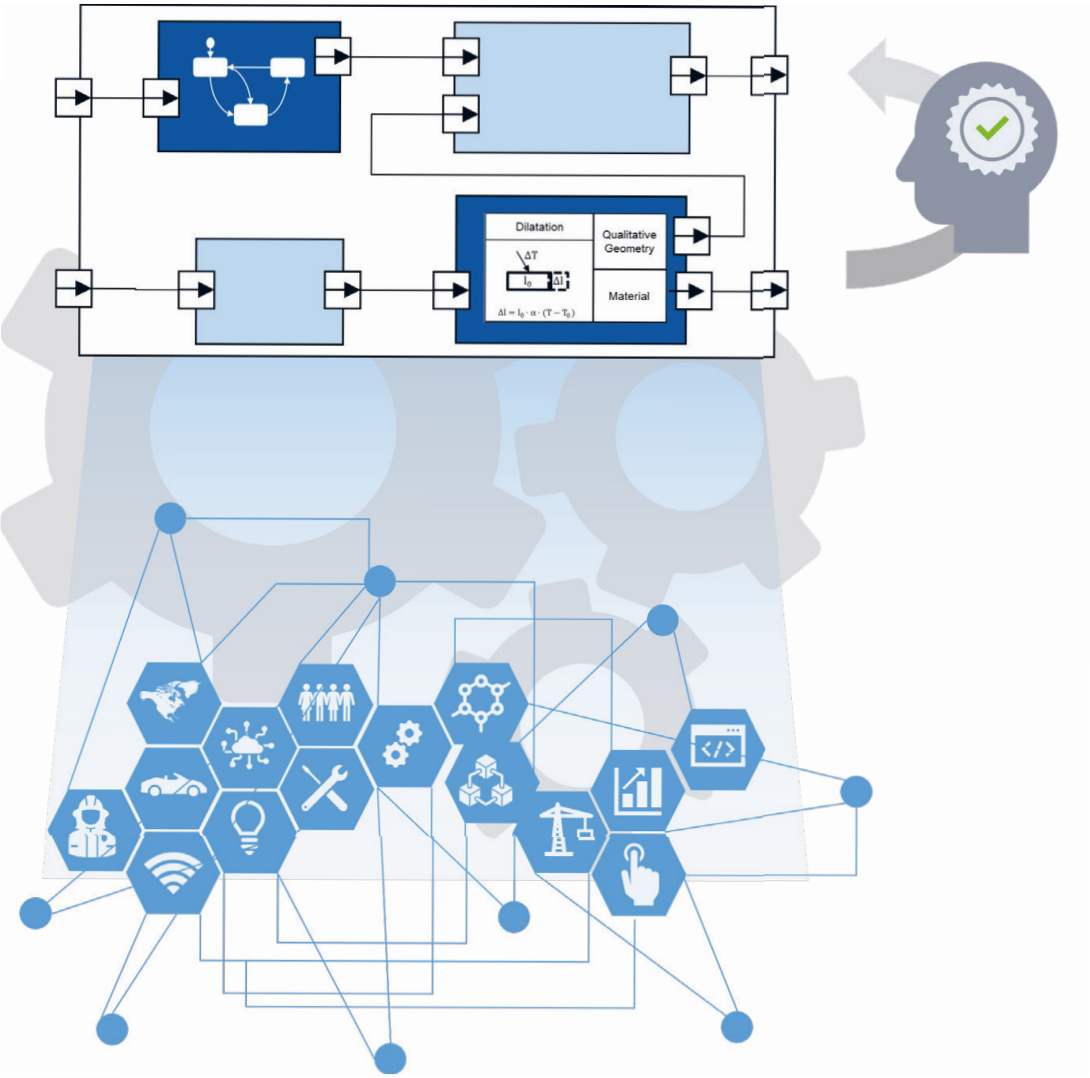
SE | RWTH AACHEN UNIVERSITY

Imke Helene Nachmann

Functional Modeling of Cyber-Physical Systems



Aachener Informatik-Berichte,
Software Engineering

Band 59

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

Functional Modeling of Cyber-Physical Systems

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

vorgelegt von

M.Sc.
Imke Helene Nachmann, geb. Drave
aus Hamburg, Germany

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

Tag der mündlichen Prüfung: 27. März 2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

[Nac25] I. Nachmann:
Functional Modeling of Cyber-Physical Systems.
Aachener Informatik-Berichte, Software Engineering, Band 59,
ISBN 978-3-8191-0107-6, Shaker Verlag, Jul. 2025.

Aachener Informatik-Berichte, Software Engineering

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

Band 59

Imke Helene Nachmann
RWTH Aachen University

Functional Modeling of Cyber-Physical Systems

Shaker Verlag
Düren 2025

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

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

Printed in Germany.

Print-ISBN	978-3-8191-0107-6
PDF-ISBN	978-3-8191-0081-9
ISSN	1869-9170
eISSN	2944-6910

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

Eidesstattliche Erklärung

Imke Helene Nachmann erklärt hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden. Hiermit erkläre ich an Eides statt

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

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

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

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

5. Alle wesentlichen Quellen von Unterstützung wurden benannt;

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

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

   - [DRW+20] I. Drave, B. Rumpe, A. Wortmann, J. Berroth, G. Hoepfner, G. Jacobs, K. Spuetz, 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, pp. 79-89, ACM, Oct. 2020.
   - [ZJS+21] T. Zerwas, G. Jacobs, K. Spuetz, G. Hoepfner, I. Drave, J. Berroth, C. Guist, C. Konrad, B. Rumpe, J. Kohl: Mechanical Concept Development Using Principle Solution Models. In: IOP Conference Series: Materials Science and Engineering, G. Jacobs, S. Stein (Eds.), Volume 1097:012001, IOP Publishing, Feb. 2021.
   - [HJZ+21] G. Hoepfner, G. Jacobs, T. Zerwas, I. Drave, J. Berroth, C. Guist, B. Rumpe, J. Kohl: Model-Based Design Workflows for Cyber-Physical Systems Applied to an Electric-Mechanical Coolant Pump. In: IOP Conference Series: Materials Science and Engineering, G. Jacobs, S. Stein (Eds.), Volume 1097:012004, IOP Publishing, Feb. 2021.

- [HNZ+23] G. Hoepfner, I. Nachmann, T. Zerwas, J. K. Berroth, J. Kohl, C. Guist, B. Rumpe, G. Jacobs: Towards a Holistic and Functional Model-Based Design Method for Mechatronic Cyber-Physical Systems. In: Journal of Computing and Information Science in Engineering (JCISE), Volume 23(5), Mar. 2023.

Köln, 25.07.2024

Imke Nachmann

Abstract

Engineering Cyber-Physical Systems faces many challenges including the demand to integrate many different forms of functionalities and features. Traditional engineering processes are structured by the physical components and software modules of the final system and the engineering activities are concerned with evolving these components. In these approaches, the information, which components realize which functions and, in particular, which components interact in which way to implement a function, most often remains implicit. This hinders not only the collaboration of experts from different domains but also an agile approach to engineering which becomes efficient through automation, *e.g.,* of V&V tasks. The shift towards functional requirements gives rise to a conceptual gap between their abstract descriptions and the very detailed descriptions of the system's implementation. So far, the involved engineering domains have established an understanding of what a system's function is, but these understandings have not been consolidated and integrated in a suitable interdisciplinary modeling technique. Therefore, we propose a functional development paradigm that promotes development by finding realizations of cyber-physical functions, which transform energy, matter, and data, rather than implementing physical products or software modules directly. Therein, we model a CPS as a network of interacting timed stream processing functions, which specify the desired logical behavior and the physical behavior. By applying this modeling technique to formalize the design process in mechanical engineering, we show how to integrate the functional understanding of software and mechanical systems, which paves the way for agile and holistic model-driven engineering of CPSs. Further, we propose requirements and a meta-model for implementing modeling languages to model these functions.

Kurzfassung

Die heutigen Cyber-Physischen Systeme müssen immer mehr Funktionalitäten erfüllen, was ihre Entwicklung vor neue Herausforderungen stellt. Die Aktivitäten traditioneller Entwicklungsmethoden fokussieren die Entwicklung physikalischer Teile oder Software Module und nicht der angeforderten Funktionen, deren Realisierung die Interaktion dieser Komponenten erfordert. Die Information, welche Komponenten, bzw. welche Interaktion welcher Komponenten die geforderte Funktion erfüllt bleibt in der Regel implizit und erschwert die Kollaboration der Experten aus den verschiedenen Domänen. Auch die Validierung und Verifikation der angeforderten Funktionen wird durch das nicht Vorhandensein dieser Information in formaler Form erschwert. Zwischen den informell und abstrakt beschriebenen funktionalen Anforderungen und der Dokumentation ihrer Implementierungen entsteht eine konzeptionelle Lücke. Die in der Entwicklung Cyber-Physischer Systeme involvierten Domänen haben jeweils ein eigenes Verständnis dessen etabliert, was die Funktion eines Systems ist. Diese Verständnisse wurden jedoch noch nicht zu einer holistischen Modellierungstechnik konsolidiert. Diese Arbeit stellt das funktionale Entwicklungspardigma vor, welches den Fokus der Entwicklung auf das Finden von Realisierungen der cyber-physischen Funktionen, die ein System, das Energie, Material und Daten definiert, legt. Dies steht im Kontrast zu traditionellen Ansätzen, die die Entwicklung von Komponenten zur Erfüllung funktionaler Anforderungen anstreben. Darin wird ein Cyber-Physisches System als ein Netzwerk aus interagierenden gezeiteten stromverarbeitenden Funktionen verstanden und modelliert. Um dieses formale Verständnis des Systems in der Entwicklung praktisch anwenden zu können wird außerdem ein Meta-Modell vorgestellt und in Form eines SysML-Profils implementiert, das die wesentlichen Anforderungen und Konzepte zur Entwicklung von Modellierungssprachen integriert. Diese Technik wird dann angewandt um einen Konstruktionskatalog aus dem Maschinenbau zu formalisieren und so aufzuzeigen, dass die Modellierung es schafft, eine Brücke zwischen Softwareentwicklung und Maschinenbau zu schaffen.

Danksagung

An dieser Stelle möchte ich allen danken, die mich während der Entstehung dieser Dissertation und insbesondere im Verlauf des Promotionsprozesses begleitet und unterstützt haben – eines Weges, der nicht nur fachliches Engagement, sondern auch Ausdauer, Frustrationstoleranz und persönliche Hingabe erfordert hat.
Mein besonderer Dank gilt meinem Doktorvater, Prof. Dr. Bernhard Rumpe. Unsere konstruktiven Diskussionen haben maßgeblich zur inhaltlichen Ausrichtung und letztlichen Fertigstellung dieser Arbeit beigetragen. Ebenso danke ich für die Möglichkeit, im Rahmen zahlreicher Industrie- und Forschungsprojekte Verantwortung, sei es in der Bearbeitung, Akquise oder Leitung, zu übernehmen. Die inhaltliche Vielfalt, die ich in meiner Arbeit am Lehrstuhl erfahren durfte, hat meine persönliche und intellektuelle Entwicklung in entscheidender Weise geprägt.
Mein aufrichtiger Dank gilt außerdem Univ.-Prof. Mag. Dr. Manuel Wimmer für die Übernahme der Zweitbegutachtung, Prof. Dr. Stefan Decker für den Vorsitz der Prüfungskommission sowie Prof. Dr. Sebastian Trimpe für seine Rolle als drittem Prüfer. Die inhaltliche Auseinandersetzung im Rahmen der Prüfungsvorbereitung war nicht zuletzt durch die spannenden Vorlesungen und Übungen eine bereichernde Erfahrung.
Für die kollegiale Unterstützung während meiner Zeit am Lehrstuhl danke ich dem gesamten Team des Lehrstuhls für Software Engineering an der RWTH Aachen herzlich. Ein besonderer Dank gilt Louis Wachtmeister, Sebastian Stüber, Christian Kirchhoff, Deni Raco, Evgeny Kusmenko, Steffen Hillemacher, Hendrick Kausch, Florian Drux und Nico Jansen, die mir wertvolles Feedback zur Dissertation gegeben haben. Für die hervorragende Zusammenarbeit möchte ich mich außerdem bei David Schmalzing, Judith Michael, Vincent Bertram, Lukas Netz, Simon Varga, Arkadii Gerasimov, Max Stachon, Oliver Kautz, Andreas Wortmann, Matthias Markthaler, Michael von Wenckstern, Timo Greifenberg, Marita Breuer, Galina Volkova und Sylvia Gunder bedanken.
Mein tief empfundener Dank gilt auch meiner Familie und meinen Freunden. Meinen Eltern, Volker und Andrea Drave, sowie meiner Schwester Svenja Dick danke ich von Herzen für ihre stetige Unterstützung und Ermutigung.
Zuletzt möchte ich meinem Ehemann Philipp Nachmann danken – für seine unermüdliche Geduld, seine Kraft und seine Bereitschaft, mir insbesondere nach der Geburt unserer Tochter den nötigen Raum und die Zeit zu geben, dieses Vorhaben zu realisieren.

Contents

Chapter 1

Introduction

1.1 Motivation and Context: Cyber-Physical Systems

Technical systems have become a part of our daily lives since the industrial revolution in the 19th century, and have continuously evolved through the 20th and 21st centuries. Inventions or discoveries such as electricity, the Ottomotor, or Zuse's Z3 have brought our society forward unlike anything else. In the second half of the 20th century innovations in these systems were coupled to physical parts. One selling argument for a vehicle, for example, was an engine with as much power as possible. Nowadays, however, the focus of the customer or the system user who dictates the innovation driver has changed: Systems are considered innovative once they offer a broad range of functionalities and features [DGH+19, CKY05]. Considering, again, the automotive industry as an example, the selling argument has shifted towards functions such as assisted, or (semi-) automated driving, connectivity, *e.g.,* for entertainment, or telephone service [DGH+19]. The implementation of these functions requires the interaction of software, mechanical, and electrical subsystems [GRSS11]. Modern systems have, therefore, become cyber-physical, meaning that they comprise software, mechanical and electrical subsystems that need to interact to achieve the desired functionality [Lee08]. Hence, Cyber-Physical Systems (CPSs) are technical systems whose functionalities arise from the interaction of computational with physical processes [BS08, Lee08, Pto14]. Engineering such systems is highly complex due to the high levels of interactivity among software, mechanical, and electrical subsystems. Therein, each subsystem is highly complex by itself.

1.1.1 The Problem-Implementation Gap in CPS Engineering

A part of the complexity of engineering CPSs arises because in traditional systems engineering development is concerned with implementing the physical parts or assemblies, software modules, or electrical circuits instead of implementing the system's functions [DRW+20, GRSS11]. As the demand for systems to implement smart, pervasive *functions* increases this prevalent approach to decomposing CPSs under development, and distributing the development tasks accordingly, creates accidental

complexities [FR07] during the engineering process. Therein, it becomes unclear which system components contribute to the implementation of the functional requirements within the overall system. This ambiguity gives rise to a conceptual problem-implementation gap [FR07] between (documentations of) the functional requirements and the implementation [BGK+09]. This aggravates the development process in all phases: During system design and implementation, functional dependencies remain implicit and may not be regarded such that unwanted side effects may occur during late development stages when changing the system design is expensive. The V&V of these systems has become a hot research topic since validating the high-level functional requirements is most often delayed to late development stages when expensive prototypes are available. System failures or errors detected at this stage may imply major changes to the system's design and re-validating the revised version of the system, again, requires expensive prototypes. Requirements stated in natural language with little or no standards that prescribe their structure, further hinder this process, because their meaning may be ambiguous, and the lack of structure often prevents automated tests, especially at system level. Additionally, traditional approaches to systems engineering are document-based [DGH+19] which further prevents to systematize, and support the development activities, *e.g.,* by automated tests, automated consistency checks of development artifacts, or through a common system understanding among all stakeholders.

Model-Driven Engineering (MDE) provides an approach to overcome a problem-implementation gap [FR07] such as the one between the functional requirements and system components. Figure 1.1 illustrates the prevalent concepts of MDE. In MDE, *models* that abstract from the technical details of a system's implementation are the primary development artifacts. The abstraction enables engineers to focus on their field of expertise. The modeling *language* provides a syntax that captures the necessary elements of the engineering domain. The formal semantics of the modeling languages in which these models are created prescribe a meaning of the models, *i.e.,* the primary development artifacts in mathematical terms [HR04]. Ideally, this meaning is commonly understood among humans and computers, and, therefore, establishes a common system understanding among experts of different backgrounds [FR07]. Interpreters and generators exploit the formality imposed by the language and provide automatic analyses and syntheses, which allows, *e.g.,* to link the functional requirements that describe the problem domain to (models of) the implementations that represent the solution space [FR07].
MLE is a field that is concerned with creating these modeling languages such that they enable involved stakeholders to create and understand the models intuitively and unambiguously [CFJ+16]. A part of this process is to find a mathematical theory that serves as a formal semantics for models in the language. Following [Esc20], the

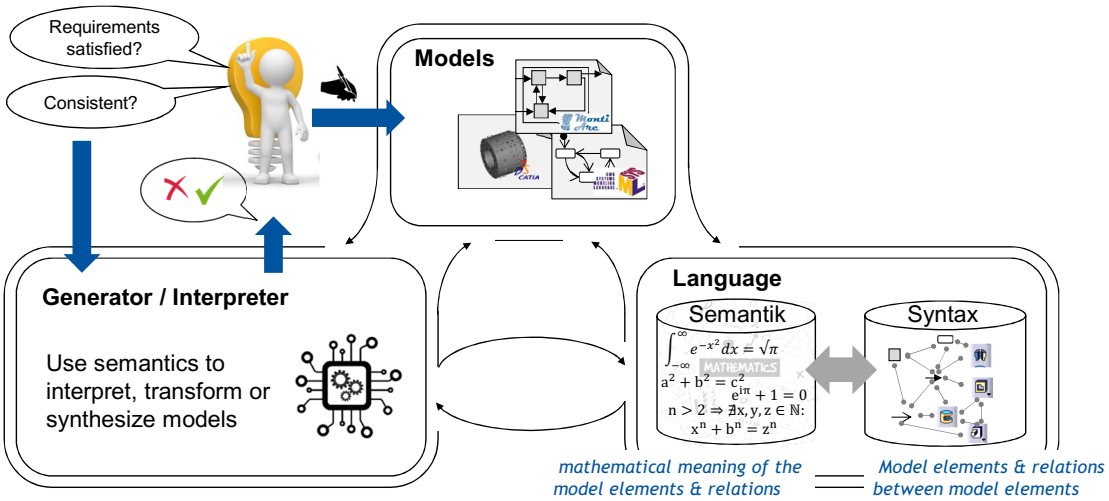


Figure 1.1: Modeling languages allow making models explicit and manageable. Semantics define the meaning of syntax in mathematical terms which enables the implementation of generators or interpreters for automatic analyses, and syntheses.

semantics put the ideas obtained from observations of the real world into mathematical words. Modeling languages in MDE enable experts to express aspects of their domain in these words without needing deep knowledge of the mathematical theory. Model interpreters implement a mapping from the model to a set of sentences in the theory, and answer questions about the properties of the modeled system, or synthesize other models such as test cases based on this interpretation. The formal semantics thereby equips each model in the language with a precise meaning that humans and computers understand equally.

1.1.2 Functional Model-Driven Engineering of Cyber-Physical Systems

The idea of functional systems engineering is similar to an idea from mechanical design methodology which understands a CPS as a function that transforms streams of energy, matter, and data, which Figure 1.2 illustrates [Kol85, BG21]. We call such a function a Cyber-Physical Function (CPF). Very similar to software architectures and models thereof, a system's CPF is hierarchically decomposed, *i.e.,* an engineer derives a specification of such a function from the requirements and throughout the course of the development process subdivides it into smaller and easier to solve sub-problems. Considering this idea in a model-driven approach to CPS engineering, such models of the hierarchical structure of the system's CPF become the primary development artifacts. This allows putting the innovation driver of CPSs, *i.e.,* functions, into focus,

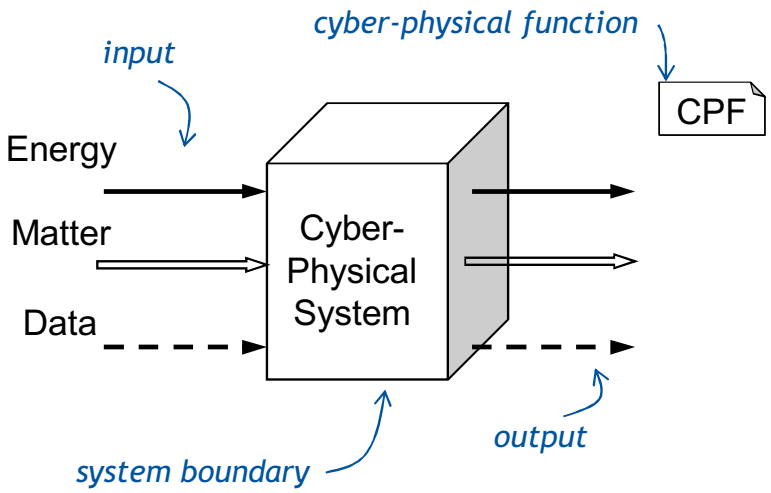


Figure 1.2: Notion of a Cyber-Physical System introduced in mechanical design theory [KK98].

and facilitates coping with the complexities that arise from the functional interactions among the system's components. The idea of understanding a system in terms of its functions is common also in software, and electrical engineering [Bro10, Alu15, Pto14]. However, this common understanding of a system has not yet been exploited to define systems modeling languages that are holistically understood among the experts of these heterogeneous domains, and that have formal semantics at the same time.

Employing functional models as part of an MDE methodology for systems engineering enables identifying which domains need to cooperate in order to implement the required functions and pave the way to automate, *e.g.,* change propagation, or V&V activities. However, functional modeling techniques currently lack an unambiguous and domain-independent definition of the term function, as well as appropriate modeling languages including a mathematical theory that serves as the semantic domain.

Contribution. This dissertation proposes a functional and model-driven approach to CPS engineering that that puts the innovation driver of modern systems engineering [DGH+19], *i.e.,* CPFs, into focus.

Focus [Rum96, BS01, Bro10, RR11] is a theory that is applied in functional model-driven approaches to software and embedded systems engineering that formalizes the idea illustrated in Figure 1.2. We hypothesize that modeling a CPS as a function that processes streams of energy, matter and data based on a Focus semantics will overcome the problem-implementation gap. To this effect, we provide an object-oriented modeling technique that enables to describe types of energy matter and

data. Further, we derive a theory of cyber-physical functions by synthesizing the approaches from [SRS99] and [Bro12] to regard both discrete and dense streams at the same time. This formalizes our understanding that by processing streams(in the sense of [Bro12]) of energy, matter and data, a CPS defines a CPF and that the CPF encapsulates the physical and logical structures of an implementation in which software and hardware interact. We show that this understanding can be formalized using the theory of TSPFs over continuous and discrete time
domains [SRS99, BS01, Bro13, Bro10, Bro12]. This theory provides an unambiguous meaning to specifications of desired functionalities. In the development process refining these specifications iteratively makes these specifications a reflection of the current development stage.
We aim to provide an integrated methodology that provides a formal specification of the system under development, independent of the domain it is implemented in. By formalizing informal functional requirements, these specifications bridge the gap between the functional requirements and specifications of how the system performs its transformations. To this effect, we propose the functional development paradigm as a set of principles derived from the application of Focus to the domain of CPSs. Further, we propose a formal methodology that implements these principles by using the modeling techniques for CPFs and cyber-physical types. In the proposed methodology, models of the system's function formalize the functional requirements and are then iteratively refined throughout the development.
We then show how the modeling technique enables modeling functional
structures [BG21] of mechanical systems such that this model is understood in both software and mechanical engineering. Eventually, this will enable the decomposition of a system into reusable components provided in model libraries. Since the modeling technique rests on an existing sound theory that has been applied in practical software engineering of CPSs [AVT+15a], existing analyses for software or embedded systems can be reused to enable agile development driven by automation.
To showcase the proposed approach in practice, we shed light on the engineering of modeling languages that regard the mechanical development activities in CPS engineering. The meta-model first published in [DRW+20] provides a language engineering perspective on the terms, concepts, and relations of mechanical design methodology. For modeling languages, such as SysML for Functional Mechanical Architectures (SysML4FMArch) [DRW+20] which implements this meta-model, it is possible to implement automatic validation and verification procedures. As the meta-model, SysML4FMArch was first published in [DRW+20]. The second part of this dissertation adapts the meta-model to abstract from the technical details of an implementation of the modeling languages. It provides a basis for defining modeling languages to integrate the engineering of physical components in the functional MDE of CPSs. We show how SysML4FMArch implements this version of the meta-model and provide details on its implementation in the modeling tool MagicDraw.

1.2 Research Questions

The contribution of this dissertation outlined above is guided by the following research questions.

Research Question 1. *How can functional specifications regard the transformation of energy, matter and data at once to serve as an abstract, domain-independent model of a CPS and how can the design of physical components be supported by such a specification?*

This question subdivides into the following questions. Each chapter of this dissertation provides answers to one of these questions.

RQ1 What are the characteristics of streams of energy, matter, and data and how can these characteristics be described?

RQ2 How can the theory of TSPFs [Bro10, Bro12, SRS99] describe the functions of CPSs?

RQ3 What are the constituents of an MDE methodology that targets the development of system functions?

RQ4 How can functional MDE integrate mechanical engineering activities?

RQ5 What are the constituents of a useful modeling language for specifying CPSs from a functional point of view? And what does such a modeling language look like?

RQ6 How can these functional specifications facilitate dimensioning and testing to support agile development of CPSs?

1.3 Preliminaries

This section provides preliminaries on the following:

- Section 1.3.1 introduces and defines the notation and conventions used throughout this dissertation to establish a mathematically grounded understanding of the functions of a CPSs.
- Section 1.3.2 summarizes the idea of MDE and defines what a formal methodology is in this context. The section details the principles that a formal MDE methodology must implement in order to be practically applicable. The preliminaries provided in this section are used in Chapter 4.

- Section 1.3.3 provides insight into the basic setup of an automotive cooling system which serves as a recurring example throughout this dissertation.
- Section 1.3.4 summarizes the elements of MontiArc's graphical syntax reused and enhanced to include logical statements in this dissertation to represent the functional structure of CPSs.
- Section 1.3.5 summarizes the elements of SysML as specified in the standard [Man19] which are used to define SysML4FMArch a modeling language in the form of a SysML profile for the functional and model-driven engineering of CPS presented in Chapter 6.

1.3.1 Notation and Conventions

Throughout this dissertation, we use the following notation: $\mathbb{R}_+$ denotes the set of positive real numbers including 0. The symbol $\mathbb{N}$ denotes the natural numbers, $\mathbb{N}_0$ denotes the natural numbers including 0. Let $x \in \mathbb{R}$, then $|x|$ denotes the absolute value of x. Let $n \in \mathbb{N}$. For a vector $v \in \mathbb{C}^n$, we denote its transpose by $v^T \in \mathbb{C}^{1\times n}$. The symbol $\wp(M)$ denotes the power set of a set M, and $\wp^{fin}(M)$ denotes the set of all finite subsets of M.

Logics As in [Bro10], we utilize logical formulas to express interface assertions, *i.e.,* logical statements about the relation between the inputs and outputs of a function. To this effect, we utilize the standard notation, with the connectives $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not"), $\Rightarrow$ ("implies"), $\Leftrightarrow$ ("if and only iff"). As quantifiers we use $\forall$ ("for all"), $\exists$ ("exists"). Propositional logic is subject in many basic literature on mathematical logic. A definition for a formal language to state formulae in propositional logic can be found *e.g.,* in [BM97]. Further, $v_1, v_2, \ldots$ with indexes $i \in \mathbb{N}$ denotes an infinite collection of variables. Also, we use parentheses) and (, and the symbol = which denotes the "equal sign".

Sets, Intervals, Sequences Sets play an important role when specifying the functional behavior of CPSs, and when defining types of energy, matter, and data. Let X, Y be arbitrary sets. Then, $X \times Y$ denotes the Cartesian product of X and Y, *i.e.,* the set of all pairs (x, y) with $x \in X$ and $y \in Y$. For $z \in X \times Y$ we denote by z_X the X-component of z and by z_Y the Y-component of z. Further, $\#X$ denotes the cardinality of the set X. to indicate that Y is a superset of X, we write $X \subseteq Y$. A set X with a partial order $<$ is called *discrete* iff for all $x \in X$ the set $\{y \in X \mid y < x\}$ is finite [Bro12]. The set X is called *dense* iff for all $x, y \in X$ such that $x < y$ there exists $z \in X$ such that $x < z < y$.
We consider the set of real numbers $\mathbb{R}$ to be equipped with the generic ordering "$<$" (see, *e.g.,* [Foe16]).

For Intervals, we use the following set of notations: Let $a, b \in \mathbb{R} \cup \{-\infty, \infty\} = \mathbb{R}_\infty$ with $a \leq b$

- $[a, b]$ denotes the closed interval from a to b, *i.e.,* the set $\{r \in \mathbb{R}_\infty \mid a \leq r \leq b\}$
- $]a, b[$ denotes the open interval from a to b, *i.e.,* the set $\{r \in \mathbb{R}_\infty \mid a < r < b\}$
- in analogy $]a, b]$ and $[a, b[$ denote the half-open intervals from a to b.
- the numbers a and b are called the *boundaries* of the respective intervals
- we say that an interval is finite, iff $-\infty < a, b < \infty$.

For sets, we use the following set of notations: Let M be a set

- M^∞ denotes the set of all infinite sequences of elements in M
- M^* denotes the set of all finite sequences of elements in M
- $M^\omega = M^\infty \cup M^*$ denotes the set of all finite and infinite sequences of elements in M

Functions We consider functions in the mathematical sense. Throughout this dissertation we will see that a CPF that processes energy, matter, and data can be considered such a function. That is, a function is a mapping between two sets that assigns to each element in the one set, called the domain, exactly one element in the other set, called the codomain or the image.

Definition 1.1 (Function [Rum96])**.** *A function f on the sets X and Y is a triple (X, Y, f) where $f \in X \times Y$ is such that for each $x \in X$ there exists at most one image $f(x) \in Y$.*

Here, the set f defines a relation between the domain and the codomain. Typically, we also write $f : X \to Y$ to denote a function instead of the tuple-notation. Functions can be composed such that for functions $f : Y \to Z$, and $g : X \to Y$ the relation $h = f \circ g$, where $h(x) = f(g(x))$ for all $x \in X$ defines another function $h : X \to Z$.
For functions, we use the following set of notations: Let M, N be sets and $S \subseteq M$ and $f : M \to N$ be a function

- $f|_S$ denotes the restriction of f to elements in S
- $f(S)$ denotes the image of f under S, *i.e.,* the set $\{n \in N \mid \exists s \in S : f(s) = n\}$
- $[M \to N]$ denotes the set of all functions from M to N and we write $F : [M \to N]$ or $F : \mathbb{P}(M \to N)$ to denote a set F of functions from M to N.

- $\mathrm{dom}(f)$ denotes the domain of f, *i.e.,* $\mathrm{dom}(F) \stackrel{\text{def}}{=} M \setminus \{m \in M \mid \nexists f(m) \in N\}$, and $\mathrm{img}(f) = N \setminus \{n \in N \mid \nexists m \in M : f(m) = n\}$ denotes the image of f.

Definition 1.2 (Cauchy Continuous)**.** *Let $(X, ||\ ||_X), (Y, ||\ ||_Y)$ be metric spaces (cf. Definition B.5) and $f : X \to Y$ be a function. Then f is called* continuous *on X iff for all $\varepsilon \in \mathbb{R}_+$ there exists $\delta \in \mathbb{R}_+$ such that for all $x, x' \in X$ it holds that*

$$||x - x'||_X < \delta \Rightarrow ||f(x) - f(x')||_Y < \varepsilon.$$

Smoothness is a characteristics of functions that we use to distinguish between hybrid streams that represent energy, fluid material, or signals. We define these properties as follows:

Definition 1.3 (Smooth Function [SRS99])**.** *Let $f : I \to M$ be a function for $I \subseteq \mathbb{R}_+$. Then, f is called* smooth *iff f is infinitely often differentiable, or, in case $M \not\subseteq \mathbb{R}^n$ for $n \in \mathbb{N}$, iff f is constant on I. The set $D(f) \subseteq \mathbb{R}_+$ denotes the points in time, where f is not smooth, i.e., discontinuous*

1.3.2 Constituents of a Formal Model-Driven Engineering Methodology

MDE is an engineering paradigm in which development decisions are made based on the information stored in models. Therein, a model is a purposeful representation of an original system that is reduced or abstracted from its size, detail, and/or functionality [Sta73]. The purpose determines what the model is used for in the development process. In MDE models reflect the state of the system under development at any point in time during the development process. They serve multiple purposes, in particular, not only documentation purposes, but enable to automate specific tasks in the development process. To this effect, the modeling language defines modeling elements that represent aspects of the system in the real world that are the focus of the development.

Definition 1.4 (Modeling Language [Rum96, HR04, Kau21])**.** *A modeling language is a triple $(M, S, [\![\ ]\!])$, where*

- *M is a set of well-formed models*
- *S is a semantic domain, and*
- *$[\![\ ]\!] : M \to \wp(S)$ is the semantic mapping that assigns each well-formed model a meaning, i.e., a set of elements from the semantic domain.*

Systems engineering is concerned with the development of highly complex systems that cannot be described by models in a single language [HKR$^+$07]. These systems are described by multiple models in heterogeneous languages [Rum13]. The UML is a

typical example for a *systems modeling language* that includes multiple languages to model software systems. Language composition tackles this challenge [Völ11] and enables to define a systems modeling language as the composition of multiple modeling languages that address specific aspects of the system under development.

Model Composition allows describing a system by decomposing it into smaller, and easier-to-develop subsystems. In the functional development paradigm, composition refers to decomposing the system along its CPFs rather than its geometric components. For this composition to be traceable and intuitive, model composition must be thoroughly defined, in particular, such that it is compatible (associative) with the semantic mapping. Because modern systems are compositional comprising multiple components which define a CPF, composition needs to be performed on both the syntactic and the semantic level [Rum13, HKR$^+$07]. The composition operator on the syntactic domain and the semantic domain must be compatible [HKR$^+$07]: Let (M, S, Sem) be a modeling language, and $\oplus : M \times M \to M$ be a syntactic composition operator, and $\otimes : S \times S \to S$ a semantic composition operator. For two models $m1, m2 \in M$ it must hold that

$$[\![m1 \oplus m2]\!] = [\![m1]\!] \otimes [\![m2]\!]. \tag{1.1}$$

In the following, we assume that $(\mathcal{M}, \mathcal{SM}, [\![\ ]\!])$ is a modeling language with a syntactic composition operator $\oplus$ and a semantic composition operator $\otimes$. The semantics of a set of models is then the semantics of the composition of the models in the set, *i.e.,* let $M \subseteq \mathcal{M}$ be a set of models. Then $[\![M]\!] = [\![\oplus M]\!] = \otimes [\![M]\!]$.

Model Consistency assures that the set of models that describe the status of the system under development do not contain contradictions. Because CPSs are so complex which leads to a very high number of models present in their development, automated checks to identify contradictions as soon as possible are crucial for efficient development.

Definition 1.5 (Consistency of a Set of Models [Rum96])**.** *A set of models $M \subseteq \mathcal{M}$ is consistent, iff their semantics is not empty, i.e.,*

$$M \text{ is consistent} \Leftrightarrow [\![M]\!] \neq \emptyset.$$

Besides consistency, the set of documents that describes a system under development must not be *redundant*. In practice, the danger of two or more models describing the same aspect or extract of a system lies in the possibility that these models evolve separately from each other which will lead to the inconsistency of the set of all documents. Further, it is highly inefficient to work on two models describing the same thing. Therefore, an agile approach to CPS engineering must allow the detection of redundant models automatically.

Definition 1.6 (Implication for Models [Rum96]). *A set of models $M \subseteq \mathcal{M}$ implies another model $m \in \mathcal{M} \setminus \{m\}$ if adding m to M does not yield a change in the semantics of M, i.e.,*

$$M \models m \Leftrightarrow [\![M]\!] = [\![M \oplus m]\!] = [\![M]\!] \otimes [\![m]\!].$$

The implication relation reveals a formal condition that determines whether or not a model is redundant [Rum96].

Refinement formalizes development steps that transform models. On the way from high-level system specifications that describe *what* a system shall do, to a technical description of *how* the system performs a transformation, models are iteratively refined. That is, through each development step the models evolve to regard more and more details that become available during the engineering process [DKMR19]. A model m refines a model m', iff every element in the semantics of m is also an element in the semantics of m' [Rum96, Bro10].

Definition 1.7 (Refinement). *Let $m, m' \in \mathcal{M}$ be two models. Then, m* refines *n iff $[\![m]\!] \subseteq [\![m']\!]$.*

In systems engineering, decomposition is a means to overcome complexity and stems from the principle of divide-and conquer. It refers to a system, or more accurate for this work, a function is specified by the composition of many sub-functions. To support engineering in distributed teams, refinement must be compositional, *i.e.,* composition must be defined such that the following property is met: Let $m, m', n \in \mathcal{M}$ be models and assume that $m \models m'$. Then, $[\![m]\!] \otimes [\![n]\!] \subseteq [\![m']\!] \otimes [\![n]\!]$ and $[\![m \oplus n]\!] \subseteq [\![m' \oplus n]\!]$, *i.e.,* replacing the model m' by a refined version in the composition with n yields a refinement of the overall composition.
In contrast to a refinement, a refactoring means a transformation of a model such that its syntax remains unchanged. Formally, a refactoring is an implication in both directions:

Definition 1.8 (Refactoring). *Let $m, m' \in \mathcal{M}$, and G be two models. Then, the model m* refactors *the model n iff $[\![m]\!] = [\![m']\!]$.*

Formal Methodologies systematize MDE by enabling the application of formal methods not only for interpreting models but also for defining relations among different models possibly written in multiple modeling languages. We adapt the notion of a formal methodology has been defined in [Rum96]. The methodology for the functional MDE of CPSs is based on the ideas presented in [Rum96] and aims to implement the principles published there. The formality of the methodology allows to implement it with different modeling languages, tools and processes.

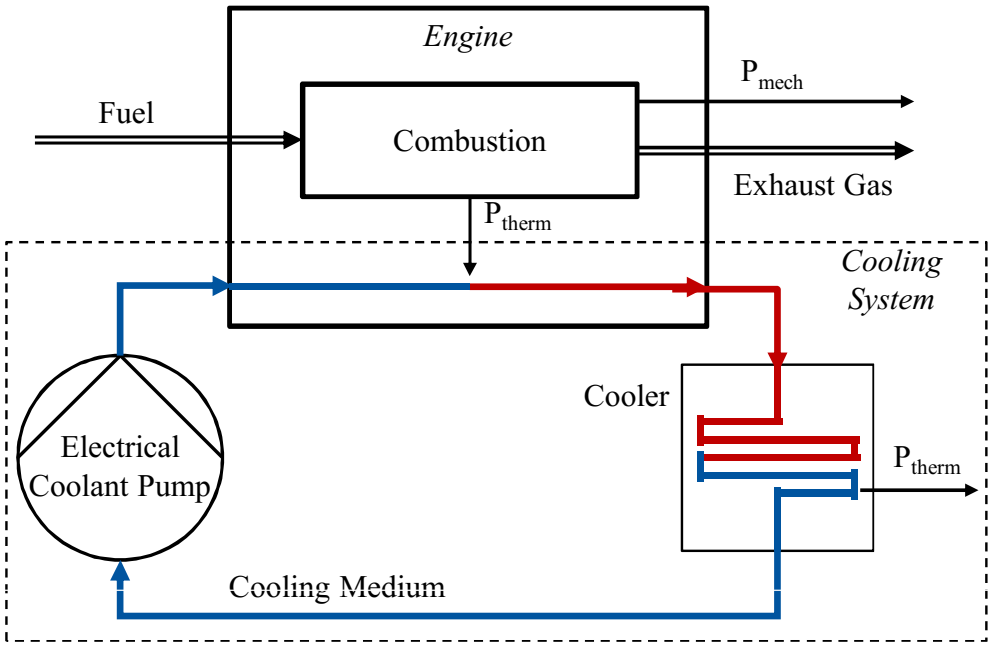


Figure 1.3: Diagrammatic illustration of an automotive combustion engine with a cooling system [DRW+20].

Definition 1.9 (Formal Methodology [Rum96])**.** *A formal methodology consists of*

- *a (systems) modeling language that defines kinds of models together with a semantic domain and a semantic mapping for each kind,*
- *a set of development steps that transform the models that are also formally grounded, and*
- *a set of guidelines of when and with which aim to apply each transformation.*

1.3.3 The Automotive Cooling System

As in [DRW+20], the automotive electrical coolant pump which is a subsystem of the automotive cooling system will serve as a recurring example throughout this dissertation. The section summarizes the setup and working principle of such a system to enhance the understanding of the examples.
The drive system of a vehicle converts input energy into mechanical energy and transfers the mechanical energy onto the road, where friction 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. For example, combustion drives convert the chemical energy held by fuel into mechanical energy. In contrast, electric drives, convert electrical to mechanical energy. The engine of a vehicle is responsible for converting an incoming energy to mechanical energy. Combustion drives, for instance, convert the chemical energy held by fuel into mechanical energy. In contrast, electric drives, convert electrical to mechanical energy. Figure 1.3 illustrates the principle setup of a combustion engine diagrammatically. By combustion, a portion of the chemical energy held by the incoming fuel is transformed into thermal energy which causes the pressure in the combustion chamber to increase. The released exhaust gas holds the rest of the chemical energy while the increasing pressure acts on the surface of the engine's piston as mechanical energy (P_{mech}) causing the piston to move. However, some of the thermal energy (P_{therm}) is released as heat and causes the engine's temperature to increase. Once above a maximum threshold, the engine overheats and stops functioning. To prevent the engine from overheating, it has to be cooled, *i.e.,* the excess heat has to be dissipated. In automotive systems, water cooling systems, as sketched at the bottom of Figure 1.3, often 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 [Ste94], the circulating cooling medium absorbs the combustion heat at the engine. The cooler releases the heat from the cooling medium to the surrounding air to cool it down. For convection to work, the cooling medium needs to circulate, *i.e.,* keep moving. In the technical implementation shown in Figure 1.3, the circulation is achieved by a cooling medium pump, which uses electrical energy to circulate the cooling medium. Because the coolant pump and its control to adjust the current power it generates, have a great impact on the engine's efficiency [JDL+17] and because its failure leads to expensive adjustments [JDL+17] its engineering is a critical task in automotive development. The hydrodynamic processes together with a software-based control, however, make the cooling system complex [Gü10]. Validating a design of the pump requires considering other system components [Gü10], *e.g.,* the engine, and changes in the environment [KHSE15], *e.g.,* the temperature.

