press (May 2017)
3. Chen, X., Kang, E., Shiraishi, S., Preciado, V.M., Jiang, Z.: Digital behavioral twins for safe connected cars. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. pp. 144–153. ACM (2018)
4. Choudhury, I., El-Baradie, M.: Machinability assessment of inconel 718 by factorial design of experiment coupled with response surface methodology. Journal of Materials Processing Technology **95**(1-3), 30–39 (1999)
5. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering. pp. 37–54. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007)
6. Glaessgen, E., Stargel, D.: The digital twin paradigm for future NASA and US Air Force vehicles. In: 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference 20th AIAA/ASME/AHS Adaptive Structures Conference 14th AIAA. p. 1818 (2012)
7. Greifenberg, T., Look, M., Roidl, S., Rumpe, B.: Engineering Tagging Languages for DSLs. In: Conference on Model Driven Engineering Languages and Systems (MODELS'15). pp. 34–43. ACM/IEEE (2015)
8. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University (February 2012)
9. Hai, R., Geisler, S., Quix, C.: Constance: An intelligent data lake system. In: SIGMOD Conference (2016)
10. Hopmann, C., Heinisch, J., Tercan, H.: Injection molding setup by means of machine learning based on simulation and experimental data. In: ANTEC 2018 Conference and Tradeshow. Orlando, Florida, USA (2018)
11. Hopmann, C., Jeschke, S., Meisen, T., Thiele, T., Tercan, H., Liebenberg, M., Heinisch, J., Theunissen, M.: : Combined learning processes for injection moulding based on simulation and experimental data. In: Proceedings of the 33rd International Conference of the Polymer Processing Society (PPS33). Polymer Processing Society, Cancun, Mexico (2017)
12. Hopmann, C., Wahle, J., Theunissen, M., Heinisch, J., Bibow, P., Lammert, N., Kessler, F.: : Flexibilisation of injection moulding manufacture through digitisation. In: 29th International Colloquium Plastics Technology. Shaker Verlag, Aachen (2018)
13. Klocke, F., Abel, D., Hopmann, C., Auerbach, T., Keitzel, G., Reiter, M., Reßmann, A., Stemmler, S., Veselovac, D.: Approaches of self-optimising systems in manufacturing. In: Brecher, C. (ed.) Advances in Production Technology, pp. 161–173. Springer International Publishing, Cham (2015)

14. Kudlik, N.: Reproducibility of the plastic injection moulding process: RWTH Aachen University, Dissertation. Verlag Mainz, Wissenschaftsverlag (1998)
15. Leng, J., Zhang, H., Yan, D., Liu, Q., Chen, X., Zhang, D.: Digital twin-driven manufacturing cyber-physical system for parallel controlling of smart workshop. Journal of Ambient Intelligence and Humanized Computing **10**(3), 1155–1166 (2019)
16. Mahnke, W., Leitner, S.H., Damm, M.: OPC Unified Architecture. Springer Science & Business Media (2009)
17. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)
18. Nikolakis, N., Alexopoulos, K., Xanthakis, E., Chryssolouris, G.: The digital twin implementation for linking the virtual representation of human-based production tasks to their physical counterpart in the factory-floor. International Journal of Computer Integrated Manufacturing **32**(1), 1–12 (2019)
19. Qi, Q., Zhao, D., Liao, T.W., Tao, F.: Modeling of cyber-physical systems and digital twin based on edge computing, fog computing and cloud computing towards smart manufacturing. In: ASME 2018 13th International Manufacturing Science and Engineering Conference. pp. V001T05A018–V001T05A018. American Society of Mechanical Engineers (2018)
20. Rao, N.S., Schott, N.R.: Understanding plastics engineering calculations: Hands-on examples and case studies. Hanser and Hanser Publications, Munich and Cincinnati, Ohio (2012)
21. Schmitz, M., Hopmann, C., Röbig, M., Pelzer, L., Topmöller, B., Wurzbacher, S.: Jenseits menschlicher fähigkeiten. modellgestützte prozesseinrichtung durch vollvernetzte produktion im spritzgießen. Kunststoffe **109**(9), 142–145 (2019)
22. Schnerr-Häselbarth, O.: Automation of online quality control in injection moulding of plastics: RWTH Aachen University, Dissertation. Mainz, Aachen (2000)
23. Shen, C., Wang, L., Li, Q.: Optimization of injection molding process parameters using combination of artificial neural network and genetic algorithm method. Journal of Materials Processing Technology **183**(2-3), 412–418 (2007)
24. Söderberg, R., Wärmefjord, K., Carlson, J.S., Lindkvist, L.: Toward a digital twin for real-time geometry assurance in individualized production. CIRP Annals **66**(1), 137–140 (2017)
25. Tao, F., Zhang, M.: Digital twin shop-floor: a new shop-floor paradigm towards smart manufacturing. Ieee Access **5**, 20418–20427 (2017)
26. Tercan, H., Guajardo, A., Heinisch, J., Thiele, T., Hopmann, C., Meisen, T.: Transfer-learning: Bridging the gap between real and simulation data for machine learning in injection moulding. In: 51st CIRP Conference on Manufacturing Systems / Edited by Lihui Wang, vol. 72, pp. 185–190. Elsevier (2018)
27. Thein, K.M.M.: Apache kafka: Next generation distributed messaging system. International Journal of Scientific Engineering and Technology Research **3**(47), 9478–9483 (2014)
28. Vaculik, R.: Improved control of part quality in injection molding based on statistical process models: RWTH Aachen University, Dissertation. Verlag der Augustinus-Buchh., Aachen (1996)
29. Wortmann, A., Barais, O., Combemale, B., Wimmer, M.: Modeling languages in Industry 4.0: an extended systematic mapping study. Software and Systems Modeling pp. 1–28 (2019)
30. Zhang, H., Zhang, G., Yan, Q.: Digital twin-driven cyber-physical production system towards smart shop-floor. Journal of Ambient Intelligence and Humanized Computing pp. 1–15 (2018)

Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems

Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, Andreas Wortmann
Software Engineering, RWTH Aachen University, Germany, www.se-rwth.de

ABSTRACT

Digital twins emerge in many disciplines to support engineering, monitoring, controlling, and optimizing cyber-physical systems, such as airplanes, cars, factories, medical devices, or ships. There is an increasing demand to create digital twins as representation of cyber-physical systems and their related models, data traces, aggregated data, and services. Despite a plethora of digital twin applications, there are very few systematic methods to facilitate the modeling of digital twins for a given cyber-physical system. Existing methods focus only on the construction of specific digital twin models and do not consider the integration of these models with the observed cyber-physical system. To mitigate this, we present a fully model-driven method to describe the software of the cyber-physical system, its digital twin information system, and their integration. The integration method relies on MontiArc models of the cyber-physical system's architecture and on UML/P class diagrams from which the digital twin information system is generated. We show the practical application and feasibility of our method on an IoT case study. Explicitly modeling the integration of digital twins and cyber-physical systems eliminates repetitive programming activities and can foster the systematic engineering of digital twins.

CCS CONCEPTS

• **Software and its engineering** → *Architecture description languages*; **Integration frameworks**; • **Computer systems organization** → *Embedded and cyber-physical systems.*

KEYWORDS

Model-Driven Software Engineering, Cyber-Physical Systems, Digital Twins, Information Systems, Software Architecture

ACM Reference Format:
Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, Andreas Wortmann . 2020. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3365438.3410941

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410941

1 INTRODUCTION

Motivation. There is an increasing demand for the fast and agile creation of digital twins [56, 67], namely digital representations of cyber-physical systems (CPSs), in a variety of disciplines, *e.g.,* marine [33, 39, 70], smart manufacturing [68, 69], avionics [35, 41], building information and energy management [21, 31, 40], automotive [10, 11] or health care [15, 32, 37]. Such a digital twin (DT) comprises models, data traces, (aggregated) data representations, and services to represent, monitor, control, or even optimize the observed CPS. Digital twin information systems (DTISs) with a set of graphical user interfaces provide a convenient and effective way to manage a CPS [34]. The DTIS would be responsible for displaying the data and allowing for interaction with both users and the CPS. The CPS then handles all the cyber-physical elements and shares its data with the DTIS. As such, DTISs can serve as viable bases for representing and monitoring CPSs, *i.e.,* acting as their DTs. Clearly, the DTIS and CPS have to share a great number of interfaces to be able to communicate about data and models.

Open Challenges. Until now, large parts of the connections between such interfaces had to be crafted manually. These implementation tasks do not require high cognitive performance of the developers but are, due to the number of interfaces, time-consuming, and hence, error-prone. As the tasks and the artifacts to be developed are highly repetitive, this is a good candidate for improvements [64]. Following the idea of model centered architecture [43, 44], models can be used for the flexible definition of any kind of system interfaces and communication. Through making these interfaces and their connections explicit in suitable models, creating these repetitive artifacts can be automated. This improves efficiency and consistency in engineering DTs for CPSs. Although model-driven software engineering (MDSE) provides the necessary methods to generate these connections, these methods have not yet been applied to integrated development and connection of DTs to CPSs.

Contribution. In this paper, we address the challenge of reducing the effort for engineering the communication interfaces between cyber-physical systems and digital twins implemented as information systems. To this end, the paper conceives a model-driven method for the integration of CPSs and DTISs using a novel, domain-specific tagging language that decouples the development of both systems. This separates the concerns involved, and many related development tasks can then be fully automated.

Our contribution, hence, consists of

- A development process for the model-driven integration of CPSs and DTISs.
- A model-driven solution for the generative extension of architecture models and class diagrams (CDs) with elements that keep their data synchronized.

- A method for clearly separating business logic and synchronization infrastructure in model-driven systems using DTs.

Structure of the paper. In the following, Section 2 introduces preliminaries on concepts, modeling languages, and tools used in the remainder. Section 3 presents the requirements for our system. Section 4 introduces our running example, the automatic fire extinguishing system, and shows how it can be represented with different types of models. Section 5 presents how to enhance models with further information about component communication and how to generate the synchronization infrastructure between a DTIS and the corresponding CPS. Section 6 shows the application of our method in a case study. Section 7 relates our approach to other approaches and Section 8 discusses it. Section 9 concludes.

2 BACKGROUND

This section introduces our notion of digital twins, the MontiArc architecture description language, which we leverage to model the architecture of CPSs, the MontiGem code generation framework for the efficient engineering of DTISs, and the tagging language framework used to combine the CPSs with the DTISs.

2.1 Digital Twins

DTs are digital representations of cyber-physical assets or processes that enable advanced control, decision making, and optimization. They are used in a variety of domains, including avionics, automotive, and smart manufacturing [12, 26, 68].

While the use of DTs promises to improve the use of CPSs in many ways, intensional definitions of DTs are rare and vague, such as (1) "An always in sync digital model of existing manufacturing cells throughout the life cycle" [66], (2) "[...] virtual product models, which are frequently referred to as DTs" [60], or (3) "[...] a set of virtual information constructs that fully describes a potential or actual physical manufactured product from the micro atomic level to the macro geometrical level" [26]. Such approaches to definitions often use the term model—opposed to the commonly accepted definition of Stachowiak [63]—in a sense that the reduction property (*i.e.*, the model is an abstraction of the original for a specific purpose) cannot be adhered to. Often, these definitions also focus on very specific applications, such as "manufacturing cells" or "product models." Hence, a commonly accepted definition still is lacking.

Based on a joint effort within the German "Internet of Production"[1] cluster of excellence research project of 200 researchers of 25 departments conducting research in artificial intelligence, computer science, innovation research, labor science, mechanical engineering, and production technology [61], we conceived the following definition on the constituents of DTs that is liberated from specific applications, focuses on its contents, and separates data and models:

A 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 respects to the original system.

From this, it follows that (1) A DT is not a model itself: instead it comprises models of the system it represents. These can be the engineering models used to build the developed system, models derived from these, or abstractions of the data traces observed by the DT. (2) A DT can be made active by invoking its services, which may comprise databases, user interfaces, analyses, and even the interaction with other systems. (3) A DT leverages its models and data traces to converge the observations coming from the represented system and from itself.

This supports the investigation of a variety of DTs, such as development digital twins used during the development of the (to be) represented system, usage DTs that represent the system as operated, diagnostic DTs that support detailed analysis of the represented system in its context, and many more.

In this respect, our contribution focuses on efficiently modeling DTs comprising data structures representing properties of CPSs by relating interfaces of the CPSs' architecture models to data structure properties. The data structures are used by a DTIS that may aggregate and abstract this data prior to visualization and further use. These DTs offer services through their software architecture as well as through human interaction with the DTIS.

[1] Internet of Production: https://www.iop.rwth-aachen.de/

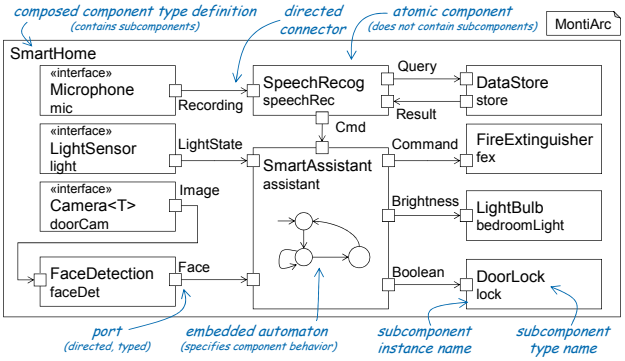


Figure 1: MontiArc architecture of a smart home system

2.2 MontiArc

MontiArc [18, 27] is an extensible [17, 57] architecture description language [45] for the efficient engineering of CPSs. The language comprises modeling elements for atomic and composed component types that exchange messages via the directed and typed ports of their interfaces. Atomic component types yield embedded behavior (*e.g.*, automata) models or general-purpose language (GPL) implementations that define their behavior directly. Composed components are hierarchically composed and yield topologies of subcomponents. Hence, their behavior emerges from their subcomponents' behavior.

Figure 1 illustrates MontiArc's most important modeling elements on the software architecture for a smart home: the system boundaries are defined by the `SmartHome` component type that contains ten subcomponents of different component types, such as the subcomponent `mic` of component type `Microphone`. The subcomponents `mic`, `light`, and `doorCam` sense the smart home's environment and send their data either into post-processing components (such as `speechRec` or `faceDet`) or directly into the central component `assistant`. Based on these inputs, the central controller, *i.e.*, the `assistant`, decides on the next actions and activates the actuators `fex`, `bedroomLight`, and `lock` on the right.

MontiArc supports various features to facilitate architectural programming, such as generic data type parameters for component types, interface components, component type inheritance, component parametrization, injection of component instances into composed components, or dynamic reconfiguration [29]. In Figure 1, the component type `Camera`, *e.g.,* defines a generic data type parameter that can be used to define the data type of its single outgoing port. For the component instance `doorCam`, this parameter bound to the data type `Image`. `Camera`, as well as the component types `Microphone` and `LightSensor`, also is an abstract component type that does not yield an implementation by design. Instead, this type is replaced by a platform-specific component type that extends it and yields a specific implementation before deployment.

MontiArc models can be translated to Java [58] for educational purposes, to Python [4] for service robotics, and to C++ for Internet of Things (IoT) systems. Through modular language engineering, the MontiArc language and its code generation capabilities can be extended with novel language elements and transformations [17].

2.3 MontiGem

MontiGem [6, 23], the generator framework for enterprise information systems, uses models to generate complete (enterprise) information systems [24]. Different UML/P [59] languages, such as CDs and the object constraint language (OCL), are used as sources. Further domain-specific languages (DSLs) are incorporated for code generation such as the GuiDSL, a graphical user interface (GUI) description language. Using these models, MontiGem generates the data structure, database schema, and (web-)pages, including corresponding data views (ViewModels). Together with the basic runtime environment for the frontend, *i.e.,* the user interface, and the backend, *i.e.,* the data processing, of the information system (IS), the generated code forms an executable application which is extendable by handwritten code.

We derive the database schema and data structure in front- and backend from CDs. This ensures consistency between front- and backend by construction. We use OCL as a restriction language on the data structure and generates validators for data inputs. Commands handle the communication between front- and backend and also depend directly on the CD input. Additional structure and behavior commands can be defined. GUI models describe the layout of the generated (web)page, as well as the used ViewModels. Those ViewModels map the data structure to specified GUI models enabling the generation of views with specific extracts from the data. This enables defining the ViewModels in place, where they are to be displayed. To improve usability and speed up the development process, a set of standard GUI models does not need to be defined manually but can be generated based on the domain models (CDs). This provides an overview of all used data classes but still allows for adaption and extension of the (web)pages using handwritten GUI models and/or code. Additionally, we use a tagging language [25, 42] to enrich the domain model described in the CD. This DSL enables the use of different generator configurations, *i.e.,* what should be generated, or adds implementation-specific information to CD or GUI models.

The MontiGem generator framework enables the generation of a complete IS using only domain-specific CDs citeGMN+20 but

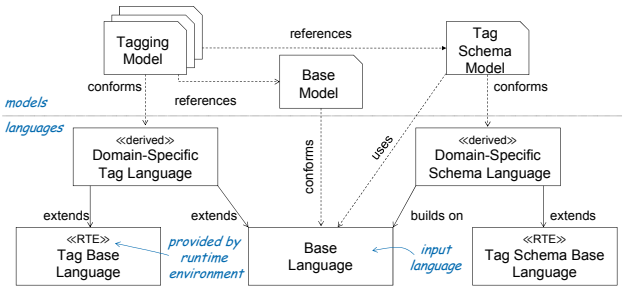


Figure 2: Architecture of the tagging languages based on [25]

allows to use further DSLs for detailed behavior. To allow the inclusion of further DSLs, *e.g.,* behavior and goal models [49] or privacy concepts [48] is in discussion. By now, the resulting IS presents stored data and provides operations to create, edit, or delete data sets. MontiGem is used in several application areas, such as finance cockpits [7, 23], IoT or energy management dashboards. Each specific implementation adapts and extends the generated code with domain-specific logic and additional functionality.