1.3.4 The Architecture Description Language MontiArc

This preliminary section presents MontiArc's graphical syntax which is for presentation purposes. MontiArc is a powerful ADL that comes with a lot of infrastructure for code generation, *e.g.,* for simulations, and analyses in the context of V&V. The theory of TSPFs over discrete time domains [BS01, RR11] provides the foundation on which MontiArc's semantics is built. We reuse MontiArc's graphical syntax to model CPFs as TSPFs over continuous time domains as MontiArc already includes the necessary language constructs for types, channels, and components. Here, we include an expression language to specify the behavior of the components as interface assertions and a version of hybrid automata that enables to specify functional behavior in a

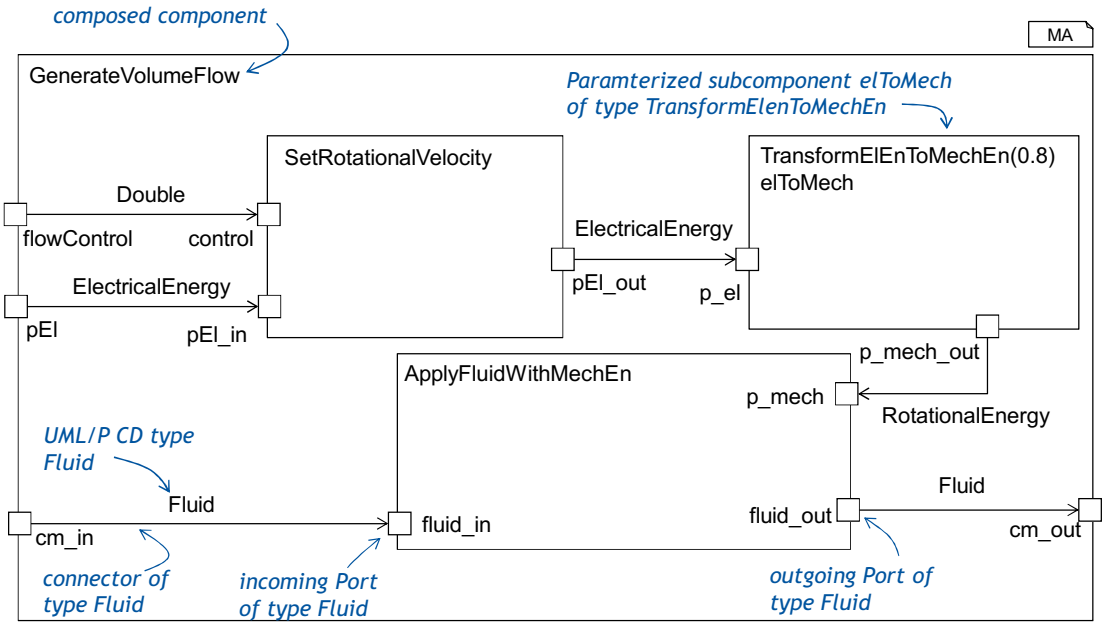


Figure 1.4: A MontiArc [Hab16] example that models the architecture of a hydraulic pump.

state-based manner. The semantics of these models is established in Chapter 3. This dissertation provides an extension of the theory of TSPFs over discrete time domains [BS01, RR11], the foundation on which MontiArc's semantics is built, to enable extending architecture-centric MDE of software systems to CPSs. The infrastructure provided by MontiCore [HKR21] allows to extend MontiArc to include respective language concepts easily.

MontiArc [Hab16, HRR10, HRR12b] enables an architecture centered approach to software engineering and implements the (discrete) Focus semantics [BS01] for software systems. The language follows the C&C paradigm [Kus21] to. *Components* encapsulate a functionality. The interface of a component comprises typed ports and message exchange between components is represented by connectors between these ports. In MontiArc, components are hierarchically arranged to represent the decomposition of a (software) system into functions that are easier to implement. The top-level functionality is the composition of these sub-components. MontiArc has been extended to enable modeling the behavior of components, *e.g.*, in the form of automata [Wor16]. MontiArc is implemented with the MontiCore language workbench [HKR21], which offers mechanisms for language composition that enable to integrate different behavioral languages. Figure 1.4 shows an example of a MontiArc architecture that includes all elements of the graphical syntax which we will reuse to

model the composition of CPFs. The component `GenerateVolumeFlow` is decomposed into three subcomponents. In MontiArc, it is possible to omit the name of a component instance, which is the case for the other two subcomponents of type `SetRotationalVelocity` and `ApplyFluidWithMechEn`. The interface of the topmost component `GenerateVolumeFlow` contains four ports: The three input ports `flowControl` of type `Double`, `pEl` of type `ElectricalEnergy`, and `cm_in` of type `Fluid`, as well as the output port `cm_out` also of type `Fluid`. The types `ElectricalEnergy`, and `Fluid` are custom types that are defined in a UML/P CD [Rum16, Sch12]. The messages that the component receives via the `flowControl`, and `pEl` ports are passed to the `control`, and `pEl_in` ports of the `SetRotationalVelocity` component, respectively.

1.3.5 SysML

The SysML profile introduced in [DRW+20], which is detailed in Section 6.2 is tailored for mechanical engineering and provides a language for modeling the functional structure of a mechanical system, which is understood as a reusable foundation of the mechanical product development cycle in [BG21]. This section provides a summary of the relevant SysML elements that are extended or reused in SysML4FMArch based on [DRW+20].
SysML is a general-purpose modeling language family for systems engineering [Man19] that includes modeling languages for behavior, structure, and requirements. The SysML reuses and extends a subset of the UML 2.5 [Man15] for describing aspects of system software and hardware in an integrated way [JKPB12]. The structure modeling languages include, *e.g.,* BDDs, and IBDs that describe the structure, interfaces, and properties of blocks. While BDDs model the external structure and dependencies among blocks, IBDs model the internal structure of a block. Figure 1.5 shows illustrative examples for a BDD and an IBD which contain most of the SysML modeling elements reused or specialized in the profile presented in Section 6.2: *Blocks* extend UML classes and model system decomposition, system interaction, and other system properties such as values [Man19]. The properties of blocks are organized in compartments. The *values*-compartment lists a block's *ValueProperties*, which are *ValueTypes* having composite aggregation, *e.g.,* `nW` of the block `Hydrodynamics`. *PartProperties*, listed in the *parts*-compartment, are blocks that have composite aggregation [Man19], *e.g.,* `wheel` of `WheelCyl`. *ConstraintBlocks* are specific blocks for integrating analyses, *e.g.,* of reliability, but also to specify physical constraints as mathematical formulas [Man19]. *ConstraintProperties* of a block are ConstraintBlocks and have composite aggregation, *e.g.,* `centriForce` of `Hydrodynamics` in Figure 1.5. *ConstraintParameters* are the ValueProperties of ConstraintBlocks and model the variables of such expressions. *ProxyPorts* are properties of a block and make features or internal parts of the block available for other components. They do not

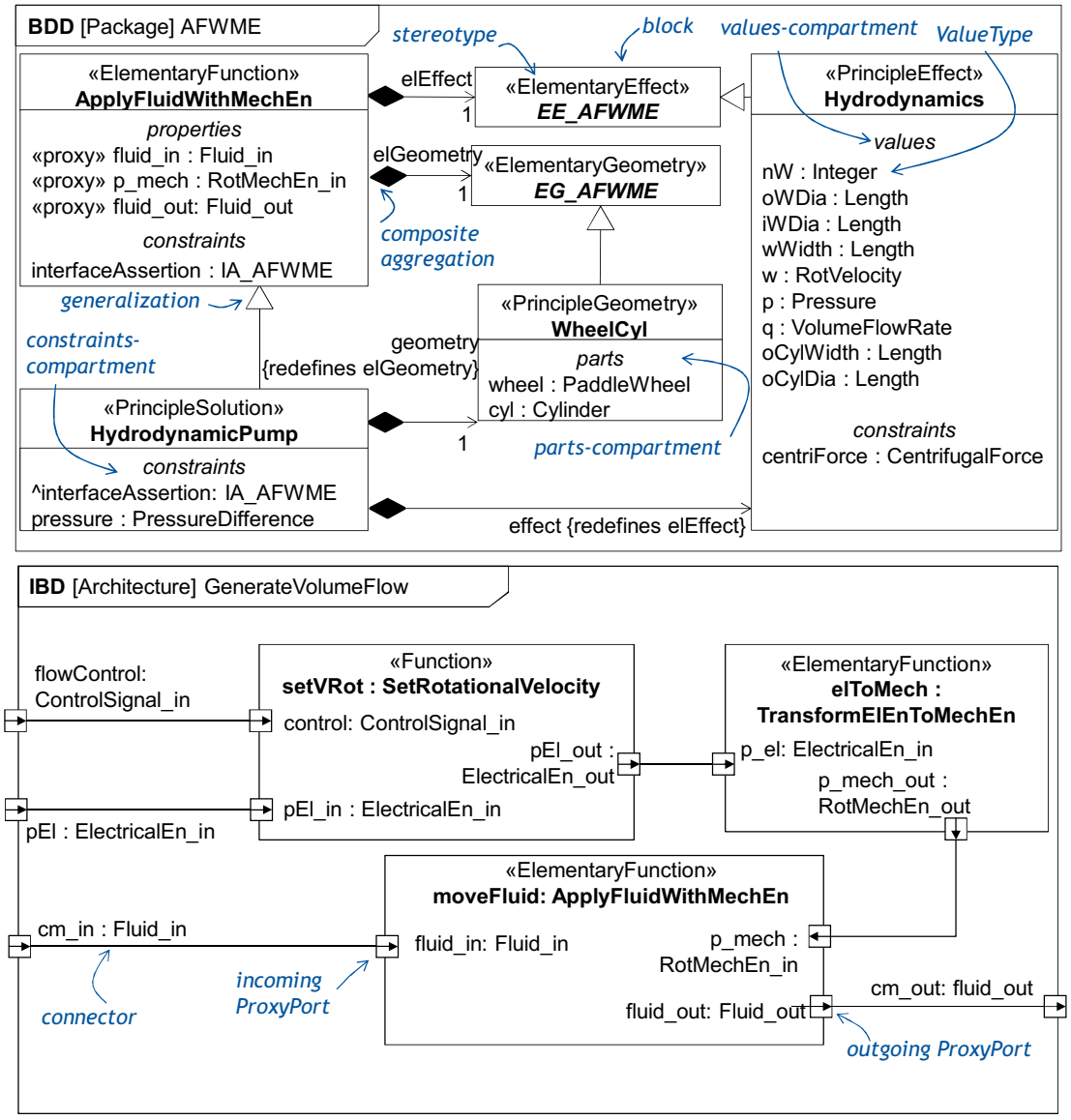


Figure 1.5: Top: SysML BDD of the functional architecture of the running example. Bottom: IBD of `GenerateVolumeFlow`. For details on stereotypes and contents see Chapter 6. [DRW+20]

model separate parts of the system and neither exhibit behavior nor comprise internal parts [Man19]. *InterfaceBlocks* provide the types for ProxyPorts and hold the stereotype «proxy», *e.g.,* `fluid_in` of the block `ApplyFluidWithMechEn` typed by the InterfaceBlock `Fluid_in`. InterfaceBlocks specify the elements that flow between a block and its environment through *FlowProperties* with direction *in, out* or, *inout* [Man19]. Section 6.2 gives more insight into the applications of InterfaceBlocks. An internal structure, *i.e.,* the interconnection of a composition of blocks, is modeled by an IBD that is associated with the composed block. The bottom of Figure 1.5 shows an IBD that models the functional structure of an automotive coolant pump, which will be further explained in the upcoming chapters. 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`, indicate this unique direction through an arrow. For instance, `p_el` typed by the InterfaceBlock `ElEnergy_in` in the IBD in Figure 1.5, has only FlowProperties of direction `in`. *Parametric diagrams* are restricted IBDs that show only the usage of ConstraintBlocks. *BindingConnectors* are connectors with stereotype «equal» that specify the equality of the numeric values of the properties at both ends [Man19]. As an extension of UML [Man17], SysML provides infrastructure to create profiles by defining stereotypes as extensions of meta-classes or as sub-stereotypes [Man19], which we exploit to define SysML4FMArch in Section 6.2.

1.4 Publications

The contributions of this thesis are the result of extensive research over the course of multiple years. The results on the application of SysML as a modeling language to bring the formal modeling technique into practice come from a long-standing cooperation with experts from mechanical engineering. The SysML profile SysML4FMArch which is presented in detail in Chapter 6 together with the meta-model that captures concepts from mechanical design theory are a result of this cooperation. Also, the model of the automotive cooling system and the concepts for modeling testing and dimensioning procedures in SysML have emerged from our collaboration. Therefore, this dissertation is partially based on the following publications of which I, Imke Nachmann, am the author or co-author:

- [DRW+20] I. Drave, B. Rumpe, A. Wortmann, J. Berroth, G. Hoepfner, G. Jacobs, K. Spuetz, 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, pp. 79-89, ACM, Oct. 2020.

- [ZJS+21] T. Zerwas, G. Jacobs, K. Spuetz, G. Hoepfner, I. Drave, J. Berroth, C. Guist, C. Konrad, B. Rumpe, J. Kohl: Mechanical Concept Development Using Principle Solution Models. In: IOP Conference Series: Materials Science and Engineering, G. Jacobs, S. Stein (Eds.), Volume 1097:012001, IOP Publishing, Feb. 2021.

- [HJZ+21] G. Hoepfner, G. Jacobs, T. Zerwas, I. Drave, J. Berroth, C. Guist, B. Rumpe, J. Kohl: Model-Based Design Workflows for Cyber-Physical Systems Applied to an Electric-Mechanical Coolant Pump. In: IOP Conference Series: Materials Science and Engineering, G. Jacobs, S. Stein (Eds.), Volume 1097:012004, IOP Publishing, Feb. 2021.

- [HNZ+23] G. Hoepfner, I. Nachmann, T. Zerwas, J. K. Berroth, J. Kohl, C. Guist, B. Rumpe, G. Jacobs: Towards a Holistic and Functional Model-Based Design Method for Mechatronic Cyber-Physical Systems. In: Journal of Computing and Information Science in Engineering (JCISE), Volume 23(5), Mar. 2023.

1.5 Thesis Organization

This thesis provides answers to the research questions posed in Section 1.2. Analyzing and providing (formal) definitions for energy, matter and data is inevitable as a basis for unifying the engineering domains. The research question RQ1 "What are the characteristics of streams of energy, matter and datahow can these characteristics be described?" which is the focus of Chapter 2 addresses this. The chapter elaborates on the characteristics of energy, matter and data in the context of CPS engineering. To integrate the provided notions in a respective modeling technique, the chapter defines a CD-like notation for modeling types of energy, matter and data which enables to distinguish entities by their physical or logical characteristics.
Chapter 3 addresses the research question RQ2 "How can the theory of TSPFs [Bro10, Bro12, SRS99] describe the functions of CPSs?". This chapter synthesizes a theory of CPFs from two versions of Focus: The first [Bro12] allows discrete and hybrid streams at the same time but specifies functional behavior in a set-based manner [RR11] which is not practical for functional CPS engineering. The second [SRS99] allows only hybrid streams but specifies functional behavior as sets of TSPFs which is the appropriate technique in our setting.
Chapter 4 defines the functional development paradigms by deriving five principles from some of the contributions that use Focus to define a formal semantics for modeling languages in software or embedded systems engineering. This set of principles can be understood as a framework for defining a functional approach to the development of a system in any systems engineering domain. The chapter then defines

a formal methodology based on the contributions on cyber-physical types and on CPFs provided in Chapter 2 and Chapter 3.
To address the research question RQ4 "How can functional MDE integrate mechanical engineering activities", Chapter 5 elaborates on mechanical design methodology which understands a (mechanical) system as illustrated by Figure 1.2. The chapter puts the concepts into the words of the theory of CPFs provided in Chapter 3 and showcases the applicability of the modeling technique in the mechanical domain by formalizing most of the functions from a mechanical design catalog [KK98].
Chapter 6 addresses the research question RQ5 "What are the constituents of a useful modeling language for specifying CPSs from a functional point of view? And what does such a modeling language look like?" by providing an updated version of the meta-model and the SysML profile from [DRW+20].
Chapter 7 evaluates the approach by providing two extensive modeling examples. The automotive electric coolant pump example was initiated in [DRW+20] and is adapted to the new version of the profile, here. Further, this chapter addresses the research question RQ6 "How can these functional specifications facilitate dimensioning and testing to support agile development of CPSs".
Finally, Chapter 8 concludes the thesis and gives an outlook to future work.

Chapter 2

Cyber-Physical Types

While software systems interact by exchanging (data) messages, CPSs define functions in the sense of Definition 1.1 that exchange energy, matter, and data via their interfaces. An *entity* represents an instantaneous interaction among components that involves the exchange of energy, matter, or data. Functional specifications define the behavior of the functions defined by a CPS as transformations of streams of energy, matter and data. This dissertation proposes, among others, to specify these transformations in mathematical terms through interface assertions and hybrid automata. These mechanisms require to talk about the characteristics of different types of energy, matter, and data. Similar to data messages, entities of energy and matter have a type that defines the common characteristics of a set of entities. In the physical or mechanical domains, these characteristics of energy and material types are modeled by physical quantities which are described by units. In the software domain, classes define the characteristics of data messages of a specific type by attributes and methods. Here, we propose to transfer these ideas to the physical domain and to describe kinds of entities through attributes and methods.
This chapter, therefore, addresses the research question RQ1 "What are the characteristics of streams of energy, matter, and data how can these characteristics be described?".

2.1 Preliminaries on Types and Classes

A formal modeling technique for types is introduced in [Rum96] where a type is modeled by a class with attributes and methods. In this dissertation, we utilize classes to represent types of entities that are exchanged among the functions of a CPS. To this effect, we summarize the notions from [Rum96].
Let $\mathcal{C}^U$ be a universe of class names, and $\mathcal{U}$ be a universe of entities. The mapping $\text{class} : \mathcal{U} \to \mathcal{C}^U$ assigns each entity a class. The relation $\preccurlyeq \in \mathcal{C}^U \times \mathcal{C}^U$ defines a partial order on $\mathcal{C}^U$ which represents the class hierarchy. We say that an entity $m \in \mathcal{U}$ is of type $c \in \mathcal{C}^U$ iff $\text{class}(m) \preccurlyeq c$ [Rum96]. Let $\mathcal{T}$ be a set of types. The mapping $\text{ent} : \mathcal{T} \to \wp(\mathcal{U})$ assigns each type a set of entities. A cyber-physical type becomes a set

of entities assuming that $\mathcal{C}^U \subseteq \mathcal{T}$ and for all $c \in \mathcal{C}^U$ it holds that
$\text{ent}(c) = \{m \in \mathcal{U} \mid \text{class}(m) \preccurlyeq c\}$.
Using classes to specify types allows us to define the characteristics of entities in the type via attributes and methods which are modeled using typed variables. Now, let $\mathcal{V}$ be a set of names for variables, and let methods be a set of names for methods. To this effect, the mapping $\text{attributes} : \mathcal{C}^U \to \wp(\mathcal{V})$ assigns each class a set of attributes, and the mapping $\text{methods} : \mathcal{C}^U \to \wp(\text{Meth})$ assigns each class a set of methods. Let $c \in \mathcal{C}^U$ be a class, $a \in \text{attributes}(c)$ be an attribute, and $b \in \text{methods}(c)$ be a method. Then, for all entities $m \in \mathcal{U}$ with $\text{class}(m) \preccurlyeq c$ we write $m.a$ or $m.b$ to denote the value of the attribute a or a call of the method b on the entity m. Every method obtains a set of arguments via the mapping $\text{args} : \text{Meth} \to \wp^{fin}(\mathcal{V})$ [Rum96]. The former includes the integers, reals, characters, Booleans, *etc.*, while the latter is defined based on the elementary types according to the records and variants paradigm [BS01].

2.1.1 Cyber-Physical Class Diagrams

A common notation for classes are CDs, which we will use to model cyber-physical types. Here, we use a CD-like notation to specify classes which we call Cyber-Physical Class Diagrams (CPCDs).

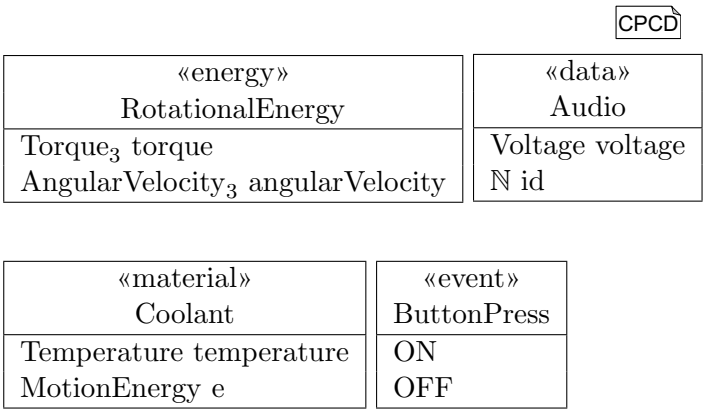


Table 2.1: Examples for the specification of cyber-physical types. The illustration shows all modeling elements used for specifying cyber-physical types in this dissertation.

Table 2.1 shows examples for specifications of cyber-physical types as classes. The illustration includes all notational elements. In the cyber-physical context, we need to indicate the kind of type, *i.e.,* energy matter, and data. The type of the kind will impose restrictions on which streams accurately represent a flow of the entities of that

type which is detailed in Section 3.2. The classes are therefore stereotyped.
As Section 2.2.4 elaborates, events are a special form of data, which we model as enums for which we utilize a respective stereotype. Here, the attributes of the energy-type MotionEnergy are given by attributes(MotionEnergy) = {velocity, force}, and their types are determined by type(velocity) = $\mathbb{R}^3(\mathrm{m\,s^{-1}}) \stackrel{\text{def}}{=} \mathrm{Velocity}_3$, and type(force) = $\mathbb{R}^3(\mathrm{N}) \stackrel{\text{def}}{=} \mathrm{Force}_3$. Here, the types are physical quantities which we model as $\mathbb{D}^n(U)$ where $\mathbb{D}$ is a number domain, $n \in \{1, 2, 3\}$ is the dimensionality, and U is an SI-unit. This notation is taken from [Kus21] and explained in detail in Section 2.2.1. The tag in the upper right corner identifies a type specification as a CPCD.
An extensive CD language together with formal semantics is defined, *e.g.,* in [NRSS22, RRS23]. The notational elements presented in Table 2.1 suffice for modeling cyber-physical types in the context of this thesis. Since modeling types is not the focus of this work, we will utilize the very simplified version of CDs. Since all elements are present in the language defined in [NRSS22], it is possible to apply the formal semantics detailed there to CPCDs. Also, we have not yet needed to specify methods for types, which is possible, by including the common CD-like notation for methods.

2.2 Modeling Cyber-Physical-Types

The previous section has established a general notion of a type together with a modeling technique. This section elaborates on the characteristics of cyber-physical types. Among others, we provide details on what physical quantities are and how they are used to model energy and matter types as classes with attributes (and possibly also methods) in terms of their characterizing quantities. Data types are commonly understood in software engineering and we integrate the prevalent notion in the cyber-physical setting.

2.2.1 Physical Quantities, Units, and Position

The characteristics of energy and material types are typically described by physical quantities. Physical quantities are the “words” used to describe physical phenomena and express a quantitative relation, or a comparison between physical systems [FR76, FR83]. Examples of physical quantities are length, energy, or momentum. A unit describes a measurement procedure that assigns a numerical value to a physical quantity, which allows describing a physical quantity as the product of a number and a unit [Tay08]. Unit systems define the set of units and thereby a set of available measurement procedures, which in turn defines a set of physical quantities, *i.e.,* those that are quantifiable by the available set of measurements. By defining units for physical quantities, the differentiation between ”base quantities” such as, *e.g.,*

length, and "derived quantities" such as, *e.g.,* force, becomes meaningful [FR83]. As each unit stands for an available measurement procedure, it makes sense to define units for those quantities that are well presentable through experiments, easy to access, and reproducible [FR83]. The derived quantities that do not exhibit these properties can then be calculated from the base quantities, and their units are defined accordingly. The unit system assigns base and derived units to these quantities, respectively [Tay08]. Therein, a derived unit is obtained from the products of powers of the base units [Tay08]. A product of two physical quantities is, therefore, defined by the product of the numeric values and the product of their units. In our context, we use physical quantities to model the characteristics that define a type of energy, or matter. Mathematically, physical quantities are captured as variables. Therein, scalar physical quantities only have a magnitude, while vectorial quantities are also associated with a direction [Arf85].

Modeling Physical Quantities Physical quantities are described by a number and a unit [Tay08]. Literature that uses or deals with physical quantities distinguishes between vectorial and scalar quantities [Arf85], where the former distinguish by the number being a vector in the n-dimensional real- or complex-valued vector space, where $n \in \mathbb{N}$. For scalar physical quantities, the number is, as the name suggests, a scalar, *i.e.,* a real or complex number. We define the SI-unit system following the standard [Tay08] and the formalization proposed in Appendix D.

Definition 2.1 (SI-Unit System)**.** *The system of SI-units denoted SI, is the unit system generated by the following set of base units:*

$$\{1_{SI}, \mathrm{s}, \mathrm{m}, \mathrm{kg}, \mathrm{A}, \mathrm{K}, \mathrm{mol}, \mathrm{cd}, \mathrm{s}^{-1}, \mathrm{m}^{-1}, \mathrm{kg}^{-1}, \mathrm{A}^{-1}, \mathrm{K}^{-1}, \mathrm{mol}^{-1}, \mathrm{cd}^{-1}\}.$$

Every unit in the SI-unit system has a representation as a product of powers of the base units (*i.e.,* the SI-unit system is generated by the base units). The units in the SI-unit system that are not base units are called *derived* units. A DSL that implements the concept of a physical quantity consisting of a number and a unit is provided in [Kus21] which also provides insights into a language engineering point of view for SI-units. The DSL also implements the multiplication of physical quantities and addition of physical quantities with compatible units. The concepts presented in Appendix D provide a formal meaning to this notation. We reuse the language introduced in [Kus21] for physical quantities. It integrates common primitive types and SI-units. The language offers the following number domains:

- the natural numbers $\mathbb{N}$,
- the natural numbers including 0, *i.e.,* $\mathbb{N}_0$,
- the integers $\mathbb{Z}$,

- the rational numbers $\mathbb{Q}$,
- the real numbers $\mathbb{R}$,
- the complex numbers $\mathbb{C} = \{a + ib \mid a, b \in \mathbb{R}\}$, and
- the Booleans $\mathbb{B}$.

We denote this set of types by $\mathbb{D} = \{\mathbb{N}, \mathbb{N}_0, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{B}\}$. Following the ideas introduced with the SI-DSL from [Kus21], physical quantities are considered primitive types and consist of a number and a unit.
These types are written in the form $\mathbb{D}^n(u)$, where $u \subseteq SI$ is an SI-unit and $n \in \mathbb{N}^k$ for $k \in \{1, 2, 3\}$ indicates the quantity's dimensionality. We denote the set of all such physical quantities by $\mathbb{D}^n[SI]$, where $\mathbb{D}$ is a number domain. The type system from [Kus21] offers mechanisms to define vectors, matrices, and cubes which allows to represent vectorial quantities accurately. To improve the readability of type specifications, we utilize names for physical quantities, *i.e.,* we define Current $\stackrel{\text{def}}{=} \mathbb{R}(\text{A})$. We utilize subscripts to let the modelers choose the dimensionality of vectorial quantities, *i.e.,* Temperature $\stackrel{\text{def}}{=} \mathbb{R}(\text{K})$ for $n \in \{1, 2, 3\}$. The rest of this dissertation uses the physical quantities defined in Appendix A.
Note, that in the implementation provided in [Kus21], the provided numeric domains do not include $\mathbb{R}$ and $\mathbb{C}$ which cannot be implemented by a computer. Instead, modelers can choose the rationals $\mathbb{Q}$ or the Gaussian rationals $\mathbb{Q}[j] = \{a + jb \mid a, b \in \mathbb{Q}\}$ as approximations, respectively. The set $\mathbb{D}^n[SI]$ denotes the set of physical quantities. As in [Kus21], we consider physical quantities to extend the set of primitive types.

Definition 2.2 (Primitive Type)**.** *A primitive type is a cyber-physical type $P \in \mathcal{T}$ from which all other types are constructed. Here, the set of primitive types is defined as*

$$\mathcal{P} \stackrel{\text{def}}{=} \mathbb{D}^n \cup \mathbb{D}^{n \times m} \cup \mathbb{D}^n[SI]$$

where $\mathbb{D} = \{\mathbb{N}, \mathbb{N}_0, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}, \mathbb{B}\}$ and $n \in \{1, 2, 3\}, m \in 0, 1, 2, 3$.

To every SI-unit that is not a base unit, there exist alternative notations. Two units $u, v \in SI$ fall into the same equivalence class of SI-units iff there exists a set of base units such that u can be expressed as a product of powers of the base units and v (*cf.* Section D.1). The mapping $\text{unit} : \mathcal{U} \to \mathbb{D}^n(SI)$ assigns each entity an SI-unit.

Position Modeling CPSs requires talking about position. Physics and mechanical engineering often consider the position of an object as a part of its state. For the functional modeling of CPSs, however, this interpretation is not appropriate because the CPF encapsulates the state, *i.e.,* the state of the system is not visible to an outsider. The position of a system, however, can be determined by an outside observer.

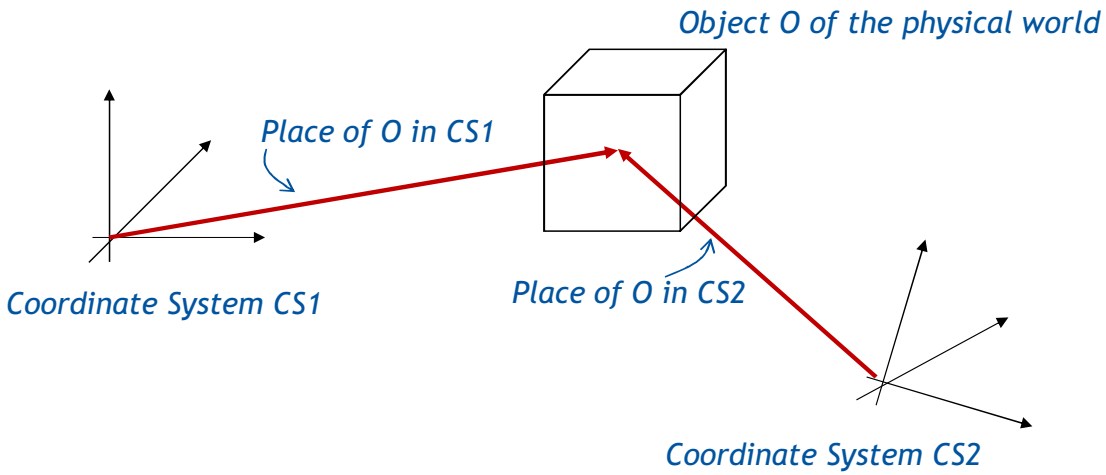


Figure 2.1: Place is the physical quantity that describes the position of an object in the real world with respect to a coordinate system.

In our modeling approach, the position of the object is determined at its interface: We consider physical entities, *i.e.,* energy and matter, to have a position. The interface of a CPF that receives or transmits these entities must be present at their position.
Part of every CPS is a physical structure that comprises physical parts, and assemblies (consider, *e.g.,* the chassis, engine, wheels, *etc.* of a car) that interact with the computational structures to realize these functionalities. Implementing and integrating these structures requires talking about the position of each physical part of the system. In physics, the position is often considered part of the physical state of an object. In a functional interpretation, the state is not visible from the outside. Because the position is visibile for outsiders, we do not consider it a part of the system's state. Physical quantities are always measured at a physical position and matter is always present at a physical position [Sti89]. Euclidean vector spaces (*cf.* Appendix B) are a common mathematical modeling tool to describe the position. In physics, place is a physical quantity that describes the position of an object with respect to a specific coordinate system. Therein, the modeler defines a (Cartesian) coordinate system (*cf.* Appendix B), and provides a model of a position through a coordinate vector. The value of the quantity Place is a vector in $\mathbb{R}^n$ for $n \in \{1, 2, 3\}$, which is a translation from the origin of the coordinate system to the (center of) the object (*cf.* Appendix B). Descriptions of physical systems heavily depend on the choice of the coordinate system, because computations, such as dimensioning or other simulations take the geometric models as input to reveal answers to questions regarding optimal dimensions, materialistic properties, or predictions. Models of the physical world often get more complex when analyzed in three-dimensional space; therefore, when analyzing a model

of the physical world, model users may abstract from one or the other dimension. Depending on how the coordinate system is chosen by the modeler, the place of an object may differ. The value of the physical quantity place depends on the coordinate system a model user has chosen. A common way of dealing with multiple coordinate systems is to set a global coordinate system. Through change of basis [KM03], vectors in coordinates of other coordinate systems can be transformed into the global coordinate system, which Figure 2.1 illustrates.

Modeling Physical Position Considering the cyber-physical *functions* of a system, the position of parts and assemblies belongs to the physical implementation of the system. The function thus encapsulates their position. However, a functional specification must talk about the position of the entities, *e.g.,* for specifying the transport of an object [KK98]. In this dissertation, every entity has a position that can be referenced in specifications.
We model the position by a primitive type $\text{Position}_n = \mathcal{A}_n(\mathbb{R}(\text{m})) \cup \xi$. Each position entity is then equipped with the SI-unit meter, *i.e.,* we consider the vectors to have a unit, which implies that distances can also be determined with the unit m. The special character ξ denotes the irrelevance of position which is the case, *e.g.,* for data messages. The variable $n \in \{1, 2, 3\}$ indicates the dimensionality of the space considered in the specifications. It makes sense to define the primitive type Position in this way instead of just as $\mathbb{R}(\text{m})$ because setting up a modeling project will require defining the coordinate system in which all positions throughout the project will be defined and, also, respective vector operations to be predefined.
The mapping $\text{pos}_n : \mathcal{U} \to \text{Position}_n$ assigns each entity a position in a global (*i.e.,* project/company/system/... wide) coordinate system. We consider the position as a point in the Euclidean affine space (see Appendix B).
Also, we do not consider other number domains than the real numbers, as positions are typically measured in this domain. In an implementation, it is certainly reasonable to use $\mathbb{Q}$. This position represents the position of the entity in the real world. Extending the mapping to apply to streams of entities is straightforward: Let $s \in [TD_s \to \mathcal{U}]$ be a timed stream, then $\text{pos}_n(s)$ is defined as the timed stream $TD_s \to \mathcal{A}_n(\mathbb{R}), t \mapsto \text{pos}(s[t])$.
As an example, consider a function that changes the position of an incoming entity $m1$. A specification of the behavior of such a function is

$$m1 = m2 \qquad \&\& \tag{2.1}$$

$$\text{pos}_3(m2) = \text{pos}_3(m1) \cdot v. \tag{2.2}$$

$$\tag{2.3}$$

Here, we represent that the output of the function $m2$ is unchanged with respect to all characteristics except for its position. The position of the output entity $m2$ is linearly translated by $v \in \mathbb{R}^3(m)$ which is *e.g.,* a data message.

2.2.2 Energy

In general, energy is a physical quantity that describes the capacity to do work. The law of energy conservation states that energy is never created nor destroyed [FR76]. Energy can only be stored by the system or *exchanged* between the system and its environment via the system's interface. In this case only, energy appears in a specific *form* [FR76]. The form can be determined by observing other physical quantities change, such as electrical current, forces, or velocity [Sta10, FR76, KK98]. The exchange of energy in a specific form is always bound to a change of at least two physical quantities, one being an intensive quantity, the other an extensive quantity [FR76, GB07], the so called energy components. In physics, quantities are distinguished into extensive and intensive quantities. For our purposes the simplified definitions of extensive are sufficient: The value of an extensive quantity increases proportionally with the number of particles [Sti89]. Another simplified version of the definition states that a quantity is extensive iff it doubles its value when combining two equal systems [FR76]. Examples of extensive quantities are mass and entropy. A quantity is intensive iff its value may stay constant over two consecutive aggregate phases [Sti89]. This is because the amount of energy a system receives or transmits is determined by the forces contributing to a change in the value of an external parameter together with the amount this parameter has changed.

Definition 2.3 (Energy Component [KK98])**.** *The physical quantities that an exchange of energy in a specific form is bound to, are the* energy components *of that form of energy.*

The exchange of energy is bound to the change in the value of the extensive variable and the amount of transferred energy is proportional to that change. The proportionality factor is the intensive quantity [FR76]. Mathematically, a form of energy is defined by the following general form [FR76]:

$$\text{form of energy} = \text{intensive quantity} \cdot d(\text{extensive quantity}). \tag{2.4}$$

It is very interesting that literature about mathematical models of streaming energy such as [FR76] in principle uses streams in the sense of Focus to model streams of energy: A stream of energy describes the amount of energy that is transferred within a fixed interval of time [FR76]. Considering in Equation 2.4, the change of the extensive quantity *over time* yields the stream of energy [FR76], *i.e.,*

$$\text{stream of energy} = \text{intensive quantity} \cdot \frac{d(\text{extensive quantity})}{dt}. \tag{2.5}$$

In this definition, the product has the unit of energy (Equation 2.4) or power (Equation 2.5), respectively.

In the running example of an automotive cooling system (*cf.* Section 1.3.3) a pump transforms electrical energy into rotational motion. In this example, the exchange of energy causes rotational motion [FR76]. The physical quantity that specifies this form of energy is angular momentum, or torque, which is also bound to a conservation law. That is, whenever the angular momentum of a system increases or decreases, it has to receive or transmit angular momentum from or to its environment, respectively [FR76]. The amount of rotational energy that the system receives or transmits per unit of time is determined by the product of the change in angular momentum, and the angular velocity of the system. The pairs of physical quantities that define a flow of energy in a specific form are predefined [FR76].
The messages that CPS components exchange may carry energy. Consider, for example, a sound signal which transmits information through pressure energy which is exchanged between the sender, *e.g.,* a speaker, and a receiver, *e.g.,* a human listener. Therefore, we consider the mapping $\text{power} : \mathcal{U} \to \text{Power}$, where $\text{Power} \stackrel{\text{def}}{=} \mathbb{R}(\text{W})$ to describe the amount of energy, a message transmits per unit of time.

Modeling Energy Types Whenever system components exchange or process energy it appears in a specific form and as mentioned above the exchange of energy is modeled by two physical quantities, *i.e.,* the energy components that determine the form of energy and how much energy is transferred. An energy type describes such a form of energy. A (standardized [FR76]) pair of physical quantities defines a type of energy, such that their product yields the change of energy.

Definition 2.4 (Energy Type)**.** *An energy type is represented by a class* $c \in \mathcal{C}^U$ *such that there exist attributes* $p, q \in \text{attributes}(c)$ *with* $\text{type}(p) = P$, $\text{type}(q) = Q$, *where* $P, Q \in \mathbb{D}^n[SI]$ *are physical quantities such that* $\text{unit}(P)\,\text{unit}(Q) = \text{W} \stackrel{\text{def}}{=} \text{kg}\,\text{m}^2\,\text{s}^{-1}$.

In the theory of [FR76], a form of energy is fully defined by a standardized pair of intensive and extensive quantities. Definition 2.4 considers a type of energy to be represented through a class that has attributes of the respective types of the intensive and extensive quantities that define this type of energy. Rotational energy, for example, is a form of energy whose exchange causes rotational motion [FR76] and is defined by the product of torque and angular velocity, *i.e.,*
$\text{RotationalEnergy}_n = \text{Torque}_n \times \text{Velocity}_n$, where the energy components are defined as $\text{Torque}_n \stackrel{\text{def}}{=} \mathbb{R}^n(\text{N}\,\text{m})$, $\text{AngularVelocity}_n \stackrel{\text{def}}{=} \mathbb{R}^n(\text{rad}\,\text{s}^{-1})$, and $n \in \{1, 2, 3\}$ indicates the dimensionality. In this dissertation, we leave choosing the dimensionality n to the modeler as it will have a major impact on the specifications of physical phenomena. This allows modeling the form in which the energy is transmitted while also enabling us to talk about the amount of energy that is transmitted per unit of time, *i.e.,* the power, which is calculated as the product of the two quantities. In the following, let $\mathcal{E}$ denote the set of all energy types.

Example 2.1. *Electrical energy is a form of energy that is bound to the physical quantities current and voltage. These quantities are defined as* $\text{Current} \stackrel{\text{def}}{=} \mathbb{R}(\text{A})$ *and* $\text{Voltage} \stackrel{\text{def}}{=} \mathbb{R}(\text{V})$. $\text{ElectricalEnergy} = \text{Voltage} \times \text{Current}$ *represents energy that is exchanged in the form of electricity. The cooling system in the running example (cf. Section 1.3.3) uses a coolant to dissipate the heat from the components of the vehicle. To model the functions that perform the dissipation, we consider the coolant's temperature and motion energy. These quantities are defined as* $\text{Temperature} \stackrel{\text{def}}{=} \mathbb{R}(\text{K})$ *and* $\text{MotionEnergy} \stackrel{\text{def}}{=} \text{Force}_3 \times \text{Velocity}_3$ *[FR76] with* $\text{Force}_3 \stackrel{\text{def}}{=} \mathbb{R}^3(\text{N})$ *and* $\text{Velocity}_3 = \mathbb{R}^3(\text{m}\,\text{s}^{-1})$. *A pump will cause the fluid coolant to flow. The pump uses an electric drive that causes a paddle wheel to rotate. Functionally, the pump transforms electrical to rotational energy. Rotational energy is a form of energy that is bound to the physical quantities torque and angular velocity. These quantities are defined as* $\text{Torque}_3 \stackrel{\text{def}}{=} \mathbb{R}^3(\text{N}\,\text{m})$ *and* $\text{AngularVelocity}_3 \stackrel{\text{def}}{=} \mathbb{R}^3(\text{rad}\,\text{s}^{-1})$. *Table 2.2 shows the classes that represent electrical energy, rotational energy, motion energy, and coolant.*

CPCD

«energy» ElectricalEnergy
Current current
Voltage voltage

«energy» RotationalEnergy
Torque_3 torque
AngularVelocity_3 angularVelocity

«energy» MotionEnergy
Force_3 force
Velocity_3 velocity

«material» Coolant
Temperature temperature
MotionEnergy e

Table 2.2: Example for the specification of the energy type that represents rotational energy.

Related Work: Bond Graphs Bond Graphs are a mathematical technique for modeling the exchange of energy among components [GB07] that emerged in the domain of control engineering. Bond Graph models are used to design and analyze the processes in dynamic or control systems [GS96, Alu15]. Thus, Bond Graphs describe the expected behavior of a system design in terms of physical laws. There exist many state-of-the-art tools that offer functionalities for the generation of symbolic representations, for model inversion, parametric identification or even for deriving simulations or other design aids [GB07].