2.4 Tagging

In this work, we use tagging to connect CPSs with their DTs. Tagging [25] is a language engineering technique that enables the non-invasive annotation of existing models of a given base language through models of a domain-specific tagging language automatically derived from the base language. Through this, domain experts can reuse established syntax of the base language in the tagging models for annotating it and do not need to convolute the base model with these annotations. In consequence, this increases the reusability of the base models.

As depicted in Figure 2, tagging is based on a common tag base language (bottom left), which predefines various tag types, and a common tag schema base language (bottom right), which prescribes the structure of tag schemata. Based on these and the base language (bottom middle), the tagging code generators derive a domain-specific schema language and a domain-specific tag language. Models of the former govern the type, number, and shape of tags in conforming tag models. This, *e.g.,* enables annotating the base models with non-functional properties [42] or adding communication information [20]. In general, for a single base language, multiple tag schemata can be defined, and models of the domain-specific tagging language then are validated against the schema they reference. Models of the domain-specific tag language refer to a base model they annotate and to a tag schema model they conform to.

3 REQUIREMENTS

Within the last decade, we have gained experience in various domains including avionics [36, 72], automotive [9, 22], robotics [5, 57], smart homes [50, 65], and manufacturing [47, 48, 71]. These domains are facing the same challenges in creating a connection between a CPS and a DTIS. To automate engineering of these connections, we identified the following requirements based on an analysis of popular IoT tools such as Arduino IoT Cloud, Amazon AWS, or Microsoft Azure (see Section 7 for details):

(R1) *The CPS and the DTIS shall be able to synchronize any data type known to both the CPS and the DTIS.* Until now, integrating CPS and DTIS demands for error-prone handcrafting to map each datatype from one system to the other one from various languages.

(R2) *The communication infrastructure that keeps the CPS and the DTIS synchronized shall be completely generated from corresponding models.* Until now, the integration of both demands repetitive handcrafting and is error-prone.

(R3) *The handwritten artifacts (e.g., models, code) specifying the CPS and DTIS shall not contain information about their integration and the integration of the systems shall not presuppose the content of the handwritten artifacts.* Component developers and system architects of the CPS and frontend and backend developers of the DTIS should be able to work independently on the design of these systems, including modeling aspects. R3 ensures the independent modeling of CPS and DTIS. In addition, R3 ensures that the integration can be applied to legacy artifacts that were not created with DTs in mind.

(R4) *The DTIS shall enable users to manually override the specified behavior of the CPS temporarily or permanently.* This is important to be able to handle exceptional situations. Thus, a user's manual intervention should be possible and override the automatic behavior of the system.

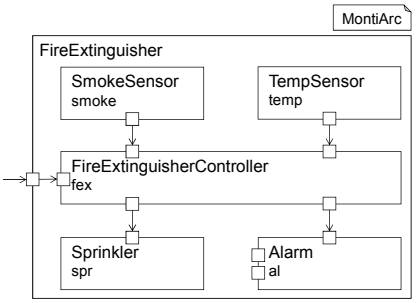
(R5) *The CPS should support heterogeneous platforms as long as they can communicate with the Internet.* To work with platform-independent versions offers hardware flexibility.

The following sections discuss each of these requirements in detail and show how these requirements are met.

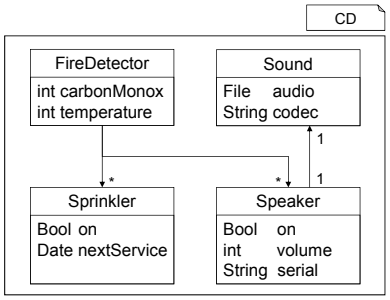
4 EXAMPLE: AUTOMATIC FIRE EXTINGUISHING SYSTEM

In IoT environments, systems need to interact with the real world and connect to DTISs to receive the goals of their users. In the following, we use an automatic fire extinguisher in a smart home environment (*cf.* Figure 1) as a running example. This example is motivated by Google's fire alarm system *Nest Protect*[2]—though our example is simplified for better comprehension. Our simplified version of the architecture is based on the fire alarm architecture shown in [46]. Figure 3 depicts the models used to specify this automatic fire extinguishing system: (a) the MontiArc architecture of the CPS and (b) a CD describing the DTIS's data structures.

The CPS architecture shows two sensors (top), a central controller (middle), and two actuators (bottom). The sensors measure the carbon monoxide concentration in air and the current temperature. This raw data is sent to the central controller, which then decides whether there is a fire or not. If a fire is detected, the controller can react by turning on the sprinklers or triggering a fire alarm. To do so, the controller sends commands to the actuators via its outgoing ports and the attached connectors. While the sprinkler only needs to be prompted to switch on, the `Alarm` component also requires a sound file with the alarm tone and a volume level at which the alarm should be played. The architect, however, did not specify how this information should be provided. This underspecification is reflected by the two ports on the left side of the `alarm` component that are not connected to another component. Allowing such underspecification is crucial in the development process as it allows to defer design decisions to a later stage of the development process where more information about the system is available. However, to generate code from the architecture model, the gaps resulting from underspecification have to be filled.



(a) Underspecified CPS architecture describing a fire extinguishing system that detects fire based on smoke and temperature sensors and uses this information to trigger an alarm and turn on a sprinkler



(b) Domain model of the DTIS describing the data structure used to monitor the fire extinguishing system in online platform

Figure 3: Automatic fire extinguisher system. The MontiArc model (a) describes the logical software architecture of IoT devices. The CD (b) describes the IS data model.

The DTIS domain model shows four data classes that might be used in a DT of the CPS. For example, turning up the `volume` of the `Speaker` in the domain model should cause the volume of the real `Alarm` to increase **(R1)**. Similarly, if the temperature sensor detects a temperature change, the temperature information in the DTIS needs to be updated. While the DTIS's domain model represents a view on the same system, the data structure is different, as the DTIS may contain information that is not required by the CPS architecture, omit data used by the CPS, and structure the data differently. For example, the DTIS domain model also includes a `Date nextService` storing the due date of the next required maintenance. Though this might be valuable information to the user who interacts with the DTIS, the sprinklers themselves do not need this information.

The two models are used as input for MontiArc and MontiGem to generate code that is executed on the IoT devices and in the backend of the DTIS. However, these models do not define the interfaces between the CPS architecture and DTIS, *i.e.,* the CPS does not know how to exchange data with the DTIS and vice versa.

[2]Nest Protect: https://store.google.com/product/nest_protect_2nd_gen_specs

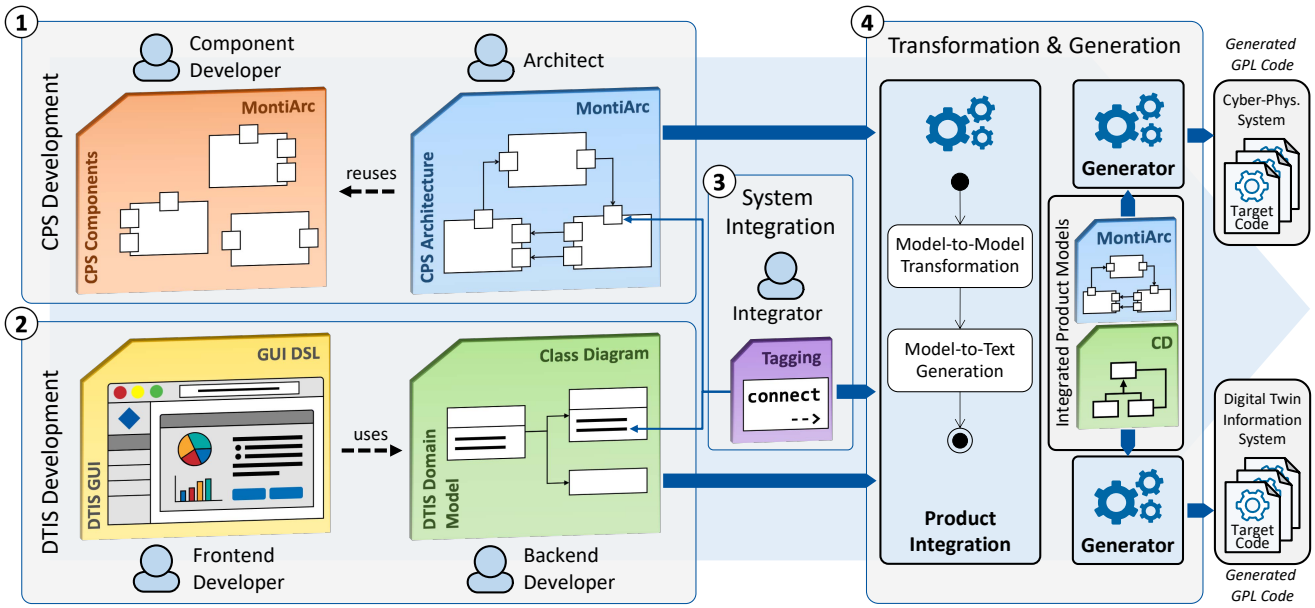


Figure 4: Process overview: The CPS architecture and the IS can be developed independently of one another. The integrator connects the models by tagging elements in the architecture and the IS. Based on these connections, model-to-model transformations create the necessary interfaces between the architecture and the IS.

Implementing connections between both systems is a repetitive and time-consuming task. Clearly, the automatic generation of such interfaces and their automatic integration into existing systems is an attractive option for the development of such systems **(R2)**.

5 INTEGRATING CYBER-PHYSICAL AND INFORMATION SYSTEMS

Our process for developing integrated CPSs and DTISs consists of four activities (*cf.* Figure 4), the first two of which can be performed in parallel: (1) Developing the CPS architecture; (2) Developing the DTIS; (3) Integrating the CPS with the DTIS; and (4) Generating the CPS and DTIS.

The first two steps consist of developing a set of models from which the software running on the CPS's devices and the DTIS can be generated. The two systems can, but do not have to, be developed independently of each other **(R3)**. As the systems may be developed independently of each other, our process can be applied to already existing systems as well as to greenfield, *i.e.,* newly developed, systems including both the development of the CPS and the DTIS. In the third step, the models generated in the first two steps are integrated. The fourth step generates GPL code from the models. This is fully automated.

Step (1): The CPS architecture development starts with developing a set of reusable software components—in our case using the MontiArc architecture description language. Next, the architect connects the components to create an integrated architecture of the CPS. While first developing a set of reusable components independently of the architecture is useful, it is not required to apply our method. It is also possible to start developing architectures for specific products and then later decide which components are worth maintaining independently of the product.

Step (2): The DTIS development includes the development of the front- and backend. The frontend depends on accessing data provided by the backend. Nevertheless, the front- and backend can be developed (partly) in parallel. In our case, the DTIS is developed using MontiGem, *i.e.,* we use class diagrams to describe the domain model, *i.e.,* the data structures used by the backend.

Step (3): Once the CPS architecture and the DTIS have been developed, the integrator tags ports of the CPS architecture with attributes of the domain model and vice versa. This tagging is conceptually based on [25]. Section 5.1 describes this in more detail.

Step (4): Using the tagging created in Step 3, the CPS architecture, and the DTIS's data model as input, a model-to-model transformation extends the CPS and DTIS by the necessary communication and synchronization infrastructure. This step is fully automated **(R2)**. The process for transforming the CPS architecture is described in Section 5.2, and the process for transforming the DTIS is described in Section 5.3. The transformation results in integrated product models using the same MontiArc and class diagram languages that were used by the input models. This allows forwarding the resulting models of the integrated CPS and DTIS to MontiArc's and MontiGem's code generators that produce the necessary GPL code to be executed on the CPS devices and the server that provides the DTIS.

5.1 Tagging CPS Architectures and IS Domain Models

DTs need to stay synchronized with the original system. While the logic that the DTIS may apply to the data of the DT is application-specific, the task of keeping the data values of two systems in sync is repetitive and, therefore, automatable. If both the CPS and the DTIS are developed in a model-driven fashion, tagging can be

utilized to select the model elements of both systems that should stay synchronized.

The tagging model serves two purposes: First, it specifies how to match DTs, *i.e.,* objects instantiated from the domain model, to the physical devices they represent. Secondly, it specifies which ports of the CPS architecture should connect to which attributes of the domain model. Ports are architecture elements that components use to exchange data with other components. Hence, they are ideal candidates for injecting data into or extracting data from the system. Attributes in the domain model are used to store the actual data values. Accordingly, tagging attributes of the domain model and connecting them to ports of the architecture allows specifying which data reflects the state of the CPS. Figure 5 shows such a tagging model.

The two purposes of the tagging model leave two tasks for the integrator (Figure 4) who is responsible for connecting the CPS and the DTIS: (1) Identify which attributes of the domain model are designed to store device identifiers or detect there is no such attribute for a certain device type. (2) Select which attributes of the objects instantiated from the domain model should be synchronized to which ports of the CPS architecture.

For the first task, the integrator needs to find out which attributes of the domain model are intended to store device identifiers of the CPS. If the integrator finds an attribute that stores a device identifier, (s)he specifies it as an identifier in the tagging (ll. 4-5 of Figure 5). Manually specifying the identifier enables the user of the DTIS to create digital twins for future devices. If the user sets the values of these attributes to hardware identifiers of the devices, the system can match the actual devices to their DTs once the devices first go online. Here, objects of the `Speaker` class are identified by the attribute `Speaker.serial`, which stores the serial numbers of the physical devices. Objects of the `Sound` class are identified by the `serial` attribute of the `Speaker` object, which references the `Sound`. This is possible as there is exactly one `Speaker` for every `Sound` (Figure 3(b)).

If the integrator does not find an attribute that stores a device identifier, (s)he can choose to automatically instantiate DTs for devices once they first connect to the DTIS (l. 8 of Figure 5). The `auto identify` keyword is followed by the qualified name of a class from the domain model. This class is then instantiated every time a device with a port that should be synchronized to one of the class's attributes first connects to the DTIS. Each physical device is identified by a unique hardware-specific identifier of the device, *e.g.,* the MAC address of the network interface. The DTIS uses this identifier to connect the ports of the architecture to the new instance of that class, but the identifier will not be part of the data model.

To ensure that any communication with a device is always assigned to the same digital twin, the DTIS must know a permanent device identifier for each device. For the second task, the integrator needs to specify which attributes of the domain model should be synchronized to which ports of the architecture. The remaining lines of Figure 5 (ll. 9-18) show how to connect ports of the CPS architecture to attributes of the domain model of the DTIS and vice versa. If data from the CPS architecture should be sent to the DTIS, the CPS architecture will update the DTIS whenever a new message is sent through the port. For this, the integrator specifies

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
       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.

6  // Automatically create and link a digital twin when
7  // a device first connects to the IS
8  auto identify FireDetector
       class name from domain model.

9  // Send data from the CPS architecture to the IS
10 connect port smokeSensor.value
11     --> attr FireDetector.carbonmonoxy
       architecture component instance   outgoing port
12 connect port temperatureSensor.value
13     --> attr FireDetector.temperature
       "Use messages from temperatureSensor's port 'value' to update the temperature attribute of the FireDetector class"
       class name from domain model   attribute name