Control systems are very often described by systems of differential equations. The idea for creating the technique emerged from the observation that dynamic systems generate similar equations across different energy domains (*e.g.,* electrical, fluid, or mechanical). From a mathematical point of view, these systems are therefore analogous[GB07]. Bond Graphs consist of components that exhibit energy ports and bonds between these ports. The bonds specify the exchange of energy among the components and the ports define an energy interface of a component [GS96]. Very similar to our approach, Bond Graphs consider a form of energy to be described by a pair of two variables such that their product is power [GS96]. A Bond Graph model is a graphical representation of a system of differential equations that describe the exchange of energy within a dynamical system. Because of the analogy of the different energy domains, Bond Graphs provide a universal modeling technique that is understood among the experts of these domains. The modeling of types is certainly analogous to the Bond Graph and our approach. However, Bond Graphs are a modeling technique to represent the actual behavior of a system. That is, an engineer thinks of the design of a system, *i.e.,* a constellation of geometric parts, or (controller) software modules, and creates a Bond Graph model of this setup to analyze whether his/her idea meets the requirements. This is very different from our approach which aims to describe the *desired* behavior of a system while abstracting from a technical design. Another major difference between Bond Graphs compared to our approach is that a Bond Graph model describes a set of differential equations. These models may be hierarchical in the sense that components encapsulate other components, but interpreters flatten the hierarchy and derive a system of equations from the flattened model [GS96]. Solvers that are used, *e.g.,* to generate simulations of the behavior, will produce different results when the hierarchy is not flattened. In contrast, our approach describes a system as the composition of functions in the mathematical sense (*cf.* Section 1.3.1). For functions that are not decomposed, we use logical formulae to describe properties of the function's transformation. Both approaches follow the principle of underspecification: A system of differential equations possibly has many solutions, and there exist possibly many functions that exhibit the properties described by a logical formula or automaton.

2.2.3 Matter

In physics, everything that takes up space, *i.e.,* has a volume and has a mass, is matter [SB91]. Mass and volume are, therefore, inherent properties of any piece of matter, independent of its manifestation. The presence or absence of matter can therefore be determined by the mass or volume being zero. Materials are types of matter that are distinguished by material characteristics which are described by a finite subset of physical quantities, *i.e.,* the material constants [Sti89], to which, *e.g.,* the density or specific heat capacity belong. A generic scheme for the definition of material constants is described in [Sti89]. For a specific material, the values of the

characterizing material constants do not change during system runtime. Depending on the surrounding temperature, pressure, electric and magnetic field, there exist fixed values of these constants which hold during system runtime.
When cyber-physical components exchange matter, physical entities flow between the geometric parts of the system. These entities are *made of* such materials. A major concern in material engineering is to find or create materials such that designated material constants have optimal values concerning an engineering objective. In this field, materials are the "systems" that define functions. From a systems engineering point of view, these functions result from the requirements imposed by a technical solution to implement a higher-level system function. This dissertation focuses on modeling and developing these functions to enable the development of a conceptual solution (a principle solution) which includes a notion of geometry and allows deriving the materialistic requirements.
Due to the equivalence of mass and energy expressed through the famous equation $E = mc^2$, exchanging matter is inherently bound to an exchange of energy [PBFG07], which does not necessarily hold vice versa. For example, whenever a system changes its volume it transmits or receives matter and the system's pressure changes its value. The latter indicates that the system also receives or transmits an amount of so-called pressure energy determined by the product of pressure and volume [FR76]. In this case, the volume specifies this form of energy, and the pressure specifies the amount of transmitted energy [FR76]. Specifying a flow of matter often requires specifying the type of transmitted energy as well as the material that is transmitted via the respective physical quantities. The type of the exchanged matter determines the characteristics that are relevant for the processing of the material by the system's function.

Modeling Material Since mass and volume are inherent properties of materialistic entities, we define mappings that represent these real-world properties:

- $\text{mass} : \mathcal{U} \rightarrow \text{Mass}$, which returns the mass of a material as an element of the physical quantity mass represented by real numbers with the SI-unit kilogram , *i.e.,* $\text{Mass} \stackrel{\text{def}}{=} \text{kg}$, and
- $\text{vol} : \mathcal{U} \rightarrow \text{Volume}$ which returns the volume of a material as an element of the physical quantity volume represented by the real numbers with the SI-unit cubic meter, *i.e.,* $\text{Volume} \stackrel{\text{def}}{=} \text{m}^3$.

Matter is characterized by physical quantities that define a material through material constants, the geometric shape of a discrete piece of matter, or characterize a fluid material. For specifying the functions of a CPS that processes material, it makes sense to let the modeler choose the relevant physical quantities. In some cases, the model needs to reflect that the exchange of matter is bound to an exchange of energy. For example when modeling the transfer of heat away from the engine of a vehicle through

a cooling medium, as in the running example (*cf.* Section 1.3.3). The modeler can express that, for example by adding an attribute to the class defining the material that has an energy type that models the respective form of energy. In the running example a material type with an attribute of a type that models the energy form heat to indicate that coolants carry heat.

Definition 2.5 (Material)**.** *A material Mat is a type that is represented by a class $c \in \mathcal{C}^U$ such that for all $a \in$ attributes(c) it holds that* type$(a) \in \mathbb{D}^n[SI] \times \mathcal{E}$*. The set of all material types is denoted $\mathcal{M}at$.*

Since the exchange of material is inherently bound to an exchange of energy, the power of a material message is defined as the sum of the powers of its energy-attributes, *i.e.,* $\text{pow}(m) = \sum_{e \in \text{attributes}(m):\text{type } e \in \mathcal{E}} m.e > 0\text{W}$ for all $m \in \mathcal{U}$ with $\text{type}(m) \in \mathcal{M}at$.

Example 2.2. *Automotive cooling systems utilize a coolant, i.e., a flowing fluid, to absorb heat, e.g., from the engine's components, and transport it to a radiator which cools down the coolant while releasing the heat into the surrounding air. A pump controls the volume flow rate of the coolant, i.e., it applies energy in the form of compression to the coolant. In this process, energy is exchanged between the pump and the fluid in the form of compression energy which is determined by the physical quantities pressure with SI-unit* Pa *and angular velocity with the SI-unit* $\text{rad}\,\text{s}^{-1}$*. Table 2.3 shows a CPCD that specifies the coolant and the energy type compression.*

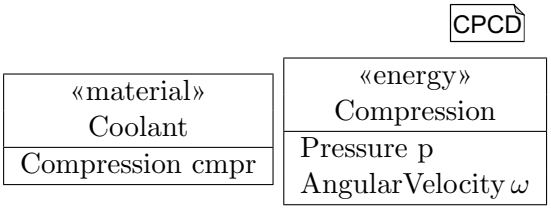


Table 2.3: Specification of the types that represent a coolant material and compression energy (see [FR76] for a definition of compression energy).

2.2.4 Data

Besides energy and material, CPSs also process information, *i.e.,* data. A characteristic of software systems is that they solely process information in the form of discrete data messages [Bro10].
When two systems exchange data, they exchange discrete messages [BS01]. Such systems are most likely software systems. Electric or mechanical systems, on the other hand, process materials and energy which appear discrete in the case of items,

continuous in the case of fluids, gases, or energy, and piecewise continuous in the case of signals.
The exchange and modeling of discrete data are well understood in the domain of software engineering. Data types are those that contain information, in the cyber-physical context, this information is often about something from the physical world, *e.g.,* a measurement. To this effect, data types are composed of physical quantities and primitive types. Data types must not have attributes whose type is an energy or material type. Further, data types contain a special message $\xi \in T$ that represents the absence of a message.

Definition 2.6 (Data Type)**.** *A data type is represented by a class* $c \in \mathcal{C}^U$ *such that for all* $a \in \text{attributes}(c)$ *it holds that* $\text{type}(a) \in \mathbb{D} \cup \mathbb{D}^n[SI]$*. The set of all data types is denoted by* $\mathcal{D}$*. All data types include the empty message, i.e.,* $\xi \in D$ *for all* $D \in \mathcal{D}$*.*

Since data messages are solely information, they do not exhibit physical properties, thus, it holds that for all $T \in \mathcal{D}$ and all messages $d \in T$ that $\text{pow}(d) = 0\text{W}$, $\text{vol}(d) = 0\text{m}^3$, and $\text{mass}(d) = 0\text{kg}$.
Note, that the meaning of a data message differs from that of other entities: The data message carries information about a phenomenon from the real world while physical quantities, energy, or material entities represent the actual phenomenon. This will make a significant difference when modeling cyber-physical types and their relations which is discussed in Chapter 6.

Example 2.3 (Audio Data)**.** *Audio data encodes the course of a voltage that represents information from the recording. Here, we model audio data as a class with stereotype «data» that holds two attributes, i.e.,* $\text{attributes}(\text{Audio}) = \{voltage, id\}$*, with types* $\text{type}(voltage) = \text{Voltage} \stackrel{\text{def}}{=} \mathbb{R}(\text{V})$ *and* $\text{type}(id) = \mathbb{N}$*. Table 2.4 shows this class.*

«data» Audio
Voltage voltage
$\mathbb{N}$ id

Table 2.4: Specification a type that represents audio data.

Events Mathematically, we consider an event as a logical observation on a timed stream (a timed stream is a function $s : TD \to \mathcal{U}$, where $TD \subseteq \mathbb{R}_+$ is a time domain. See Section 3.1):

Definition 2.7 (Event [Bro12]). *An event is a predicate* $e : TS \times \mathbb{R}_+ \to \mathbb{B}$*, where* TS *is the set of all timed streams. We say that the event* e occurs *in the stream* $s \in TS$ *at time* $t \in \mathbb{R}_+$ *iff* $e(s,t)$ *holds.*

An event models an observation within a timed stream and conveys information about the order and/or time slice in which the observation has occurred. Definition 2.7 defines the interpretation of event messages in the functional specifications considered in this dissertation.
In this dissertation, we specify the extraction of event information from a dense or discrete stream by a CPF which is detailed in Section 3.4.3. The CPFs that define the occurrence of an event in another stream can be interpreted as *sensors*.

Modeling Events In the functional specification of a CPS, in particular at an early stage during the development, it may be necessary to abstract from the stream on which the event occurs, and simply define that a system reacts on a certain event. Consider for example a system that offers the user the functionality to switch it on or off. At a very early stage of development it may not be relevant or known how this functionality is implemented and therefore the kind of stream that is input to the function that senses the event is not known, yet. To this effect, we consider event types to solely give an event a name, and event streams as streams of event messages. Let $\mathcal{E}v$ be the set of all events.

Definition 2.8 (Event Type). *An event type is a set of events* $E \subseteq \mathcal{E}v$*. For all events* e *it holds that* $\mathrm{pow}(e) = 0\mathrm{W}, \mathrm{vol}(e) = 0\mathrm{m}^3$*, and* $\mathrm{mass}(e) = 0\mathrm{kg}$.

Enums solicit for modeling event types, where the literals represent names of events. Table 2.5 provides an example for an event type definition.

Table 2.5: Definition of an event type that represents a button press.

This integrates the principle of underspecification for modeling events: Modelers can define events as types without referencing the stream in which the events modeled by the type occur. At a later stage in the development when the kind and type of this stream in which the events occur is known, the modeler can create a CPF that models a "sensor" to sense the occurrence of the event. The Example 3.7 of an electrical switch

that generates a stream of electrical energy upon a button press event in Section 3.4 illustrates this. The function defines the button press event to occur in a stream of force iff the incoming force is large enough.

2.3 Summary: Cyber-Physical Types

In the cyber-physical context, we need to consider the "physical" properties of entities defined by the following mappings on $\mathcal{U}$:

- $\text{pow} : \mathcal{U} \to$ Power which indicates the amount of energy a message transports per unit of time
- $\text{mass} : \mathcal{U} \to$ Mass which indicates the mass of a message
- $\text{vol} : \mathcal{U} \to$ Volume which indicates the volume a message takes up in space.

With these properties, we can distinguish the kind of type of an entity: In the following table, let T be a type and $m \in T$.

Type	**Form**	**Position**	**Power**	**Mass**	**Volume**
Energy	$\mathbb{D}^n[SI]^2$	$\text{pos}(m) \neq \xi$	$\text{pow}(m) > 0\text{W}$	$\text{mass}(m) = 0\text{kg}$	$\text{vol}(m) = 0\text{m}^3$
Material	$\mathbb{D}^n[SI]^n \times \mathcal{E}$	$\text{pos}(m) \neq \xi$	$\text{pow}(m) \geq 0\text{W}$	$\text{mass}(m) \neq 0\text{kg}$	$\text{vol}(m) \neq 0\text{m}^3$
Data	$\mathcal{P} \times \mathcal{M}at \times \mathcal{E}$	$\text{pos}(m) = \xi$	$\text{pow}(m) = 0\text{W}$	$\text{mass}(m) = 0\text{kg}$	$\text{vol}(m) = 0\text{m}^3$

Table 2.6: Classification of the different kinds of types.

A type of energy is defined by a tuple of two physical quantities, one is the extensive and one the intensive quantity. Their product is the physical quantity of energy. Thus, the power transmitted by an entity of energy is greater than zero, while the mass and volume of such an entity are zero. For material, the type consists of attributes that hold an energy type plus additional attributes that represent material constants, *i.e.,* physical quantities that define fixed characteristics of the material. Data entities convey information about things from the real world. Therefore, a data type may contain attributes whose type is primitive (including physical quantities but also the general primitive number types and Booleans). The position of a data message is the empty set, as it represents a piece of information. For the same reason, the power, mass, and volume are all equal to zero.

Chapter 3

A Theory of Cyber-Physical Functions

This chapter addresses the research question RQ2 "How can the theory of TSPFs [Bro10, Bro12] describe the functions of CPSs?". The FOCUS theory provides a formalism to represent software or embedded systems as message-exchanging components. Therein, a component is represented by an interface that consists of a set of input channels, a set of output channels, and a behavior. The latter is a set of so called Timed Stream Processing Functions (TSPFs) which are higher-order functions that map histories of the input channels to histories of the output channels. For each channel, a history describes the incoming messages (or, in our context, entities) over time on that channel.
Formalizing the idea of a CPF illustrated in Figure 1.2, we here, consider a functional specification to model a CPF through sets of typed input and typed output channels together with the system's behavior. The previous Chapter 2 has introduced the notion and modeling technique for cyber-physical types that are used here to provide types for the interface of a CPF. Timed streams represent the exchange of entities via the channels. Channel histories map timed streams to each channel of the system's interface and thereby represent the interaction history of that component. Section 3.4 elaborates that the type of each channel, in a functional specification, includes an identifier to define the kind of stream that represents the exchange of messages of that type over time. Sets of TSPFs is the most expressive way to specify functional behavior [RR11]. Section 3.4 introduces interface assertions, hybrid automata or architectural specification to specify the behavior of a CPF as a set of TSPFs.

3.1 Timed Streams

Timed Streams describe the communication histories of channels, *i.e.,* communication links within a system (*cf.* next paragraph), the flow of values assumed by a variable of a system, or a sequence of actions executed [Bro01].
From now on, let $\mathcal{U}$ be a universe of entities, that includes a pseudo entity $\xi \in \mathcal{U}$, that represents the empty entity, or the absence of any entities. Some stream specifications also use a time tick, denoted $\surd$ which is a pseudo entity to include timing information in a discrete stream. A timed stream is a function $s : TD_s \to \mathcal{U}$ on a time domain

$TD_s \subseteq \mathbb{R}_+$. We consider the interval of a timed stream s to be the smallest interval that contains the time domain of s, *i.e.,*

$$\text{interval}(s) = [\inf(TD_s), \sup(TD_s)[, \tag{3.1}$$
$$\tag{3.2}$$

where $\inf(TD_s)$ exists because $TD_s \subseteq \mathbb{R}_+$ which is bounded and we consider $\sup(TD_s) \in \mathbb{R}_+ \cup \{\infty\}$.
We distinguish between different kinds of discrete and dense streams [Bro01, RR11], where a stream is called *discrete* iff its time domain is discrete and *dense* iff its time domain is dense. The kinds of streams are listed in Table 3.1. The kind of a stream s is denoted $\text{kind}(s)$. A stream s is called *continuous* iff its time domain is an interval in $\mathbb{R}_+$, and $\text{img}(s)$ is a metric space (*cf.* Definition B.5) such that s is Cauchy continuous (*cf.* Definition 1.2) on the set TD_s [Bro12]. We use streams to represent the exchange of energy, matter, and data over time. In this dissertation, we model CPSs as sets of functions that process discrete and continuous timed streams at the same time. In this setting, the idea is that for a discrete timed stream $s : \mathbb{N} \to \mathcal{U}^*$, the sequence $s(t)$ for $t \in \mathbb{N}$ denotes the sequence of entities communicated during the time interval $[(t-1)\delta, t\delta[$, where $\delta \in \mathbb{R}_+$ is a time granularity [Bro12]. The prefix of s until time t is denoted $s \downarrow t$, given by $s|_{TD_s \cap [0,t[}$, which is the restriction of s to the interval $[0, t[$ [Bro12]. The set TS denotes the set of all timed streams, and $TS|_{[0,t[}$ the set of all prefixes of timed streams until time t. Table 3.1 provides a summary of the stream categories defined and discussed in [RR11].

Topology	**Kind of Stream**	**Basic Form**
discrete	event stream	$\mathcal{U}^\omega$
discrete	time-synchronous stream	$\mathbb{N} \to \mathcal{U}$
discrete	timed event stream	$\mathcal{U}^\omega \cup \{\surd\}$
discrete	time slice stream	$\mathbb{N} \to \mathcal{U}^\star$
dense	hybrid streams	$\mathbb{R}_+ \to \mathcal{U}$
dense	signal set streams	$\mathbb{R}_+ \to \wp(\mathcal{U})$
super dense	super dense streams	$\mathbb{R}_+ \to \mathcal{U}^\star$

Table 3.1: A classification of timed streams according to [RR11].

In this dissertation, we will use time-slice streams, timed event streams and hybrid streams to model the flows of energy matter and data. Here, note that timed event streams are only well-defined iff infinite streams contain infinitely many ticks. This condition implies that, between two time ticks, there may appear only finitely many entities.

3.1.1 A Complete Partial Order of Timed Streams

A distinguishing feature of the Focus theory is the compositionality of functional properties. For the methodology, we are proposing in Chapter 4 the compositionality of refinement is particularly important. In the context of CPS engineering it is crucial that these properties are compositional in the sense that refining a component of a functional specification yields a refinement of the composed function. This supports the "divide and conquer" principle which in the engineering process leads to the decomposition of complex tasks to cope with the complexity. To support this principle efficiently, it is inevitable that each of the resulting sub-tasks must be developed individually. Development means enriching functional specifications with newly available or adapted information, refining a sub-function must yield a refinement of the overall function. To prove that the composition of two or more CPFs yields again a CPF even in the case of feedback composition and also that refining a component of a decomposed CPF yields a refinement of the overall CPF, we need to set up the semantic domain of our models such that these properties are met.
To this effect, we utilize Scott's domain theory [SG90, Win93, SRS99]. For this, Appendix C provides the mathematical foundation. Defining a relation on timed streams that forms a CPO enables us to transfer the notions of monotonicity, continuity, least upper bounds, and least fix points to CPFs. Thereby, we can apply the results about fix points by Knaster-Tarski [Tar55] and Kleene [Kle52] to CPFs which implies the well-formedness of composition and the compositionality of refinement.
To do so, we will transfer the prefix-relation of discrete channel histories, *e.g.,* defined in [Rum96] to our setting where we consider both discrete and dense streams.
This requires a notion of time shift, as the time domains of a discrete and a dense stream are not "aligned" because the natural numbers in the time domains of discrete streams refer to the number of time slices that can happen at any point in the real-time ($\mathbb{R}_+$). This notion is already introduced in [Bro12].

Definition 3.1 (Time Shift [Bro12])**.** *Let $s : TD_s \to \mathcal{U}^\omega$ be a timed stream, and let $u \in \mathbb{R}_+$ be a specific point in time. The time shift of s by u is denoted $s^{TM}u$, its interval and time domain are given by*

$$\begin{aligned} &\text{interval}(s) = [t, t'[\Rightarrow \text{interval}(s^{TM}u) = [t+u, t'+u[\\ &TD_{s^{TM}u} = \{t+u \mid t \in TD_s\}. \end{aligned}$$

Now, let $t \in TD_s$ (this implies that $t + u \in TD_{s^{TM}u}$), then

$$(s^{TM}u)(t+u) = s(t).$$

The idea of the prefix relation for discrete streams used, *e.g.,* in [Rum96] is that a stream is a prefix of another stream if there exists a third stream that when

concatenated with the first, yields the second stream. The notion of concatenation is introduced in [Bro12] for the general notion of timed streams that we use here.

Definition 3.2 (Concatenation [Bro12])**.** *Let* $s : TD_s \to \mathcal{U}^\omega$ *with* interval$(s) = [t, t'[$ *and* $s' : TD_{s'} \to \mathcal{U}^\omega$ *be timed streams. We assume that* $t' < \infty$ *because otherwise* $s^\frown s' = s$*. Then,*

$$\text{interval}(s'^{TM}t') = [t''', t''[\Rightarrow \text{interval}(s^\frown s') = [t, t''[$$
$$TD_{s^\frown s'} = TD_s \cup TD_{s'^{TM}t'}.$$

The concatenation is defined as follows: Let $t^* \in TD_{s^\frown s'}$*, then*

$$t^* \in TD_s \Rightarrow s^\frown s'(t^*) = s(t^*)$$
$$t^* \in TD_{s'^{TM}t'} \Rightarrow s^\frown s'(t^*) = (s'^{TM}t')(t^*).$$

The above Definition 3.2 provides the basis to translate the prefix-relation from discrete Focus, found *e.g.,* in [Rum96] to our setting:

Definition 3.3 (Order of Timed Streams)**.** *Let* $s : TD_s \to \mathcal{U}^\omega$ *and* $u : TD_u \to \mathcal{U}^\omega$ *be timed streams. Then,* $s \sqsubseteq u$ *iff there exists a timed stream* $v : TD_v \to \mathcal{U}^\omega$ *such that* $s^\frown v = u$*.*

The above order is a slight adjustment from the traditional order on discrete streams [Bro97, BDD+93, Rum96] for our setting which allows a timed stream to be discrete or dense. The set of timed streams TS together with the relation defined in the Definition 3.3 defines indeed a CPO (see Appendix C for the respective definitions).

Proof. We have to show that $\sqsubset$ is reflexive, antisymmetric and transitive.
Reflexive: Let $\langle\rangle : \emptyset \to \mathcal{U}^\omega$. Then, $s^\frown\langle\rangle = s$ which implies that $s \sqsubseteq s$.
Antisymmetric: Let s and u be timed streams, respectively, such that $s \sqsubseteq u$ and $u \sqsubseteq s$. We know that because $s \sqsubseteq u$ there exists a timed stream v such that $s^\frown v = u$ and because $u \sqsubseteq s$ there exists another timed stream v' such that $u^\frown v' = s$. Now assume that interval$(s) = [t, t'[$ and interval$(u) = [t'', t'''[$. Then,

$$TD_u = TD_{s^\frown v} \tag{3.3}$$
$$= TD_s \cup TD_{v^{TM}t'} = TD_u \tag{3.4}$$
$$= TD_u \cup TD_{v'^{TM}t''} \cup TD_{v^{TM}t'} \tag{3.5}$$

Therefore, $TD_{v'^{TM}t''} \subseteq TD_u$ and $TD_{v^{TM}t'} \subseteq TD_u$. By Definition 3.2 it follows that

$$(s^\frown v)(t) = s(t) = u(t).$$

for all $t \in TD_{s^\frown v}$.

Transitive: Let s, u, v be timed streams such that $s \sqsubseteq u$ and $u \sqsubseteq v$. We know that there exists timed streams v', v'' such that $s^\frown v' = u$ and $u^\frown v'' = v$. Therefore, $(s^\frown v')^\frown v'' = v$, which implies $s^\frown(v^\frown v'') = v$. Thus, the stream s is also a prefix of the stream v.
The order is complete considering that the empty stream $\langle\rangle$ with $TD_{\langle\rangle} = \text{interval}(\langle\rangle) = \emptyset$ is the least element with respect to $\sqsubseteq$.
Further, there exists a least upper bound for each chain in the set of timed streams because the interval of each timed stream is open on its right end. □

The order relation $\sqsubseteq$ from Definition 3.3 can be extended to channel histories by pointwise application: Let C be a set of channels. Let $x, y \in \vec{C}$ be two channel histories. Then, it holds that $x \sqsubseteq y$ iff for all $c \in C$ it holds that $x(c) \sqsubseteq y(c)$. This makes the domain of channel histories together with the extended prefix-order also a CPO.

3.2 Cyber-Physical Streams

Timed streams are the mathematical foundation that we use to model the exchange of entities of energy, matter, and data. This section introduces the notions of streams of energy, matter, and data and defines their distinguishing properties formally.

3.2.1 Energy streams

An energy stream represents the exchange of energy among a CPS or one of its components and its environment. Literature about mathematical models of streaming energy such as [FR76] in principle already uses timed streams to model the flow of energy: A stream of energy describes the amount of energy that is transferred within a fixed interval of time [FR76]. Therefore, the unit of a stream of energy is energy per time, which is Watt and the respective physical quantity is power [FR76]. The amount is determined by observing the change in the intensive and the extensive physical quantity that define the form in which the energy is exchanged (*cf.* Section 2.2.2). Abrupt changes in the values of these physical quantities are, in the real world, not possible. Observing these physical quantities over time, therefore, yields a smooth curve. Signals or data that carry information about measurements of these values may have jumps or even be discrete. A stream of energy is modeled by a dense stream (*cf.* Section 3.1, Table 3.1) whose image (*cf.* Definition 1.1) is an energy type. To accurately represent the exchange of energy, a stream of energy is smooth (*cf.* Definition 1.3) everywhere because it takes (possibly very little) time for a physical quantity to change its value.

Definition 3.4 (Energy Stream)**.** *An energy stream is a total function $\mathbb{R}_+ \to \mathcal{E}$ ($\mathcal{E}$ denotes the set of all energy types as declared in Section 2.2.2) that is smooth everywhere, i.e., an energy stream is a hybrid stream that is smooth everywhere.*

The smoothness of this kind of stream is well-defined because every physical quantity is a metric space as is detailed in Appendix D. Energy types do not contain a special message that denotes the absence of any messages, because when a system does not receive or transmit energy, it simply receives or transmits zero Watts, *i.e.,* one of the two physical quantities that define the energy type then carry a value that is zero. That is, we model the absence of energy received or transmitted on a channel $c \in C$ with $\text{type}(c) = P \times Q$, during a period of time $[t, t'[$ such that for all channel histories $x \in \vec{C}$ it holds that the measured value of one of the characterizing quantities is zero during $[t, t'[$, *i.e.,* $(x(c)_P \downarrow [t, t'[) \cdot (x(c)_Q \downarrow [t, t'[) = 0$.
Let the streams $s_P : \mathbb{R}_+ \to P$, and $s_Q : \mathbb{R}_+ \to Q$ denote the restriction of the image of s to P, and Q, respectively. For the power transmitted by a timed energy stream $s \in [\mathbb{R}_+ \to E]$ it then holds that

$$\text{pow}(s) = s_P \cdot s_Q \in [\mathbb{R}_+ \to \text{Power}]$$

where $s_P \cdot s_Q : \mathbb{R}_+ \to \mathbb{R}(\text{W}), t \mapsto s_P(t) \cdot s_Q(t)$ denotes the timed stream that carries the power messages.
In case either P, or Q is a vectorial quantity, while the other is scalar, the multiplication $\cdot$ becomes the vector space multiplication and in case P and Q are vectorial quantities, the multiplication-operator $\cdot$ denotes the scalar product in $\mathbb{R}^n$ (*cf.* Appendix B).

3.2.2 Item Streams

Streams of matter represent the exchange of matter within a system and are modeled by timed streams whose image is a material type. Streams of matter include periods of time, where the material is absent or present. From a modeling perspective, the characteristics that determine this flow depend on whether the matter appears solid and is exchanged as piecewise items – imagine, for example screws that a conveyor belt transports to a machine tool which uses them to connect parts – or fluid that flows continuously.
Therein, a flow of items is conceptually very similar to a flow of data messages, *i.e.,* discrete. A material is absent at time $t \in \mathbb{R}_+$ in a material stream iff $\text{mass}(s(t)) = 0\text{kg}$ or $\text{vol}(s(t)) = 0\text{m}^3$. Fluid material behaves significantly different than solid, piecewise material that needs to be regarded in functional specifications. As a material can appear both as fluid or item, the modeler specifies the kind of stream in the CPF specification. The CPCD notation only offers a stereotype «material».
Items flow discretely. In that, item streams are very similar to the data streams handled by software systems. That is, there are *points* in real time, where an item is present, and *periods* in real time, where the item is absent. Discrete streams (*cf.* Table 3.1), therefore, represent item streams accurately. Here, we utilize time slice streams, which are isomorphic to time-synchronous streams [RR11] to represent the

exchange of items. For models of systems that process only items and/or data, for example, the discrete Focus theory that was established to support the engineering of software systems is sufficient.

Definition 3.5 (Item Stream)**.** *An item stream is a total function* $\mathbb{N} \to \mathcal{M}at^*$ *(*$\mathcal{M}at$ *denotes the set of all materials as declared in Section 2.2.3), i.e., an item stream is a time slice stream.*

Here, the pseudo entity $\{\xi\} \in \mathcal{U}$ represents the absence of items at a certain time in the stream. Note, that the set of time-synchronous streams $[\mathbb{N} \to \mathcal{U}]$ is isomorphic to the set of time-synchronous streams [RR11], *i.e.,* both are valid to use for item streams. When specifying streams of matter, deciding which kind of stream is appropriate may not always be obvious. Sand, for example, consists of many small rocks. The kind of matter stream, however, is set by the modeler in a functional specification. In the type definition, the stereotype «material» is sufficient.

3.2.3 Fluid Streams

Fluids flow continuously, meaning that the amount of fluid changes smoothly. These streams are, therefore, modeled by hybrid streams of the form $s : TD_s \to \mathcal{M}at$ such that the hybrid streams $\text{vol}(s) : TD_s \to \text{Volume}$ and $\text{mass}(s) : TD_s \to \text{Mass}$, resoectively are smooth everywhere. The smootheness implies that these streams are infinitely often differentiable (Definition 1.3), and, thus, enable to derive the mass or volume flow as $d\,\text{mass}(s)/dt : TD_s \to \mathbb{R}(\text{kg}\,\text{s}^{-1})$ or $d\,\text{vol}(s)/dt : TD_s \to \mathbb{R}(\text{m}^3\,\text{s}^{-1})$, respectively. Similar to a stream of energy, a stream of fluid matter thereby represents the amount of matter that passes through (an interface of a function) per unit of time. Note, that the or is non-exclusive and that smoothness is well-defined for these streams because every physical quantity is a metric space (see Appendix D). The attributes that define the characteristics of fluid matter streams may not change smoothly. Whenever the fluid ceases to be present at that channel, the mass or vol functions change to 0.

Definition 3.6 (Fluid Stream)**.** *A fluid stream is a hybrid stream* $s : \mathbb{R}_+ \to M$*, where* $M \in \mathcal{M}at$ *is a material such that at least one of the two streams*

$$\text{vol}(s) : \mathbb{R}_+ \to \text{Volume} \text{ or} \tag{3.6}$$

$$\text{mass}(s) : \mathbb{R}_+ \to \text{Mass} \tag{3.7}$$

$$\tag{3.8}$$

is smooth everywhere.

Note that the two streams vol and mass provide information about a material stream and can be considered as signals that are introduced in the next section.

To model the exchange of materials between the functions of a CPS, it is sufficient to differentiate between solid, piecewise materials and fluids because the flow of gasses is conceptually equivalent to the flow of fluids which is also how it is modeled on literature from this domain [FR76].

3.2.4 Data Streams

A data stream represents the exchange of information among CPSs or CPS components. Similar to matter, the exchange of information depends on how it is represented within the system.
Software systems communicate by exchanging discrete data messages. Therefore, we represent data streams by discrete time slice streams.

Definition 3.7 (Data Stream)**.** *A data stream is a total function* $\mathbb{N} \to \mathcal{U}^\star$*, i.e., a data stream is a time slice stream.*

Note, that the definition of a data stream works also in case of the absence of messages, because the universe of entities $\mathcal{U}$ includes an entity ξ which denotes the empty entity. The Focus theory has been initiated and designed to formalize the characteristics of these systems in mathematical terms which provides a semantic domain for models of software systems. The theory provides ways to distinguish among many different kinds of discrete streams. However, we assume that Definition 3.7 is expressive enough for modeling the processing of discrete data messages by CPSs, as the set of time slice streams $[\mathbb{N} \to \mathcal{U}^\star]$ is isomorphic to the set of time-synchronous streams $[\mathbb{N} \to \mathcal{U}]$ [RR11], *i.e.,* both are valid to use for data streams. Technically, data streams also include the untimed event streams $\mathcal{U}^\omega$ because the set of time-synchronous streams $[\mathbb{N} \to \mathcal{U}]$ and the set of untimed event streams are also isomorphic [RR11]. Also, timed event streams of infinite messages $\mathcal{U}^\infty \cup \{\surd\}$ that use the $\surd$ to separate time slices are technically included because this subset is isomorphic to the subset of infinite timed event streams [RR11]. Both represent the exchange of messages in a discrete way, but their interpretation differs [RR11]. In analogy to the stream kinds that represent the exchange of energy and matter, data streams represent the exchange of data messages.

3.2.5 Signals

In the cyber-physical context, data is not always discrete. *Signals* represent continuous data processed, *e.g.,* by controllers or sensors. In a signal, this the relevant information is represented by physical quantities, whose values convey information, and provide a technical implementation for exchanging data [BG21]. Here, we consider signals to be piecewise continuous functions that represent the measurement of the value of the physical quantity that carries the desired information. Technically, signal streams are often implemented through streams of energy.

Definition 3.8 (Signal Stream)**.** *A signal is a total function $\mathbb{R}_+ \to \mathcal{U}$ that is piecewise smooth, i.e., a signal is a hybrid stream that is piecewise smooth.*

3.2.6 Event Streams

Events represent discrete information that trigger system behavior. For example, an event may be a spike in the value of a physical quantity upon which the system initiates an action, or that causes a change in the aggregate phase of a material. An *event stream* is a sequence of event messages, where each message marks the occurrence of the event.

Definition 3.9 (Event Stream)**.** *An event stream is a finite or infinite sequence of messages, i.e., an event-stream is an element of the set $\mathcal{U}^\omega \cup \{\surd\}$.*

Event streams abstract from timing information [RR11], *i.e.,* an event stream does not convey any information on the time of occurrence of an event within a time slice. It solely conveys information on the order of the occurrence of the events within a time slice. Timed event streams include the time-tick $\surd$ which denotes the beginning or end of a time slice. The length of the time slice, however, is left open [RR11].

The timed streams considered in this dissertation hold a kind that defines the properties of the flow of entities represented by the stream. From now on, we denote the set of available stream kinds by $\mathcal{K} = \{\text{energy}, \text{fluid}, \text{item}, \text{data}, \text{signal}, \text{event}\}$ and define a mapping $\text{kind} : TS \to \mathcal{K}$ that assigns each timed stream a kind.

A timed stream that does not fall into the categories of energy, fluid, item, data, signal, or event is not considered a valid representation of a stream of entities in this dissertation.

3.3 Timed Stream Processing Functions and Behavior

In functional development, a specification of the desired function of a CPS is the target of all development activities. CPFs are modeled as components with an interface together with a specification of the behavior of that function. The behavior describes mathematically how the underlying system shall transform the streams incoming through the input channels to the outgoing streams of the output channels. This section provides a formal definition of the term CPF together with a set of properties that these functions must fulfill to provide an accurate functional specification for a CPS. In the sequel, this section provides different techniques to specify the behavior of a CPF.

3.3.1 Channels and Histories

A *channel* is an identifier for an interaction link that is identified by a *variable*, *i.e.,* the channel's name. Channels define the interface of a CPF. Via a channel, a system or system component receives or transmits entities. Here, channels are typed, *i.e.,* there is a type assignment that defines the type of entities that enter or leave the system via the interaction link together with the type of the stream which defines the temporal behavior of the flow of these entities. Let S be a set of types. A type assignment for a set C of channels is a mapping $\text{type} : C \to S \times \mathcal{K}$, where $\mathcal{K} \subseteq =$ {event, time-synchronous, timed event, time slice, hybrid, signal set, super dense} (*cf.* Table 3.1) is the set that defines a set of stream kinds. A Channel history represents the interaction history of a channel.

Definition 3.10 (Channel History [Bro12])**.** *A* channel history *is a mapping* $x : C \to TS|_{[0,\infty[}$ *such that*

- $x(c) \in \left[TD_{x(c)} \to \text{type}(c)_1\right]$ *is a timed stream for all channels* $c \in C$ *[SRS99] and*
- $\text{kind}(x(c)) \in \text{type}(x(c))_2$.

where $\text{type}(x(c))_i$ *denotes the i-th element in the tuple* $\text{type}(x(c))$ *(*$i \in \{1, 2\}$*) A set of channels is called* typed *iff there is a type assignment defined for the set of channels.*

Similar to a channel history, a *channel snapshot* is a mapping $\hat{x} : C \to TS|_{[t,t'[}$ for times $t, t' \in \mathbb{R}_+$ such that $t < t'$. By $\widehat{C}$ we denote the set of all channel snapshots of the channels in C [Bro12].
Let C, C' be two distinct sets of channels. As in [SRS99], we define the addition of two channel histories $x \in \vec{C}, y \in \vec{C'}$ that coincide on $C \cap C'$, *i.e.,* $x|_{C'} = y|_C$, as $(x + y)(c) = x(c)$ if $c \in C$, and $(x + y)(c) = y(c)$ if $c \in C'$.

3.3.2 Behavior

Let I, O be sets of channels, respectively. A tuple $\mathcal{I} = (I, O)$ forms an *interface*, where the first entry represents the set of input channels and the second entry the set of output channels. The behavior of such a function is then specified by a set of valid functions that map the histories of the input channels to the histories of the output channels. A *TSPF* on the interface $\mathcal{I}$ is a function $f : \vec{I} \to \vec{O}$ that is continuous in the sense of Definition C.5 with respect to the prefix order on timed streams (Definition 3.3). These functions provide for each input history exactly one output history [SRS99]. For CPSs, it is not realistic that components produce an output before receiving the respective input. Causality is the property of TSPFs that models this.

Definition 3.11 (Causality [Bro12])**.** *A TSPF* $f : \vec{I} \to \vec{O}$ *is* causal *iff the output until time t is determined entirely by the input until time t, i.e.,* $x \downarrow t = y \downarrow t$ *implies that* $f(x) \downarrow t = f(y) \downarrow t$*, for all* $x, y \in \vec{I}$*, and all* $t \in \mathbb{R}_+$*.*

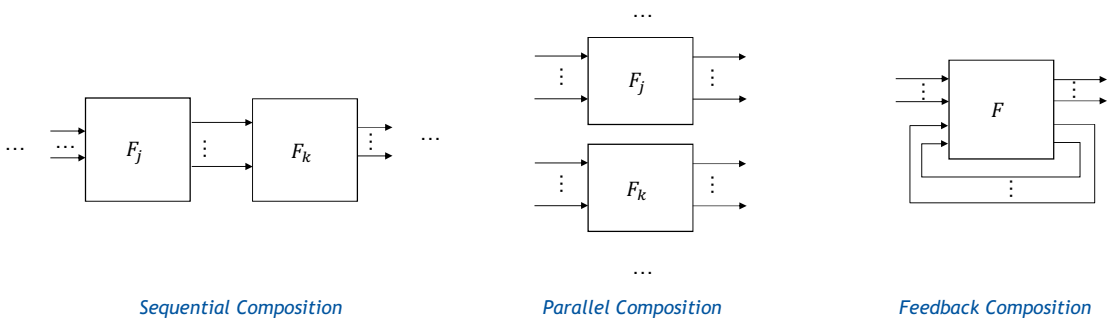


Figure 3.1: Different kinds of composition [SRS99].

Strong causality requires components to take some time to process an input. The function of a CPS processes discrete and dense streams at the same time. Mathematically, a system receives infinitely many inputs on a dense channel (*i.e.,* a channel of a dense kind) between two inputs received on a discrete channel. Therefore, we could say that the discrete channels "slow down" the transformations of the dense channels. Strong causality is the mathematical property that formalizes this.

Definition 3.12 (Strong Causality)**.** *A TSPF $f : \vec{I} \to \vec{O}$ is* strongly causal *iff it is delayed, i.e., iff there exists a delay $\delta > 0$ such that the output until time $t + \delta$ is completely determined by the input until time t for all $x, y \in \vec{I}$, all times $t \in \mathbb{R}_+$, i.e., $x \downarrow t = y \downarrow t$ implies that $f(x) \downarrow (t + \delta) = f(y) \downarrow (t + \delta)$ [Bro12].*

We denote the set of strongly causal TSPFs with delay δ by $\vec{I} \xrightarrow{\delta} \vec{O}$. For two delays $0 < \delta_2 \leq \delta_1$, it holds that $[\vec{I} \xrightarrow{\delta_1} \vec{O}] \subseteq [\vec{I} \xrightarrow{\delta_2} \vec{O}]$. As in the traditional Focus on discrete streams, it follows that strongly causal TSPFs are continuous in the sense of Definition C.5, and causality is the property that reflects continuity in our setting. Sets of strongly causal TSPFs specify the *behavior* of a CPF, *i.e.,* they define a valid set of transformations of input to output streams.

Definition 3.13 (Behavior)**.** *A* behavior *on an interface (I, O) is a set of TSPFs $B \subseteq [\vec{I} \to \vec{O}]$.*

A behavior $B \subseteq [\vec{I} \to \vec{O}]$ on an interface (I, O) is called *deterministic*, iff $\#B = 1$.

3.3.3 Composition

Composition plays an, if not the most important role in the Focus theory. To cope with the increasing complexity of CPSs we need to decompose the system into subsystems and be able to develop these subsystems independently [FR07]. Following the divide and conquer principle, pretty much all engineering domains have established

a notion of decomposing the system under development and following this decomposition when distributing the development tasks. In the physical world decomposing and reassembling a CPS in this way works because a system composed of subsystems is, again, a system. Also, we can observe CPSs in the real world that feed back their outputs as inputs (provided the outputs are of a matching type and kind). These systems are also CPSs. In our approach, we model CPSs in terms of the function, *i.e.,* the CPF, it defines. The theory we are proposing models these functions as an interface together with a set of TSPFs over channel histories of the interface which defines the function's behavior. To obtain a proper definition of composition, we need to prove that the composition of CPFs is again a CPF, and since we define a CPF as a set of TSPFs over discrete and hybrid streams, we need to show that feedback composition is well-defined for these functions.
In general, there are three forms of composition: (1) sequential composition, (2) parallel composition, and (3) feedback composition. Figure 3.1 illustrates these forms of composition, which are formally defined, *e.g.,* in [BDD+93] or [Rum96]:

Definition 3.14 (Sequential Composition of TSPFs)**.** *Let $f : \vec{I_f} \to \vec{O}$, and $g : \vec{O} \to \vec{O_g}$ be TSPFs. Then, the* sequential composition *of f and g is defined as*

$$(f \circ g) : \vec{I_f} \to \vec{O_g} : x \mapsto g(f(x))$$

The sequential composition of two (strongly) causal TSPFs is again (strongly) causal: Let f, g be the two TSPFs from Definition 3.14 and assume they are strongly causal. Because f is strongly causal, there exists $\delta_f > 0$ such that $x \downarrow t = y \downarrow t$ implies that $f(x) \downarrow t + \delta_f = f(y) \downarrow t + \delta_f$ for all $t \in \mathbb{R}_+$. Since g is also strongly causal, there exists also $\delta_g > 0$ such that this implies $g(f(x)) \downarrow t + \delta_f + \delta_g = g(f(y)) \downarrow t + \delta_f + \delta_g$ for all $t \in \mathbb{R}_+$.
The sequential composition of two behaviors is defined as the set of TSPFs that results from the element-wise application of the $\circ$-operator: Let $F : \vec{I_F} \to \vec{O}$ and $G : \vec{I} \to \vec{O_G}$ be two behaviors. Then,

$$F \circ G = \{f \circ g \mid f \in F, g \in G\} \subseteq \vec{I_F} \to \vec{O_G}$$

is a behavior. Because $\circ$ on TSPFs preserves strong causality, and since all elements of F and G are strongly causal, also $F \circ G$ is strongly causal.