14 // Send data from the IS to the CPS architecture
15 connect attr Sprinkler.on          --> port sprinkler.on
16 connect attr Speaker.on            --> port alarm.on
17 connect attr Speaker.volume        --> port alarm.volume
   "If 'Speaker.volume' changes, inject the new value into alarm's 'volume' port"
18 connect attr Speaker.sound.audio --> port alarm.sound
       class name from domain model   (association +) attribute   component instance   port
```

Figure 5: Tagging model connecting the architecture and the domain model from Figure 3. Ports of the architecture are mapped to attributes of the domain model and vice versa.

the sending port and the receiving attribute (ll. 9-13). To this effect, a message containing the fully qualified name of the port, the identifier of the device, and the new value is generated and sent to the DTIS. Inversely, if updates of an attribute are to be forwarded to a port of the CPS (ll. 14-18), the DTIS generates such a message whenever the attribute in the DTIS is updated and sends it to the corresponding device.

5.2 Cyber-Physical System Architecture Transformation

Our method leverages model-to-model transformations to extend (possibly underspecified) CPS architectures by components that carry out the communication and synchronization with the DTIS. Figure 6 conceptually depicts this transformation. The input architecture consists of a sensor, a controller, and an actuator. The tagging defines with ports need to send data to or receive data from the DTIS. Accordingly, for each such tag, a component is generated. The transformation distinguishes between three cases: Tagging (1) outgoing ports, (2) incoming ports without connectors, and (3) incoming ports with connectors.

Case (1): The tagged port is an outgoing port that should send data from the architecture to the DTIS. The generated sender component (Figure 7) has an incoming port that takes data of the type given to the component as a generic type parameter **(R1)**.

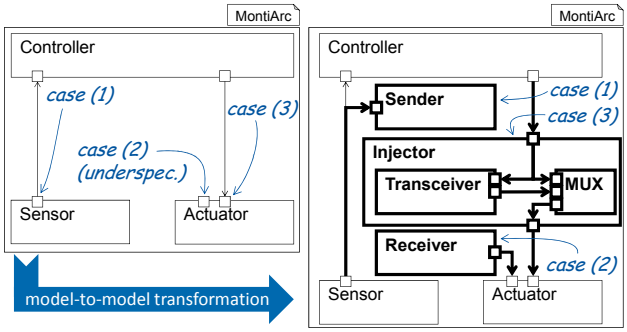


Figure 6: Architecture transformation that inserts components to keep the system synchronized with the information system. Generated elements illustrated in bold. Generic parameters omitted from all generated elements for better readability.

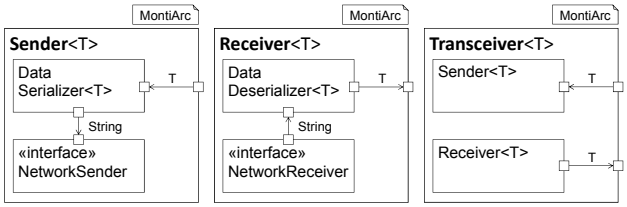


Figure 7: Prototypes for generated composed components that handle the communication with the IS.

The generator uses the type of the tagged port to instantiate the sender component and, thus, ensures matching types. A generated connector then connects the tagged port to the generated component. As soon as the generated sender component receives data, it serializes and forwards the data, together with identifiers of the emitting device and port, to the DTIS. The `Sender` component specifies the `NetworkSender` as an interface component. This component is exchanged during the deployment process by a hardware platform-specific version. This ensures portability across hardware platforms **(R5)** by exchanging the interface components that require platform-specific network functionalities by platform-specific variants during deployment of the architecture.

Case (2): The tagged port is an incoming port without connections. The process for injecting data into the architecture is inverse to the process of extracting data: If the tagged port should process data from the DTIS, the generated receiver component (Figure 7) has an outgoing port with a generic type **(R1)**. The generator again uses the type of the tagged port to instantiate the generated component and, thus, guarantees that the type of the port of the receiver component matches the tagged port. A generated connector connects the outgoing port of the generated receiver component to the tagged (incoming) port. If the `NetworkReceiver` component inside the generated receiver component receives data from the DTIS, it creates a message on its outgoing port that is then deserialized and forwarded to the tagged port.

Case (3): The tagged port is an incoming port with a connector. In this case, a more complex injector component is generated that replaces the connector and synchronizes incoming messages with the DTIS. This injector has two subcomponents: A transceiver and a multiplexer (MUX). The transceiver can be realized by combining

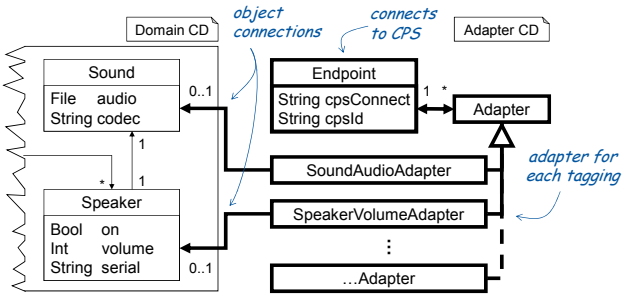


Figure 8: Additional endpoint for the example domain (Figure 3(b)). Each mapping has its own adapter to transform given data. Generated elements illustrated in bold.

the generated sender and receiver components from the previous two cases (Figure 7). This is done by using them as subcomponents of the transceiver, where connectors forward the data to or from the ports of the enclosing transceiver component. The generated sender and receiver keep the device synchronized with the DTIS.

The MUX gives manual decisions of the user priority over automatic decisions by the CPS, as the user's manual intervention expresses the explicit decision to override the automatic behavior of the system **(R4)**. If the MUX receives valid input from both of its incoming ports, the port connected to the DTIS is preferred. The MUX does not accept any value from the original system until the transceiver component explicitly releases the connection by sending an empty message. This is done to prevent immediately overriding the user's messages.

5.3 Information System Transformation

The generated sender and receiver components of the CPS communicate with *endpoints* of the DTIS. Endpoints store the necessary communication-related information about the connection to the CPS (`cpsConnect` in Figure 8), *e.g.,* a socket. Moreover, an endpoint maps a port of the CPS architecture to the respective *adapter* of the DTIS. Adapters are responsible for processing incoming data and monitoring data updates in the DTIS. For each `connect` statement of the tagging, we extend the DTIS with an adapter.

Depending on the tagging, the adapter either processes data received from the CPS (ll. 10-13, Figure 5) or monitors data updates that need to be sent to the CPS (ll. 15-18, Figure 5). An adapter that processes data received from the CPS determines to which object in the data source of the DTIS, *e.g.,* database, the data belongs. Figure 9 describes the process of how an object is loaded. The first step is to find the adapter that knows how to load the data from the data source and which parts (attributes) of the object to load. If the adapter is found, the object can be loaded directly. If the adapter cannot be found, that is, it has not yet been created, it is created together with the object in the DTIS data source and connected to it. Now that the adapter is created, the object can be retrieved. After the loading of the connected data object, the adapter then updates the data source accordingly. An adapter that informs the CPS about data updates listens to changes in the data objects. Changes in data objects can either originate from a user or external sources, *e.g.,* data imports. On every change, the adapter creates a message and forwards the data to the CPS via the endpoint.

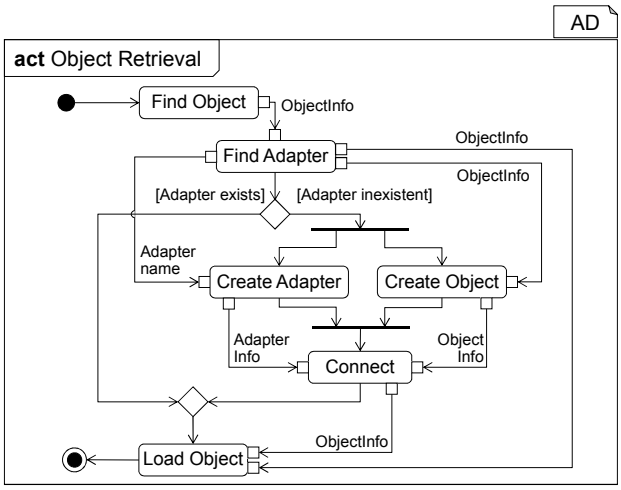


Figure 9: Object retrieval with adapters

Endpoints handle the DTIS's communication with the CPS and can be created either manually by the user or automatically when a CPS device first connects to the DTIS and no corresponding endpoint exists. The connection information of a device (`cpsConnect`) is set when the device first connects to the DTIS. As IoT devices may be mobile and the network topology cannot be assumed to be static, the endpoint updates this information whenever a CPS sends a message to the DTIS. To enable these updates, devices of the CPS include a unique device identifier (`cpsId`), *e.g.,* a name or serial number, in their messages to the DTIS.

This identifier can also be used to link devices to existing data objects when they first connect to the DTIS. To this effect, an attribute in the existing data class can be set in the tagging model (ll. 4-5, Figure 5). This also enables creating DTs of devices even before their counterparts in the CPS first connect to the DTIS. A special variant of this is choosing an attribute of a connected object. This only works for *-to-1-associations as the object has to be uniquely identified. If the identifier of the CPS device is unknown to the DTIS, a new object is created. If the tagging model specifies automatic mapping (l. 8, Figure 5), a new object is created whenever a device first connects.

The application itself does not need to know any of the extensions to the backend data structure, because the data is transformed by the respective adapter to match the internal data structure. The existing data structure or database schema used by the application does not need to be changed; it is only extended to handle the communication (Figure 8). These extensions are non-invasive and do not require the system to know the adapters.

6 CASE STUDY

Our case study uses the fire extinguisher example (Section 4), which is based on Google's *Nest Protect*. The goal is to build an extensible fire extinguisher system that can be monitored and controlled at runtime from an online dashboard. For monitoring the system, sensor values need to be sent to a DTIS. For controlling the system, the state of the actuators and its representation in the DTIS needs to be synchronized and the DTIS should be enabled to influence actuators. From an engineering perspective, the synchronization of the systems should be realizable with as little effort as possible. Since writing high-level communication protocols is a repetitive task, the communication between the DTIS and the CPS should be generated to a large extent instead of handwritten **(R2)**. The CPS was implemented using three Raspberry Pi 4B **(R5)**. Two of them were connected to a gas sensor and a siren, and one was connected to a temperature sensor. All of them executed C++ Code generated from MontiArc models. The sprinklers were only virtually present to prevent damage to our laboratory.

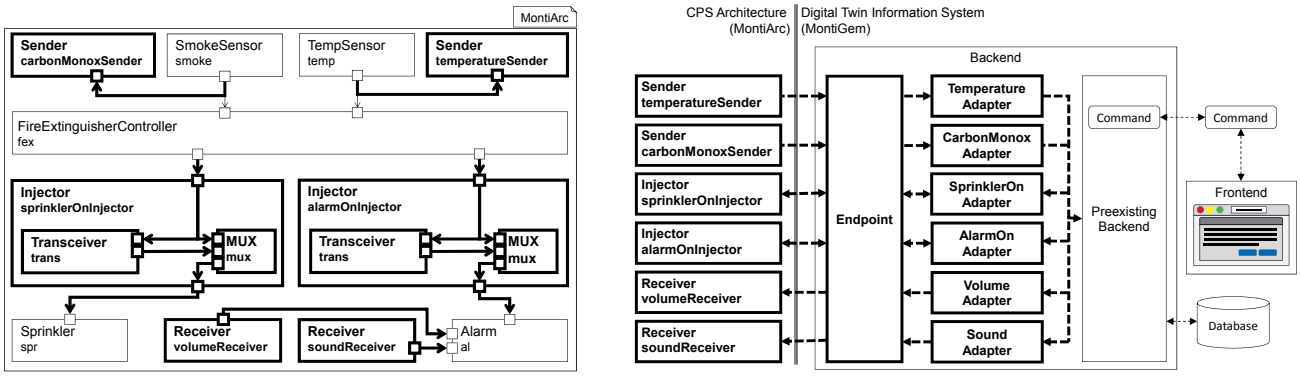
We integrate the architecture model (Figure 3(a)) of the CPS and the domain model (Figure 3(b)) of the DTIS using the tagging model from Figure 5. A major advantage of our approach is that the integration of the two systems only takes nine statements. This not only makes very efficient use of the engineers' time but also enables rapid prototyping of DTs as no time is needed for implementing communication infrastructures to keep the CPS and DTIS in sync.

Using the tagging model, we transform the CPS architecture and extend the DTIS according to this tagging model (*cf.* Section 5.2). Through this, we automatically include appropriate endpoints for the communication with the cyber-physical devices. The resulting architecture (Figure 10(a)) shows generated elements in bold. Clearly, all concepts from the original architecture are still present in the resulting architecture except for the connectors between the controller and the actuators. The two absent connectors are replaced by connectors to the added `Injector` components that tap the data and forward it to the DTIS. The new sender and receiver components connect to the already existing components of the architecture to extract or inject data. Figure 3(a) shows two underspecified ports that leave the decisions open how the `Alarm` component gets the desired volume and the sound played in case of an alarm. The generated `VolumeReceiver` and `SoundReceiver` components eliminate this underspecification. The fact that the original components remain unchanged ensures that the behavior description of the components, which relies on communication via the ports, does not need to be changed.

Figure 10(b) shows the changes in the DTIS based on the specified elements defined by the tagging model (Figure 5). Adapters for each mapping are added to the existing infrastructure of the DTIS. Once again, the additions do not interfere with the system's business logic, but only add the ability to update the internal data objects and send data update from the DTIS to the respective CPS. It is not necessary to modify the system's data- or view-logic. The added parts interact with the preexisting infrastructure. To load data objects which are sent to the CPS, the adapters are used to load the specific data from the database. The adapters then transform the domain data to a format that suits the communication with the CPS. When data is sent from the CPS to the DTIS an adapter transforms the message to an internal data object and stores it in the database.

7 RELATED WORK

Multiple tools support the model-driven engineering of IoT systems. ThingML [28, 51] enables defining devices and their logic using a C-like DSL that is used to generate C, Java, or JavaScript artifacts. Ericsson's Calvin [54] enables defining the architecture of IoT applications in a MontiArc-like syntax using the CalvinScript



(a) Transformed CPS architecture including DT communication infrastructure.

(b) Adapters and DTIS adapter endpoint added to the existing MontiGem DTIS.

Figure 10: Automatic fire extinguisher system. Generated elements are in bold. (Generic type) parameters are omitted for better comprehensibility.

DSL. MDE4IoT [19] uses the Foundational Subset for Executable UML Models (fUML) and the action language for foundational UML (ALF) [53] to describe IoT applications and possible deployments to physical devices. SysML4IoT [30] describes how to develop adaptive IoT systems using a SysML-based DSL. Node-RED [2] provides a graphical editor that allows connecting the in- and outputs of software components. Node-RED comes with a library of predefined components to access, *e.g.,* Twitter or Amazon S3 cloud storage. Similarly, the Ptolomy-based CapeCode [13] offers a user interface for graphically combining software components as reusable building blocks.

While some of these systems enable specifying message exchanges and serialization, they lack mechanisms for automatically synchronizing them to DTISs or defining DTs **(R2)**. Though it might be possible to execute some of the actors defined in the above languages inside the DTIS, none of these languages is designed to define a DTIS. Therefore, synchronization with the DTIS data structures is possible **(R1)**, but requires considerable manual effort **(R2)** and that the DTIS and CPS developers agree on messages or topics for synchronization **(R3)**. Especially, this requires the developer of the CPS architecture to have in-depth knowledge about an already existing DTIS and vice versa. In our approach, only the integrator is required to know both systems and only on a high level of abstraction.

Even with popular IoT solutions, such as IBM's Bluemix Cloud platform, connecting devices as simple as a temperature sensor, can be unnecessarily complicated [38]. Following our method, reading and synchronizing the values of a sensor to the data structure in the DTIS requires only two statements in the model (*cf.* ll. 8-11, Figure 5). Moreover, as all of the above approaches to IoT development platforms are based on components exchanging data with each other, our method could be applied to any of the above systems if the generated sender and receiver components were adapted to the platforms' respective interfaces.

The robot operating system (ROS) [55] serves to develop (distributed) robotic software as collections of loosely connected nodes that perform computations and exchange messages over topics (typed message buses). These topics can exchange messages of complex data types known to participating nodes **(R1)** and a generic communication infrastructure takes care of handling message handling. However, sending and reacting to messages has to be handwritten **(R2)** and requires developers to agree on topics **(R3)**. As such, ROS architectures can already represent small-scale IoT applications. While the communication infrastructure of ROS is generic, the data types that can be communicated are, similar to our approach, defined in (rosmsg) models that resemble C++ structs. From these, platform-specific implementations of the data types are generated. However, ROS does not feature any notion of system representation aside from logging and debugging information that could be considered a representation of the CPS.

AutoFocus 3 [8] is a modeling framework based on the Focus [14] calculus to describe the architectures of embedded systems. As such, it covers modeling from requirements to logical and technical architectures to their deployment. It neither facilitates engineering of DTISs to represent DTs, nor connecting DTs to the modeled CPSs. None of the platforms provide specific infrastructure for users to overwrite values out-of-the-box **(R4)**.

Many popular commercial IoT platforms support the development of digital twins but lack means for the model-driven development and integration of CPSs and DTISs. Examples include the arguably largest cloud providers: Microsoft Azure's "device twin" [3] and Amazon AWS's "device shadow" [1]. Both of them exchange data with the CPSs using a combination of JSON and MQTT to synchronize values known to both the CPS and the cloud **(R1)**. Those messages can also be used for manually overwriting values **(R4)**. Structurally, the DT services offered by Azure and AWS resemble the tripartite division into CPS, DTIS and integration of our development process with one important difference: While AWS and Azure require lots of error-prone low-level programming for communicating with the DTIS and synchronizing values, our model-driven approach can automatically generate this infrastructure **(R2)**. Hence, our approach decouples the business logic of the systems from the communication and synchronization tasks required to create a DTs. Thus, we argue that our systems are easier to understand and maintain as the implementation of the business logic is not cluttered with code needed by the infrastructure **(R3)**. In contrast, the Arduino IoT Cloud takes works at a lower level of abstraction. It allows users to define variables of primitive data

		IoT / Architecture Tools										
		This Solution	ThingML [28, 51]	Calvin [54]	MDE4IoT [19]	Node-RED [2]	CapeCode [13]	ROS [55]	AutoFocus 3 [8]	Microsoft Azure	Amazon AWS	Arduino IoT Cloud
Requirements	R1	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	P^1
	R2	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
	R3	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	R4	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	P^2
	R5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	P^3

[1] Only primitive data types (optionally with unit)
[2] Users can set values, but devices can immediately reset them
[3] Only Arduino-compatible hardware

Table 1: Comparison to related work. ✓= fulfilled, ✗= not fulfilled, P = partially fulfilled

types online **(R1)** for which it then generates the necessary code to keep them synchronized with the Arduino IoT Cloud **(R2)**. This, however, pollutes the business logic with synchronization-specific code and requires the developers to use the variables defined by the generated code **(R3)**. While all other tools usually only require support for a specific GPL, the Arduino IoT Cloud is limited to Arduino-compatible devices **(R5)**.