Definition 3.15 (Parallel Composition of TSPFs)**.** *Let $f : \vec{I_f} \to \vec{O_f}$, and $g : \vec{I_g} \to \vec{O_g}$ be TSPFs such that the output channels are disjoint, i.e., $O_f \cap O_g = \emptyset$. In this case, the interfaces (I_f, O_f) and (I_g, O_g) are* compatible for parallel composition*. Further, let $x_f \in \vec{I_f}$ and $x_g \in \vec{I_g}$ be two input channel histories. Then, the* parallel composition *of f and g is defined as*

$$(f||g) : I_f \vec{\cup} I_g \to O_f \vec{\cup} O_G : x \mapsto \begin{cases} f(x), & x \in \vec{I_f} \\ g(x), & x \in \vec{I_g} \end{cases}$$

Parallel composition preserves strong causality if both components are strongly causal, which can be proven in a similar manner as the preservation of strong causality by sequential composition. Similar to sequential composition, parallel composition is extended to behaviors by the element-wise application of the composition operator: Let $F : \vec{I_F} \to \vec{O_F}$ and $G : \vec{I_G} \to \vec{O_G}$ be again two behaviors with interfaces that are compatible for parallel composition. Then the parallel composition of F and G is defined as the behavior

$$F||G = \{f||g \mid f \in F, g \in G\}.$$

It follows from the compositionality of strong causality that also, $F||G$ is strongly causal.

Definition 3.16 (Feedback Composition [Rum96])**.** *Let $f : \vec{I} \to \vec{O}$ be a TSPF that is strongly causal, and let $B \subseteq I \cap O \neq \emptyset$. Then,* feedback composition *on the channels in B, denoted $\mu_B f : (\overrightarrow{I \setminus B}) \to (\overrightarrow{O \setminus B})$ is the smallest function with respect to the order $\sqsubseteq$ on timed streams [Rum96] such that for all $i \in (\overrightarrow{I \setminus B})$ there exists a channel history $s \in \vec{B}$ with*

$$(\mu_B f)(i) + s = f(i + s).$$

Feedback composition preserves strong causality because $(i + s) \downarrow t = (i' + s') \downarrow t$ implies that $f(i + s) \downarrow t + \delta = f(i' + s') \downarrow t + \delta$ for all $i, i' \in \overrightarrow{I \setminus B}$ and $s, s' \in \vec{B}$.
In equivalence to sequential and parallel composition, feedback composition is extended to behaviors by element-wise application of the definition: Let $F : \vec{I_F} \to \vec{O_F}$ be a behavior such that there exists $B \subseteq I \cap O \neq \emptyset$. Then, the feedback composition $\mu_B F$ is defined as the smallest element (with respect to the order $\sqsubseteq$ on timed streams) of the feedback compositions of the elements in F, *i.e.,*

$$\mu_B F = \sqcup\{\mu_B f \mid f \in F\}.$$

Feedback composition, again, yields a behavior because feedback composition preserves strong causality.
While sequential and parallel composition do not require the involved TSPFs to be strongly causal, feedback composition is only well-defined for strongly causal TSPFs. Because strong causality implies that the TSPF is continuous with respect to Definition C.5, the fix point theorem of Knaster-Tarski [Tar55] implies the existence of the least fix point $\mu_B f$.

3.3.4 Refinement

Starting the development activities from greenfield or brownfield [DGM+21], the information about the system under development that is available in the development process will never be complete. Therefore, descriptions of the CPF under development

can also never be complete. As the set of requirements on the system evolves throughout the entire *life*-cycle of the system, as errors or failures are detected, it may not become complete at all [Rum16]. Thus, describing a CPF's behavior by a deterministic and fully specified relation that allows only one possible implementation is hard if not impossible. Further, in some cases, specifying the CPF's behavior as non-deterministic is even more accurate than a deterministic specification considering delays, or energetic fluctuations during system operation, the geometric degrees of freedom, or desired variability of the system. Thus, describing a CPF must allow explicating a lack of information, and non-determinism. A description of a CPF may include *underspecifications* which allows for a *range* of possible implementations. The theory above allows to specify the behavior of a CPF as a *set* of TSPFs which represents this underspecification: The set defines a solution space for the implementation of the behavior. Further, using sets enables one to regard non-determinism by including different kinds of behavior on the same input history. Capturing the system as an underspecified CPF facilitates inventing new engineering solutions to existing challenges and paves the way for innovation. In this dissertation, we consider an open-world assumption [DKMR19, NRSS22] regarding underspecification: Interpretations of the models shall consider whatever is not modeled to not be restricted. Section 4.2 defines a functional model-driven methodology for engineering CPSs that builds on the principle of iterative refinement. Compositionality and including controlled underspecification in the specifications are prerequisites for this approach. The methodology formalizes the idea that the development activities aim at lifting the underspecification in the models where information is missing. A new version of a model refines another, iff it contains less underspecification than the previous version. Throughout the development, the models that represent the current development state of the system are iteratively refined by adding the information that is obtained during the development to the models. Therein, refinement is a formal relation among two specifications that represents moving on to the next development step. In this dissertation we consider the development to proceed through iterative refinement, *i.e.,* each new version of a model contains more information than the previous version [DKMR19]. Further, we consider the two different types of refinement, *i.e.,* property refinement and interaction refinement, described in [Bro12].
The idea is that everything that was proven for a system that conforms to the previous model should hold for a consecutive version.

Definition 3.17 (Property Refinement [Bro12])**.** *Let $F \subseteq [\vec{I} \to \vec{O})$ be a behavior. The behavior $F' \subseteq [\vec{I} \to \vec{O}]$ is called a* property refinement *of F iff $F' \subseteq F$.*

Property refinement describes the notion of including new information in the specification of a CPF's behavior *e.g.,* when new information is available in the development process [DKMR19].

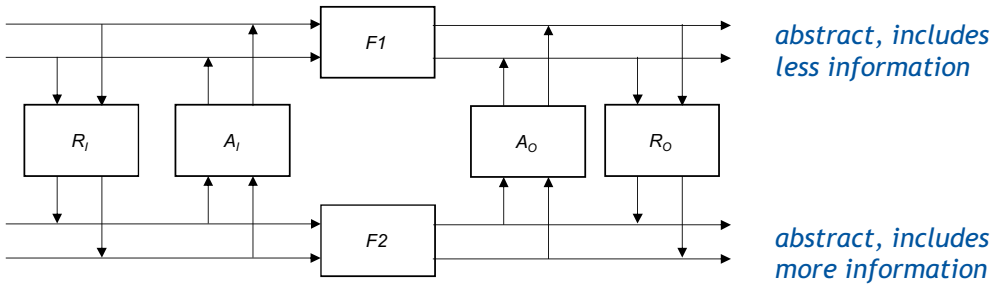


Figure 3.2: Illustration of interaction refinement, found similarly in [Bro12].

Property refinement is compositional [Bro12], meaning that given a refining behavior F' of a behavior F (*i.e.,* $F' \subseteq F$) and another behavior G, then it holds that $F' \circ G \subseteq F \circ G$, $F'||G \subseteq F||G$, and also $\mu_B F' \subseteq F$. This follows naturally from the definition of the composition operators on behaviors as the element-wise application of the composition on TSPFs.

Interaction refinement corresponds to the notion of including new information in the interface of a functional specification. It enables consistently (1) changing the names and the number of the input and the output channels of a component, and (2) changing the types of the channels in a component's interface [Bro12]. Let C, C' be two sets of channels. Interaction refinement is described by a pair of two functions $A \in [\vec{C'} \to \vec{C}]$ and $R \in [\vec{C} \to \vec{C'}]$. The function A specifies the abstraction, *i.e.,* defines a relation between the channels of the new, more concrete component, while the function R defines a relation between the channels of the abstract component to the concrete component. To accurately represent this, we require that $(R \circ A)(x) = x$ for all concrete channel histories $x \in \vec{C'}$. We will refer to the tuple (A, R) as a *refinement pair*.

Definition 3.18 (Interaction Refinement [Bro12])**.** *Let* $F1 \in [\vec{I_1} \to \vec{O_1}]$ *and* $F_2 \in [\vec{I_2} \to \vec{O_2}]$ *be two behaviors. We call* $F2$ *an* interaction refinement *of* F_1 *iff there exist two refinement pairs* (A_I, R_I) *and* (A_O, R_O)*, where* $A_I \in [\vec{I_2} \to \vec{I_1}]$*,* $R_I \in [\vec{I_1} \to \vec{I_2}]$*,* $A_O \in [\vec{O_2} \to \vec{O_1}]$*, and* $R_O \in [\vec{O_1} \to \vec{O_2}]$ *such that* $A_O \circ F_2 \subseteq F1 \circ A_I$*, i.e., the sequential composition* $A_O \circ F2$ *is a property refinement of* $F1 \circ A_I$*.*

Because property refinement is compositional, also interaction refinement is compositional. That is, assuming that $F1$ is a component either in a sequential, parallel or feedback composition, replacing $F1$ by $F2$ yields an interaction refinement of the composition which follows directly from the fact that property refinement is compositional.

CPF
name n (list of parameters P)
in list of typed input channels $I = [i_1, i_2, \ldots, i_n \mid n \in \mathbb{N}_{<\infty}]$ out list of typed output channels $O = [o_1, o_2, \ldots, o_m \mid m \in \mathbb{N}_{<\infty}]$
Behavior specification $B : \vec{I} \times \vec{O} \to \mathbb{B}$

Table 3.2: Specification scheme for CPFs based on [Bro10].

3.4 Specifying Cyber-Physical Functions

A specification formalizes functional system requirements and thereby provides an underspecified model of the desired system. During the development process, such specifications are iteratively refined to, finally, obtain a specification of how the system implements the desired functionality. This section elaborates on how to specify the behavior and interface of a CPF. Similar to [BS01], we utilize interface assertions, HIOSs, and architectural specifications to obtain a functional model of a CPS.
An abstract syntax for a functional specification defines a CPF as a tuple comprising an interface and a behavior.

Definition 3.19 (Cyber-Physical Function)**.** *A CPF is a tuple $F = [I, O, B]$, where*

- *(I, O) is an interface,*
- *$B \in [\vec{I} \to \vec{O}]$ is a behavior.*

For a concrete syntax we use the specification scheme is depicted in Table 3.2 which is adapted from [Bro10]. Here, we also allow parameters to enable flexible definitions of functional components. Such a scheme denotes a CPF specification that is not further decomposed, which is called an elementary specification in [BS01]. We use MontiArc's C&C-like notation introduced in Section 1.3.4 to specify a CPF as the composition of other CPFs. This is elaborated in Section 3.4.3. The CPF-tag in the upper right corner allows uniquely identifying the specification as a CPF specification. In both the tabular notation and the MontiArc notation, the channels must have a type and a kind indicated by the stereotype and the stereotype must fit the kind of the channel's type. This defines the type assignment for the input and output channels of the modeled function.

3.4.1 Specification by Interface Assertions

Interface assertions are formulae in predicate logic over the identifiers of the input and output channels [Bro12]. They provide one technique to specify the behavior of a CPF as a set of TSPFs: Being a logical formula, the interface assertion defines a predicate

$p : \vec{I} \times \vec{O} \to \mathbb{B}$. The predicate holds for a pair of input and output histories iff the pair fulfills the interface assertion. Thus, the predicate defines a behavior as the set of TSPFs that fulfill the predicate via $F = \{f \in [\vec{I} \to \vec{O}] \mid \forall i \in \vec{I} : \ p(i, f(i))\}$.
In this formalism, the behavioral specification of the tabular scheme in Table 3.2 is given by a formula in predicate logic over the set of channel names and possible internal variables. The formula defines a predicate on TSPFs. Those TSPFs that fulfill the predicate are in the semantics of the specification. The behavior of a CPF is, therefore, the *set* of TSPFs for which the predicate defined by the interface assertion holds.
The semantics of a CPF specification in the scheme of Table 3.2, denoted $[\![n]\!]$ is the set of behaviors that fulfill the predicate defined by the interface assertion, *i.e.,*

$$[\![n(P)]\!] \stackrel{\text{def}}{=} \{f \in [\vec{I} \to \vec{O}] \mid \forall x \in \vec{I} : \ B(x, f(x))\}$$

Interface assertions are a specification technique that is suited to specifying transformations of only dense streams, only discrete streams, or of hybrid transformations.

Specifying transformations of dense streams in interface assertions In mechanical engineering, it is common practice to specify the transformation of energy by the power balance stating that the outgoing power must not be greater than the incoming power which derives from the law of energy conservation. Example 3.1 and Example 3.2 give examples for the specification of two CPFs that transform energy and matter by an interface assertion.

Example 3.1 (Electric Drive)**.** *An electric drive converts electrical energy to rotational energy. Table 3.3 provides the specifications of the corresponding channel types and of the CPF defined by an electric drive which could be part of the functional specification of the corresponding mechanical system using the mechanical design methodology introduced in [Kol98] The specification uses the power balance: Abstracting from energetic losses, the outgoing power must be equal to the incoming power. Section 5.3 discusses the formulation of interface assertions, modeling energetic losses, and delay.*

Example 3.2 (Connect Fluid with Energy)**.** *Centrifugal pumps employ, e.g., a paddlewheel [HJZ+21] which rotates to bring fluid material into motion. That is, to apply motion energy to the fluid, which, in the case of the paddle wheel solution is provided as rotational energy. Table 3.4 specifies the material type water and a CPF that causes fluid water to flow. Again, we utilize the power balance to specify how the function applies the fluid with energy and abstract from potential energetic losses: The power carried by the outgoing fluid must be equal to the power received by the function carried by the input fluid and rotational energy. The variable ε again enables to specify a threshold for allowed energetic losses. The specified CPF generates an output stream of a fluid material kind that carries energy such that the power of that stream of energy*

CPCD

«energy» ElectricalEnergy	«energy» RotationalEnergy
Current current Voltage voltage	Torque_3 torque AngularVelocity_3 angularVelocity

CPF

ElectricDrive($\mathbb{R}_+ \delta$)

in «energy» ElectricalEnergy e out «energy» RotationalEnergy r
$\forall t \in \mathbb{R}_+ :$ $\text{pow}(e)(t) = \text{pow}(r)(t + \delta)$

Table 3.3: Specification of the CPF defined by an electric drive.

carried by the fluid is obtained by adding the power of the incoming stream of motion energy to the stream of energy carried by the incoming fluid. The ε allows energetic losses (see Section 5.3 for a discussion on energetic losses in CPF specifications).

Specifying hybrid transformations of dense and discrete streams Energy and fluid matter are represented by dense hybrid streams, *i.e.,* by the set $[\mathbb{R} \to \mathcal{U}]$. A characteristic of a CPS, however, is its hybrid nature because these systems process energy, matter, and data, *i.e.,* dense and discrete streams at the same time. CPFs that process, *e.g.,* energy and data, are hybrid components because they process energy and data streams which are represented by dense and discrete streams at the same time. The upcoming Section 3.4.2 details how to specify such CPFs using HIOSs. Interface assertions, however, can also be used to specify true hybrid CPFs. The example of a power amplifier which is based on an example from [Bro12], given in our notation in Example 3.3 illustrates this.

Example 3.3 (Power Amplifier [Bro12])**.** *The example of an amplifier is a specialized version of the example in [Bro12]. A power amplifier increases the power of an incoming energy stream by the last value provided on a data input channel. Because the kind of the input channel is data, the streams that represent histories of this channel are discrete. The specification in+Table 3.5 specifies this functionality. The parameter δ defines a time granularity. The interface assertion states that the power generated on the output port $p2$ in the interval $[t\delta, (t+1)\delta[$ is determined by the power received on the input port $p1$ during the time interval $[(t-1)\delta, t\delta[$ multiplied by the last actual value received on the data input port x. The specification is taken from [Bro12].*

CPCD

«energy» MotionEnergy
Velocity velocity Force force

«material» Water
MotionEnergy e

CPF

ApplyWaterWithRotationalEnergy($\mathbb{R}_+\delta$)

in «energy» RotationalEnergy m, «fluid» Water $w1$ out «fluid» Water $w2$
$\forall t \in \mathbb{R}_+ :$ $\text{pow}(w2.e)(t+\delta) = \text{pow}(w1.e)(t) + \text{pow}(m)(t)$

Table 3.4: Specification of a CPF that causes a fluid coolant to flow.

CPF

PowerAmplifier($\mathbb{R}_+\delta$)

in «energy» Power $p1$, «data» $\mathbb{R}_+$ x out «energy» Power $p2$
$\forall t \in \mathbb{R}_+ :$ $p2(t+\delta) = p1(t) \cdot lt(x,t)$ where $T(x,t) = \{n \in TD_x \mid n \leq t-\delta\} \neq \emptyset \Rightarrow lt(x,t) = x(\max(T(x,t)))$ $T(x,t) = \emptyset \Rightarrow lt(x,t) = 0$

Table 3.5: Specification of a CPF that causes a fluid coolant to flow.

Add CPF
in «data» $\mathbb{Z}$ $d1$, «data» $\mathbb{Z}$ $d2$ out «data» $\mathbb{Z}$ d
$\forall t \in \mathbb{N}$: $d(t) = d1(t-1) + d2(t-1)$

Table 3.6: Specification of a CPF that adds up two integers.

Specifying transformations of discrete timed streams Discrete timed streams, so far, are mainly considered in the Focus theory. The theory has been developed with discrete timed streams in mind as it emerged in the domain of software engineering to support the MDE of software and software intensive systems. These systems interact by exchanging discrete data messages, where the exchange of continuous entities as energy or fluid matter does not play a role. Literature on Focus provides various examples of such specifications, therefore, we provide a simple example of a component that adds two numbers.

Example 3.4 (Add)**.** *Consider the addition of two integer numbers. A CPF that performs such a transformation takes two integer data streams as input and produces one integer data stream as output. The behavior is such that at any point in discrete time, the output is the sum of the last two messages received on the input streams. Table 3.6 provides a CPF specification for a component that adds two integers. Note that data streams are discrete and therefore, the time domain of the streams in the specification is the set of natural numbers* $\mathbb{N}$*.*

3.4.2 Specification by Hybrid Automata

Hybrid I/O Automata (HIOS) provide a common modeling tool to specify the behavior of hybrid systems that process discrete and dense streams at the same time [Bro12, Alu15, Pto14]. HIOSs define computations, *i.e.,* infinite runs of transitions, by defining a stream of states and an output history for each input channel history [Bro12]. In this sense, a HIOS provides much more detail on how the system shall perform a transformation while abstracting from details of the technical implementation. This specification technique is particularly suited to specify the CPFs defined by sensors and actuators that sense or enact upon the occurrence of an event in a stream.

Hybrid I/O State Machines (HIOS) Automata are a powerful mathematical tool to describe behavior in a state-based manner. HIOSs accurately describe the behavior of

hybrid systems, *i.e.,* those systems that process discrete, and dense streams at the same time [Bro12, Alu15, Pto14]. Various definitions of this kind of state machine exist, we utilize a version that defines them as a generalization of Mealy machines proposed in [Bro12]: A HIOS is a tuple $(\Sigma, \Lambda, \Delta, I, O)$, where

- Σ is a (possibly infinite) set of states,
- $\Lambda \subseteq \Sigma$ is a set of initial states,
- $\Delta : (\Sigma \times \widehat{I}) \to \wp(\Sigma \times \widehat{O})$ is a state transition function,
- I and O are sets of input and output channels, respectively.

Such a machine produces for every pair $(\sigma, \alpha) \in \Sigma \times \widehat{I}$ a pair $(\sigma', \beta) \in \Delta(\sigma, \alpha)$ containing the successor state $\sigma \in \Sigma$ and the output channel snapshot $\beta \in \widehat{O}$ that is produced by the state transition prescribed by Δ [Bro12].
A *computation* of a HIOS is a triple of three infinite streams $(x, \sigma, y) \in \widehat{I}^\omega \times \Sigma^\omega \times \widehat{O}^\omega$, where $x = x_1 x_2 \ldots \in \widehat{I}^\omega$, $y = y_1 y_2 \ldots \in \widehat{O}^\omega$, and $\sigma = \sigma_0 \sigma_1 \ldots \in \Sigma^\omega$ such that $(\sigma_{i+1}, y_{i+1}) \in \Delta(\sigma_i, x_{i+1})$ for all $i \in \mathbb{N}_0$.
We use the he following discretization that associates a discrete stream of channel snapshots with each channel history adapted from [Bro12] to define the semantics of a HIOS CPF specification based on computations: Consider a HIOS $(\Sigma, \Lambda, \Delta, I, O)$. Let $t_0, t_1, \ldots \in \mathbb{R}_+^\omega$ be a sequence of times, and let $x \in \vec{I}$ be an input channel history. Then, we can associate with x a unique discrete event stream $\hat{x} \in \hat{I}^\omega$ of channel snapshots (see Section 3.3.1) via

$$\hat{x}_{i+1} = x|_{[t_i, t_{i+1}[}$$

for $i \in \mathbb{N}_0$. A given HIOS generates for every sequence of input channel snapshots a sequence of states $\sigma_i \in \Sigma^\omega$ and a sequence of output snapshots $\hat{y} \in \hat{O}^\omega$ by choosing σ_{i+1} such that

$$(\sigma_{i+1}, \hat{y}_{i+1}) \in \Delta(\sigma_i, \hat{x}_{i+1}).$$

Then, there is a channel history $y \in \vec{O}$ that is uniquely associated to $\hat{y}$ via

$$y|_{[t_i, t_{i+1}[} = \hat{y}_{i+1}.$$

The channel history $y \in \vec{O}$ is obtained by concatenating the snapshots $\hat{y}_{i+1}$ for all $i \in \mathbb{N}_0$. This defines streams of channel snapshots $\hat{x} = \hat{x}_0, \hat{x}_1, \ldots \hat{I}^\omega$ and $\hat{y} = \hat{y}_0, \hat{y}_1, \ldots \in \hat{O}^\omega$ and a stream of states $\sigma = \sigma_1, \sigma_2, \ldots \in \Sigma^\omega$. The tuple $(\hat{x}, \sigma, \hat{y})$ is then a computation of the HIOS. The semantics of the HIOS specification in Table 3.7 is the set of all such computations:

$$[\![(\Sigma, \Lambda, \Delta, I, O)]\!] \stackrel{\text{def}}{=} \{(x, \sigma, y) \in \hat{I}^\omega \times \Sigma^\omega \times \hat{O}^\omega \mid (\sigma_{i+1}, y_{i+1}) = \Delta(\sigma_i, x_{i+1}) \forall i \in \mathbb{N}_0\}.$$

There are different ways to define the time series that enables this discretization. The approach proposed in [Bro12] does so by imposing the following restriction on the HIOS:
A HIOS is called *time-based* iff there exists a mapping time : $\Sigma \to \mathbb{R}_+$ such that for all $\sigma, \sigma' \in \Sigma$, $\alpha \in \widehat{I}$, and $\beta \in \widehat{O}$ such that $(\sigma', \beta) \in \Delta(\sigma, \alpha)$ it holds that $\text{time}(\sigma) < \mathit{time}(\sigma')$. A HIOS is a δ-step timed state machine iff it is time-based and for all $(\sigma', \beta) \in \Delta(\sigma, \alpha)$ with $\text{time}(\sigma) = j\delta$ and $\text{interval}(\alpha) = [j\delta, (j+1)\delta[$ it holds that $\text{time}(\sigma') = \text{time}(\sigma) + \delta$ and $\text{interval}(\beta) = [j\delta, (j+1)\delta[$ for all $j \in \mathbb{N}$. In [Bro12] the discretization above is defined for δ-step timed state machines for which $t_i = i\delta,\ i \in \mathbb{N}_0$ provides a convenient way to define computations of HIOS.
The HIOSs definition given in [Cam14] is much more complex and enables to define the sampling dynamically. It is possible but out of the scope of this dissertation to adapt these HIOSs and integrate them into CPF specifications.

HIOS Specifications of CPFs To utilize HIOSs for specifying the behavior of a CPF as a set of TSPFs, we embed a graphical notation for hybrid automata similar to those introduced in [ACH+95, Cam14] into the tabular scheme to specify functional behavior in this way. Table 3.7 shows an example of such a specification for a thermostat that includes all elements of the notation. Example 3.5 provides details on the example and on the relation to the HIOS definition introduced above.
The semantics of a CPF specification that uses HIOSs is then the set of all TSPFs that assign to each input history the output history obtained from the computation of the HIOS. The semantics of a CPF whose behavior is specified by a HIOS, is then the set of TSPFs that assign to each input channel history $\vec{x} \in \vec{I}$ the output channel history $\vec{y} \in \vec{O}$ such that the associated discrete streams of channel snapshots and states (x, σ, y) is a computation of the HIOS. That is, given a fixed series of times $t_0, t_1, \ldots \in \mathbb{R}_+^\omega$ that uniquely determines $\hat{x}$ and $\hat{y}$ for all $x \in \vec{I}$ and $y \in \vec{O}$,

$$[\![\text{n(P)}]\!] = \{f \in [\vec{I} \to \vec{O}] \mid \forall x \in \vec{I}\ \exists y \in \vec{O}\ \exists \sigma \in \Sigma^\omega :\ (\hat{x}, \sigma, \hat{y}) \in [\![(\Sigma, \Lambda, \Delta, I, O)]\!] \wedge f(x) = y\}. \tag{3.9}$$

(3.10)

These ideas were introduced in [Bro12].

Example 3.5 (Thermostat)**.** *The model in Table 3.7 specifies the CPF defined by a thermostat similar to the example presented in [ACH+95]. The parameters of the specification are all temperatures which is a physical quantity. Here,*
Temperature $\stackrel{\text{def}}{=} \mathbb{R}_+(\text{K})$ *models the physical quantity temperature whose SI-unit is the Kelvin.*
Once the room temperature drops below a minimal value min ∈ Temp*, the heater is turned on and provides the temperature* tmp*. Once the room temperature reaches a*

CPF

Thermostat($\mathbb{R}_+$(K) minT, $\mathbb{R}_+$(K) maxT, $\mathbb{R}_+$(K) tp_0, $\mathbb{R}_+$(K) roomConst, $\mathbb{R}_+$(K) heatProvided)

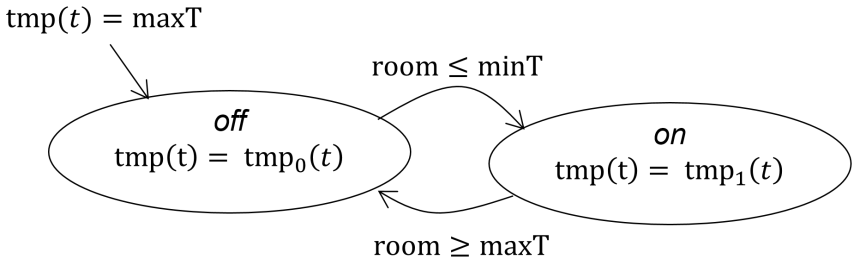
in «data» $\mathbb{R}$(K) room out «energy» $\mathbb{R}$(K) tmp
 $\text{tmp}_0(0) = \text{maxT}$ $\text{tmp}_0(t) = \text{tp}_0 \exp^{-\text{roomConst} \cdot t} \quad \forall t \in \mathbb{R}_+$ $\text{tmp}_1(0) = \text{minT}$ $\text{tmp}_1(t) = \text{tp}_0 \exp^{-\text{roomConst} \cdot t} + \text{heatProvided}(1 - \exp^{-\text{roomConst}\, t}) \forall t \in \mathbb{R}_+$

Table 3.7: Specification of the CPF defined by a thermostat with a HIOS which was similarly published in [ACH+95].

feel-well level of $\max \in \text{Temperature}$*, the heater is turned off, and the output temperature of the heater decreases. The model has a data input that carries information about which temperature is measured in the room and an energy output that describes how the (physical) room temperature shall change. The minimal and maximal temperatures* min*, and* max*, as well as the initial room temperature* T_0*, the provided heating power* heatProvided*, and the room constant* roomConst *are parameters of the specification. The set of input channels and the set of output channels both contain one element, i.e.,* $I = \{\text{room}\}$*, and* $O = \{\text{tmp}\}$*, where* $\text{type}(\text{room}) = \mathbb{R}_+(\text{K})$*,* $\text{kind}(\text{room}) = \text{data}$*, and* $\text{type}(tmp) = \mathbb{R}_+(\text{K})$*,* $\text{kind}(\text{tmp}) = \text{energy}$*, respectively. The set of states consists of elements of the form* $\Sigma = \{off, on\}$*. The physical room temperature changes according to the specifications within the states as specified by* tmp_0 *and* tmp_1*. The set of initial states contains the element* (off)*, and the state transition function is given as*

$$\Delta(d, room) = \begin{cases} (off, \text{tmp}_0 \,|_{[t,\delta(t,\text{room})[}) & d = on,\ lt(\text{room}, t) \geq \text{maxT} \\ (on, \text{tmp}_1 \,|_{[t,\delta(t,\text{room})[}) & d = off,\ lt(\text{room}, t) \leq \text{minT} \end{cases},$$

where

- $\delta(t, x) = \min\{t' > t |\ x(t') \in \{\text{minT}, \text{maxT}\}\}$ *represents the next time that the measured room temperature received on the input port reaches the value* minT *or* maxT *respectively, and*
- $lt(x, t) = x(\max\{t' \in \mathbb{R}_+ \mid t' \leq t, x(t') \in \mathbb{R}(\text{K}) \setminus \{\xi\}\})$ *is the last value received on the channel* x *before time* t*.*

Example 3.6 (Electrical Switch)**.** *An electrical switch enables switching electrical devices on and off. It defines a subsystem that interrupts or connects a stream of electrical energy. In this example, we consider a mechanical switch that takes as input a signal indicating that a force acts on a mechanical button and a stream of electrical energy. If the switch is turned on, it provides a stream of electrical energy with the same power as comes in on the input channel, or provides a stream of electrical energy with* 0W*. In this specification, the interface is defined as the set of input channels* $I = \{\text{f}, \text{in}\}$*, and output channels* $O = \{\text{out}\}$*. The function is parametrized with the minimal force needed to press the switch* $f_{\min}$*. The set of states* Σ *contains the discrete states* on*, and* off *and the continuous states* out_0*, and* out_1 *describing the evolution of the output energy* out*. Therein, the set of initial states comprises only the element* $(\text{off}, \text{out}_0)$ *The state transition function,* Δ *is defined via*

$$\Delta(d, f) \begin{cases} (off, \text{out}_0 \,|_{[t,\delta(t,f)[}) & d \in \{\text{on}\},\ ||f(t)||_3 \geq ||f_{\min}||_3 \\ (on, \text{out}_1 \,|_{[t,\delta(t,f)[}) & d \in \{off\},\ ||f(t)||_3 \geq ||f_{\min}||_3 \\ (on, \text{out}_1 \,|_{[t,\delta(t,f)[}) & d \in \{on\},\ ||f(t)||_3 < ||f_{\min}||_3 \\ (off, \text{out}_0 \,|_{[t,\delta(t,f)[}) & d \in \{off\},\ ||f(t)||_3 < ||f_{\min}||_3 \end{cases}$$

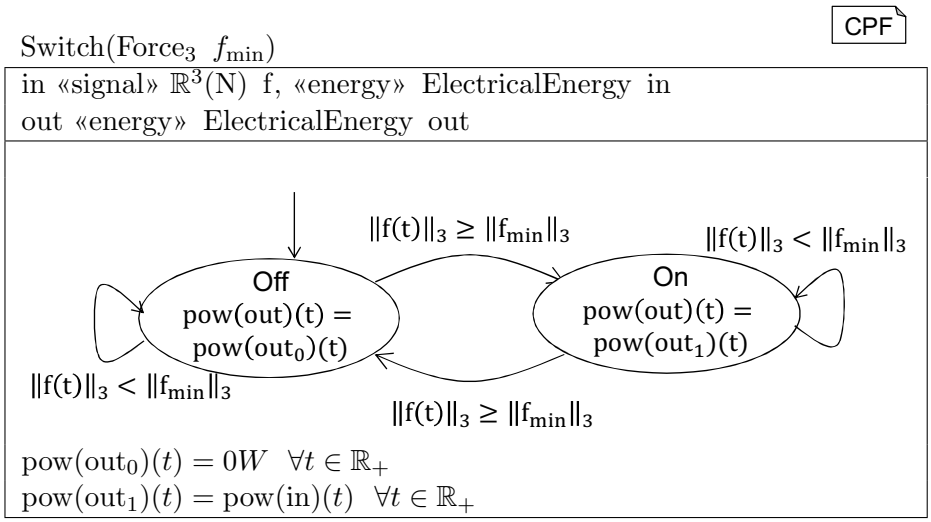


Table 3.8: HIOS specification of the CPF defined by an electrical switch.

where $\delta(t, f) = \min\{t' > t|\ ||f(t)||_3 \geq ||f_{\min}||_3\}$ *represents the next time after* t*, when the absolute value of the incoming force* f *exceeds the minimal force* $||f_{\min}||_3$ *needed to activate the switch.*

3.4.3 Architectural Specification

Architectural specifications define the functional behavior through the functional composition of multiple CPFs. In [Bro10], this specification technique is called the compositional specification. That is, functional composition in the mathematical sense (*cf.* Section 1.3.1). The specification, again, defines the functional behavior as a set of TSPFs, *i.e.,* those that result from the composition of the composed functions. Based on [BDD+93], Section 3.3.3 has introduced sequential, parallel, and feedback composition for hybrid TSPFs, *i.e.,* TSPFs that operate on channel histories with both discrete and dense image domains. For compositional specifications, we utilize a C&C [Kus21] notation, that is very similar to MontiArc [HRR12a] (*cf.* Section 1.3.4). The compositional specification declares the stream kind of connectors in addition to their types allows referencing physical quantity types by the number domain and the SI-unit, and holds the CPF tag. Figure 3.3 provides the compositional specification of a pump which is a slight adaptation of the example provided in [Kol85]. The specification composes the CPFs defined in Table 3.8, Table 3.3, and Table 3.4. In this case, the specification represents a parallel composition of these three functions. In the notation, the interfaces are represented by named ports and connected by typed

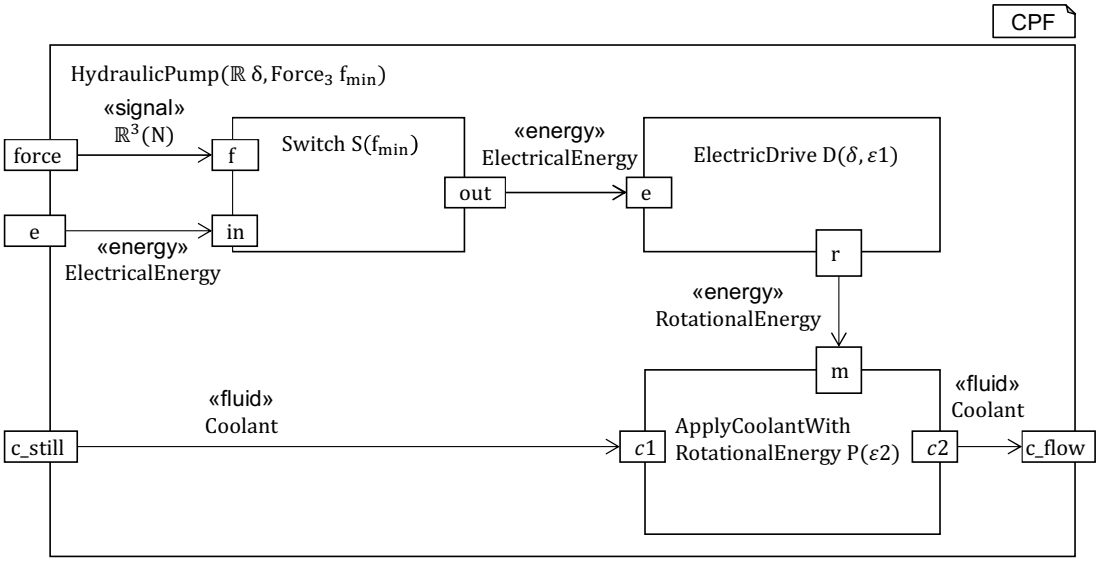


Figure 3.3: Architectural specification of the CPF defined by a pump similar to [Kol85].

connectors which is the notation of MontiArc.

Rotational energy is the form of energy that is characterized by torque (N m) and angular velocity (rad s^{-1}). Turning the pumping system off corresponds to requesting the pump to operate at a power of zero Watt, while turning the pumping system on corresponds to its operation at a power that is greater than zero Watt. This power demand is proportional to the conveying capacity [GBG18]. In contrast to [Kol85, Kol98], we specify the function with only one input signal telling the power the pumping system shall operate at. Figure 3.3 shows the formalization of the pumping system provided in [Kol85, Kol98] using the definitions from Example 3.1, 3.2, and Example 3.6. Instead of symbols or a substantive and verb to describe the functional behavior which is common in mechanical design theory (*cf.* Chapter 5), we utilize logic to specify the behavior of these subfunctions [Bro10]. The switch subfunction takes a signal stream as input that represents the force applied to a button by the user. Here, the force is taken as a signal, as the force carries the information of whether or not the user would like to turn the system on or off. If the switch is turned on, it provides the electrical energy a delay of $\delta > 0$ it has received as input. The drive function converts the electrical energy to motion energy. The specification uses the law of energy conservation and abstracts from losses. That is, at all points in time t, the power exchanged through motion energy until time t is less than or equal to the power received as electrical energy until time t. Power is determined by the product of the two physical quantities that describe the respective form of energy. The pump function

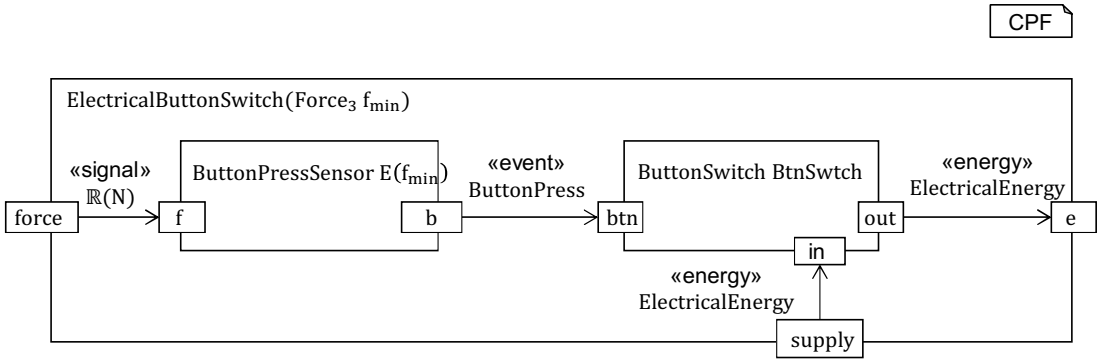


Figure 3.4: Decomposed specification of the electrical switch.

applies the motion energy to the fluid, by converting it to compression energy. However, the compression is carried by the fluid, therefore the incoming and outgoing streams are fluid streams. The specification uses the law of energy conservation for the drive function. However, the function can also be realized in another domain, and we, therefore consider the more abstract, and general specification of this function, here.

Example 3.7 (Electrical Switch with Button Press)**.** *For switching on or off electrical devices often have a mechanical button. Pressing the button turns the system on or off depending on whether it has been off or on, respectively. To specify this CPF, we decompose the specification in Table 3.8 as shown in Figure 3.4. To this effect, Table 3.9 defines a CPF that provides electrical energy once it receives a button press event. The data type* `ButtonPress` *has been defined previously in Table 2.5. Further, Table 3.10 specifies the CPF of a physical button. It can be interpreted as the CPF defined by a sensor that measures the amount of force that a user applies to the physical interface of the function which was already outlined in Section 3.2.6. The function takes a force as input. Once this force exceeds a minimal force* $f_{\min}$*, the function sends a* `ButtonPress.PRESS` *event on the event output port* `b`*.*

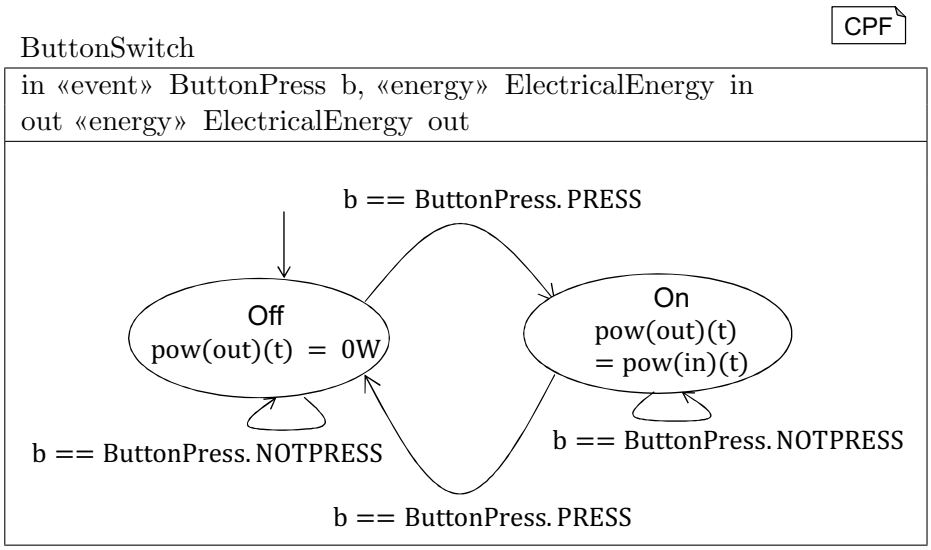


Table 3.9: HIOS specification of the CPF defined by a switch that reacts to a button press event.