While our concept is designed for combining MontiArc and MontiGem, there is no conceptual limitation that forbids adapting the generated MontiArc components to use the interfaces offered by Azure and AWS. Adding support for them only requires implementing cloud-specific `NetworkSender` and `NetworkReceiver` components and adapting the `DataSerializer` to use the JSON structures expected by the respective cloud (*cf.* Figure 7).

8 DISCUSSION

One of the main advantages of our solution is the separation of concerns that is achieved by defining communication and synchronization related structures separately from the business logic of the application. This makes the models easier to understand because developers who encounter them for the first time can concentrate on the business logic without being distracted by the technical details of synchronizing values. Also, this enables generating the necessary infrastructure to keep the CPS synchronized with the DTs. This eliminates a repetitive and error-prone task for the developers.

Our separation of concerns comes at the cost of the integrator needing to have a high-level understanding of models for both the CPS and DTIS. While this is a simple task for small systems, it can quickly become complicated as systems become more complex. Therefore, ideally, the integrator is not a single person, but a group consisting of at least one developer of both the CPS and the DTIS. Commercial solutions like the "spatial graph" used by Microsoft Azure's DT require similar roles. In Azure's spatial graph, each device can contain multiple sensors producing data of a certain type. The devices have to be aware of this information and react to requests created based on the information in the spatial graph. This leads to problems if there is a mismatch between the sensors offered by the actual device and the sensors specified in the spatial graph. Since our solution directly utilizes the models used to create the CPS and DTIS, we can detect potential inconsistencies caused by this integration step automatically before deploying the system.

While we think a model-driven approach to developing CPSs and DTISs offers many advantages [16], we do acknowledge that many real-world systems do not use a model-driven approach [62]. DTISs, in particular, are today often programmed by hand. Therefore, it is necessary to leave open the option of communicating with those systems. By allowing customization of the communication mechanisms through abstract components, our solution can also easily be adapted to communicate with popular commercial solutions like Amazon AWS instead of our MontiGem DTIS. This would be done by providing network components (*cf.* Figure 7) that use the APIs of the commercial or handwritten solutions.

In some situations, however, it might also be useful to convert between the data types offered by the CPS and the data types used by the DTIS. For example, the CPS might process values of a temperature sensor given in Fahrenheit, while the DTIS stores temperature in Celsius. Currently, the data types used by the CPS have to match the data types used by the DTIS. As future work, we plan to allow conversions and transformations that are applied during the synchronization.

Furthermore, the logic of the synchronization can be further investigated. Currently, the CPS gives priority to the messages coming from the DTIS. However, the component that sends a message to the tagged port is not aware of this process and therefore does not change its behavior. Thus, it would immediately overwrite the value set by the user in the DTIS. To prevent this, the user can (temporarily) lock a value in the DTIS. As long as the lock is set, messages from the CPS are then ignored in favor of the last value set in the DTIS for this port. This prevents user-set values from being overwritten by the CPS. This process, however, may not be desired for all use cases. If the CPS should adapt its behavior to match the user-set values, a more complex synchronization is required.

Moreover, our evaluation only shows the general feasibility of the approach. For productive use, further investigations regarding the scalability would be necessary. To ensure scalability on the server side, common load distribution methods can be used. On the CPS device side, the available processing power and network bandwidth limit the number of values synchronized with the DTIS.

9 CONCLUSION

Creating DTs for a system comprises creating models of the system, means to process and represent data received from that system, and connecting the represented system to its DT. The latter usually involves manually programming the connection using a communication framework of choice, such as MQTT [52]. This is tedious, error-prone, and complicates the analysis of connections. Our method to connect DTs with DTISs facilitates their integration and separates concerns in DT development by decoupling the development of the CPS architecture and the DTIS. The generated infrastructure consists of consistent-by-construction interfaces between CPS and DTIS that synchronize both systems and accelerates developing DTs for CPSs. Overall, explicitly modeling the integration can facilitate the systematic engineering of DTs.

REFERENCES

[1] Device Shadow Service for AWS IoT. [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html. Last checked 28. April 2020

[2] Node-RED—Low-code programming for event-driven applications. [Online]. Available: https://nodered.org. Last checked 28. April 2020

[3] Understand and use device twins in IoT Hub. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins. Last checked 28. April 2020

[4] Adam, K., Butting, A., Heim, R., Kautz, O., Rumpe, B., Wortmann, A.: Model-Driven Separation of Concerns for Service Robotics. In: Int. Workshop on Domain-Specific Modeling (DSM'16). pp. 22–27. ACM (October 2016)

[5] Adam, K., Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Executing Robot Task Models in Dynamic Environments. In: Proc. of MODELS 2017. Workshop EXE. CEUR 2019 (September 2017)

[6] Adam, K., Michael, J., Netz, L., Rumpe, B., Varga, S.: 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, vol. P-304, pp. 59–66. Gesellschaft für Informatik e.V. (May 2020)

[7] Adam, K., Netz, L., Varga, S., Michael, J., Rumpe, B., Heuser, P., Letmathe, P.: Model-Based Generation of Enterprise Information Systems. In: Fellmann, M., Sandkuhl, K. (eds.) Enterprise Modeling and Information Systems Architectures (EMISA'18). CEUR Workshop Proceedings, vol. 2097, pp. 75–79. CEUR-WS.org (May 2018)

[8] Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. ACES-MB&WUCOR@ MoDELS **1508**, 19–26 (2015)

[9] Bertram, V., Maoz, S., Ringert, J.O., Rumpe, B., von Wenckstern, M.: Component and Connector Views in Practice: An Experience Report. In: Conf. on Model Driven Engineering Languages and Systems (MODELS'17). pp. 167–177. IEEE (September 2017)

[10] Biesinger, F., Weyrich, M.: The facets of digital twins in production and the automotive industry. In: 23rd Int. Conference on Mechatronics Technology (ICMT). pp. 1–6. IEEE (2019)

[11] Blech, J.: Towards digital twins for the description of automotive software systems. Electronic Proceedings in Theoretical Computer Science **312**, 20–28 (Jan 2020). https://doi.org/10.4204/eptcs.312.2

[12] Boschert, S., Rosen, R.: Digital twin—the simulation aspect. In: Mechatronic futures, pp. 59–74. Springer (2016)

[13] Brooks, C., Jerad, C., Kim, H., Lee, E.A., Lohstroh, M., Nouvelletz, V., Osyk, B., Weber, M.: A Component Architecture for the Internet of Things. Proc. of the IEEE **106**(9), 1527–1542 (September 2018)

[14] Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Heidelberg (2001)

[15] Bruynseels, K., Santoni de Sio, F., van den Hoven, J.: Digital twins in health care: Ethical implications of an emerging engineering paradigm. Frontiers in genetics **9**, 31 (2018). https://doi.org/10.3389/fgene.2018.00031

[16] Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling **19**(1), 5–13 (2020). https://doi.org/10.1007/s10270-019-00773-6

[17] Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A.: Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In: European Conf. on Modelling Foundations and Applications (ECMFA'17). pp. 53–70. LNCS 10376, Springer (July 2017)

[18] Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Architectural Programming with MontiArcAutomaton. In: In 12th Int. Conf. on Software Engineering Advances (ICSEA'17). pp. 213–218. IARIA XPS Press (May 2017)

[19] Ciccozzi, F., Spalazzese, R.: MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In: 10th Int. Symp. on Intelligent Distributed Computing (October 2016)

[20] Dalibor, M., Jansen, N., Kirchhof, J.C., Rumpe, B., Schmalzing, D., Wortmann, A.: Tagging Model Properties for Flexible Communication. In: Proc. of MODELS 2019. Workshop MDE4IoT. pp. 39–46. IEEE (September 2019)

[21] Francisco, A., Mohammadi, N., Taylor, J.: Smart city digital twin-enabled energy management: Toward real-time urban building energy benchmarking. Journal of Management in Engineering **36**(2), 04019045 (2020). https://doi.org/10.1061/(ASCE)ME.1943-5479.0000741

[22] Gatto, N., Kusmenko, E., Rumpe, B.: Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In: Burgueño, L., Pretschner, A., Voss, S., Chaudron, M., Kienzle, J., Völter, M., Gérard, S., Zahedi, M., Bousse, E., Rensink, A., Polack, F., Engels, G., Kappel, G. (eds.) Proc. MODELS 2019. Workshop MDE Intelligence. pp. 196–202 (September 2019)

[23] Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., Rumpe, B., Varga, S.: Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Michael, J., Bork, D. (eds.) Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers. pp. 22–30. CEUR Workshop Proceedings (February 2020)