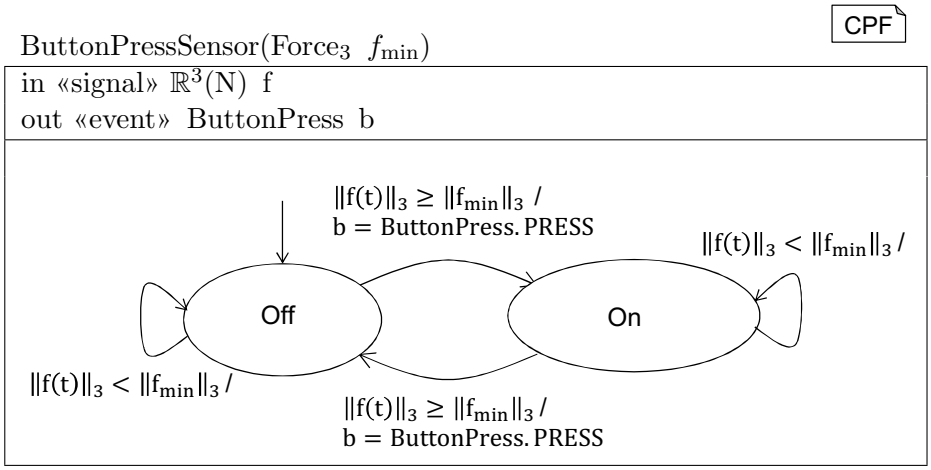


Table 3.10: The ButtonPressEvent specifies the occurrence of a ButtonPress in a stream of force. It enhances the definition of the event type ButtonPressEvent by specifying when the events occur in the stream of force.

Chapter 4

A Methodology for Functional Model-Driven Engineering of CPSs

Engineering innovative systems has become a complex task due to many reasons [FR07]. One of them are interdisciplinary development teams and the shift toward functions being the innovation drivers continue to increase this complexity significantly. As [Rum96] emphasizes, traditional engineering disciplines apply norms, standards, and guidelines known throughout the entire domain to systematize, structure and support the engineering process at all phases. An example of such a standardized process is the product development process described in [GBP+21a] for mechanical engineering. Due to the interdisciplinarity of systems engineering, methodologies in this field must regard the peculiarities of all of the domains. What systems engineering methodologies mostly have in common is that they utilize models to describe the system under development focusing on various aspects of the system. This dissertation is concerned with introducing an abstraction layer between the requirements that are mostly functional and descriptions of the system's implementation. To answer the research question RQ4 "What are the constituents of an MDE methodology that targets the development of system functions?", this chapter defines the functional development paradigm by the five principles. So far, we have defined these principles for the domain of CPSs. Based on that, we define a formal model-driven methodology that follows the functional development paradigm.

4.1 The Functional Development Paradigm

In traditional software engineering, a programming paradigm facilitates developers to create code in high quality, by giving aid and instructions for creating solutions to complex problems [Lei21]. The paradigm defines the focus of the development activities and proves guidelines on how to evolve the elements of focus to finally obtain a solution to the problem. Similarly, an engineering paradigm puts one or more aspects of the object to be engineered into focus to guide, structure, and link development activities. A software engineering paradigm is characterized by the (formal) definition of the set of aspects that should guide the development. Examples are object orientation or MDE.

The model-driven paradigm is a paradigm in which development decisions are made based on the information stored in models that is characterized by the three aspects (1) model, (2) modeling language, and (3) methodology. A *model* is a purposeful representation of an original system that is reduced or abstracted from its size, detail, and/or functionality [Sta73]. The purpose determines what the model is used for in the development process. In MDE, models reflect the state of the system under development at any point in time during the development process. They serve multiple purposes, in particular, not only documentation purposes, but enable to automate specific tasks in the development process. Establishing the model-driven paradigm requires the definition of a *modeling language* in which models are created by the developers. Therein, a modeling language consists of the definition of a set of well-formed models, a semantic domain which is defined based on a sound mathematical theory, and a semantic mapping that assigns each well-formed model a set of meanings from the semantic domain [Kau21, HR04].
The *methodology* defines a (formal) framework that structures the way and time models are created during the development process. In terms of [Rum96] a formal methodology consists of 1. a set of model kinds, 2. a formal semantics for each kind of model, 3. a set of development steps that transform the models that are also formally grounded, and 4. a set of guidelines of when and with which aim to apply each transformation.
This chapter characterizes the functional development paradigm by four aspects that structure the model-driven development of systems, define of a semantic domain for a functional modeling language that is detailed in Chapter 6 and set up a methodology that enables functional model-driven engineering.

4.1.1 The Five Principles of Functional Development

Nowadays, innovations in modern systems arise from an increasing number of functions and features that are implemented by interacting software, mechanical, and electrical components or subsystems. Engineering such innovative systems while maintaining accuracy and efficiency throughout the entire development process requires to overcome the conceptual gap between requirements and implementations in a way that enables the efficient collaboration of experts from heterogeneous domains. The application of agile principles is necessary to support large-scale engineering of CPSs which requires automating tasks such as change management, testing or dimensioning which is the process of finding optimal values for geometric properties. The efficiency is, among others, driven by reuse and iterative refinement which decrease the amount of accidental complexities and improve the documentation of information through standardization and automation. Decomposing the system along the innovation driver enhances the reuse of known and validated solutions, while it minimizes the need for communication among teams because the interdependencies among the components are clarified in a model. In the engineering of distributed software or embedded systems,

functions are already the target of well-established development methodologies. Surprisingly, mechanical design theory has also recognized that making a (mechanical) system's function the target of the development has many advantages, *e.g.,* by structuring the creative process of finding geometric designs such that physical effects come into action in a way that a dedicated purpose is met [BG21, Kol98].
The *functional development paradigm* developed in this dissertation thesis, therefore, promotes five principles: (1) *streams* of energy, matter, and data are the inputs and outputs of a CPS, (2) a CPS transforms these streams and, thereby, defines a *function* which has become the main innovation driver and is, thus, considered the target of the development, (3) controlled *underspecification* enables to, among others, include information as soon as it is available and to identify the lack of information (automatically). Further, it allows to include uncertainty and to restrict the solution space such that its exploration becomes manageable. (4) The task of developing a CPS must be decomposed into smaller tasks to develop subsystems to cope with the complexity, thus the *composition* of two or more CPFs yields a CPF, and (5) *refinement* allows to narrow the solution space iteratively and include newly obtained information in the models directly. As refinement respects composition, this approach contributes to coping with the complexity of CPSs as it enables to engineer sub-tasks individually. These aspects have been formally defined in the previous Chapters 2 and 3. The following section will define a formal methodology based on the theory that follows the principles of the functional development paradigm.

4.2 Formal Methodology for Engineering CPSs

The preliminary Section 1.3.2 introduces the foundations of formal methodologies. This section uses these foundations to define a formal methodology for the functional engineering of CPSs. It includes two kinds of models, *i.e.,* functional specifications of CPFs and type definitions by CPCDs. Functional specifications define the interface and behavior [Bro12] of a CPF. The types of channels in the interface are defined by a CPCD. The respective modeling languages were introduced in Chapter 2 and 3, respectively. The semantic composition of CPFs has been introduced in Section 3.3.3. Their syntactic composition is introduced in Section 3.4.3. A way of composing CPCDs semantically sound is introduced in [LRSS23].
Development steps are transformations of the set of models which means, either models are added to this set, or models in the set are transformed in some way. At any time, the set of all models describes the current development state of the system. To prevent ambiguities, this set must remain consistent in the sense of Definition 1.5 and free of redundancies. Therefore, it is necessary to perform (automated) checks for redundancies and inconsistent models after each transformation. Also, our methodology applies the idea of the SMARDT methodology [DKMR19] that once a

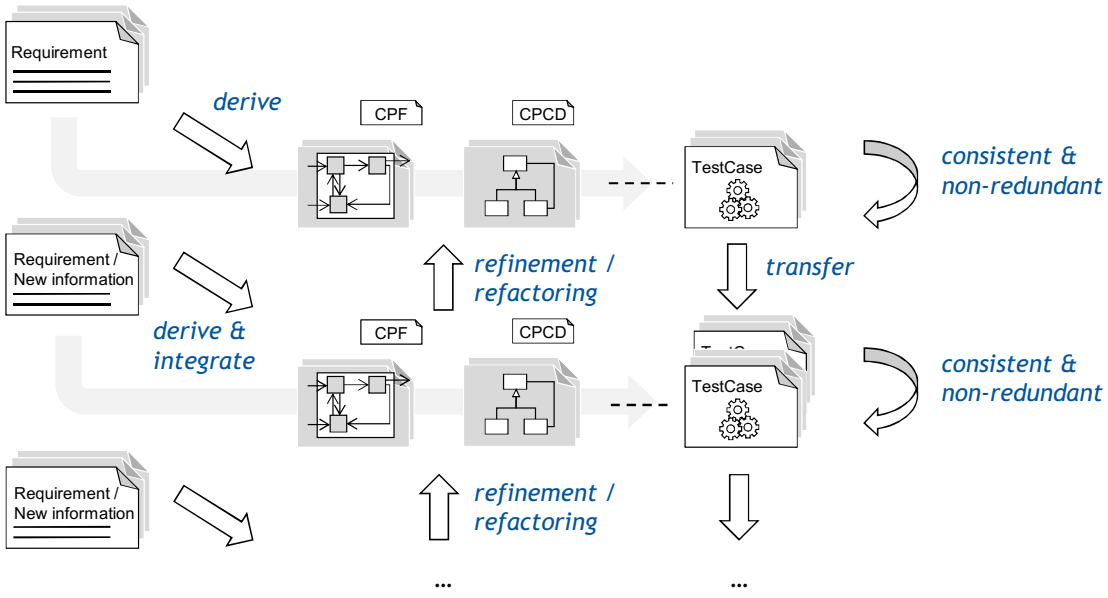


Figure 4.1: Illustration of a development step in the functional MDE methodology for CPSs.

test case is created it must hold throughout the rest of the development. Figure 4.1 illustrates the idea of the methodology: Based, *e.g.,* on requirements, engineers create a functional model of the system. In this process, modelers interpret *e.g.,* textual requirements, and derive a (set of) CPF specification(s) together with the necessary types in a CPCD. A functional model of the system in its current state of development is given by the CPF and the CPCD. The CPFs and CPCD are enhanced with a set of test cases that allow validating whether the models represent a system that corresponds to the requirements. In each development step, the existing set of test cases is transferred as they are and run to validate correctness with respect to the former set of requirements [DKMR19]. This set is complemented by new test cases to validate correctness with respect to the new or adapted models. Checks for redundancy and consistency are performed after each transformation of these models. A transformation is accepted iff the set of models is free of inconsistencies and redundancies and iff the transformation yields a refinement or a refactoring of the original models. Performing a transformation corresponds to a development step and produces a new version of the original model. Each development step aims to either *refactor*, *i.e.,* change the model's syntax without changing its semantics (*cf.* Definition 1.8), or *refine*, *i.e.,* changing the model's syntax such that the semantics of the new version is a subset of the former (*cf.* Definition 1.7), the specification. A formal methodology provides a set of development steps together with guidelines on

when to apply them [Rum96] (*cf.* Definition 1.9). Transformations of CPCDs and guidelines on when to apply them are provided in [Rum96].

4.2.1 Transformations of the Interface of a Functional Specifications

In our methodology, functional specifications are given by CPFs which were introduced in Chapter 3. Transformations of these models can be performed by changing the interface or by changing the behavior specification. Both kinds of transformations can be performed within a single development step. However, for checking refinement, it makes sense to analyze the transformations with respect to refinement separately. Similar to [SRS99], we consider transformations of the interface to prepare certain transformations of the behavior that require new input or output channels.
Concerning the interface, the modeler can modify a CPF by

- renaming a channel,
- adding a channel,
- removing a channel,
- changing the type of a channel,
- changing the kind of a channel,
- adding a parameter,
- removing a parameter.

Naturally, the specification of the function's behavior is only syntactically well-defined iff it uses the variables that define the function's interface or a parameter. Therefore, removing channels or parameters that are used in the behavioral specification of the CPF does not yield a syntactically sound model.
In the following, let $F = [I, O, B]$ and $F' = [I', O', B']$ be two CPFs.

Renaming channels or changing the type or kind of a channel formally corresponds to defining two functions $R_I \in [\vec{I} \to \vec{I'}]$ that renames input channels and $R_O \in [\vec{O} \to \vec{O'}]$ that renames output channels. The transformation yields a refinement, iff there exists $A_I \in [\vec{I'} \to \vec{I}]$, $R_I \in [\vec{I} \to \vec{I'}]$ and $A_O \in [\vec{O'} \to \vec{O}]$, such that (A_I, R_I) and (A_O, R_O) are refinement pairs making F an interaction refinement of F' in the sense of Definition 3.18.

Adding input channels or parameters or removing output channels or parameters that are not (yet) used in the behavior specification of the CPF can be seen as a preparation for changing the interface of a CPF. This becomes apparent with the interface adaption from [SRS99], that formalizes adding new input channels and removing output channels that are not used: Let $F = [I, O, B]$ be a CPF, let J be an extension of the input channels, *i.e.,* $I \subseteq J$, and let P be a subset of the output channels, *i.e.,* $P \subseteq O$. Then $F' = [J, P, B \updownarrow_J^P]$, where

$$B \updownarrow_J^P = \{g \in \{\vec{J} \to \vec{P}\} \mid \exists f \in F : \ \forall x \in \vec{J} : \ g(x) = f(x|_I)|_P\}$$

As we consider the behavioral specifications to remain unchanged, we obtain refinement pairs by defining respective embedding and restriction functions: Let A_I and R_O be the restrictions of $\vec{J}$ to $\vec{I}$ and of $\vec{O}$ to $\vec{P}$, respectively, and let A_O and R_I be respective embeddings, *i.e.,* $A_O : P \to O, x \mapsto x$ and $R_I : I \to J, x \mapsto x$. Then, (A_I, R_I) and (A_O, R_O) are refinement pairs such that also $A_O \circ F' \subseteq F \circ A_I$ holds. Thus, F' is an interaction refinement of F.

4.2.2 Transformations of the Behavior

Besides an interface, a CPF consists of a behavior which is a set of TSPFs that can be specified by interface assertions, by HIOSs, or by the composition of multiple CPFs. Development steps may include transformations of the behavioral specification of a CPF. Behavioral specifications are most likely necessary if the interface of a specification has been changed or if new information, *e.g.,* about yet underspecified transition becomes available in the development process.

Interface assertions are formulae in predicate logic [BM97] that specify the set of TSPFs that includes all valid transformations of the input streams to the output streams performed by a CPS. To precipitate a development step, the modeler may transform the interface assertion of a CPF by adding or removing statements from the interface assertion. Whether or not one yields a refactoring or refinement, *i.e.,* a development step depends on whether or not the added or removed statement implies the existing interface assertion. To understand this, note that the implication of the interface assertions corresponds to the set inclusion of the behaviors: Let $F = [I, O, B]$ and B be the behavior defined by the interface assertion $p : \vec{I} \times \vec{O} \to \mathbb{B}$. Then $p(i, f(i))$ holds for every channel history $i \in \vec{I}$ and for every TSPF $f \in B$. Now, let $q : \vec{I} \times \vec{O} \to \mathbb{B}$ be another predicate and let $B' = \{f \in [\vec{I} \to \vec{O}] \mid \forall i \in \vec{I} : \ q(i, f(i))\}$. Assume that $p \Rightarrow q$, then it follows directly that $B' \subseteq B$ because $p(i, (f(i)))$ implies that $q(i, f(i))$ holds for all $i \in \vec{I}$ and all$f \in \vec{I} \to \vec{O}$. The interface assertion q is a refactoring of p iff also $q \Rightarrow p$ holds.
To give an example of a behavioral refinement imposed by adding a statement, let p, q be the interface assertions from above and assume that B is the behavior defined p.

Assume, the modeler adapts the interface assertion such that B' is defined by $p \wedge q$. The transformation yields a refinement iff $p \wedge q \Rightarrow p$ but $\neg(q \Rightarrow p \wedge q)$. In this case, because $(p \wedge q)(i, f'(i))$ holds for all channel histories $i \in \vec{I}$ and all TSPFs $f' \in B'$, also $p(i, f(i))$ holds due to the implication and therefore $f' \in B$. If the implication would also hold vice versa., *i.e.,* $p \wedge q \Rightarrow q$, adding q in conjunction would yield a refactoring because then also $f \in B$ would imply $f \in B'$.

HIOSs specify functional behavior in a state-based manner. The semantics of this kind of specification is defined via the computations of these automata. Development steps can be performed by changing the elements of the HIOS by the following transformations:

- adding a state,
- removing a state,
- renaming a state,
- refining a state definition,
- adding a transition,
- removing a transition,
- refining a transition,
- extending the set of initial states,
- reducing the set of initial states.

The contribution in [Rum96] provides a calculus that enables one to decide whether such transformations imply a refinement of the original automaton for spelling automata. This calculus is not directly applicable to HIOSs. Since defining such a calculus is out of the scope of this work, we say that these transformations yield a refinement, iff the transformation of the HIOS yields a refinement of the original automaton. As we have introduced a trace semantics for HIOSs that defines the semantics of a HIOS as the set of computations defined by a HIOS, the transformed HIOS refines the original HIOS iff the transformed set of computations is a subset of the original set of computations.

Decomposing a functional specification enables to follow the divide and conquer principle. For the syntax of a CPF, decomposition enables to keep the interface assertions or the HIOS specifying the function's behavior compact. An example of a decomposition is given in Example 3.7. Because the theory that provides the semantic domains for CPFs is compositional, decomposition enables to refine the individual components to obtain a refinement of the composed specification. A decomposition aims at refactoring or refining the existing functional specification to transform a complex specification following the divide and conquer principle. After a decomposition step, the development tasks can be distributed, *e.g.,* to different teams for further development.
In general, there exist three ways to decompose a function defined by the different kinds of composition: parallel, sequential, or with feedback (*cf.* Section 3.3.3). Whether the decomposition yields a refinement or refactoring depends on whether the interface assertion that specifies the behavior of the original function holds also for the composed behaviors, or if the set of TSPFs derived from the computations of a HIOS includes the set of TSPFs that result from the composition.
When to decompose is often a subjective decision. The contribution in [KRW20] proposes to initiate an (automated) decomposition based on an influence relation among the channels of the interface. Therein, an input channel influences an output channel iff changes in an input history cause a change in the output history at the output channel. The contribution is defined for time-synchronous port automata that transform discrete streams. Mechanical design methodology which is further detailed in Chapter 5 proposes to decompose a function until it is entirely described by the composition of a finite set of elementary functions. The latter are a set of standardized functions that mechanical design theory considers to be complete in the sense that all functions of mechanical systems can be described by. The semantics of composition are defined in Section 3.3.3 and Section 3.4.3 provides the basics on how to model decomposed CPFs.

4.3 Related Work

There exists a range of literature on (MDE) methodologies for CPSs. Some of these approaches [BBD+21, Vog15, DGH+18, HNZ+23, SJZK23] describe a system as a (set of) functions that process energy, matter, and/or data, that decompose the system into smaller parts to overcome complexity and that include underspecification to be able to consider multiple implementations and narrowing the solution space iteratively to find an optimal solution. What these methodologies have in common is that they separate the development into layers that represent activities at different stages of development. Systematizing and generalizing these activities in a process model is certainly important, however when it comes to categorizing models into these layers the hustle

begins. The practice has shown that a general definition of these layers that predefines the kinds of models or their contents of information that lead to discussions of when to create a specific model or on which layer a certain model belongs. These discussions hinder the progress of the development. Therefore, the methodology presented here is formal and defines the kinds of documents for defining and evolving a functional view of the system. Our methodology simply predefines the refinement/refactoring relation between two versions of the system's functional model to reflect the integration of new information or improvements of the model's syntax. Practical applications of this methodology can define layers to categorize the models and systematize their creation throughout the development specific to the project, company, or system. To prevent discussions of when to place which model at which layer, it makes sense to equip each layer with one or more predicates that allow placing the models unambiguously and possibly automatically.
In contrast to SMARDT [DGH+18], SPES [BBD+21], montego [SJZK23] and other methodologies such as [HNZ+23], we do not predefine layers of development. The "layers" in Figure 4.1 represent iterations of the development and we solely require the models of later iterations to be refinements or refactorings of previous iterations. When applying mechanical design theory and respective methodologies, such as proposed in [Kol98, BG21] it may be reasonable to differentiate among functions and solutions because it is possible to define when a function becomes a solution unambiguously. In Chapter 5 we will define a solution to be a function whose behavior is defined by logical formulae that represent physical laws from a predefined set (*e.g.,* a design catalog such as [KK98]). This way, a function becomes a solution as soon as a modeler chooses a physical effect described by a (set of) physical laws from a library to describe the behavior of a function. However, in other domains of CPS engineering, it may not be so clear when such layers can be distinguished.

Software Platform Embedded Systems (SPES) Conceptually, the methodology that we have outlined has many similarities with the SPES Methodology [BBD+21] as it is also based on the formal theory of Focus and defines a system from a functional perspective as a set of TSPFs. This methodology takes a functional point of view to structure the development and also follows the principles of decomposition and underspecification to manage the complexity of the engineering process. The semantics applied for the models in the model-driven approach fostered by SPES is based on the traditional discrete Focus [BS01] meaning that the components in the models of the system interact by exchanging discrete data messages. SPES has been developed for software or software intensive systems which are accurately discribed by TSPFs that process only discrete streams. To separate the concerns of all the stakeholders involved in the engineering of these systems and also for managing the artifacts that document the status of the development, SPES defines the four viewpoints: Requirements,

functional viewpoint, the logical viewpoint, and the technical viewpoint [GJRR22b]. The requirements viewpoint constitutes models of system requirements and supports the requirements engineering activities. The methodology also offers mechanisms to capture requirements in a structured and traceable manner. The functional viewpoint describes the system's functions as a hierarchical network while the logical viewpoint describes the system as a composition of logical components. As in our methodology, the idea of SPES is that these functional models are derived from the requirements. The technical viewpoint combines models of software and hardware components. [GJRR22b] The functional viewpoint describes the functions at a very high level of abstraction that can be derived directly from the requirements [BBD+21]. Therein, the logical viewpoint describes a logical realization of the functions described in the functional viewpoint [BBD+21]. In some circumstances, the differentiation between these two levels may be clear, however, in others [GJRR22b] it remains ambiguous. Therefore, our methodology forgoes to define general levels of abstraction. The SysML-profile proposed in [GJRR22b] provides graphical DSLs for each of these viewpoints.

Specification Method for Requirements, Design, and Test (SMARDT) Concerning the integration of test cases and the creation of models of the system under development, the idea of our methodology is very close to that of SMARDT [DGH+18]. SMARDT divides the development into four layers, where the models of each layer are enhanced by a (set of) test cases. The four layers defined by SMARDT are: Layer one includes requirements and use cases for the different functionalities modeled using the respective SysML elements. Layer two contains abstract functional specifications in the form of SysML activity diagrams, state charts, sequence diagrams, and IBDs Layer three adds implementation-specific aspects, such as *e.g.,* timing, to the functional description. Layer four comprises the implementation artifacts in the form of code, hardware specifications, *etc.*. Similar to our approach, the idea is that models on the lower levels refine the models on the higher levels, which reflects that they contain more information about the system under development. Then, the test cases defined on every layer must be passed for all models of the same and all lower levels. As in our approach, test cases from the higher level remain relevant for each lower level. This way, SMARDT aims to assure consistency and correctness throughout the entire development process.

The Montego Method emerged from the domain of mechanical engineering [SJZK23] and assimilates the approach presented in [DRW+20] and [HNZ+23]. The development is subdivided into requirements, functions, solutions and product and prescribes the usage of SysML for modeling the objectives of each layer. The Montego method employs the ideas from Koller's design methodology which allows us to differentiate

between functions and solutions unambiguously, because solutions are considered as functions whose behavior is described by physical laws from a predefined set. The contribution in [SJZK23], however, does not provide a semantics, *i.e.,* meaning of the models. As prevalent in mechanical design methodology, the functional models including solutions mostly serve documentation purposes. The approach uses model execution to trigger external simulation models that fill external CAD models. The execution uses the functional relations in the model as trace links, *i.e.,* it interprets the functional flows as interrelation among these models. In terms of [Vog15], the method follows the dimension of modeling to structure the development artifacts and does not yet use the dimension of formalization to add precision to the meaning of the models and their elements.

Formal Model-Based Requirements Engineering The methodology presented in [Vog15] formalizes requirements using the discrete Focus theory, *i.e.,* streams are considered as functions $\mathbb{N} \to \mathcal{U}$, to integrate formal specifications in the requirements engineering process. The approach considers a function to be "an intent to use a system in a specific usage context or for a certain purpose" [Vog15]. The focus of [Vog15] is to introduce modeling and formalization in the requirements engineering phase of the development of software (intensive) systems.

Multi-Paradigm Modelling In the field of multi-paradigm modeling, a paradigm "acts as a pattern for describing a whole class of artifacts sharing similar characteristics or designates a framework that encapsulates theories inside a scientific domain" [ABH+21]. The framework proposed in [ABH+21] offers mechanisms to formally define a paradigm as a set of properties and to check automatically whether a candidate satisfies the properties. Here, we have defined the functional development paradigm informally as a set of three principles, *i.e.,* function, composition, and underspecification. A functional MDE methodology follows these principles, whenever the target of the development are CPFs. That is, models of functions (in the mathematical sense of Definition 1.1) are the primary source of information during the development process. These models exhibit underspecification which is lifted iteratively by refinement throughout the development process, *i.e.,* a new version of a model must be a refinement of its previous version (*cf.* Figure 4.1). A model is underspecified, whenever its semantics contains more than just one element. In the functional MDE approach that is proposed in this dissertation, we consider specifications of CPFs to describe a *set of* TSPFs over discrete and hybrid streams (*cf.* Section 3.3). Throughout the development, this set is narrowed, *e.g.,* by adding constraints to a behavioral specification. The principle of composition implies that models of functions are hierarchically decomposed, *i.e.,* a function can be described by the functional composition (in the mathematical sense, *cf.* Section 1.3.1) of other functions. This

decomposition enables structuring the development along the functional decomposition to overcome the complexity of engineering CPSs.
The framework from [ABH+21] enables to check whether a candidate satisfies the properties that define a paradigm. For the paradigm proposed here, that would imply defining formal checks on whether the candidate defines functions as a set of TSPFs over discrete and hybrid time domains defines a refinement relation among such functions such that a function refines another iff the set of TSPFs of the former are a subset of the TSPFs defined by the latter function, and defines mechanisms to describe a function as a composition of other functions. Formally defining these, is certainly interesting but out of the scope of this dissertation.
In the context of MDE, multi-paradigm modeling provides solutions to deal with the different formalisms of models, *i.e.,* models created using heterogeneous tools and languages by experts from heterogeneous domains [SW21, VLM02]. In contrast to our approach, multi-paradigm modeling focuses on the smooth integration of heterogeneous models in the engineering environment following the principle that everything should be modeled explicitly at the most appropriate level of abstraction with the most appropriate modeling language [ABH+21, SW21]. The paradigm suggested here, could be seen as one paradigm for one set of models employed in an MDE environment for CPSs.

Chapter 5

Formalizing Design Catalogs as Libraries of Physical Functions

A major challenge in CPS engineering is the collaboration of experts from different backgrounds in the fields of software, mechanical and electrical engineering [Rod91, DRW+20]. The development artifacts from each domain are heterogeneous and most often highly detailed, such that experts from different backgrounds or domains cannot understand them. This issue has triggered research on MDE of CPSs in all three domains and brought forward modeling techniques, and theories [BS01, Alu15, Pto14], as well as methodologies [Kol85, Rod91, Rot01, BG21]. What prevalent MDE approaches have not yet achieved, is the integration of the domain of mechanical engineering [HNZ+23]. A question pursued in the making of this dissertation is, therefore, the research question RQ2.1 "Can this theory formalize the concepts from mechanical design theory defined in [Kol98, KK98, BG21]?" which is addressed in this chapter. Mechanical engineering is "the study of physical machines that may involve force and movement. It is an engineering branch that combines engineering physics and mathematics principles with materials science, to design, analyze, manufacture, and maintain mechanical systems." [ME2]. Although approaches such as [BS01, Pto14, Alu15] specify the system as networks of interacting CPFs, they abstract from the aspects, mechanical engineering is concerned with, *e.g.,* varying physical effects and the system's geometry to realize a system's functionality. These approaches assume the physical principles and basic geometric setups of the physical implementations to be present and do not consider their variation to optimize an implementation in this regard.

The functional development paradigm proposed in Section 4.1 promotes to align the development along the functions defined by a system that can be derived from requirements which have become the innovation driver for CPSs. Mechanical design theory is a branch of mechanical engineering that promotes using functional descriptions that abstract from physical processes and geometry as the starting point of a systematic search for physical implementations. This chapter aims to align these similar approaches to describing a system by taking advantage of the similarities in the interpretation of a system as a network of interacting functions, and their notation

from the domains of software [BS01] and mechanical engineering [BG21]. To show that the modeling technique established in Chapter 2 and Chapter 3 enables expressing functional structures from mechanical design methodology in a model with formal semantics, we derive interface assertions for a subset of the functions listed in Koller's design catalog [KK98]. The catalog provides a collection of known elementary functions. According to [Kol98], these functions are to mechanical engineers what basic arithmetic operators like $+, -, \cdot, /$ or the Boolean and set operators like $\wedge, \vee, \cup, \cap$, *etc.*. are to software engineers. The Koller catalog then provides lists of physical effects that realize the transformations defined by the elementary functions that can be used to combine a physical solution to implement the elementary function. The following sections summarize the main ideas of mechanical design theory, that a system can be described as a network of interacting functions as well as the existence of elementary functions and how solutions can be composed from physical effects to realize these functions together with a quantitative description of geometry. In Section 5.2 we provide descriptions of elementary functions from [KK98] in the scheme of Table 3.2 which use interface assertions to formalize the behavioral descriptions of these functions. Koller suggests decomposing system functions hierarchically until the lowest level contains solely elementary functions. Providing specifications of these elementary functions enables the application of the methodology outlined in Section 4.2 for projects that apply the design methodology defined by Koller in [Kol98]. Chapter 6 provides a SysML profile as a modeling language for mechanical engineering experts in which we have digitalized a part of the Koller catalog [KK98].

5.1 Mechanical Design Methodology

"Engineering Design is the process that ensures the technical feasibility of a product concept and the required "functions" of the product. In engineering design, engineers select the process, specify the materials, and determine shapes to satisfy technical requirements and to meet performance specifications." [Swa00]
To systematize this process and to reduce complexity, mechanical design methodologies like [PBFG07, Rot02, Rod91], or [Kol98] promote describing the system through a *functional structure*. This is a hierarchical structure of interacting functions that utilizes descriptions of *physical effects* to link these functions to geometric components [BG21, VDI97]. Therein, a conceptual solution is composed of the solutions of these sub-functions.With the rising number of functional requirements and the trend of functionalities and features becoming the innovation driver of CPSs, research in this field has gained momentum with the aim of narrowing the problem implementation gap (*cf.* Section 1.1.1) [ZJK+22, ZJS+21, ZRJ+22, JKB+22].
A part of the research in this field deals with documenting elements, such as functions, physical effects, or geometries, that occur repeatedly during the development process in

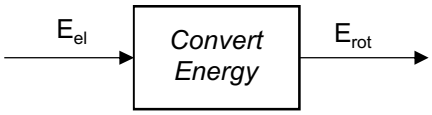


Figure 5.1: Graphical notation of a function that converts electrical to rotational energy as in [Kol98, BG21].

so-called design catalogs [KK98, Rot94, VDI82] to make knowledge about these elements reusable and to enable their systematic variation. In practice, however, mechanical engineers rarely explicate or even model functional structures or even use design catalogs during development [DRW+20]. Most often, engineers keep the link between functions and the geometric components implicit, *i.e.,* in their minds. Further, the functional structures remain informal, *i.e.,* there is no mathematical theory that gives these models an unambiguous unique meaning. Consecutively, the knowledge of such design catalogs, *e.g.,* [KK98, Rot01], is not provided in an unambiguous format, and experts may not be able to use the knowledge efficiently (*cf.* Section 5.1.3).

5.1.1 Functional Structures

In mechanical design theory, a functional structure describes the function of a system as a network of smaller functions. The task of a design engineer is, to conceptualize a physical product that is manufacturable and fulfills the defined function [Ruc17]. Mechanical engineers have to do so implicitly by enforcing physical phenomena to occur which is only possible by manipulating the shape, *i.e.,* form, geometry, material and state of the physical system [Ruc17]. Therefore, mechanical engineers understand a functional specification as a design task, *i.e.,* the task of finding a physical product defined in terms of geometry, material, and physical properties that implements the function. Here, the term function refers to the general and desired relation between the input and output of a system to fulfill a task [BG21]. In this dissertation, we will refer to the functions considered in mechanical engineering as physical functions. In the following, we use the terms “function” and “design task” interchangeably.
The idea of a physical function is strikingly similar to the notion of systems and functions formalized in the theory of TSPFs [BS01, Bro12]: A mechanical system or a part of it is delimited by a boundary, through which physical quantities can enter and leave the system as *functional flows* [BG21]. The *function* of the delimited system processes the incoming functional flows to the outgoing flows. Functional flows are distinguished as flows of *energy*, *matter*, and *data* [PBFG07, UE03]. Figure 1.2 from [Kol85] illustrates this notion and underlines the similarities to the notion of a system formalized by the theory of TSPFs [BS01, Bro12]. Due to these similarities, we can safely consider functions targeted by mechanical engineering as CPFs in the sense

of Definition 3.19. The function of a mechanical system breaks down into several sub-functions that interact by exchanging functional flows [BG21] which implements the principle of composition (*cf.*Section 3.3.3) from the functional development paradigm. The functional synthesis which is the first step in the design process [Kol98] is concerned with decomposing the overall function into sub-functions. These impose an easier-to-solve design task. The result of the functional synthesis is a functional structure that is visualized as a compositional functional specification in the sense of Section 3.4.3. However, in these descriptions, the functional behavior is not expressed by the formalism presented in Section 3.4 but informally through images or textual notation. In [KK98, Kol98, BG21], what we would call the behavior of these functions,*i.e.,* a description of what the function does, is described by the scheme

$$\langle \text{verb} \rangle \ \langle \text{substantive} \rangle \tag{5.1}$$

where the substantive specifies the physical quantity that is transformed and the verb specifies in what way the quantity is transformed, *i.e.,* it specifies the activity of the system under development whose function is specified. The latter is also referred to as the *operation* of the function in [Kol85]. The main function of the coolant pump in the automotive cooling system (*cf.* Section 1.3.3) is to "apply a fluid with mechanical energy". In this example, the operation is "apply"‘ and the physical quantities are "fluid" and "mechanical energy". The function imposes the incoming mechanical energy (P_{mech} in Figure 1.3) upon the incoming cooling medium, such that the cooling medium leaving the system boundary has a higher pressure. Figure 5.1 shows a graphical notation of a physical function used in [Kol98, BG21]. Typical graphical notations [BG21, Rot02, Rod91, Kol98] assimilate C&C-notations with boxes representing functions and arrows between these boxes representing the functional flows. The type of flow is indicated solely by a more or less standardized name that is attached to the arrow.

5.1.2 Design Catalogs, Elementary Functions and Principle Solutions

Mechanical design methodology has introduced design catalogs to further systematize the product development by storing proven solutions for recurring design tasks [BG21, GA09]. These design catalogs provide a "collection of known and established solutions to design tasks or sub-functions" [BG21]. The definition, structure, and kinds of design catalogs have even been standardized by the German VDI in [VDI91]. At the end of the functional synthesis, the methodology in [Kol85] proposes to consult such a design catalog and describe the sub-functions as composition of the functions listed in the design catalog to enable reusing the stored knowledge for finding suitable solutions. Design catalogs typically provide ideas for or building blocks of conceptual solutions to recurring functions. Catalogs with different

targets exist, *e.g.,* for machine elements [Rot94], or mechanic connections [Rot96]. The Koller catalog [KK98], which lists 350 physical effects that are mapped to the elementary functions they are suited to fulfill, is a popular contribution in this field.

Elementary Functions Research in the field of mechanical design theory has commonly come to the conclusion that there exists a finite set of basic operations [KK98] a system can perform to transform energy, matter, and data [BG21, Rot00, Rod91, Kol98]. In [KK98, Kol98], a basic operation is an "activity in the course of a physical process that cannot be further subdivided into different activities" [Kol98]. According to [Kol98], these operators assimilate the arithmetic and Boolean operators from which electrical circuits and at their lowest technical level software systems are built. An elementary function is then defined as a function described by a basic operation together with the *physical quantities* the function shall transform in the scheme of Equation 5.1 [KK98, Kol85]. For elementary functions Koller introduces a set of symbols that denote these basic operations in the graphical notation. Figure 5.2 shows the symbol for the basic operation "convert" which is a diagonal line from the lower left bottom to the upper right bottom of the box that represents the function.
The methodology proposed in [Kol98] is based on the assumption that there exists only a finite set of elementary functions and that all mechanical systems can be defined as a composition of such elementary functions. The leaves of a functional structure are functions that are not further decomposed but the types (informally described by names) most likely describe a type of energy or material but not a single physical quantity. Even though such functions are not listed in the design catalog [KK98], they are conceived as elementary functions by the methodology. To utilize the knowledge from a design catalog, the engineer has to find an elementary function that is listed, *e.g.,* in Koller's design catalog [KK98] that lists a set of physical effects.
To this effect, we propose the following definitions:

Definition 5.1 (Elementary Function)**.** *An elementary function is a CPF that is not composed of other functions and there exists a finite set of physical effects to realize the function.*

Typically, there is more than one physical effect with which a single elementary function can be realized.
In both, the graphical and the textual notations sketched in Figure 5.1 or Figure 5.2 and Equation 5.1 proposed in [Kol98, KK98], channels are not explicitly typed. The kind of arrow indicates the kind of stream, *i.e.,* energy, matter, or data, and the quantity is derived implicitly by the name of the channel. elementary functions in [KK98] solely process physical quantities (which are primitive types) and not energy or matter which are composite types. To find appropriate elementary functions to

decompose a leaf-function in a functional structure requires the engineer to apply implicit knowledge of the physical quantities that define a type of energy or material. This challenges the application of design catalogs in practice which is further detailed and illustrated by an example in Section 5.1.3.

Principle Solutions In [Kol85], a principle solution defines a physical principle together with a geometric structure to realize a function [Kol98]. Therein, a physical principle to realize an elementary function is determined by a physical effect and an effect carrier. The former describes a physical phenomenon that realizes the transformation specified by the elementary function while the latter specifies the material or space to realize the elementary function [Kol85]. A physical effect describes a physical implementation of the behavior of an elementary function.

Definition 5.2 (Physical Effect [Kol85])**.** *A physical effect is a physical phenomenon, physical occurrence, or process of a physical event and provides a causal relation between the inputs and outputs of a function.*

In mechanical engineering, physical effects can be quantified through physical laws [BG21, Kol85, Rot00]. physical effects are quantified by physical laws, which can be seen as standardized equations or, in our terminology, standardized interface assertions for the behavior of solutions.

Definition 5.3 (Physical Law [Rot00])**.** *A physical law is a quantitative relation between physical quantities that involves material constants under certain circumstances.*

The material constants in the quantitative relation of a physical law reference properties of the geometric interface of the component. This is where the methodology closes the problem-implementation gap (*cf.*Section 1.1.1): The geometric interface is composed of so called *active surfaces*. The physical effect acts between these surfaces to implement the desired functionality. From a geometric point of view, the cyber-physical streams of which the CPSs interface is composed, enter the function through these surfaces.
For the conversion of electrical to mechanical energy in the automotive cooling system (see Section 1.3.3), for example, the effect catalog lists, *e.g.,* the *Biot-Savart effect* (*cf.* paragraph 7.2.3). A principle solution defines how a physical effect, given a qualitative geometry with certain material properties [BG21] implements an elementary function. In our modeling methodology, we understand the elementary function as a component that encapsulates the implementations describes by one to many principle solutions. Therein, a principle solution refines the functional behavior by adjusting its specification such that it represents one of finitely many physical effects and also offers an basic geometric implementation. There exist various ways to describe geometry mathematically, *e.g.,* through quadrics [Fis13] or B-splines [DR08].

As the specification of the geometric view of a CPF is out of scope of this dissertation, we will not provide a formal technique for modeling the geometric parts of a solution. The SysML-profile SysML4FMArch introduced in Chapter 6 that provides a modeling language for physical functions and, in particular, for elementary functions, provides modeling elements to define the geometric parts of a solution as a class whose attributes are typed with physical quantities. The attributes of these parts represent the geometric characteristics of the modeled geometry, *e.g.,* the radius, height, and volume of a cylinder. By associating the names of these attributes to those variables in the physical laws of the physical effect that represent geometric properties, the modeler explicates the relation between the physical effect and the geometry. Chapter 6 provides further details on this. Approaches such as [IJZK23, HJZ+21, ZJS+21] link these modeling elements to CAD models, which most likely have formal semantics based on B-Splines implemented in the corresponding tools.

Definition 5.4 (Principle Solution)**.** *A principle solution is a CPF that refines an elementary function by defining the functional behavior as a physical effect at the interface of the function and offers a basic physical implementation.*

Implementing the principle of composition (*cf.* Section 3.3.3, solutions to composed functions are composed from the solutions of the sub-functions (physical limitations apply, but [BG21] proposes techniques to identify the compatible solutions). The principle solution of the physical function "apply fluid with mechanical energy" in the running example (see Section 1.3.3) could, for example, specify that the selected effect should be implemented with the principle geometry of a rotating paddlewheel mounted within the cylinder through which the fluid flows. Section 6.2.3 provides further details on this matter.

5.1.3 Challenges of the Functional Synthesis

In the design process considered in [Kol85], design catalogs list physical effects to elementary functions that transform streams of physical quantities, energy, or matter. The result of the functional synthesis is a functional structure, *i.e.,* an architectural specification, of the system's function whose leaves are not further decomposed. The types of channels of these leaf functions are energy and material types (because the process considers only physical functions). These are composite types, *i.e.,* the classes that represent them have attributes. Koller's design catalog [KK98], however, lists only functions that process the primitive physical quantities. For applying the Koller catalog [KK98] in the design methodology of [Kol98], the design engineer has to implicitly derive from the informal name of the type of channel which physical quantities the function shall transform [Kol85] to identify useful elementary functions. This process is manual and relies on the knowledge of the engineer, *e.g.,* which physical quantities are energy components of the energy type used in the elementary function's

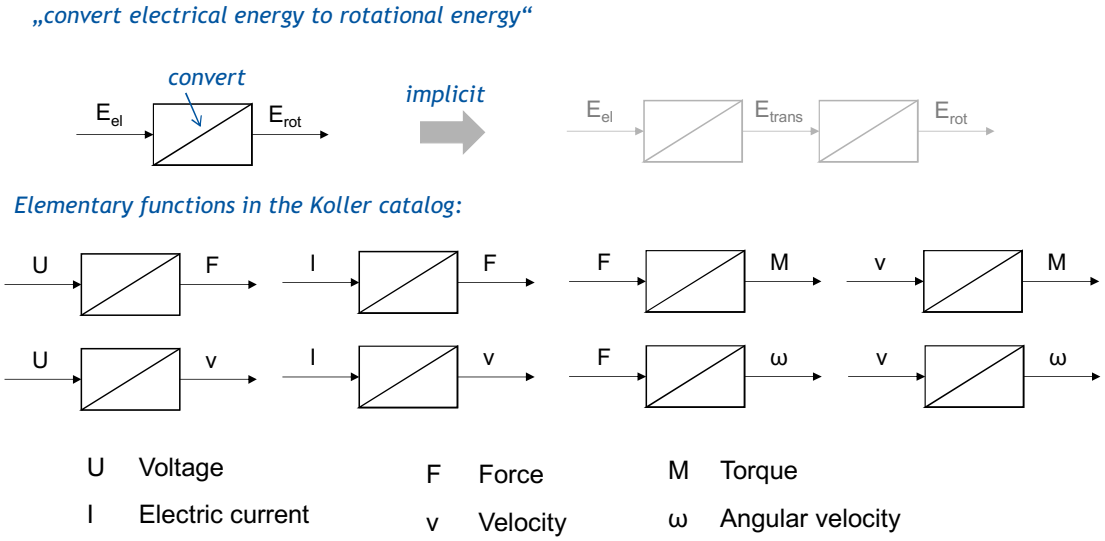


Figure 5.2: elementary functions define transformations of the general quantities energy, and material, while [Kol85, KK98] offer physical effects only to elementary functions that transform (single) physical quantities.

interface. This is similar in other methodologies such as [Rot01], *i.e.,* approaches that aim to systematize the design process still heavily rely on the engineer's knowledge and do not aim to provide automatic support for finding concepts for solutions.

Figure 5.2 illustrates this process on the example of the function "convert electrical to rotational energy". A design engineer knows that electrical energy is bound to quantities like voltage and current, and that rotational energy is bound to torque and angular velocity. The Koller catalog [KK98], however, provides only elementary functions that transform either voltage or current into either force or velocity, respectively. Figure 5.2 shows these functions at the bottom. To obtain physical effects that realize the conversion of electrical to rotational energy, the engineer (mentally) decomposes the original function implicitly into the structure at the top right of Figure 5.2. To do so, the engineer uses his/her knowledge about which quantities the exchange of electrical, translational, and rotational energy are bound to and then chooses a physical effect provided for the quantity transforming functions in [KK98]. To do so in the engineering of complex CPSs, engineers need experience, and the process is time-consuming and its results most often hardly comprehensible for other stakeholders. Modeling the functional flows through typed channels as proposed in Chapter 3 will mend this challenge: The classes that specify types of energy and material hold attributes that describe the physical quantities a CPF can transform. It is, then, clear from the model which quantities are involved in the transformation.

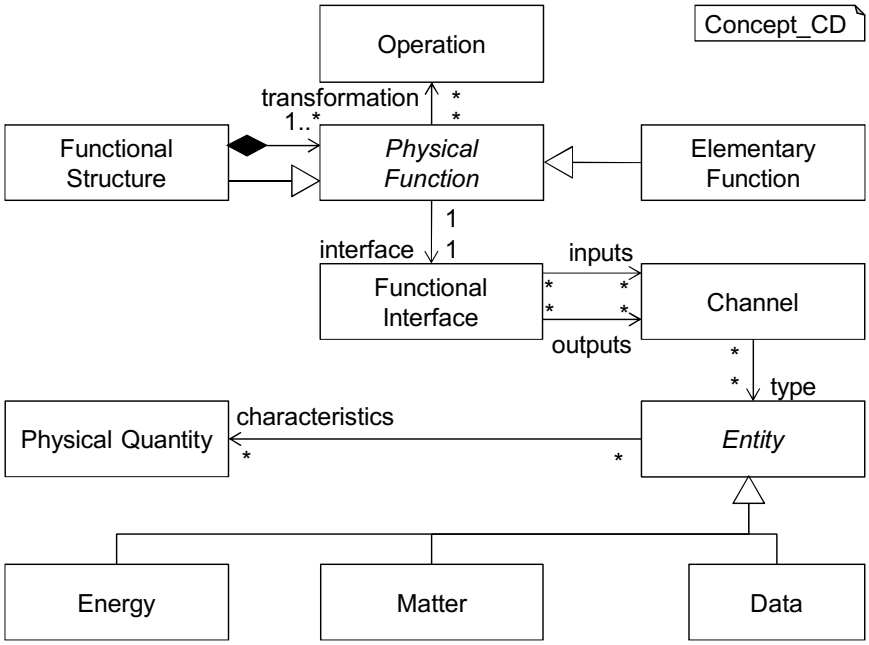


Figure 5.3: Conceptual model that captures the concepts presented in this section. The model uses UML CD notation [Rum16].

5.1.4 Summary: A Conceptual Model of Physical Functions

This section summarizes the concepts from mechanical design methodology in a conceptual model. In Figure 5.3, a *physical function* is described by an *operation* and a *functional interface* that comprises two sets of channels, *i.e.,* the inputs and outputs of the function. The interface represents the physical and logical entry or exit point of a stream of energy, matter or data. Both, the concept of *physical function* and *entity* are abstract concepts. The term elementary function is used ambiguously in [Kol98, KK98]: On the one hand, elementary functions should describe only transformations of physical quantities, but on the other hand the leaf functions of a functional structure whose types are energy or material types are also considered elementary functions. The concept of a *physical function* is abstract to reflect this. Functions can be designated as abstract which leaves the declaration of a function at the structure's leaves as elementary to the modeler. Concrete instantiations of the *entity* concept are *energy*, *matter*, and *data*. The physical entities, *i.e.,* energy, and matter, are typically described by their characteristics which are expressed in terms of *Physical Quantities* (*cf.* Section 2.2.1). These represent the different kinds of streams that physical functions process and the *physical quantities* that describe the inputs and outputs of *elementary functions*. Koller's design methodology [KK98] refers to the operation that

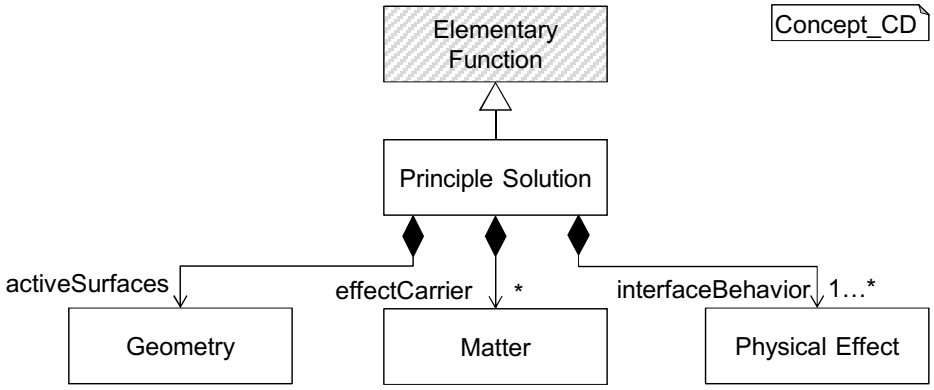


Figure 5.4: Conceptual model that captures the concept of solutions in mechanical engineering. The shaded concepts have been introduced in Figure 5.3.

describes the transformation performed by an elementary function as a basic operation and assumes that the set of all basic operations is finite. A *functional structure* is an architecture of physical functions. In the conceptual model this is represented by the composite pattern [GHJV95]: *physical function* are the components, the leaves are *elementary functions*, and the composite elements are *functional structures*.

Figure 5.4 captures the idea of a solution from mechanical engineering: *elementary functions* describe a set of *physical effects* that are suitable to realize the elementary function. The design catalog [KK98], for example, provides a list of standardized elementary functions together with a set of possible effects for each of these functions. A *principle solution* describes a physical principle to realize a function that compriese a set of *active surfaces* [Kol85, BG21, IJZK23], *i.e.,* a quantitative description of the geometric interface, an *Interface Behavior* which is described by a (set of) *physical effects* that refines the operation that has described the behavior of the elementary function, and the the *effect carrier* that defines the material or space to realize the effect [Kol85] on the active Surfaces.

In this dissertation, we describe physical functions as CPFs in the specification schemes introduced in Chapter 3. The operation is described by the name and parameters of the functional specification. Functional structures are specified using compositional specifications (*cf.* Section 3.4.3) while elementary functions are specified using interface assertions or HIOSs. The input and output quantities of a physical function are specified by the typed channels of the interface of a CPF. The kinds of channel types presented in Chapter 2 are instantiations of *Quantity* (Figure 5.3). The differentiation among kinds of types is more fine-grained than that of the quantities considered in Figure 5.3, because the formal nature of CPFs requires to declare the kinds of streams at each channel.

Example: The Functional Structure of a Pumping System Modeled as a CPF We use the example of a pumping system provided in [Kol85, Kol98] to illustrate how we specify physical functions using the modeling technique presented in the Chapters 2 and 3.
The example further illustrates the design process and how it can be enhanced by using a formal modeling technique, *e.g.,* to overcome the challenge of missing physical quantities outlined in Section 5.1.3.
The example in [Kol98] considers the task of designing a system that transports fluid from one place to another. Additional constraints impose that the driving energy is electrical energy, as a reaction to a signal the system shall turn on or off and the conveying capacity of the pumping system shall be smoothly controllable with respect to the conveying capacity.
The description of the example in [Kol85] reveals that an engineer would infer from these requirements that the pump needs to connect the incoming fluid to motion energy. The input to the function is, therefore, electrical energy, which is specified by the quantities voltage and current [FR76], and a fluid which carries an attribute telling the motion energy. Motion energy is specified by the quantities velocity and momentum [FR76]. Further, we specify an input event that tells whether the system has been turned on or off. The controllability signal will require a signal telling the desired conveying capacity. Table 5.1 shows the respective specifications of these types which have been introduced similarly in Table 2.1.

CPCD

«energy» ElectricalEnergy	«energy» MotionEnergy	«event» ButtonPress	«fluid» Fluid
Current current	Velocity velocity	ON	MotionEnergy eMot
Voltage voltage	Momentum momentum	OFF	

Table 5.1: Specification of the types needed for the compositional pump specification in Figure 5.5.

To specify the behavior of the pump, the overall function of the system, is decomposed into elementary functions. The engineer will infer that in order to get a fluid to move it needs to be connected with motion energy. An elementary function that describes this activity in [KK98] is “connect matter and energy”. At an early stage of the development, it may not be clear which kind of energy is needed for the “connection”, there is the elementary function “transform energy” which transforms the electrical energy that was required to be the driving energy. To indicate this missing piece of information, the type of the channel between the corresponding functions in Figure 5.5 is only the physical quantity Power with SI-unit W (Watt). Since the system is

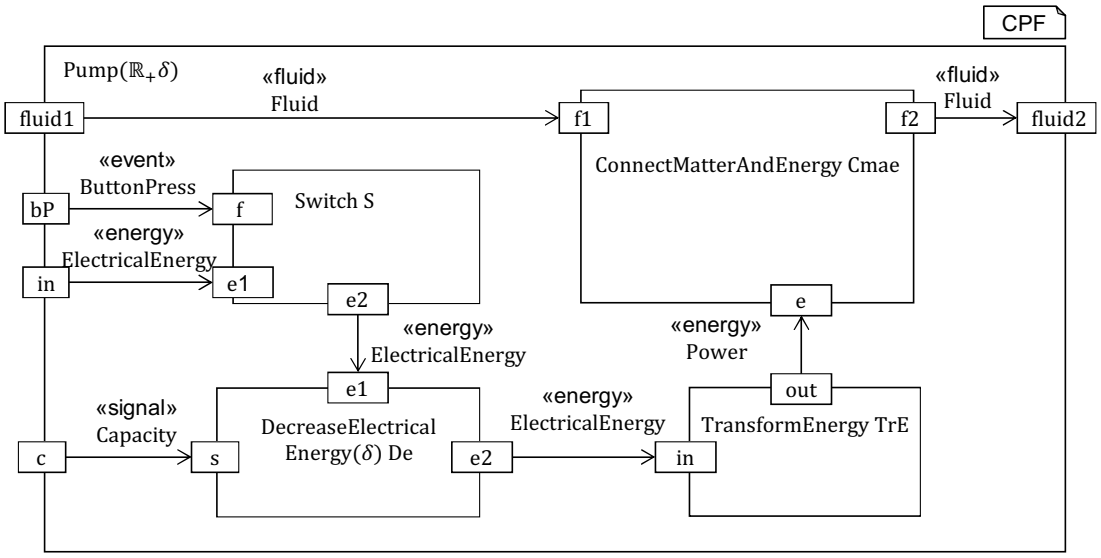


Figure 5.5: Structure of elementary functions of the pump formalized from [Kol85].

supposed to be controlled with respect to the conveying capacity and turned on or off, the compositional structure exhibits the elementary function "decrease energy" from [KK98] and the function "switch". In the conception of mechanical design theory, the switch function is a combination of the elementary functions conducting and isolating energy from [KK98], depending on the input signal, or in this case, the input event. Since there are no further details on how to combine these two functions, we utilize a HIOS-specification to specify the switch that turns on or off the electrical stream of energy in Table 5.2. Alternatively, the compositional specification of the switch given in Example 3.7 can also be used to define the switch.

We model the decrease of electrical energy by the CPF specified in Table 5.3 that generates a decreased power at the output port. To specify the elementary function we need to define how the function decreases the power in relation to the incoming Capacity signal. Here, we chose to specify that the power is decreased proportionally to the value of the signal telling the capacity. As capacity is not compatible with power, the variable $p(t)$ models a conversion factor. Note, that this is already a design decision.

The function *TransformEnergy* in Figure 5.5 transforms electrical energy. As it is not clear at this point in the development which form is needed, the output of this function is simply Power. Using an interface assertion and abstracting from all energetic losses, the function is defined in Table 5.4.

Table 5.5 specifies the function *ConnectMatterandEnergy* by an interface assertion.

CPF

Switch

in «event» ButtonPress b, ElectricalEnergy e1
out «energy» ElectricalEnergy e2

Energy $\mathrm{e}(t) = \int_0^t \mathrm{pow}\, e1(t')dt' - \int_0^t \mathrm{pow}\, e2(t')dt'$
Energy $\mathrm{loss}(t) < e(t)$

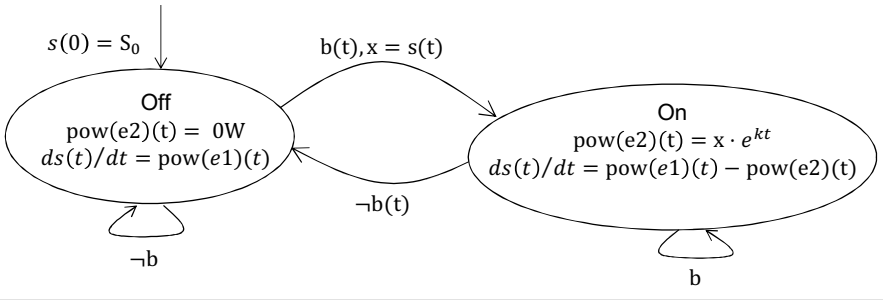


Table 5.2: HIOS specification of the CPF defined by an electrical switch.

CPF

DecreaseElectricalEnergy($\mathbb{R}_+$ δ)

in «energy» ElectricalEnergy $e1$, «signal» Capacity s, out «energy» ElectricalEnergy $e2$
$\forall t \in \mathbb{R}_+$: $\exists p(t) \in \mathbb{R}_+(\mathrm{W\,L\,h^{-1}})$: $\mathrm{pow}(e2)(t+\delta) = \mathrm{pow}(e1)(t) - p(t) \cdot s(t) \cdot t$

Table 5.3: Interface assertion specification of the elementary function that decreases electrical voltage.

CPF

TransformEnergy($\mathbb{R}_+\delta$)

in «energy» ElectricalEnergy in out «energy» Power out
$\forall t \in \mathbb{R}_+$: $\mathrm{out}(t+\delta) = \mathrm{pow}(\mathrm{in}(t))$

Table 5.4: Interface assertion specification of the energy transformation.

The specification abstracts from energetic losses and specifies that the power that drives the fluid is the sum of the power of the incoming fluid and the incoming power.

CPF

ConnectMatterAndEnergy($\mathbb{R}_+$ δ)
in «fluid» Fluid f1, «energy» Power e, out «fluid» Fluid f2
$\forall t \in \mathbb{R}_+$: $\text{pow}(\text{f2.eMot})(t+\delta) = \text{pow}(\text{f1.eMot})(t) + \text{pow}(e)(t)$

Table 5.5: Interface assertion specification of the energy transformation.

To be considered elementary functions that are also listed in [KK98], the channel types will have to be changed to reference only physical quantities. We illustrate this development step on the example of *DecreaseElectricalEnergy* which is replaced by the function *DecreaseVoltage* specified in Table 5.3. From the specification of the type

CPF

DecreaseVoltage($\mathbb{R}_+$ δ, $\mathbb{R}(\text{V h L}^{-1})$ v)
in «energy» ElectricalEnergy $e1$, «signal» Capacity s, out «energy» ElectricalEnergy $e2$
$\forall t \in \mathbb{R}_+$: $e2(t+\delta).\text{voltage} = e1(t).\text{voltage} - v \cdot s(t) \cdot t \wedge$ $e2(t+\delta).\text{current} = e1(t).\text{current}$

Table 5.6: Interface assertion specification of the elementary function that decreases electrical voltage.

ElectricalEnergy in Table 5.1 we infer that the energy components of electrical energy are *current* and *voltage*. The Koller catalog [KK98] lists effects to increase either current or voltage. At this point, we make the design decision that we will realize the function by decreasing the voltage and keeping the current constant. To this effect, we create the CPF specification *DecreaseVoltage* in Table 5.6 that adds two statements to the interface assertion of the component *DecreaseElectricalEnergy*. The second statement specifies that the voltage of the outgoing stream of electrical energy shall decrease proportionally to the capacity signal on the input port s. The third statement specifies that the outgoing current is equal to the incoming current. For this, the function yields a parameter v that represents a conversion factor from capacity to voltage.
We check whether the component *DecreaseVoltage* in Table 5.6 refines the component

DecreaseElectricalEnergy in Table 5.3. To this effect, we show that the interface assertion of *DecreaseVoltage* implies the interface assertion of *DecreaseElectricalEnergy*, *i.e.,* we show that for all $t \in \mathbb{R}_+$ it holds that

$$e2(t+\delta).\,\text{voltage} = e1(t).\,\text{voltage} - v \cdot s(t) \cdot t \wedge \tag{5.2}$$

$$e2(t+\delta).\,\text{current} = e1(t).\,\text{current} \Rightarrow \tag{5.3}$$

$$\exists p(t) \in \mathbb{R}(\mathrm{W\,h\,L^{-1}}) : \text{pow}(e2)(t+\delta) = \text{pow}(e1)(t) - p(t) \cdot s(t) \cdot t \tag{5.4}$$

where

- $e1, e2 \in [\mathbb{R}_+ \to \text{ElectricalEnergy}]$,
- $s \in [\mathbb{R}_+ \to \text{Capacity}]$,
- $v \in \mathbb{R}(\mathrm{V\,h\,L^{-1}})$.

The definition of the property function pow defined in Section 2.2.2 yields for all $e \in \text{ElectricalEnergy}$ and all times $t \in \mathbb{R}_+$ that

$$\text{pow}(e)(t) = e.\,\text{voltage}(t) \cdot e.\,\text{current}(t). \tag{5.5}$$

With that, we get that

$$\begin{aligned}
\text{pow}(e2)(t+\delta) &= e2(t+\delta).\,\text{voltage} \cdot e1(t+\delta).\,\text{voltage} && (5.6)\\
&= (e1(t).\,\text{voltage} - v \cdot s(t) \cdot t) \cdot e1(t).\,\text{current} && (5.7)\\
&= \text{pow}(e1)(t) - v \cdot e1(t).\,\text{current} \cdot s(t) \cdot t && (5.8)\\
&\stackrel{\text{def}}{=} \text{pow}(e1)(t) - p(t) \cdot s(t) \cdot t. && (5.9)
\end{aligned}$$

5.2 Formalizing the Koller Design Catalog

The Koller design catalog [KK98] is a design catalog (*cf.* Section 5.1.2) that lists physical effects to realize elementary functions. Because physical effects are quantified by physical laws which are expressed as mathematical formulae, they provide a formal relation between the input and the output of a CPF and therefore define a behavior. As physical effects most often reference variables that represent geometric characteristics, *e.g.,* length, mass, or the number of windings of an electric coil, they do not abstract entirely from the geometric view encapsulated by a CPF. Therefore, enhancing the behavioral specification of a CPF by a physical effect yields a specification that is a step closer to an implementation. Because the set of physical effects is finite, it is possible to distinguish specifications as functions or solutions. A functional specification becomes a solution once it uses physical effects from the finite set to specify the behavior. Applying the design methodology proposed in [Kol98], the

set of physical effects is listed in the design catalog [KK98]. In this dissertation, we focus on the functional specifications of the physical functions that are targeted by the development of mechanical engineering and how to link functional specifications to physical effects or models of solutions. In this section, we use the modeling technique introduced in the Chapters 2 and 3 to formalize the elementary functions listed in the Koller catalog [KK98].
The Koller catalog [KK98] uses a scheme that categorizes elementary functions according to the kinds of types of the channels in their interface.

Definition 5.5 (Interface Category of CPFs)**.** *The behavior of an elementary function*

1. *is an* energy operation*, iff its interface includes only channels typed with an energy type,*
2. *is a* material operation*, iff its interface includes only channels typed with a material type,*
3. *is an* operation between energy and material *iff its interface includes both channels typed with an energy-type and channels typed with a material type.*

The Koller catalog [KK98] relies on further dividing these categories by the basic operations of the elementary functions, *i.e.,* the specification of the transformation to be performed through a physical process by the system [Kol98]. As detailed in Section 5.1.2, these operations are standardized through symbols that represent verbs which describe these activities verbally.
The following sections will formalize this informal representation by specification schemes and examples that translate the symbol into a possible interface assertion. The interface assertions that we propose are only one possible solution to translating the symbols proposed in [KK98] to mathematical formulae. To this effect, we enhance the specification scheme given in Table 3.2 which uses placeholders for the types of the input and output channels by generic types that serve as placeholders for concrete types and are added in a list within angled brackets after the name of the CPF. Table 5.7 illustrates the notation of such schemes. The generic types enable to generalize the idea of operations from [KK98] and make one interface assertion available for different types of the input and output channels. In Koller's methodology [KK98], providing the concrete types of energy that shall be transformed by a standardized operation makes up an elementary function definition. Since the translation of verbally defined elementary operations and functions is most likely subjective, each scheme defines only a possible interface assertion into which the symbol can be translated. All interface assertions provided are given in a timed manner and use delay to assure causality even if these functions are utilized in feedback composition settings (*cf.* Section 3.1.1). Section 5.3 provides a discussion on specification styles that regard or disregard energetic losses or delays.

CPF

name n ⟨list of generic types⟩ (list of parameters P)
in list of typed input channels I out list of typed output channels O
interface assertion p

Table 5.7: Specification scheme for CPFs with interface assertions based on [Bro10].

5.2.1 Energy Operations

The energy operations listed in [KK98] are

1. Convert: Change one form of energy into another,
2. Increase/decrease: Increase or Decrease the scalar value of an energy component,
3. Change direction: Change the direction of a vectorial energy component,
4. Conduct: "Pave the way" to transfer a stream of energy from one place to another,
5. Isolate: Prevent energy from acting in a specific space,
6. Collect/split: Bring two or more streams of energy together quantitatively, and
7. Separate/blend: Sort / bring together streams of energy of differentiating properties (*e.g.*,, frequency, wavelength) properties.

Energy Conversion Converting energy refers to any activity of a system that converts one form of energy or one energy component into another [KK98]. Similar to [FR76], we use the power balance which is derived from the law of energy conservation to specify the conversion of one form of energy into another. For the conversion of energy, we provide two specification schemes: Let E1 $\in \mathcal{E}$ and E2 $\in \mathcal{E}$ be two energy types. A function converts energy of type E1 to energy of type E2 iff it has an input port $e1$ with type($e1$) = E1 and an output port $e2$ with type($e2$) = E2 such that the power at the outgoing energy channel $e2$ after a time delay δ is equal to the power that was given as input to the function via the channel $e1$. Table 5.8 provides a scheme that uses the power balance to provide a standardized interface assertion that formalizes this idea. Another way to specify elementary functions that convert one form of energy to another is to utilize a state variable that represents the energy content of the specified system at all times. A corresponding scheme is provided in Table 5.9. Integrating the power that is put into the system until time t and subtracting the power that leaves

ConvertEnergy$\langle E1, E2 \in \mathcal{E}\rangle(\mathbb{R}_+\ \delta)$ CPF
in «energy» E1 $e1$ out «energy» E2 $e2$
$\forall t \in \mathbb{R}_+ :$ $\text{pow}(e2)(t+\delta) = \text{pow}(e1)(t)$

Table 5.8: Specification scheme for elementary functions that convert the energy of one form into another that utilizes the power balance and abstracts from energetic losses.

the system until time $t + \delta$ via the output port yields this value. To specify the conversion, the scheme states that the value of the state variable must never be smaller than 0W which is reasonable because there is no such thing as a negative energy content. The delay allows the function to store energy for a time period of δ which is a parameter of the function.

ConvertEnergyStateBased$\langle \text{E1}, \text{E2} \in \mathcal{E}\rangle(\mathbb{R}_+\delta)$ CPF
in «energy» E1 $e1$ out «energy» E2 $e2$
$s \in \text{Energy}$ $\forall t \in \mathbb{R}_+ :$ $s(t) = \int_0^t \text{pow}(e1)(t') - \int_0^{t+\delta} \text{pow}(e2)(t')dt'$ $s(t) \geq 0\text{W}$

Table 5.9: Scheme for the specification of elementary functions that convert energy using an internal variable that represents the function's energetic state.

An example of an elementary function in the category of energy converters is the electric drive that is specified in Table 3.3. The component *ElectricDrive* transforms ElectricalEnergy into RotationalEnergy and uses the specification scheme from Table 5.8. The interface assertion of the CPF uses the power balance: The power generated at the output channel r by transmitting rotational energy at time $t + \delta$, where $\delta > 0$ is a time delay must be equal to the power at the input channel at time t.

Increasing/Decreasing Energy Increasing or decreasing energy refers to activities of a technical system that increase or decrease the value of an energy component (*cf.* Section 2.2.2). Here, we abstract from this information. Section 5.3 details how,

e.g., energetic losses can be included in a specification. Examples are systems that increase or decrease torque, rotational velocity, or electrical transformers [Kol85]. Saying that a function increases or decreases a stream of energy is not exactly meaningful in a physical sense, since energy is a conserved quantity and can therefore neither be created nor destroyed [FR76]. There are, again multiple possibilities to define interface assertions to specify such functions. From a functional perspective, it is often not (yet) relevant which energy component is increased or decreased, what is relevant is that the function generates more or less power. In this case, the specification scheme in Table 5.10 provides a suitable formalization.

CPF

IncreasePower$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+ \delta)$	DecreasePower$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+ \delta)$
in «energy» E $e1$ out «energy» E $e2$	in «energy» E $e1$ out «energy» E $e2$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pow}(e2)(t+\delta) > \mathrm{pow}(e1)(t)$	$\forall t \in \mathbb{R}_+ :$ $\mathrm{pow}(e2)(t+\delta) < \mathrm{pow}(e1)(t)$

Table 5.10: Schemes for the specification of elementary functions that increase or decrease the power transmitted by a stream of energy.

Another scheme to specify the increasing or decreasing power of a system can be derived from the *PowerAmplifier* defined in Example 3.3. In this specification, an additional input signal provides the value by which the power shall increase in the time interval $[t, t+\delta]$. The scheme in Table 5.11 generalizes the CPF specification from Example 3.3. The interface assertion specifies that the slope of the outgoing power on the time interval $[t, t+\delta[$ is the last value received in the time interval $[t-\delta, t[$. If the function has not received a value in this time interval the outgoing power is equal to the incoming power.
CPFs that increase or decrease a physical quantity can be specified similarly. Table 5.12 provides a specification scheme for elementary functions that increase or decrease the value of a physical quantity. The example presented in Section 5.1.4 illustrates how these specifications can be used to refine a function at the leaves of a functional structure. Let $Q_p \in \mathbb{D}^n[SI]$ be a physical quantity and $p \in \{1, 2, 3\}$ indicate the dimensionality of the quantity. Example 5.1 provides an example for the specification of an elementary function that decreases torque that uses the specification scheme from Table 5.12.

Example 5.1 (Decreasing Torque)**.** *A gearbox is a very common part of mechanical systems that contain an engine. Most often, the quantities of the energy components of the form of energy generated by the engine need to be increased or decreased to assure proper functioning of the other system components. Consider, for example, a wind*

CPF

IncreaseDecreasePowerControlled$\langle \mathrm{E} \in \mathcal{E} \rangle (\mathbb{R}_+ \delta)$

in «energy» E $p1$, «data» $\mathbb{R}$ x out «energy» E $p2$
$\forall t \in \mathbb{R}_+ :$ $p2(t+\delta) = p1(t) \cdot lt(x,t)$ where $T(x,t) = \{n \in TD_x \mid n \leq t - \delta\} \neq \emptyset \Rightarrow lt(x,t) = x(\max(T(x,t)))$ $T(x,t) = \emptyset \Rightarrow lt(x,t) = 0$

Table 5.11: Specification scheme for elementary functions that increase or decrease power determined by a control input.

CPF

IncreaseQuantity$\langle \mathrm{Q_p} \in \mathbb{D}^n[SI] \rangle (\mathbb{R}_+ \delta)$

in «energy» $\mathrm{Q_p}$ $e1$ out «energy» $\mathrm{Q_p}$ $e2$
$\forall t \in \mathbb{R}_+ :$ $\|\|q2(t+\delta)\|\|_p > \|\|q1(t)\|\|_p$

DecreaseQuantity$\langle \mathrm{Q_p} \in \mathbb{D}^n[SI] \rangle (\mathbb{R}_+ \delta)$

in «energy» $\mathrm{Q_p}$ $q1$ out «energy» $\mathrm{Q_p}$ $q2$
$\forall t \in \mathbb{R}_+ :$ $\|\|q2(t+\delta)\|\|_p < \|\|q1(t)\|\|_p$

Table 5.12: Schemes for the specification of elementary functions that increase or decrease the value of a physical quantity.

turbine as defined in [MNN+22]. The rotor transforms wind energy into rotational energy whose energy components are torque and velocity. In these systems, the torque generated by the rotor is very high, while the angular velocity is low. The generator, which is a component of the wind turbine that converts the rotational energy to electrical energy, however, needs high angular velocity and lower torque. The gearbox is the part between the rotor and the generator that decreases the torque and increases the angular velocity. The component should be designed such that it keeps the power constant, i.e., it should minimize energetic losses. Of course, losses occur in this process but we abstract from these losses here. The functions DecreaseQuantity$\langle$Torque$\rangle$ *and IncreaseQuantity*$\langle$AngularVelocity$\rangle$ *are elementary*

functions that define suitable CPFs.
*The specification in Table 5.13 defines a CPF that takes a rotational energy as input and generates a rotational energy as output. The interface assertion states that the torque at the output channel $q2$ is reduced while remaining the power constant compared to the incoming power on channel $q1$. The interface assertion of this specification is taken from DecreaseQuantity⟨*Torque*⟩.*

CPCD	Gearbox($\mathbb{R}_+$ δ) CPF
«energy» RotationalEnergy	in «energy» RotationalEnergy $r1$ out «energy» RotationalEnergy $r2$
Torque τ AngularVelocity ω	$\forall t \in \mathbb{R}_+ :$ $\text{pow}(r1)(t) = \text{pow}(r2)(t + \delta)$ $\|\|r1(t).\tau\|\|_3 \geq \|\|r2(t + \delta).\tau\|\|_3$

Table 5.13: Specification of an elementary function that decreases the torque in a rotational energy.

Changing the Direction of a Stream of Energy Besides a unit and a value, vectorial physical quantities, such as force or momentum, have another attribute to indicate direction. An operation that physical systems perform, is changing the direction of vectorial physical quantities. For example, another function defined by a gearbox reverts the rotational direction of a gear, and thereby the direction of the angular momentum and the rotational velocity with which the gears rotate. Mathematically, changing the direction of a vector $x \in \mathbb{C}^n$ $(n > 1)$ corresponds to a linear transformation $f(x) = A \cdot x$, for a matrix $A \in \mathbb{C}^{n \times n}$. In Euclidean space which is an accurate model of our physical world [Esc20], changing the direction of a vector corresponds to a rotation. To specify this, we use the special orthogonal group $SO(n) = \{A \in GL(n, \mathbb{R}) \mid \det(A) \in \{1, -1\}\}$ as a primitive type, where $GL(n, \mathbb{R})$ is the general linear group that contains all invertible matrices with entries in $\mathbb{R}$ (see Appendix B for the definitions) [Fis13]. The special orthogonal group $SO(n)$ contains all matrices that rotate a vector around an axis. Similar to the elementary functions that increase or decrease energy, we can define schemes that specify the change of the direction of vectorial physical quantities and elementary functions that use analog interface assertions to specify the change of an energy component.

Example 5.2 (Changing the Direction of Torque)**.** *Torque and rotational velocity are vectorial physical quantities that define rotational energy [FR76]. In physical systems, a common task is not only to increase or decrease torque, or rotational velocity but also to change their direction. In a setting where the input is typed by the physical quantity,*

CPF

ChangeDirectionofQuantity$\langle \mathrm{Q_p} \in \mathbb{D}^n[SI], p \in \mathbb{N}\rangle(\mathbb{R}_+\delta,\ SO(p)A)$
in «energy» $\mathrm{Q_p}$ $e1$ out «energy» $\mathrm{Q_p}$ $e2$
$\forall t \in \mathbb{R}_+ :$ $e2(t+\delta) = A \cdot e1(t)$

Table 5.14: Schemes for the specification of an elementary function that changes the direction of a physical quantity.

the functions ChangeDirection$\langle \mathrm{Torque_n}, n\rangle(\mathbb{R}_+\delta, SO(p)A)$ *for* $n \in \{1,2,3\}$ *obtained from the scheme in Table 5.14 provide a CPF specification of this elementary function. For example, inverting the direction along the X-axis can be specified as*
ChangeDirection$\langle \mathrm{Torque_3}, 3\rangle(\mathbb{R}_+(\mathrm{W})\ \varepsilon, \mathbb{R}_+\ \delta, SO(3)\ A)$ *for* $A = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} A$
cogwheel transmission, for example, inverts the direction of the torque [KK98].
Table 5.15 specifies a CPF that changes the direction of the energy component torque of rotational energy. The specification uses the same interface assertion from the scheme in Table 5.14.

CPCD

«energy» RotationalEnergy
$\mathbb{R}^3(\mathrm{N\,m})\,\tau$ $\mathbb{R}^3(\mathrm{rad\,s^{-1}})\,\omega$

CPF

ChangeTorqueDirection$(\mathbb{R}_+\ \delta, SO(3)\ A))$
in «energy» RotationalEnergy r out «energy» RotationalEnergy r'
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pow}(r)(t) = \mathrm{pow}(r')(t+\delta)$ $r'(t+\delta).\tau = A \cdot r(t).\tau$

Table 5.15: Specification of a CPF that changes the direction of torque.

Energy Conducting Conducting energy is another elementary operation defined in [KK98] and refers to bringing a stream of energy from one place to another. This type of operation refers much more to enabling a system to conduct energy, and not directing a stream of energy along a specified path [Kol85]. Pipes or cables are (parts of) systems that enable, *e.g.,* water, or electric energy to flow, while at the same time directing these streams along a defined path.
The specification scheme in Table 5.16 formalizes this notion. In the scheme, the start

and end position are parameters of the CPF. The interface assertion requires the power that is generated at the output port to be equal (abstracting from losses) to the incoming power which formalizes the notion that the function enables a system to conduct energy.

CPF

ConductEnergy$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+\delta,$ Position start, Position end)

in «energy» E $e1$ out «energy» E $e2$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pow}(e1)(t) = \mathrm{pow}(e2)(t+\delta)$ $\mathrm{pos}(e2(t)) = \mathrm{start} \wedge \mathrm{pos}(e2(t+\delta)) = \mathrm{end}$

Table 5.16: Scheme for the specification of an elementary function that conducts energy.

A specification of the function Conducting electrical energy, for example, which is a CPF defined by a cable, is ConductEnergyConductEnergy$\langle$ElectricalEnergy$\rangle(\mathbb{R}_+\delta,$ Position start, Position end).

Energy Isolating Isolating energy means a CPF that prevents energy convection from one place to another [Kol98]. In [KK98] the isolation of energy has no output ports but only one energy input port. The corresponding interface assertion in Table 5.17 prohibits the position of the incoming energy to be in a list of positions that are a parameter of the function. For mechanical engineers, storing energy is equivalent to

CPF

IsolateEnergy$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+\delta, List < \mathrm{Position} > P)$

in «energy» E e out $\emptyset$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pos}(e)(t) \notin P$

Table 5.17: Scheme for the specification of an elementary function that conducts energy.

isolating energy [Kol98]. That is because preventing energy from convecting from one place to another requires the system to absorb energy that "tries" to convect in the "wrong direction". A spring, for example, stores energy through its elastic deformation [Kol98]: The spring stores the displacement energy applied to it by the force that causes its extension to increase. The spring stores this energy as long as the extending force prevails. Once this force decreases, the spring releases the energy,

again, in the form of displacement energy, by decreasing the extension. In this dissertation, we specify the storage of energy a little differently: In practice, systems that store energy are requested to release the stored energy at some point. The interface assertion in the scheme in Table 5.18 therefore holds an internal variable telling the energy content of the system that is required to be greater than zero Joule at all times. The energy content is determined by the amount of energy (integral over time of the power) received via the input port minus the amount of energy that leaves the system via the output port. The Boolean event input tells whether or not the system shall release the stored energy.

CPF

StoreEnergy$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+\ \delta)$

in «energy» E e, «event» Boolean b out «energy» E e'
$s \in \mathrm{Energy}, \forall t \in \mathbb{R}_+ :$ $0 \leq s(t) = \int_0^t \mathrm{pow}(e)(t')dt' - \int_0^t \mathrm{pow}(e')(t')dt'$ $lt(b,t) \Rightarrow 0 < \int_0^t \mathrm{pow}(e')(t'+\delta)dt' \leq s(t)$ $\neg lt(b,t) \Rightarrow 0 = \mathrm{pow}(e')(t+\delta)$ where $lt(c,t^*) = c(\max\{0 \leq t \leq t^* \mid c(t) \neq \xi\})$

Table 5.18: Specification scheme for elementary functions that store energy.

Table 5.19, Table 5.20, and Table 5.21 provide possible HIOS specifications of an energy source. The specified functions have two states or modes, *i.e.,* `on` and `off`. In the `off` state there is no power generated on the energy output port `e2`. In the state `on` a power is generated on the output port `e2` that decreases exponentially. The modeler can adapt the specified functional behavior by the parameters s_0 and k of the CPFs. In case the stream of power generated on the output port should have a different shape, the HIOS needs to be defined with a different formula for pow($e2$) in the `on` state. The specifications in Table 5.19 and Table 5.20 are quite similar with the exception that the power generated on the output port of the latter does not decrease. The function provides a constant power stream. The specification Table 5.21 also exhibits a charging feature. The CPF has an additional input port and the initial amount of energy s_0 is adapted during the operation: The variable s represents a state variable telling the amount of energy that is currently within the system. In the `off` state, the system can be charged, the amount of energy increases with the power that is put into the system. In the `on` state, the amount of energy is determined by the difference of the incoming and the outgoing power. When the function switches from the state `off` to the state `on`, the value of the state s is adapted to the current amount of energy.

The function defined by a simple tubular battery, for example, can be specified by the component

$$\text{Battery} \stackrel{\text{def}}{=} \text{EnergySource}\langle\text{ElectricalEnergy}\rangle(\mathbb{R}_+\delta, \text{Energy}\, w).$$

Identifying the storage of energy with energy isolation which has no output ports in [KK98] may not suffice in practice: Systems store energy in order to provide it at a different time and/or a different place. This requires energy-storing functions to specify an output port. The electrical switch defined in Example 3.6 or the button switch specification in Example 3.7 are also possible specifications of energy sources.

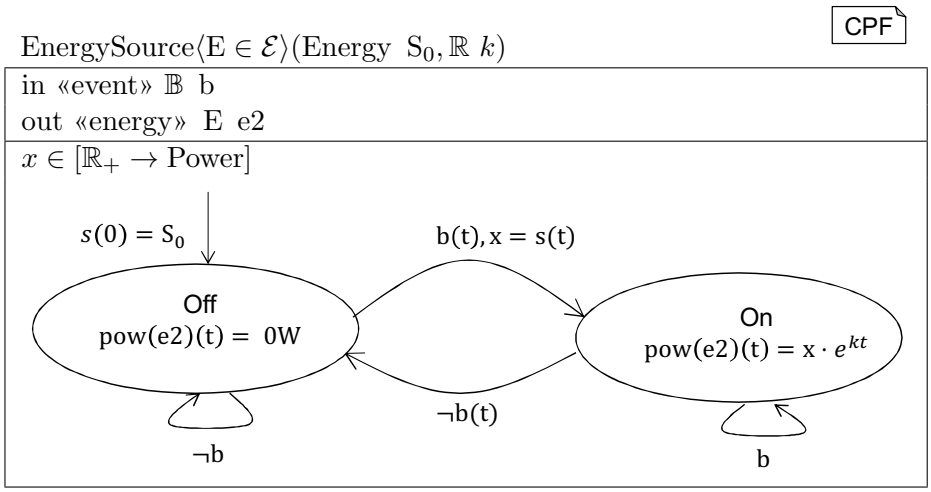


Table 5.19: HIOS specification of an energy source. The power generated at the output port if the source is in the state `off` is zero watts, while it decreases exponentially in the state `on`. The parameters S_0 and k enable modelers to adjust the exponential decrease.

Energy Splitting and Collecting Another elementary operation that can be standardized similar to energy conversion is splitting or collecting streams of energy quantitatively [Kol98, KK98]. That is, functions in this category split a stream of energy, or collect streams of energy such that the sum of the power, *i.e.,* energy per unit of time, that leaves the system via all output ports is equal to the power the system has received via its input port. Differentials, balance beams, or semipermeable mirrors are examples of mechanical devices that split energy [Kol98]. Automotive systems, often require a (mechanical) function to drive two wheels of one axle at different speeds but with the same propelling force, *i.e.,* the same torque. This

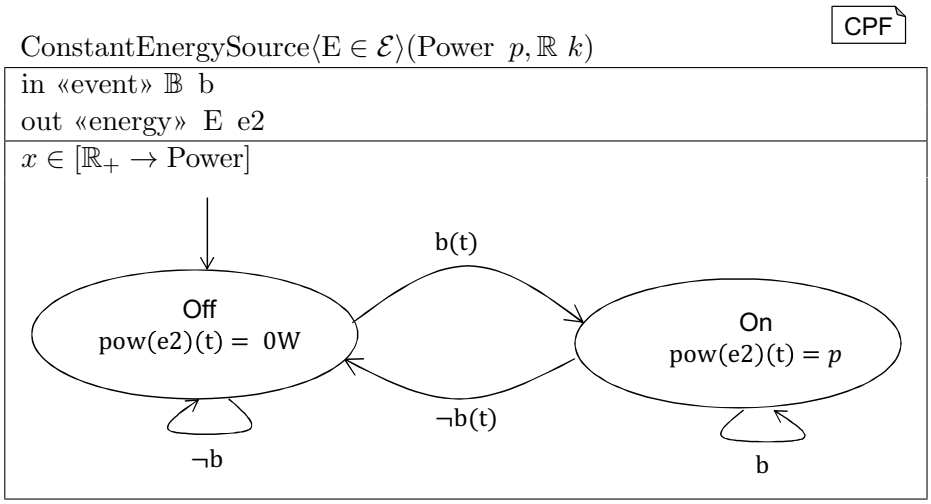


Table 5.20: HIOS specification of a constant energy source. The power generated at the output port if the source is in the state `on` is constant. The amount of power is a parameter of the function.