[24] Gerasimov, A., Michael, J., Netz, L., Rumpe, B., Varga, S.: Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In: Anderson, B., Thatcher, J., Meservy, R. (eds.) 25th Americas Conference on Information Systems (AMCIS 2020). pp. 1–10. AIS Electronic Library (AISeL), Association for Information Systems (AIS) (August 2020)

[25] Greifenberg, T., Look, M., Roidl, S., Rumpe, B.: Engineering Tagging Languages for DSLs. In: Conf. on Model Driven Engineering Languages and Systems (MODELS'15). pp. 34–43. ACM/IEEE (2015)

[26] Grieves, M., Vickers, J.: Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In: Transdisciplinary perspectives on complex systems, pp. 85–113. Springer (2017)

[27] Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University (February 2012)

[28] Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In: Proc. of the ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 125–135. MODELS '16, ACM, New York, NY, USA (2016)

[29] Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In: Software Architecture - 10th European Conf. (ECSA'16). LNCS, vol. 9839, pp. 175–182. Springer (December 2016)

[30] Hussein, M., Li, S., Radermacher, A.: Model-driven Development of Adaptive IoT Systems. In: Proceedings of MODELS 2017. Workshop ModComp. vol. 2019, pp. 17–23. CEUR, Austin, United States (Sep 2017)

[31] Jain, P., Poon, J., Singh, J.P., Spanos, C., Sanders, S.R., Panda, S.K.: A digital twin approach for fault diagnosis in distributed photovoltaic systems. IEEE Transactions on Power Electronics **35**(1), 940–956 (2020)

[32] Jimenez, J., Jahankhani, H., Kendzierskyj, S.: Health Care in the Cyberspace: Medical Cyber-Physical System and Digital Twin Challenges, pp. 79–92. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-18732-3_6

[33] Johansen, S., Nejad, A.: On digital twin condition monitoring approach for drivetrains in marine applications. International Conference on Offshore Mechanics and Arctic Engineering, vol. Volume 10: Ocean Renewable Energy (06 2019). https://doi.org/10.1115/OMAE2019-95152

[34] Josifovska, K., Yigitbas, E., Engels, G.: A digital twin-based multi-modal ui adaptation framework for assistance systems in industry 4.0. In: Kurosu, M. (ed.) Human-Computer Interaction. Design Practice in Contemporary Societies. pp. 398–409. Springer International Publishing (2019)

[35] Kraft, E.: The air force digital thread/digital twin - life cycle integration and use of computational and experimental knowledge. In: 54th AIAA Aerospace Sciences Meeting. American Institute of Aeronautics and Astronautics (2016). https://doi.org/10.2514/6.2016-0897

[36] Kriebel, S., Raco, D., Rumpe, B., Stüber, S.: Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In: Proc. of the Workshops of the Software Engineering Conf.: Workshop on Avionics Systems and Software Engineering (AvioSE'19). vol. 2308, pp. 87–94. CEUR-WS.org (2019)

[37] Laaki, H., Miche, Y., Tammi, K.: Prototyping a digital twin for real time remote control over mobile networks: Application of remote surgery. IEEE Access **7**, 20325–20336 (2019)

[38] Lekić, M., Gardašević, G.: IoT sensor integration to Node-RED platform. In: 17th Int. Symp. INFOTEH-JAHORINA. pp. 1–5 (March 2018)

[39] Liu, J., Zhou, H., Liu, X., Tian, G., Wu, M., Cao, L., Wang, W.: Dynamic evaluation method of machining process planning based on digital twin. IEEE Access **7**, 19312–19323 (2019)

[40] Lu, Q., Xie, X., Heaton, J., Parlikad, A.K., Schooling, J.: From bim towards digital twin: Strategy and future development for smart asset management. In: Borangiu, T., Trentesaux, D., Leitão, P., Giret Boggino, A., Botti, V. (eds.) Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future. pp. 392–404. Springer International Publishing, Cham (2020)

[41] Mandolla, C., Petruzzelli, A.M., Percoco, G., Urbinati, A.: Building a digital twin for additive manufacturing through the exploitation of blockchain: A case analysis of the aircraft industry. Computers in Industry **109**, 134 – 152 (2019). https://doi.org/https://doi.org/10.1016/j.compind.2019.04.011

[42] Maoz, S., Ringert, J.O., Rumpe, B., Wenckstern, M.v.: 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, vol. 1723, pp. 19–24 (October 2016)

[43] Mayr, H.C., Michael, J., Ranasinghe, S., Shekhovtsov, V.A., Steinberger, C.: Model Centered Architecture, pp. 85–104. Springer International Publishing (2017)

[44] Mayr, H.C., Michael, J., Shekhovtsov, V.A., Ranasinghe, S., Steinberger, C.: A Model Centered Perspective on Software-Intensive Systems. In: Fellmann, M., Sandkuhl, K. (eds.) Enterprise Modeling and Information Systems Architectures (EMISA'18). CEUR Workshop Proceedings, vol. 2097, pp. 58–64 (2018)

[45] Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)

[46] Mercadal, J., Enard, Q., Consel, C., Loriant, N.: A domain-specific approach to architecturing error handling in pervasive computing. SIGPLAN Not. **45**(10), 47–61 (Oct 2010). https://doi.org/10.1145/1932682.1869465
[47] Michael, J., Koschmider, A., Mannhardt, F., Baracaldo, N., Rumpe, B.: User-Centered and Privacy-Driven Process Mining System Design for IoT. In: Cappiello, C., Ruiz, M. (eds.) Proc. of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems. pp. 194–206. Springer (2019)
[48] Michael, J., Netz, L., Rumpe, B., Varga, S.: Towards privacy-preserving iot systems using model driven engineering. In: Ferry, N., Cicchetti, A., Ciccozzi, F., Solberg, A., Wimmer, M., Wortmann, A. (eds.) MDE4IoT & ModComp 2019, Model-Driven Engineering for the Internet of Things (MDE4IoT) & Interplay of Model-Driven and Component-Based Software Engineering (ModComp). pp. 15–22. CEUR-WS.org (Sep 2019)
[49] Michael, J., Rumpe, B., Varga, S.: Human behavior, goals and model-driven software engineering for assistive systems. In: Koschmider, A., Michael, J., Thalheim, B. (eds.) Enterprise Modeling and Information Systems Architectures (EMSIA 2020). vol. 2628, pp. 11–18. CEUR Workshop Proceedings (June 2020)
[50] Michael, J., Steinberger, C.: Context modeling for active assistance. In: Cabanillas, C., España, S., Farshidi, S. (eds.) Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017). pp. 221–234 (2017)
[51] Morin, B., Harrand, N., Fleurey, F.: Model-Based Software Engineering to Tame the IoT Jungle. IEEE Software **34**(1), 30–36 (January 2017)
[52] Naik, N.: Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: IEEE Int. Systems Engineering Symposium (ISSE). pp. 1–7. IEEE (2017)
[53] Perseil, I.: Alf formal. Innovations in Systems and Software Engineering **7**(4), 325–326 (2011). https://doi.org/10.1007/s11334-011-0168-x
[54] Persson, P., Angelsmark, O.: Calvin – Merging Cloud and IoT. Procedia Computer Science **52**, 210 – 217 (2015), 6th Int. Conf. on Ambient Systems, Networks and Technologies (ANT)
[55] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software (2009)
[56] Rasheed, A., San, O., Kvamsdal, T.: Digital twin: Values, challenges and enablers from a modeling perspective. IEEE Access **8**, 21980–22012 (2020)
[57] Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: 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)
[58] Ringert, J.O., Rumpe, B., Schulze, C., Wortmann, A.: Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In: Int. Conf. on Software Engineering: Software Engineering and Education Track (ICSE'17). pp. 127–136. IEEE (May 2017)
[59] Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International (July 2016)
[60] Schleich, B., Anwer, N., Mathieu, L., Wartzack, S.: Shaping the digital twin for design and production engineering. CIRP Annals **66**(1), 141–144 (2017)
[61] Schuh, G., Häfner, C., Hopmann, C., Rumpe, B., Brockmann, M., Wortmann, A., Maibaum, J., Dalibor, M., Bibow, P., Sapel, P., et al.: Effizientere produktion mit digitalen schatten. ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb **115**(special), 105–107 (2020)
[62] Sebastián, G., Gallud, J., Tesoriero, R.: Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages **56**, 100935 (2020). https://doi.org/https://doi.org/10.1016/j.cola.2019.100935
[63] Stachowiak, H.: Allgemeine Modelltheorie (1973)
[64] Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
[65] Steinberger, C., Michael, J.: Using Semantic Markup to Boost Context Awareness for Assistive Systems, pp. 227–246. Springer International Publishing (2020)
[66] Talkhestani, B.A., Jazdi, N., Schlögl, W., Weyrich, M.: A concept in synchronization of virtual production system with real factory based on anchor-point method. Procedia CIRP **67**, 13–17 (2018)
[67] Tao, F., Zhang, H., Liu, A., Nee, A.Y.C.: Digital twin in industry: State-of-the-art. IEEE Transactions on Industrial Informatics **15**(4), 2405–2415 (April 2019)
[68] Tao, F., Cheng, J., Qi, Q., Zhang, M., Zhang, H., Sui, F.: Digital twin-driven product design, manufacturing and service with big data. The International Journal of Advanced Manufacturing Technology **94**(9-12), 3563–3576 (2018)
[69] Tao, F., Qi, Q., Wang, L., Nee, A.: Digital twins and cyber–physical systems toward smart manufacturing and industry 4.0: Correlation and comparison. Engineering **5**(4), 653 – 661 (2019)
[70] Taylor, N., Human, C., Kruger, K., Bekker, A., Basson, A.: Comparison of digital twin development in manufacturing and maritime domains. In: Borangiu, T., Trentesaux, D., Leitão, P., Giret Boggino, A., Botti, V. (eds.) Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future. pp. 158–170. Springer International Publishing, Cham (2020)
[71] Wortmann, A., Combemale, B., Barais, O.: A Systematic Mapping Study on Modeling for Industry 4.0. In: Conf. on Model Driven Engineering Languages and Systems (MODELS'17). pp. 281–291. IEEE (September 2017)
[72] Zanin, M., Perez, D., Kolovos, D.S., Paige, R.F., Chatterjee, K., Horst, A., Rumpe, B.: On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In: Proc. of the SESAR Innovation Days. EUROCONTROL (2011)