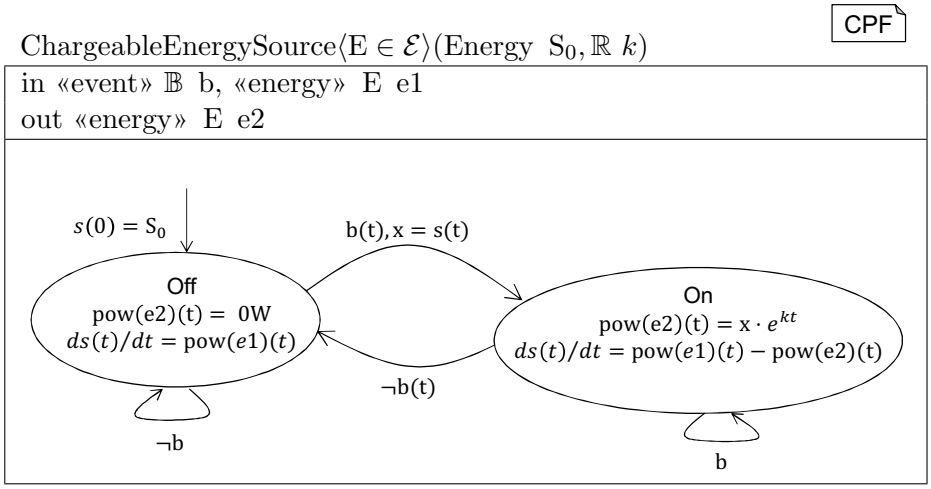


Table 5.21: HIOS specification of a CPF that stores energy and can be charged via an input energy port.

function is an elementary function in the category of energy splitters and defined by the CPF SplitEnergy⟩ RotationalEnergy⟨($\mathbb{R}_+\delta$) obtained from the scheme

in Table 5.22. An alternative specification could provide details, *e.g.,* on the desired ratio between the input and output rotational velocity by adding another input signal.

CPF

SplitEnergy$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+\delta)$

in «energy» E $e1$ out «energy» E $e2$, «energy» E $e3$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pow}(e2)(t+\delta) + \mathrm{pow}(e3)(t+\delta) = \mathrm{pow}(e1)(t)$

CollectEnergy$\langle \mathrm{E} \in \mathcal{E} \rangle(\mathbb{R}_+\delta)$

in «energy» E $e1$, «energy» E $e2$ out «energy» E $e3$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pow}(e3)(t+\delta) = \mathrm{pow}(e1)(t) + \mathrm{pow}(e2)(t)$

Table 5.22: Schemes for the specification of elementary functions that split or collect energy.

Energy separating and blending elementary functions blend two or more streams of possibly different types of energy into one, or separate a stream of energy into two or more streams of possibly different types of energy qualitatively [Kol98]. The qualitative argument by which the streams of energy are distinguished or blended concerns the value of one attribute of the energetic type. As these elementary functions operate on the attributes of the classes defining the respective energy types, we cannot derive a general scheme. Examples include functions that produce periodic energy streams with specific frequencies [Kol98].

5.2.2 Material Operations

In science and technology materials, *i.e.,* the different forms in which matter appears, distinguish by the values of the material constants or their geometric shape. Material constants describe the properties of the material depending on the environment (*cf.* **??**). Recall, that material constants are physical quantities (*cf.* Chapter 2). In Chapter 2, these properties of materials are modeled as attributes of material classes.
Mechanical systems often perform transformations of material that can be described in terms of transformations of these attributes. Material operations describe the categories in which [Kol98] distinguishes the elementary functions that transform material in this way.

The material operations listed in Koller's design catalog [KK98] are

1. Convert: Add or remove a property to or from a stream of material,
2. Increase/decrease: Increase or Decrease the scalar value of a material property (*i.e.,* a physical quantity that describes a property of the material),
3. Mate: Establish cohesion forces between two incoming materials such that they do not diverge,
4. Disconnect: Resolve cohesion forces in a material,
5. Separate / blend: Sort or blend mixtures of substances by qualitative properties,
6. Collect / split: Bring together / set apart a material according to quantitative properties (*e.g.,*, mass, volume), and
7. Conduct / isolate: Enable / prevent a stream of material to get from one place to another.

In contrast to the energy operations, using interface assertions or HIOSs to define material operations requires to reference the attributes of the classes that define a type of material. For materials, the set of attributes is not standardized as it is the case for energy types (these contain two attributes for each energy component which is explained in Section 2.2.2). Therefore, we cannot easily provide specification schemes for these operations. To illustrate how material operations can be formalized through interface assertions, we provide examples for elementary functions that transform material accordingly.

Converting Material According to [Kol98], converting material describes the category of elementary functions that add or remove a property or feature to or from a material. Examples are changing the aggregate phase of a material, or magnetizing a material. Physically, however, these conversions require to increase or decrease one or more properties that describe the material. For example, a system that changes the aggregate phase of water from fluid to solid would decrease the temperature of the water or decrease its density. Phases describe the different manifestations of a material, and the phase of a material depends on environmental conditions defined by specific physical quantities such as pressure, volume, or temperature [Sti18]. So far, it was not possible to describe these conditions through laws, and there only exist empirical observations for different materials [Sti89]. Consider, for example, the boiling point of water which cannot be calculated from the existing physical laws but has been determined through observations obtained in physical experiments [Sti89]. Thermodynamics uses, *e.g.,* material-specific p-v-T diagrams (pressure, volume,

temperature diagram), to determine the phase of a material [Sti89]. This holds similarly for magnetizing a material [Sti89].
So, technical implementations of a material converting function will define functions that increase or decrease the values of the physical quantities that describe the conditions of the environment. Therefore, we do not consider the conversion of material as a stand-alone category of elementary functions, but as a subcategory of increasing or decreasing material.

Example 5.3 (Converting fluid water to steam)**.** *Example 5.4 presents the specification of an elementary function that increases the temperature of fluid water. We can refine this function by adding a statement requesting the temperature after the time delay δ to be at* 373.2 K *which is* $100°C$*, i.e., the boiling temperature of water.*

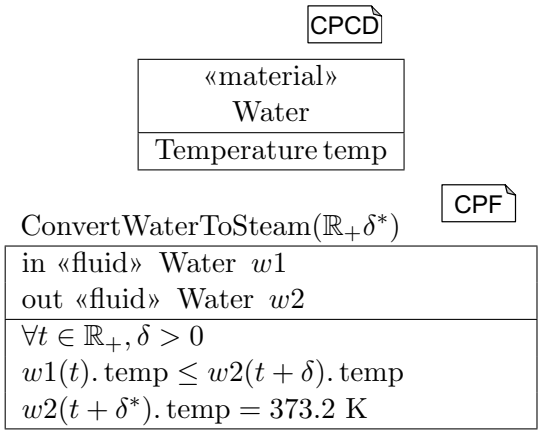


Table 5.23: Specification of the elementary function that increases the density of fluid water.

Increasing or decreasing the scalar of a physical quantity that describes a property of a material is one of the elementary operations, physical systems perform. Material types model a type of matter through attributes such as, *e.g.,* density. elementary functions in this category increase or decrease the value of these properties. Increasing, or decreasing the density, or the electrical conductivity of a material are examples mentioned in [Kol98].

Example 5.4 (Increasing the temperature of Water)**.** *Kettles increase the temperature of fluid water. In this context, we consider a material type to represent water that has an temperature attribute temp measured in the SI-unit Kelvin (*K*). The function increases the value of that attribute.*

Table 5.24 specifies an elementary function that increases the temperature of fluid water, by stating that after a delay δ at time $t + \delta$ the water temperature at the output port is greater than it was at the time t.

CPCD

«material» Water
Temperature temp

CPF

IncreaseTemperatureOfFluidWater($\mathbb{R}_+\delta$)

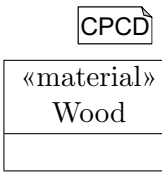
in «fluid» Water $w1$ out «fluid» Water $w2$
$\forall t \in \mathbb{R}_+$ $w1(t).\,\text{temp} \leq w2(t + \delta).\,\text{temp}$

Table 5.24: Specification of the elementary function that increases the density of fluid water.

Mate or disconnect (ger: *fügen*, *lösen*) include elementary functions that establish or resolve cohesional forces among two materials that prevents them from being separated if deseparating forces act on the mated material [Kol98]. Conjunctions of all kinds, *e.g.,* bolted, snapped, or welded, are examples of machine elements that realize the function that mates two materials [Kol98]. Physically, mating two pieces of material means establishing forces between them, such that they do not diverge excessively if a force that aims to separate the two materials is applied [Kol98].
Disconnecting of equal or different materials means resolving the cohesional forces that keep them together [Kol98]. Examples of systems that realize such functionalities are saws, sisels, knives, *etc.* [Kol98].

Example 5.5 (Wood Saw)**.** *A wood saw separates a piece of wood into two. To specify this functionality, the interface assertion in the CPF states that the sum of the volumes of the outgoing two pieces of wood must be equal to the incoming volume. in Table 5.25*

Separate or blend: Separating material means dissolving or sorting a mixture of substances into its constituting substances, while blending material corresponds to creating mixtures of substance [KK98]. Examples of systems performing the former are, *e.g.,* strainers or garbage sorting facilities, examples of material blending systems are blenders, or agitators [KK98].



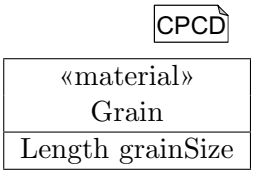
CPF

WoodSaw

in «item» Wood w out «item» Wood $w1$, «item» Wood $w2$
$\forall t \in \mathbb{N}:$ $\text{vol}(w(t)) \neq 0\text{kg} \Rightarrow \text{vol}(w1)(t+1) + \text{vol}(w2)(t+1) = \text{vol}(w)(t)$ $\text{vol}(w(t)) = 0\text{kg} \Rightarrow \text{vol}(w1)(t+1) = 0\text{kg} \wedge \text{vol}(w2)(t+1) = 0\text{kg}$

Table 5.25: Specification of an elementary function that disconnects a piece of wood into two pieces.

Example 5.6 (Strainer)**.** *Strainers sort bulk material, such as grains, by grain size. Therein, the grain size is a diameter, i.e., grain size is described by the physical quantity length whose SI-unit is meter (*m*). The example is based on [BG21]. Table 5.26 specifies the CPF of a strainer that separates bulk material into two streams of material of which one has grains of a size smaller than a fixed mesh size and the other holds grains of a size greater than or equal to the fixed mesh size.*



CPF

Strainer($\mathbb{R}$(m) meshSize)

in «item» Grain g out «item» Grain $g1$, «item» Grain $g2$
$\forall t \in \mathbb{N}$ $g.\,\text{grainSize}(t) < \text{meshSize} \Rightarrow g1(t+1) = g(t)$ $g.\,\text{grainSize}(t) \geq \text{meshSize} \Rightarrow g2(t+1) = g(t)$

Table 5.26: Specification of the CPF defined by a strainer [BG21].

Collect or split: Physical systems that bring together or split stream of material according to quantitative properties (*e.g.,* mass, or volume) belong to the category of systems that define a CPF that collects or splits material [KK98]. Collecting refers to increasing an amount of substance quantitatively, while splitting refers to decreasing the amount of substance in each of the outgoing streams [KK98]. Material collecting systems, *e.g.,* pour together two (different) kinds of material, while material splitting systems, *e.g.,* weigh out amounts of material [KK98].

Example 5.7 (Splitting or Collecting Water)**.** *In pipe systems, splitting or collecting streams of fluid materials, such as water, is a very common task. A very high-level specification of the CPF is given in Table 5.27. In principle, the interface assertion uses mass conservation to state that the incoming fluid stream of water is split into two streams equally. The formula states that the amount of water that leaves the system which is determined by the sum of the amounts of water that leave the system on each output port over time is equal to the amount of water that was put into the system.*

CPF

SplitWater($\mathbb{R}_+$ δ)

in «fluid» Water w out «fluid» Water $w1$, «fluid» Water $w2$
$\forall t \in \mathbb{R}_+$: $\int_0^t \text{vol}(w)(t')dt' = \int_0^{t+\delta} \text{vol}(w1(t'))dt' + \int_0^{t+\delta} \text{vol}(w2(t'))dt'$

Table 5.27: Specification of a CPF that splits water quantitatively.

Conduct or isolate: Functions that conduct or isolate material bring, or prevent matter to get from one place to another. The interface assertions to specify these elementary functions look very similar to the respective energy transforming elementary functions. To this effect, we provide a scheme that uses the stereotype «material» on the input and output channels. What is power for energy transformations, is volume or mass for material. We adapt the interface assertions for the corresponding energy operations by replacing all references of the power property pow by references to the volume or mass property, *i.e.,* vol or mass, respectively (Section 2.3). Using the scheme, the modeler will have to refine th stereotype «material» to be either «fluid» or «item».
A water pipe, for example, transports water from one place to another without loosing water along the way. The component Pipe $\stackrel{\text{def}}{=}$ ConductMaterial$\langle$Water$\rangle$($\mathbb{R}_+$ δ, Position start, Position end) defined in Table 5.29 specifies the function defined by a pipe as an elementary function. The interface assertion specifies the position of the outgoing fluid to be the end-position and the position of the incoming fluid to be the

CPF

ConductMaterial$\langle \mathrm{M} \in \mathcal{M}at \rangle(\mathbb{R}_+\ \delta$, Position start, Position end)

in «material» M $m1$ out «material» M $m2$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{vol}(m1)(t) = \mathrm{vol}(e2)(t+\delta)$ $\mathrm{pos}(e2(t)) = \mathrm{start} \wedge \mathrm{pos}(e2(t+\delta)) = \mathrm{end}$

Table 5.28: Scheme for the specification of an elementary function that conducts material adapted from the energy scheme Table 5.16.

start position which are both parameters of the CPF. Abstracting from losses, the interface assertion requires the volume of the outgoing fluid after a time delay δ as equal to the incoming volume. For isolating material, we adapt the respective energy

CPF

Pipe($\mathbb{R}_+\ \delta$, Position start, Position end)

in «fluid» Water $m1$ out «fluid» Water $m2$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{vol}(m1)(t) = \mathrm{vol}(e2)(t+\delta)$ $\mathrm{pos}(e2(t)) = \mathrm{start} \wedge \mathrm{pos}(e2(t+\delta)) = \mathrm{end}$

Table 5.29: Scheme for the specification of an elementary function that conducts energy.

schemes provided in Table 5.17, Table 5.18, and Table 5.19 can be adapted for this purpose and are given in Table 5.30, Table 5.31, Table 5.32.

CPF

IsolateMaterial$\langle \mathrm{M} \in \mathcal{M}at \rangle(\mathbb{R}_+\delta, \mathrm{List}\langle \mathrm{Position}_3 \rangle P)$

in «material» M $m1$ out $\emptyset$
$\forall t \in \mathbb{R}_+ :$ $\mathrm{pos}_3(m)(t) \notin P$

Table 5.30: Specification scheme for elementary functions that isolate material.

StoreMaterial$\langle \mathrm{M} \in \mathcal{M}at \rangle(\mathbb{R}_+\ \delta)$

in «material» M $m1$, «event» Boolean b out «material» M $m2$
$s \in \text{Energy}, \forall t \in \mathbb{R}_+ :$ $0 \leq s(t) = \int_0^t \text{pow}(m1)(t')dt' - \int_0^t \text{pow}(m2)(t')dt'$ $lt(b,t) \Rightarrow 0 < \int_0^t \text{pow}(e')(t'+\delta)dt' \leq s(t)$ $\neg lt(b,t) \Rightarrow 0 = \text{pow}(e')(t+\delta)\vert$ where $lt(c,t^*) = c(\max\{0 \leq t \leq t^* \mid c(t) \neq \xi\})$

Table 5.31: Specification schemes for elementary functions that store material.

MaterialSource$\langle \mathrm{M} \in \mathcal{M}at \rangle(\mathbb{R}_+\ \delta, \text{Volume}\ v)$

in «event» Boolean b out «material» M m
$s \in \text{Energy}$ $\forall t \in \mathbb{R}_+ :$ $T \stackrel{\text{def}}{=} \{t' \mid \text{vol}(m)(t') \neq 0\text{m}^3\}$ $\int_{t \in T} \text{vol}(m)(t')dt' = v$ $\neg lt(b,t) \Rightarrow \text{vol}(m)(t) = 0\text{m}^3$ where $lt(c,t^*) = c(\max\{0 \leq t \leq t^* \mid c(t) \neq \xi\})$

Table 5.32: Specification schemes for elementary functions that represent a source of material.

5.2.3 Operations between Energy and Material

Physical systems often also describe functions that process both energy, and material. Their task is, *e.g.,* to set a physical part into motion or to cool down a physical part which corresponds to dissipating heat from the part. The former belongs to the category of functions that connect material and energy, while the latter belongs to the category of functions that separate material and energy.
The operations between energy and material listed in [KK98] are

1. Connect: Apply energy to a stream of material, *e.g.,* lift material or bring material into motion, and

2. Separate a stream of an energy-afflicted material into a stream of energy, and a stream of material that is not (or less) energy-afflicted

Connect: A CPF that is an operation between energy and material that connects energy and material iff it changes the values of the energetic components of the material [KK98]. A material type comprises energetic attributes, *i.e.,* attributes with an energy type and attributes that are typed by physical quantities that describe the physical properties of the material. An example of an elementary function in this category is provided in Example 3.2 in Section 3.4.1 which specifies the the elementary function, *e.g.,* of the paddle wheel of a centrifugal pump. The paddle wheel is placed within a fluid, *e.g.,* water, and rotates which causes a pressure difference within the fluid that causes it to start moving. The function defined by the paddle wheel has two input ports and one output port: An energy input channel of type RotationalEnergy, a fluid input channel of type Fluid and a fluid output channel of type Fluid. The interface assertion uses the power balance to specify the transfer of energy.

Separate a stream of an energy-afflicted material into energy and material. Examples of this operation are cooling, lowering, breaking, or damping [Kol98]. This category of functions is defined very similarly to the operations that connect energy and material.

Example 5.8 (Radiator)**.** *The main function of a radiator is to enable a material to release heat. In a cooling system, for example, radiators are responsible for releasing the heat from the coolant. The specification in Table 5.33 expresses that at any time t and for a delay δ which is a parameter of the CPF, the power of the coolant at the output channel $c2$ plus the power at the output channel h at time $t + \delta$ is equal to the power at the input channel at time t up to an error of $\varepsilon > 0$. Since the power on the output channel, h is specified as greater than* 0 W*, the specification defines the transfer of heat from the incoming coolant.*

5.3 Discussion

The previous sections have summarized the mechanical design process from [Kol98, BG21], and formalized the Koller design catalog [KK98]. The formalization mainly concerned translating the verbal definitions of elementary operations into interface assertions or HIOSs that can be used to specify the behavior of a CPF. Further, we have provided specification schemes or examples of elementary functions for most categories, which are translations of the informal definitions in the form of a substantive and a verb prescribed in [BG21, Kol98]. For the specifications of elementary functions, we have employed timed specifications that are underspecified, *e.g.,* because they abstract from energetic losses. The following subsections discuss how

CPCD

«energy» Heat	«material» Coolant
Temperature temperature EntropyStream entropystream	Heat e

CPF

Radiator($\mathbb{R}_+$ δ)

in «fluid» Coolant $c1$ out «fluid» Coolant $c2$, «energy» Heat h
$\forall t \in \mathbb{R}_+$: $\text{pow}(c2.e)(t+\delta) + \text{pow}(h)(t+\delta) = \text{pow}(c1.e)(t)$ $\text{pow}(h)(t+\delta) > 0$ W

Table 5.33: Heat is a form of energy that is bound to the physical quantities temperature and entropy [FR76].

energetic losses, and delays can be made explicit or omitted in the specifications on the example of the CPF defined by an electric drive presented in Table 3.3.

5.3.1 Energetic Losses

The specification schemes and examples in Section 5.2.1 abstract from energetic losses. For example, the scheme in Table 5.8 that formalizes converting energy, states that the incoming power has to be equal to the outgoing power in the interface assertion. In the real world, there exist no such ideal systems that can use all the energy they receive as input. This section discusses how to include energetic losses in the interface assertions. These techniques can be applied equally to specify materialistic losses by using the vol or mass property instead of the pow property.

(Under-)specifying the Amount of lost Energy The interface assertions in the specification schemes and examples provided in Section 5.2.1 require the incoming power to be equal to the outgoing power. Consider, for example, the CPF defined by an electric drive specified in Table 3.3: The specification contains the statement

$$\text{pow}(e)(t) = \text{pow}(r)(t+\delta), \tag{5.10}$$

where e is an input port of the energy type ElectricalEnergy, and r is an output port of the energy type RotationalEnergy. The statement manifests the power balance stating that the power that leaves the system in the form of rotational energy via the output

port r must equal to the power that has come into the system in the form of electrical energy via the input port e. That is, the electric drive may not store energy, but convert it directly to rotational energy. To regard potential losses in the specification, the modeler can use underspecification or make the losses explicit. An implicit way of allowing energetic losses while underspecifying not only the kind but also the amount of losses is by replacing the statement in Equation 5.10 by the following:

$$\text{pow}(e)(t) \leq \text{pow}(r)(t + \delta), \tag{5.11}$$

Equation 5.11 specifies the CPF to conserve energy by stating that the outgoing power must be smaller than or equal to the incoming power. The specification is underspecified in the sense that it leaves open, what happens with the rest of the energy. Because energy is a conserved quantity, this energy must go somewhere, and in technical implementations of such a specification, the energetic loss will become apparent. This way, the specification leaves a margin, *i.e.,* $\varepsilon \in \mathbb{R}_+(\text{W})$ defining an "allowed" amount of energetic loss explicitly. Another implicit way to regard energetic losses in a specification is to specify a threshold that defines an amount of "allowed" loss. By replacing the statement in Equation 5.10 with

$$0 \leq \text{pow}(e)(t) - \text{pow}(r)(t) \leq \varepsilon, \tag{5.12}$$

for $\varepsilon \in$ Power, the modeler states that an amount of ε of energy may be lost during the transformation. The ε could be defined as a parameter of the CPF or as an internal variable.

Specifying the Type of lost Energy Electric drives or batteries that store electrical energy (*cf.* Table 5.18), for example, heat up during operation. That is, not all of the incoming electrical energy is converted to rotational energy, some is converted to heat, which is one form in which energy is lost during the conversion. In the engineering of electric vehicles, thermal management is of crucial importance as the vehicle's range depends heavily on its efficiency, and therefore, the energetic losses will need to be considered at some point. One way of making the loss explicit is by modeling the losses by adding energetic output ports of respective types to the interface of a CPF. Table 5.34 shows this for the electric drive. Therein, heat is a form of energy that is bound to the two quantities temperature and entropy [FR76]. Here, the specification has an extra output port of type Heat that represents a channel via which a system that implements the specified functionality may lose energy in the form of heat. The specification is still underspecified due to the $\geq$ in the specification: The amount of energy that is lost as heat must not be the entire loss of the conversion.

Not Specifying Energetic Losses The CPFs specified in this section are ideal in the sense that they abstract from losses. In the interface assertion, the equality

CPCD

«energy» Heat
Temperature t EntropyStream e

CPF

ElectricDriveWithLoss(Power $\varepsilon, \mathbb{R}_+$ δ)

in «energy» ElectricalEnergy e out «energy» RotationalEnergy r, «energy» Heat $heat$
$\forall t \in \mathbb{R}_+ :$ $\text{pow}(e)(t) - \text{pow}(r)(t+\delta) < \varepsilon$ $\text{pow}(e)(t) \geq \text{pow}(r)(t) + \text{pow}(heat)(t)$

Table 5.34: Specification of the CPF defined by an electric drive that regards energetic losses in the form of heat. Heat is an energy type represented by the two quantities, temperature and entropy stream [FR76].

in Equation 5.10 explicates this abstraction. Table 5.35 shows the CPF of an ideal electric drive, *i.e.,* the lossless conversion of electrical to rotational energy which was given previously in Table 3.3. In the real world, any conversion of energy is subject to energetic losses, therefore, this specification is not realizable.

CPF

ElectricDriveWithoutLosses($\mathbb{R}_+$ δ)

in «energy» ElectricalEnergy e out «energy» RotationalEnergy r
$\forall t \in \mathbb{R}_+ :$ $\text{pow}(e)(t) = \text{pow}(r)(t+\delta)$

Table 5.35: Specification of the CPF defined by an electric drive without accepting losses.

Making the Efficiency Explicit In mechanical engineering, efficiency is a characteristic variable for systems that convert energy. The efficiency is the instantaneous ratio between the incoming power and the outgoing power. Table 5.36 shows a specification that includes the efficiency as a parameter of the CPF. The specification allows implementations to be wrong up to an error of ε.
Another way to specify the conversion of electrical to rotational energy while making the efficiency explicit but leaving a tolerance is to specify a range of the efficiency through functional parameters that define a minimal and a maximal efficiency, as in Table 5.37. The specification references the instantaneous efficiency and states that it must be within the range at all times [GBG18].
In both specifications in Table 5.36, and 5.37, it is possible to explicate the lost energy by refining the specifications to include respective output ports as in Table 5.34.

CPF

ElectricDriveWithEfficiency($\mathbb{R}_+$ n, Power ε, $\mathbb{R}_+$ δ,)

in «energy» ElectricalEnergy e out «energy» RotationalEnergy r
$\forall t \in \mathbb{R}_+$: $\text{pow}(e)(t) - n \cdot \text{pow}(r)(t + \delta) < \varepsilon$

Table 5.36: Specification of the CPF defined by an electric drive that makes the efficiency n explicit, while leaving an error tolerance of ε.

CPF

ElectricDriveWithEfficiencyRange($\mathbb{R}_+$ δ, $\mathbb{R}_+$ $n_{\min}$, $\mathbb{R}_+$ $n_{\max}$)

in «energy» ElectricalEnergy e out «energy» RotationalEnergy r
$\forall t \in \mathbb{R}_+$: $\text{pow}(e)(t) \geq n_{\min} \cdot \text{pow}(r)(t + \delta)$ $\text{pow}(e)(t) \leq n_{\max} \cdot \text{pow}(r)(t + \delta)$

Table 5.37: Specification of the CPF defined by an electric drive that considers energetic losses through an allowed range of efficiencies.

5.3.2 Delay

A specification is delayed with delay $\delta > 0$, iff for all TSPFs in its semantics, the input until time $t \in \mathbb{R}_+$ entirely determines the output until time $t + \delta$ (*cf.* Section 3.3). The property of being delayed becomes important when specifications are composed by feedback composition which is not well-defined for non-delayed CPFs. The specifications above are all delayed. In the specifications in Table 5.35 to 5.37 the delay expresses that the system that defines the specified CPF takes some time, *i.e.,* δ, to perform the conversion of electrical to rotational energy. The specification in Table 5.35 states that the incoming power at time t must be equal to the outgoing power at time $t + \delta$. The energetic balance, *i.e.,* the difference between incoming and outgoing streams of energy, for an electric drive stated in [FR76] states this balance without delay which the specification in Table 5.38.

Similarly, stating the specifications in Table 5.34, 5.36, or 5.37 by omitting the δ parameter yields an undelayed specification. However, when composing the undelayed specifications, feedback composition is not possible, because it is not well-defined for these specifications.

ElectricDriveWithoutLossesInstantaneous CPF
in «energy» ElectricalEnergy e out «energy» RotationalEnergy r
$\forall t \in \mathbb{R}_+$: $\text{pow}(e)(t) = \text{pow}(r)(t)$

Table 5.38: Specification of the CPF defined by an electric drive without losses and stating the energetic balance instantaneously as in [FR76].

5.3.3 Related Work

Ongoing research has produced theories, and modeling languages for engineering software and electronic functions of CPS, *e.g.,* [Alu15, Pto14, BS01], as well as for designing [SFA17, Gro00], engineering [BBL+16], and operating [BSP+16] CPSs 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 [Bro12, Alu15, Pto14] or geometry are supported, the theories and languages do not support established processes or modeling concepts from other (*i.e.,* the "physical") domains, such as mechanical engineering. In the Focus theory [BS01], systems are composed of components that realize stream processing functions. As functions communicate via their interface only, they can be (de)composed and refined systematically, where refinement considers both, structure and the behavior of components [Bro18]. Architectures and the notions of subsystems and interfaces are rigorously formalized using Focus into a logical calculus for the composition of systems [Bro18]. To the best of our knowledge, none of these approaches have been used to create models with formal semantics that make the knowledge from design catalogs available for functional approaches to systems engineering.
In mechanical engineering, a variety of design catalogs to aid the design process regarding various aspects [FLD04] exist in the literature,
e.g., [BG21, Rot11, Rod91, Rot94, Rot96]. Approaches that digitize such catalogs, *e.g.,* [MEE+11, FLD04], focus on making the (extended) information from the existing design catalogs accessible by providing digital textual descriptions complemented with mathematical expressions or sketches. The provided models, however, do not have a formal semantics. Lacking a representation in a formal modeling language that also enables integrating the information within a functional architecture of a mechanical system hinders applying these approaches in a model-driven development process.

Chapter 6

A Language Engineering Perspective on Physical Functions

CPS engineering is a broad field that involves experts and companies from very different backgrounds and very different experience levels. Commonly, we mention the domains of software, mechanical, and electrical engineering. The field is even broader and also involves creative designers, accountants, jurists, *etc.*. In this dissertation, we focus on integrating the technical domains of software, mechanical, and electrical engineering. As each of these branches has, so far, established methodologies, processes, and tools to engineer their respective parts of the system, introducing functional MDE techniques will require the provision of tailored modeling languages for each domain, company, or branch. This section addresses the research question RQ5 "What are the constituents of a useful modeling language for specifying CPSs from a functional point of view? And what does such a modeling language look like?". To this effect, we have proposed a meta-model that captures concepts from mechanical design theory and implemented this meta-model in the form of the SysML-profile SysML4FMArch which was originally published in [DRW+20] This previous version of the meta-model emerged during a collaborative project with experts from software engineering and mechanical engineering. This section derives a new version of this meta-model from the concepts presented in Section 5.1 that abstracts from design theoretic and language implementation details. With this, it aims to allow for extensions that regard also the software and electrical domains in the future. The purpose of the meta-model is, therefore, to capture the concepts illustrated in Figure 5.3 and Figure 5.4 from a language engineering point of view using the words of the theory introduced in Chapter 3. This view shall enable language engineers to implement modeling languages whose semantics can be defined based on the FOCUS theory of TSPFs over discrete and dense time domains detailed in Chapter 3. Section 6.2 details how the SysML profile SysML4FMArch implements also, this new version of the meta-model so that the formal interpretation becomes explicit.

6.1 A Meta-Model for Functional Modeling Languages To Digitalize the Mechanical Design Process

In this dissertation, we consider formalizing the knowledge of design catalogs and systematizing the design process using the MDE techniques presented in Chapters 2 and 3 to provide the basis for an agile and holistic approach to systems engineering. In practice, this requires a modeling language that enables to model channel types, functions, and physical effects. To add value to the mechanical engineering part of CPS engineering, these languages must also enable linking geometric and materialistic information to these models and thereby provide a model of a conceptual solution to a function because a solution to a physical function should also provide an idea of its geometry and the effect carrier [DRW+20, HJZ+21, ZJS+21]. Models in this language together with powerful tooling that exploits their formality can be (re)used to, *e.g.,* evaluate different solutions for products at early development stages enabled by automatic functional dimensioning and testing. This section derives a meta-model [Rum17] from the concepts presented in Chapter 5 that defines modeling languages whose semantic domain can be defined based on the theory presented in Chapter 2 and Chapter 3. The meta-model is similar to the one originally published in [DRW+20] but interprets the concepts from concepts from Chapter 5 in the terminology of Chapter 2 and Chapter 3. Because the terminology is formally defined based on the theory of TSPFs over discrete and continuous-time domains (*cf.* Section 3.3), modeling languages that implement the meta-model obtain a formal semantics. This meta-model provides the conceptual basis for defining modeling languages that enable mechanical engineers to explicate functional structures and solutions composed of models of geometry and physical effects, as formal models that are systematically related and integrate with specifications of software functions in CPS engineering. The following section conceives a meta-model to capture the concepts from Figure 5.3 using object-oriented constructs for types and the notion of functions that are specified through an interface and a behavior.

6.1.1 Functional Interfaces

The *Type Kind* captures the kinds of functional flows from Section 5.1.4. In the meta-model in Figure 6.1 the construct represents the idea of a kind of type that prescribes the kinds of streams in which elements of the type can be exchanged (*cf.* Section 3.2). The concrete kinds of streams considered in our modeling theory are *Signal*, *Data*, *Event*, *Energy*, *Fluid* and *Item* (*cf.* Section 3.2). Here, we allow to underspecify the concrete type of a material flow by introducing the kind *Material Flow* as a super-construct of the *Fluid* and *Item* kinds. The meta-model does not distinguish between kinds of types and kinds of streams because it allows the creation of modeling languages for functions in which the type of stream is not declared on the channels of

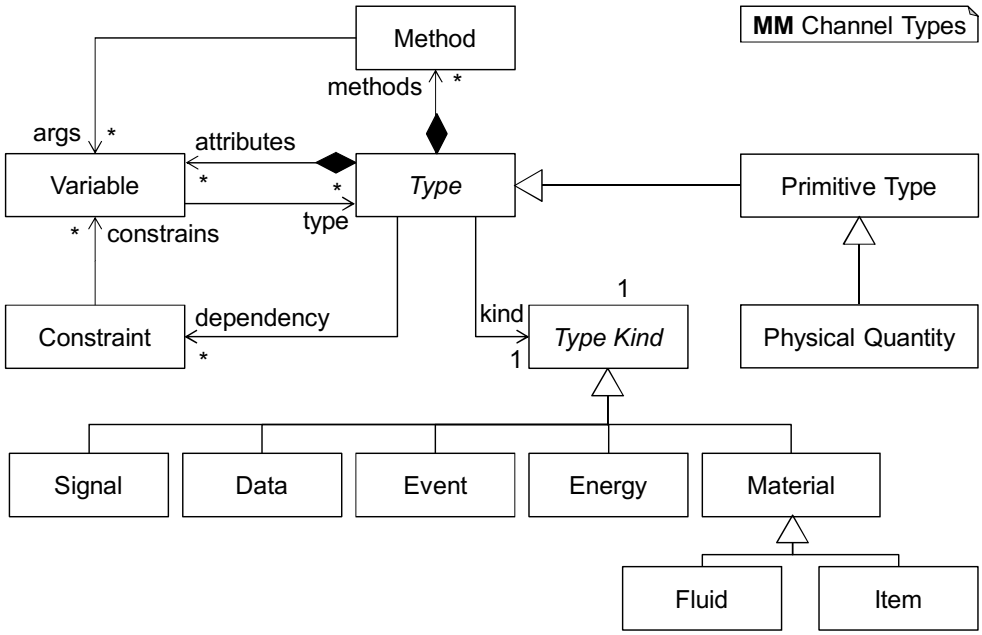


Figure 6.1: Meta-model that describes how the cyber-physical types introduced in Chapter 2 are modeled.

functional descriptions. Traditional design methodologies consider only energy, signal, and material flows [BG21, KK98], which does not suffice when interpreting functions as TSPFs (*cf.* Section 3.2). *Type*s specify the type of the messages that a function receives or transmits via its interface in terms of attributes, methods and constraints among the attributes of a type. The meta-model in Figure 6.1 offers the construct *Variable*s and *Method*s which represent the respective object-oriented concepts introduced in Section 2.1. The *Constraint* construct allows to formulate constraints among the attributes of a *Type*. These constraints represent dependencies among the attributes that represent characteristics of messages of that type. Consider, for example, a cyber-physical type that represents fluid water and assume that in the engineering context, a relevant quantity of fluid water is temperature. Since the type is supposed to represent fluid water, in a simplified setting that abstracts from physical details such as the anomaly of water [RS16], this temperature may not deceed 0°C, because at lower temperatures the water is not fluid but solid. Modeling languages for the engineering of mechanical functions need to offer constructs for specifying such constraints. A possible implementation could be in the form of OCL constraints which are also an established modeling language for constraints among the attributes of classes that represent data types [Rum16]. *Primitive Type*s are types that are the

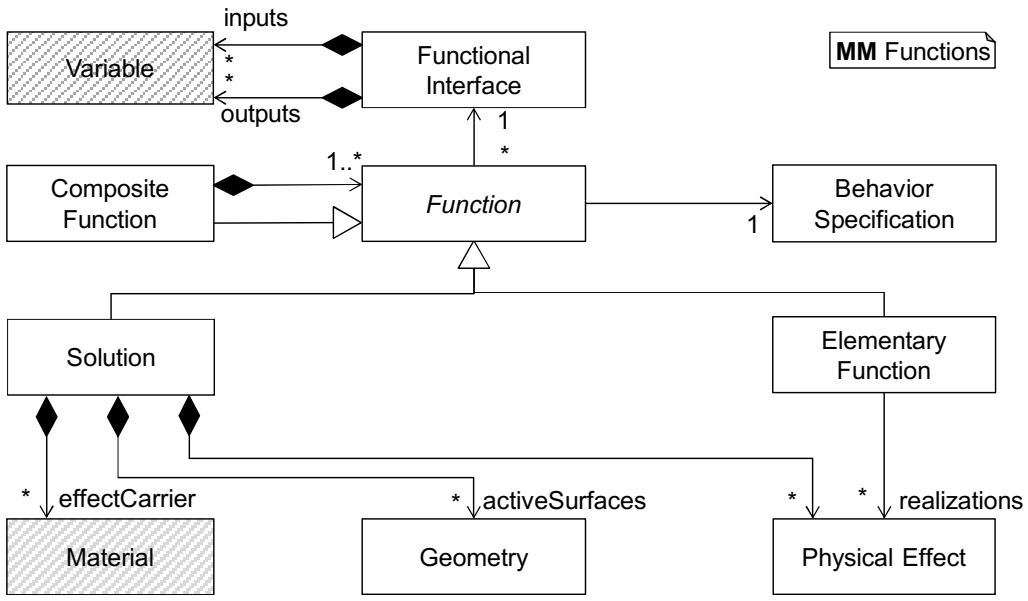


Figure 6.2: Meta-model that captures the concepts of functions and solutions from Figure 5.3 and Figure 5.4.

building blocks of all other types. We consider the set of primitive types to contain the number domains, Booleans and *Physical Quantities* as defined in Section 2.2.1.

6.1.2 Functional Architectures

The principle of decomposition (*cf.* Section 3.3.3) is neither new to software engineering [FR07, Bro07] nor to mechanical engineering [Kol85, BG21]. Both domains describe a transformation of the inputs to a function to the outputs of a function as a network of interacting sub-functions that exchange energy, matter, and data. Our meta-model describes a set of language constructs to model the functional concepts presented in Figure 5.3 in terms of the Focus modeling technique presented in Chapter 3.
Therein, a *Function* exhibits a *Functional Interface* and a *Behavior*. The interface comprises two lists of *Variable*s containing the names of the *input* and *output* channels (*cf.* Definition 3.19). Because each *Variable* has a *Type* (*cf.* Figure 6.1), the channels are typed. Being in either the input or the output set of channels, each channel obtains a unique direction. C&C languages like Simulink [MAT16], Modelica [Mod22], AutoFOCUS3 [AVT+15a], or MontiArc [Hab16] implement this: Functions can be represented by the components (in the graphical notations of each of these languages, components are represented by boxes) and their interfaces by the typed connectors.

The *Behavior* describes how the inputs are transformed to the outputs. There exist different concrete specification styles that a modeling language could offer. So far, our modeling technique offers three specification styles: Interface assertions, HIOSs, and compositional specifications (*cf.* Section 3.4). The meta-model utilizes the composite pattern [GHJV95] to capture hierarchically decomposed functions as *Composite Function*s. It is important to explicate this specification style in the meta-model as decomposition is one of the principles of the functional development paradigm (*cf.* Section 4.1). By this principle, a specification of (a solution to) a function is defined by the composition of the specifications of (solutions to) its sub-functions. In the development process, this principle allows us to break a complex engineering task down into smaller engineering tasks of manageable complexity. Therefore, composite functional specifications must be *composed* of other functional specifications. Other specification styles are defined, *e.g.,* in [BS01]. These represent the decomposed functional structures from mechanical design methodology [BG21], as well as the functional architectures from software engineering. The focus of the meta-model is the specification of modeling languages for mechanical engineers. Therefore, the leaves of a composite specification are called *Elementary Function*s and they point to a set of *Physical Effect*s. In general, there exists more than one physical effect to realize an elementary function and one physical effect may be a possible realization of many elementary functions. *Physical Effect*s capture physical phenomena as mathematical formulae over the variables that describe the input and output channels of the function the effect realizes. Consider, for example, a function that increases force. The lever effect is an effect that [KK98] indicates a possible realization for this function. The effect describes that when putting force on the longer end of a lever arm, a larger force occurs at the other. The law describes this relation by a mathematical formula. Principle solutions describe how such phenomena implement a physical function by relating the variables in the physical effect, the description of the active surfaces, and the effect carrier. Design catalogs, such as [KK98] provide a list of standardized elementary functions together with a set of physical effects, possible geometries or effect carriers that provide possible implementations of these functions. Modeling languages as specified by the meta-model allow to digitalize the knowledge from design catalogs in a formal manner that enables automation, *e.g.,* of a search engine for (optimal) solutions, in the future. Note, that this set may be empty, so *Elementary Function*s may also represent atomic functions, *e.g.,* of software systems. For the latter, the traditional MontiArc [1] is an approved modeling language that allows to define the behavior of a software function as the composition of other software functions, or through a program that can be specified by an automaton or in the form of simplified Java code.

Solutions must meet the functional specification, and therefore implement physical

[1] https://github.com/MontiCore/montiarc

functions *consistently*. A *Solution* is a *Function* that contains one or more physical effects, a set of *activeSurfaces* and the *effectCarrier* [DRW+20, IJZK23]. Speaking in terms of Section 3.3.4, a solution *refines* a function in the sense of Definition 1.7. The relation should be a true refinement, *i.e.,* the solution should contain more information than the functional specification. By including information on the physical effect, active surface, and effect carrier, the solution thereby complements the specification of the function which narrows the set of TSPFs in the semantics of a solution. Because Focus is the theory in which the models of the modeling languages specified by the meta-model are implemented, the refinement relation is compositional. That is, a solution to a composite function is obtained by providing solutions to the sub-functions. This supports the principle of decomposition in the development process as it enables the distribution of functions to development teams across the company or project because the refinement steps within each team will then yield a refinement of the decomposed function.
Current literature on design theory prescribes the list of active surfaces to comprise exactly two elements based on the idea that an effect acts between two surfaces or spaces [IJZK23]. To enable underspecifying geometric information, enhancing a solution model or supporting other settings that require to deviate from this norm, the meta-model is less restrictive in the number of active surfaces. The SysML profile presented in Section 6.2, which implements the meta-model, simply considers geometry to be a type with geometric attributes, *i.e.,* variables whose type is a physical quantity like length, area, height, *etc.* The *effectCarrier* is a *Material*-type and, therefore, defined in terms of attributes which describe the characteristics of the material and methods if necessary.

6.1.3 Discussion

The meta-model and its SysML-encoding that were published in a previous version in [DRW+20]. The work emerged during an interdisciplinary project comprising researchers from software and mechanical engineering as well as practitioners from the automotive industry. So far, the meta-model has captured and extended the notion of functional architectures prevalent in mechanical design theory [BG21, KK98]. The meta-model now captures the concepts from mechanical design theory illustrated in Figure 5.3 using the terminology introduced in Chapter 2 and Chapter 3. Modeling languages that implement the meta-model obtain a formal semantics because the elements of the meta-model are defined using the words of the Focus theory introduced in Chapter 2 and Chapter 3. We have evaluated this approach by implementing the meta-model as the SysML-profile SysML4FMArch. The modeling elements described by the meta-model provide the language constituents that modeling languages for the mechanical engineering domain need to offer. The meta-model was conceived by and in collaboration with software engineering experts who are

experienced in developing architectural DSLs for software engineering. Also, the theory that defines the semantics of the elements in the meta-model extends the discrete Focus theory. Therefore, the formal interpretations of models created with modeling languages that implement the meta-model integrate with models created with modeling languages whose semantics is built on discrete Focus.

6.2 Modeling Physical Functions and Solutions in SysML

This section details how the SysML-profile SysML4FMArch, which was published originally in [DRW+20] implements the meta-model presented in Section 6.1 by mapping the elements and associations of the meta-model to a SysML-profile. We complement the definition with additional well-formedness rules that integrate the definitions from Chapter 2 and Chapter 3 in SysML4FMArch which cannot be expressed in the graphical syntax of SysML profile diagrams. The language originally emerged during a collaborative project with experts from software engineering and mechanical engineering as an implementation of the previous version of the meta-model which was also published in [DRW+20]. Here, we enhance the definition of SysML4FMArch to also include interface assertions which enable specifying the behavior of a physical function as a quantitative relation between the inputs and the outputs. Further, we outline how the modeling language can support functional dimensioning and testing and briefly describe the implementation of SysML4FMArch as a SysML-profile in MagicDraw[2] with the SysML-plugin[3].
SysML4FMArch specifies stereotypes for SysML-elements and relations between them to encode the meta-model presented in Section 6.1. As an extension to [DRW+20], we include interface assertions to specify the behavior of physical functions in SysML4FMArch models. Section 1.3.5 summarizes the SysML elements that are reused, extended or implemented by SysML4FMArch.

6.2.1 Functional Interface

Figure 6.3 illustrates the elements of SysML4FMArch that implement the elements from the meta-model in Figure 6.1 that defines language constructs for modeling cyber-physical types. Functions interact by exchanging energy, matter or data [Kol98, BG21]. In Chapter 2 we have defined these as kinds of types because there exist different types of energy, matter and data. The kind of type belongs to a type and dictates the kind of stream (*cf.* Table 3.1) that models the exchange of messages of that type. In SysML4FMArch, *Type*s from the meta-model in Figure 6.1 are modeled by stereotyped SysML Blocks. The choice is obvious because SysML

[2]https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/
[3]https://www.3ds.com/products-services/catia/products/no-magic/addons/sysml-plugin/

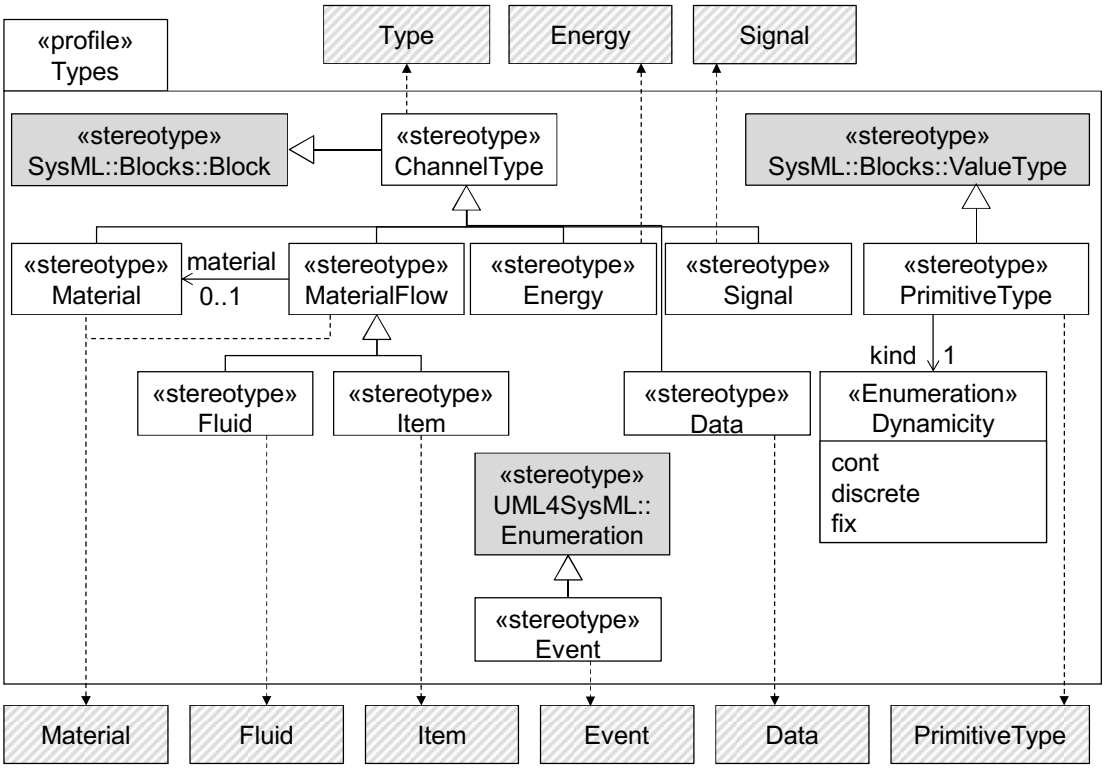


Figure 6.3: SysML4FMArch's encoding of the Functions meta-model similar to [DRW+20]. The gray elements are from the SysML standard [Man19] and the shaded boxes denote elements from the concept model depicted in Figure 6.1. Dashed arrows indicate the implementation of the concept at the arrow's end.

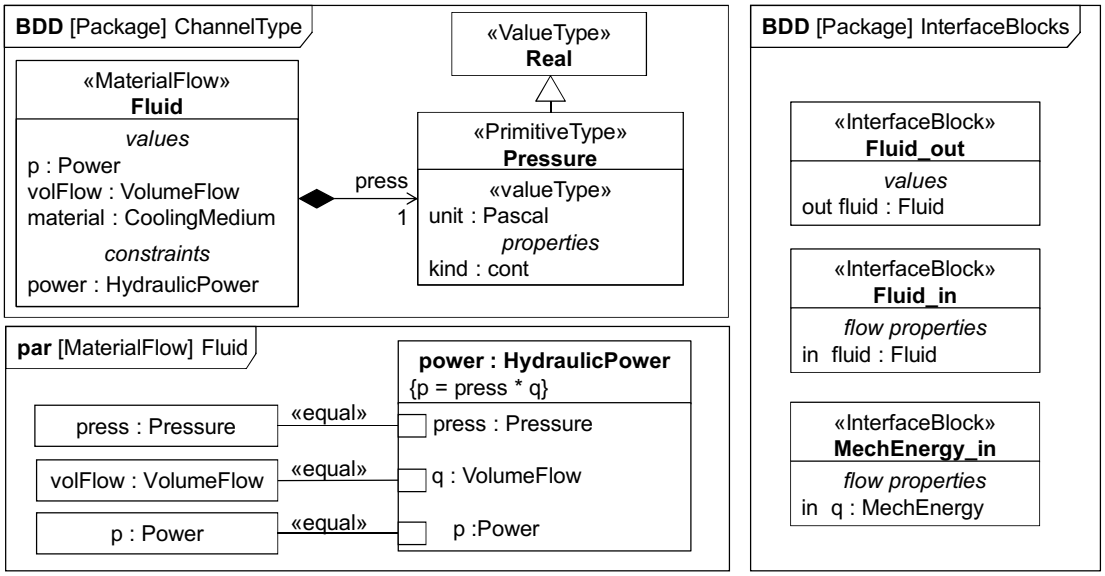


Figure 6.4: Specification of a «MaterialFlow» `Fluid` in SysML4FMArch [DRW+20].

Blocks are derived from UML classes which implement the notions presented in Section 2.1 on which also the idea of a cyber-physical type is based. Among others, *Type*s provide definitions of the types for the channels of a CPF's interface. The attributes of a *Type* are modeled by SysML ValueProperties [Man19] of a «ChannelType». For *Method*s, SysML offers the construct of Operations which are very similar to methods of classes in UML class diagrams [Man19]. In SysML4FMArch, the stereotype of a Block represents the *TypeKind* from Figure 6.1. The SysML standard already defines a library of ValueTypes that provides the numeric domains considered as primitive types (*cf.* Chapter 2) [Man19]. Also, the package offers a mechanism to equip ValueTypes with units. Every ValueType with the «PrimitiveType» stereotype for which a unit is defined, models a physical quantity. To enable channels to be typed with solely a primitive type, SysML4FMArch specifies the stereotype «PrimitiveType» which is implemented as a SysML ValueType and holds a kind property telling the topology in which to interpret the streams of a channel typed with a «PrimitiveType». Therein, «cont» represents hybrid dense streams, «discrete» represents time slice streams, and «fix» represents parameters of the system that do not change their value during runtime. The latter are used, *e.g.,* in the definition of geometries that represent active surfaces of a solution. «Signal», «Energy», «Data» and «Material», and the specific material *TypeKind*s «Item» and «Fluid» are the stereotypes to represent the corresponding kinds of types. «Event» represents the *Event* kind of type as an enumeration. The definitions of each kind of type restrict the

types of the attributes of the corresponding classes. For each of these types, we derive the following well-formedness rules from the corresponding definitions in Section 3.2. Definition 2.4 defines an energy type as a class with two attributes whose type is a physical quantity.

Well-Formedness Rule 1 (Well-Formedness of Energy Types)**.** *A Block that holds the stereotype «Energy» must have at least two ValueProperties whose type is a physical quantity, i.e., a «PrimitiveType» with a defined unit. The product of the units of these physical quantities must be* W *as specified in Definition 2.4.*

Definition 2.5 defines a material as a class whose attributes are only typed by energy types or physical quantities

Well-Formedness Rule 2 (Well-Formedness of Material)**.** *A Block that holds the stereotype «MaterialFlow», «Item» or «Fluid» must not have ValueProperties whose type is not a physical quantity or a Block with the stereotype «Energy». One of the ValueProperties must reveal information on volume or mass.*

Definition 2.6 defines a data type to be a class with attributes whose types are primitive types.

Well-Formedness Rule 3 (Well-Formedness of Data Types)**.** *A Block that holds the stereotype «Data» may only have ValueProperties whose type is a Block with the stereotype «PrimitiveType» and must not have ValueProperties whose type is a Block with the stereotype «Energy» or «Material».*

Section 3.2.5 defines a signal as a hybrid stream. As in SysML it is not possible to indicate a channel or port type with a different stereotype than its type, the stereotype «Signal» is offered in SysML4FMArch.

Well-Formedness Rule 4 (Well-Formedness of Signal Types)**.** *A Block that holds the stereotype «Signal» may only have ValuePropoerties whose type is a physical quantity. Such Blocks must not have ValueProperties whose type is a Block with stereotype «Energy», «Material» or «Data».*

Constraints between these attributes are modeled by ConstraintBlocks that include the BindingConnectors between attributes and ConstraintParameters [Man19]. By adding a ConstraintProperty of the corresponding ConstraintBlock-Type to the «ChannelType»-Block, these constraints are associated with the type.
We show an example from [DRW+20] to illustrate these constructs from SysML4FMArch. The examples in Figure 6.4 show a BDD and a parametric diagram that include these elements and show their relationships: The «MaterialFlow»-block has four attributes and models a fluid. The attribute `press` is of the real number type `Pressure` and specifies the unit `Pascal` as well as the «Dynamicity» `cont`.

ValueProperties typed by `Pressure`, therefore, change their value continuously during system runtime. The attribute `material` specifies the material of the represented fluid as a collection of typed attributes defined by the type `CoolingMedium`. The latter is provided in the complete model of the automotive cooling pump in Section 7.2. The ValueProperty `volFlow` of type `VolumeFlow` specifies the instantaneous volume flow and allows to derive information on the volume of the fluid to meet the prerequisite defined in Well-Formedness Rule 2. The ConstraintBlock `HydraulicPower`, contained in the parametric diagram at the bottom left models the physical relationship between the attributes of the `Fluid` channel type. SysML4FMArch uses the predefined SysML ProxyPorts to model the inputs and outputs of a function. The types of ProxyPorts are determined by SysML InterfaceBlocks which hold FlowProperties to specify the entities that "flow" through the port [Man19]. SysML4FMArch models are only well-formed if the InterfaceBlocks that type ProxyPorts have FlowProperties of unambiguous direction, *i.e.,* the usage of direction `inout` is not allowed and specifying FlowProperties of multiple directions in one InterfaceBlock yields an invalid model.

Well-Formedness Rule 5 (Unambiguous Port Directions)**.** *The direction of the FlowProperties of an InterfaceBlock that is used to type a ProxyPort of a «Function» must be unambiguous. The direction* `inout` *is invalid.*

The diagram on the right of Figure 6.3 shows examples for InterfaceBlocks that type ProxyPorts in our running example. Using the specification of ProxyPorts by the SysML standard [Man19] it is possible to negate the direction of a port, thus, specifying one InterfaceBlock for each direction (`in` and `out`) is not obligatory. In our example, we do so because the notation for negated ProxyPorts which simply puts a $\sim$ in front of the port name does not explicate the direction. Note that `Fluid` and `RotMechEnergy` represent the actual physical entities and not flows of information about these entities. Section 7.2.1 introduces SysML4FMArch-models of all the channel types used in the case study and throughout this paper in more detail.

6.2.2 Functions

Figure 6.5 shows how SysML4FMArch encodes the notions of *Composite Function*, *Function*, and *Elementary Function* from the meta-model that defines language constructs for specifying functions in Figure 6.2. In SysML4FMArch, the stereotype «Architecture» encodes the *Composite Function* element from the meta-model, *Functions* and *Elementary Functions* encode the corresponding elements from the meta-model (see Figure 6.5). SysML4FMArch provides the «Function»-stereotype that encodes the abstract element *Function* from the meta-model. This enables to model a function without needing to know whether or not the function will be decomposed later on which also addresses the challenge of the application of mechanical design

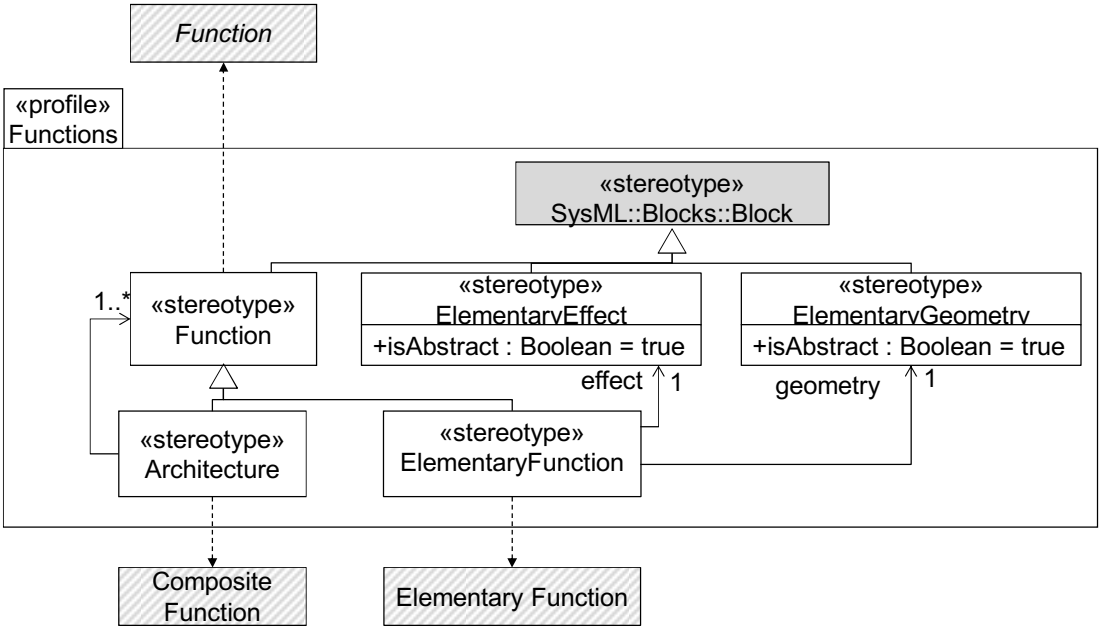


Figure 6.5: SysML4FMArch constructs for modeling functions as in [DRW+20]. The shaded boxes denote the elements defined by the meta-model in Figure 6.2 and the dotted arrows denote the implementation of the SysML4FMArch constructs of these elements.

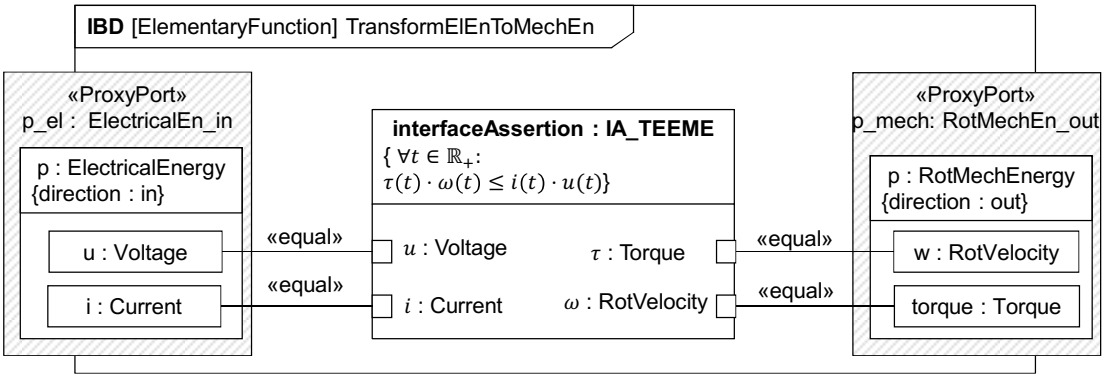


Figure 6.6: Example of a model for an elementary function in SysML4FMArch that represents the transformation of electrical energy to rotational mechanical energy from [DRW+20].

methodology outlined in Section 5.1.3. To illustrate this, consider the example «Architecture» shown in Figure 1.5. The IBD models the internal structure of the «Architecture» `GenerateVolumeFlow` which is the overall function of the system considered in our running example described in Section 1.3.3. It comprises two PartProperties of «ElementaryFunction»-type, *i.e.,* `moveFluid` and `elToMech` and a «Function» `setVRot`.
In SysML4FMArch, interface assertions [Bro18] are modeled by the ConstraintProperties of «Functions» [Man19]. For these, the standard [Man19] allows "using either formal statements in some language, or informal statements using text." Here, we use logical formulae. The IBD in Figure 6.6 is a SysML4FMArch-model of the function TEE2ME including an interface assertion. The assertion specifies functional behaviors such that the incoming electrical power, *i.e.,* the product of voltage and current, is greater or equal to the outgoing mechanical power, *i.e.,* the product of torque and rotational velocity [GB07]. SysML-ModelLibraries [Man19] enable to store the «ElementaryFunction»-blocks that specify elements from catalogs such as [KK98]. Utilizing a specialization of an «ElementaryFunction» allows refining the functions from [KK98] for specific systems while preserving consistency to the definitions in the library.

Elementary Effects and Elementary Geometries: To realize a physical function in an architecture, an engineer selects a physical effect listed in a design catalog [BG21, KK98] and derives a set of active surfaces that is sketched by the mathematical description of the effect [JKB+21]. The interconnection of the physical effect and the principle geometry refines the behavior description of a physical function by translating it to a mathematical description of a physical process, *i.e.,* the effect, together with a quantitative geometry on which the effect acts, *i.e.,* the set of active surfaces. To establish a systematic relationship between physical functions and their principle solutions, SysML4FMArch defines the abstract «Elementary Effect» and «Elementary Geometry». These elements serve as placeholders for the «PrincipleEffect» and «PrincipleGeometry» (*cf.* Figure 6.7) which represent implementations of their elementary counterparts that can be selected when creating a «PrincipleSolution» that specializes the «ElementaryFunction» (see Section 6.2.3). The example on the left of Figure 1.5 illustrates this: The «PrincipleSolution» `HydrodynamicPump` specializes `ApplyFluidWithMechEn` which redefines the elementary effect and elementary geometry to their implementations `Hydrodynamics` and `WheelCyl`, respectively.

6.2.3 Solutions

Principle solutions describe technical principles to realize a physical function as a physical effect acting on active surfaces [JKB+21]. To specify a principle solution for a

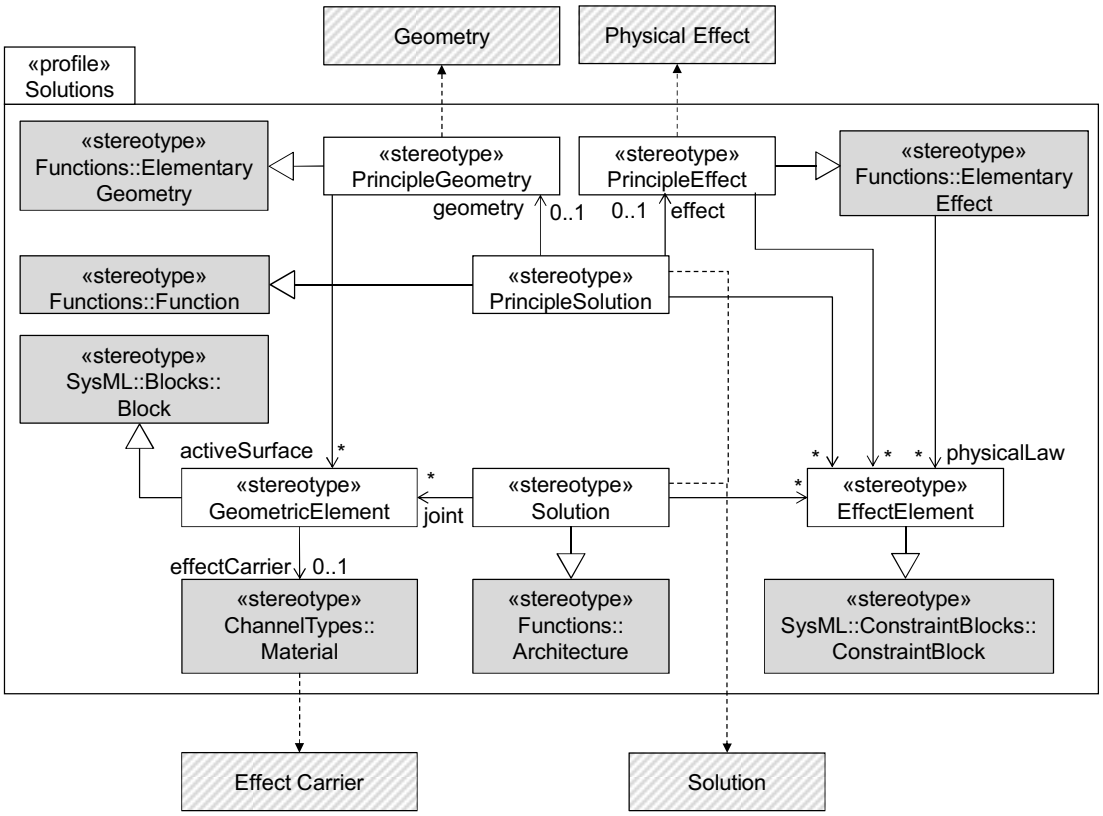


Figure 6.7: SysML4FMArch encoding of solutions (*cf.* Figure 6.2).

physical function in an architecture, the engineer chooses implementations of the elementary geometry and elementary effect owned by the physical function. Therein, the physical effect together with the geometry, provide a mathematical causal relation between the inputs and the outputs that complement the interface assertion and thereby define a refinement of the physical function's behavior. Figure 6.7 shows the encoding of the solutional elements from the meta-model in Figure 6.2 in SysML4FMArch [DRW+20].

Effect Elements and Geometric Elements: The meta-model describes a *Solution* to comprise a *Geometry* which represents the active surfaces on which a physical effect acts. SysML4FMArch enables composing such a geometry from reusable elements. To this effect, Blocks with the stereotype «GeometricElement», with typed ValueProperties, define a single active surface as a type. The ValueProperties of these elements are typed by ValueTypes with the stereotype «PrimitiveType». As mentioned in Section 6.2.1, these hold a property called dynamicity which indicates whether and how the property changes its value during system runtime. Similar to the physical relationships among the attributes of energy or material types, constraints between the ValueProperties of Blocks with the stereotype «Geometric Element» are modeled, either as BindingConnectors in case of equalities or as regular ConstraintBlocks in case of more complex mathematical relationships. Each geometric element has a PartProperty that is typed by a Block with stereotype «Material». These PartProperties model the effect carrier, *i.e.,* the material or space to realize an elementary function. ConstraintBlocks with the stereotype «Effect Element» represent physical laws or relationships between the ValueProperties of Blocks with the stereotype «PrincipleEffect». In principle, these effect elements represent mathematical constraints on the variables of the channels in a function's interface. They can be considered as a statement of an interface assertion that is derived from a physical law. Effect elements are also reusable definitions of these physical laws. Principle effects can be created by combining the physical laws from a library such that a physical process is described accurately. Techniques, *e.g.,* proposed in [JKPB12], enable to link an «EffectElement» to a simulation model and respective tools enable to trigger their execution. In SysML4FMArch, the inputs and outputs of the simulation are represented by the ConstraintProperties of Blocks with the stereotype «EffectElements». The characteristics of geometric elements are modeled by ValueProperties of Blocks with the stereotype «GeometricElements».

Principle Geometry and Principle Effect: Principle geometries comprise the active surfaces between which physical effects come into action. To this effect, our profile defines the stereotypes «PrincipleGeometry» and «GeometricElement». The active surfaces are represented by PartProperties of Blocks with the stereotype

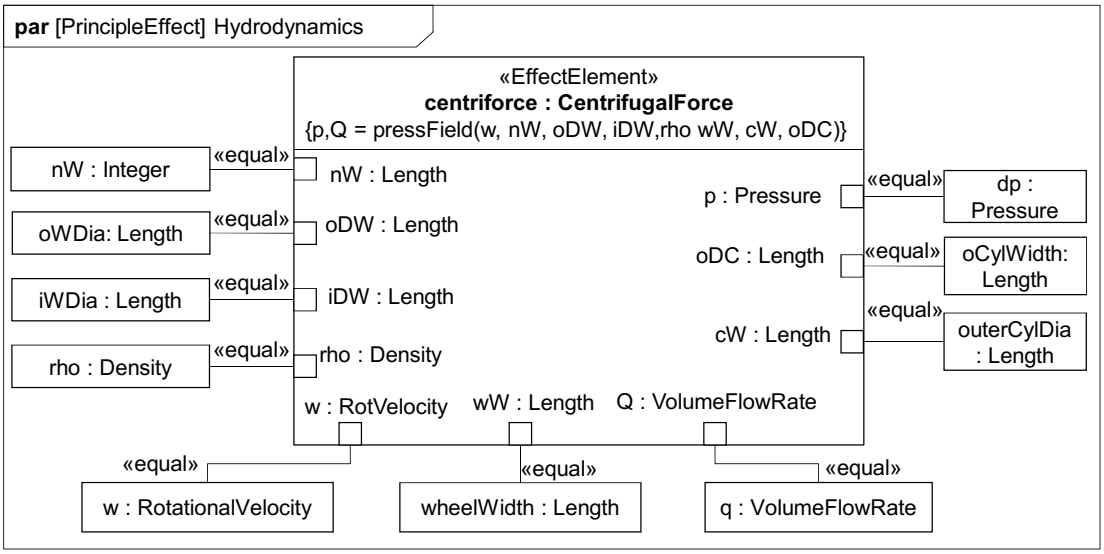


Figure 6.8: Principle effect representing the cause for turbulences in flowing fluids, modeled in SysML4FMArch.

«PrincipleGeometry» whose type is a Block with the stereotype «GeometricElement». Since a physical effect is often described by multiple physical laws (think of a system of (differential) equations), «PrincipleEffect»-elements comprise ConstraintProperties typed by a ConstraintBlock with stereotype «EffectElement». BindingConnectors connect the ValueProperties of a «PrincipleEffect»-Block to the ConstraintParameters of the contained «EffectElements» 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 6.8. The physical law

$$p \cdot Q = M \cdot \omega \tag{6.1}$$

causes turbulence, *i.e.,* a rotational velocity, within a flowing fluid [Pum10]. Here, p is the fluid's pressure, Q is the volume flow rate, ω is the rotational velocity, and M is a torque. Thus, the «PrincipleEffect» `Hydrodynamics` has ValueProperties that are typed with respective «PrimitiveTypes» that which all specify the «Dynamicity» `cont` (*e.g.,* Figure 6.3 shows the definition of `Pressure`). The torque's absolute value depends on the geometric setup through which the fluid is flowing. In the context of the running example (*cf.* Section 1.3.3), 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 passes a paddlewheel 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» `CentrifugalForce` links to an external 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 Equation 6.1. 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`.

Solutions and Principle Solutions: A principle solution inherits the functional interface, the elementary geometry, and the elementary effect as well as the interface assertion from the physical function it realizes. To implement this, we utilize SysML's generalization relation and require that principle solutions must specialize an elementary function. Redefining the elementary effect and elementary geometry of the elementary function to concrete implementations, *i.e.,* principle effect and principle geometry yields a solution to the respective elementary function [KK98]. The principle effect defines a behavior in terms of physical laws that are expressed as equations over the names of the functions ports and geometric variables. The geometric variables are declared as such in the principle geometry. To indicate that a variable of the physical effect is a geometric variable, the modeler creates a BindingConnector between the corresponding ValueProperty of the «PrincipleEffect» and the «PrincipleGeometry» This interaction of the principle effect and the principle geometry refines the behavior of the elementary function. Principle solutions are solutions to elementary functions. SysML4FMArch distinguishes between principle solutions and solutions because combining the (principle) solutions to sub-functions often requires to specify joints between the active surfaces of the sub-(principle) solutions and enhance the physical effect by equations to describe the connection. SysML4FMArch offers respective stereotypes for principle solutions that refine the stereotype «Function». A «PrincipleSolution» implements an «ElementaryFunction» and possibly redefines the inherited «ElementaryGeometry» and «ElementaryEffect» to a «PrincipleGeometry» and a «PrincipleEffect», respectively. This step may be delayed during the development process, which the multiplicity indicates. SysML4FMArch uses BindingConnectors to specify the constraints between attributes of principle geometries and principle effects as well as the function's interface. BindingConnectors between the «EffectElement»-ConstraintProperties and the «GeometricElement»-PartProperties of a «PrincipleSolution» link the ValueProperties of the geometry to the ConstraintParameters of the effect. Thereby, the modeler defines which of the variables used in the physical laws of the physical effect are geometric properties of the solution. Blocks with stereotype «Solution» represent a solution to a composite function. The stereotype therefore refines the stereotype «Architecture». Joints are PartProperties of a Block with stereotype «Solution» that represent physical connections among the

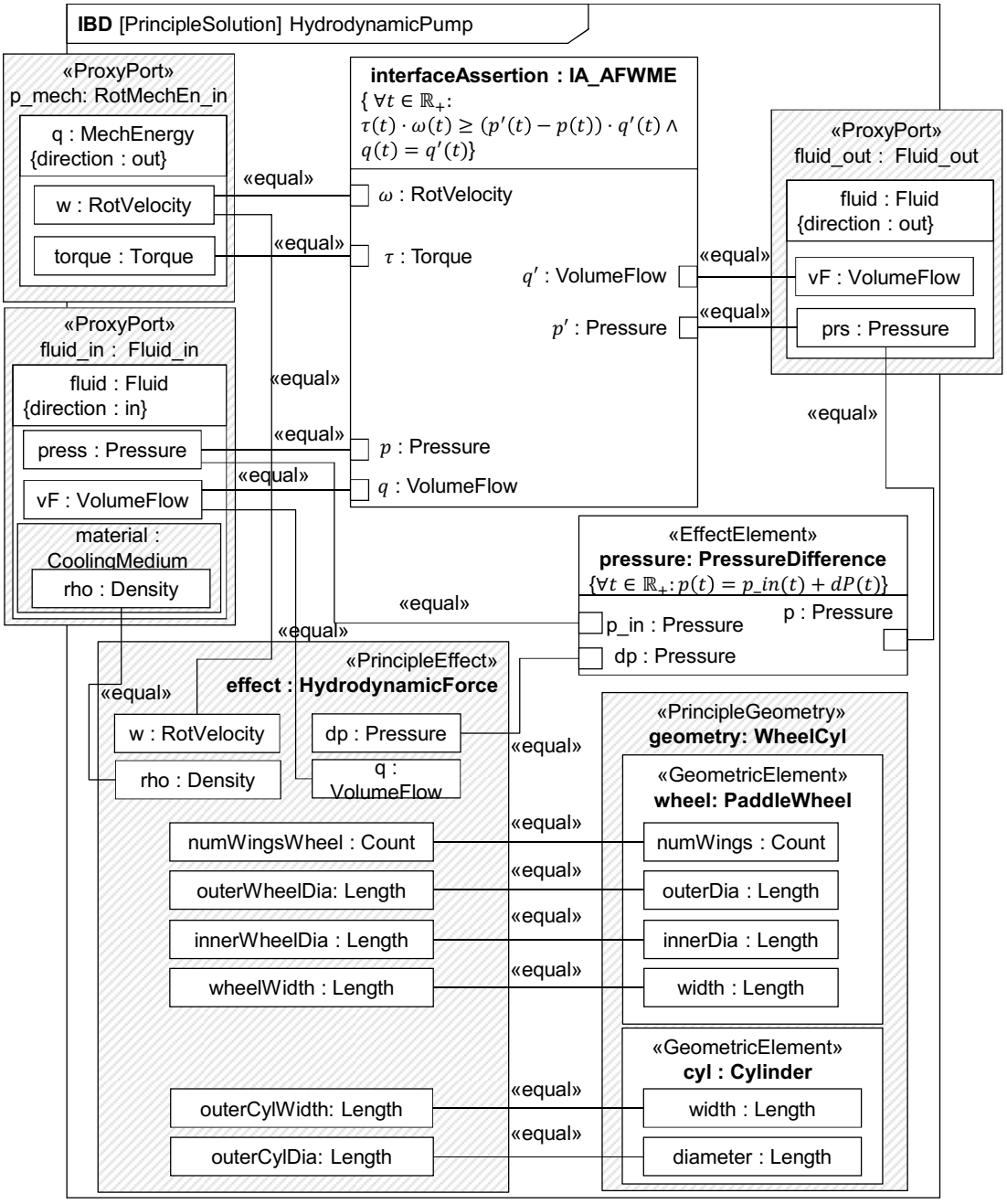


Figure 6.9: Principle solution of `ApplyFluidWithMechEn` which relies on hydrodynamics acting on a paddle wheel within a cylinder. The model represents a hydrodynamic pump.

active surfaces of (principle) solutions to the functional components of a function. The constraints between the ValueProperties of a Block with stereotype «Solution» and the contained «PrincipleSolution» components are modeled equivalently.
Figure 6.9 shows an IBD of `HydrodynamicPump`, which specializes the `ApplyFluid- WithMechEn` (*cf.* Figure 1.5). `Hydrodynamics` specializes `EE_AFWME`, the «ElementaryEffect»of `ApplyFuidlWithMechEn` (*cf.* Figure 1.5). Refining a physical function to a principle solution corresponds to creating the principle solution as a specialization of the «ElementaryFunction». An external simulation linked to the «EffectElement» modeling Equation 6.1 assumes the fluid to flow through a tubular pipe comprising a paddlewheel. The «PrincipleGeometry» `WheelCyl` specializes the «ElementaryGeometry» of `ApplyFluidWithMechEn` and has PartProperties of type `PaddleWheel` and `Cylinder`. These represent a pair of active surfaces which enforce the represented effect, and assign the attributes of the effect to distinguishable geometric shapes. The pressure of the outgoing fluid is given by 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 interface assertion uses the power balance to specify this.

6.3 Implementation of SysML4FMArch in MagicDraw

For the evaluation of the modeling approach for physical functions, we have implemented the SysML-profile SysML4FMArch in the modeling tool MagicDraw[4]. As part of our evaluation, we have modeled the automotive electrical coolant pump that was introduced as a running example in Section 1.3.3. How we modeled the system in SysML4FMArch will be detailed in Section 7.2.This section details the implementation of SysML4FMArch as defined in Section 6.2 as a MagicDraw profile.

6.3.1 Implementing Graphical DSLs in MagicDraw

MagicDraw is a modelling tool that implements the UML 2 standard [Man17]. Besides UML, the tool offers comprehensive extensions [GJRR22b] such as the SysML plugin[5] that implements the SysML standard and the CAMEO simulation toolkit[6] which, among others, provides model execution mechanisms and enables to interlink simulation or CAD models to the model elements in MagicDraw. These functionalities can be used for automating testing and dimensioning activities in the mechanical design process which is conceptualized in Section 7.2. Recently, MagicDraw's profiling mechanisms that allow defining custom graphical modeling languages have come into

[4]https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/
[5]https://www.3ds.com/products-services/catia/products/no-magic/addons/sysml-plugin/
[6]https://www.3ds.com/products-services/catia/products/no-magic/cameo-simulation-toolkit/

the focus of MLE [GJRR22b, GKR+21]. In these approaches, the languages that can be defined as profiles of the UML meta-model implemented by MagicDraw are conceived as DSLs in the sense of [Fow10]. The approach aims to bring the three pillars of MDE, into the domain of systems engineering closer together, these are the modeling languages, the methodology, and the tools [GJRR22b]. For the implementation of SysML4FMArch as a DSL in MagicDraw [GJRR22b], we have roughly followed the engineering process for developing industrial DSLs proposed in [GKR+21]. The process is divided into three levels [GJRR22b]: (1) the concept level, which includes (i) defining reusable language components that define the syntax of the modeling language [HR04], (ii) outlining a methodology to use the modeling elements such that the user's modeling goals are met, as well as (iii) defining standards and usability heuristics to achieve a good user experience. (2) The tool-specific implementation level: At this level the language engineers identify and realize the reusable DSL building blocks as MagicDraw profiles. Figure 6.10 illustrates how we have applied the conceptual model of the engineering process proposed in [GKR+21] for the engineering of SysML4FMArch.

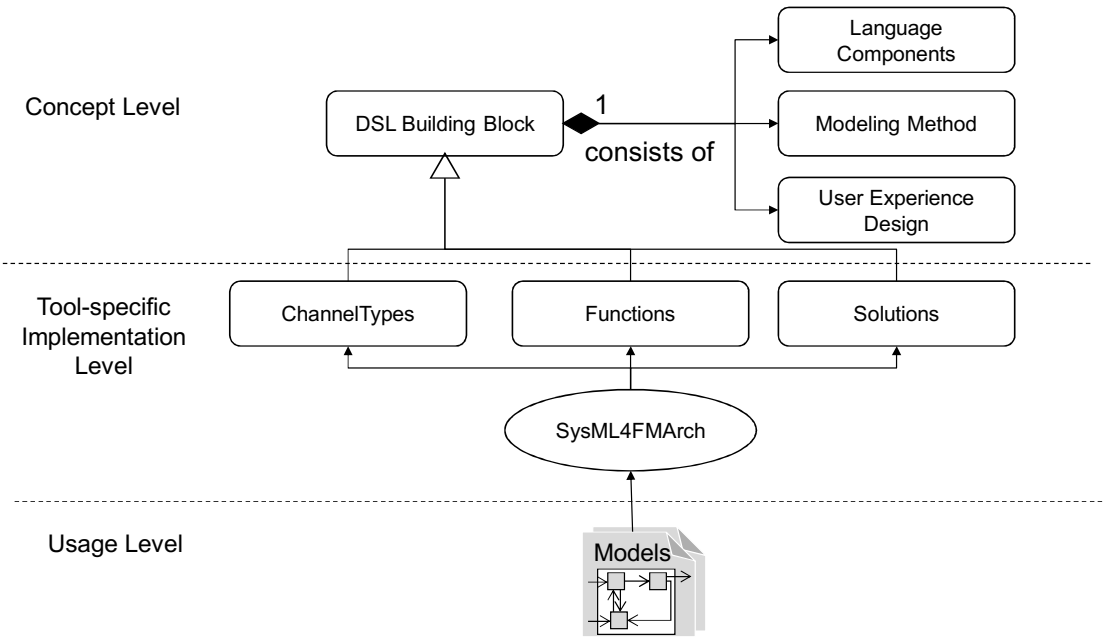


Figure 6.10: Conceptual model of the engineering process introduced in [GKR+21] as applied for the engineering of SysML4FMArch.

6.3.2 SysML4FMArch Language Components as a MagicDraw Profile

For the implementation of SysML4FMArch, we have identified three reusable DSL building blocks:

> Channel Types implement the profile presented in Figure 6.3 which represents the concepts of cyber-physical types introduced in Chapter 2
>
> Functions implement the profile presented in Figure 6.5 that represents the concepts of CPFs presented in Chapter 3, and
>
> Solutions that implement the profile presented in Figure 6.7 that puts the concepts from Figure 5.4 into a modeling language.

Together, the three building blocks form the implementation of SysML4FMArch defined in Section 6.2 as a MagicDraw language profile. Finally, level (3) is the dedicated usage level at which experts use the modeling language to model the system in MagicDraw.
For systematic distribution, MagicDraw offers mechanisms to create plugins that are installed within the application by the user. The SysML4FMArch implementation was distributed in the form of such a plugin that included the profile definition, an (incomplete) model library that digitalizes, among others, the information from the Koller catalog [KK98], and a set of Java extensions that implements warnings for certain disallowed modeling constructs that could not be out-ruled in the profile. MagicDraw profiles consist not only of stereotypes that implement UML 2 [Man17] meta-classes but also of *customization elements*. The latter allow for defining rules or context conditions for stereotypes. MagicDraw does not allow stereotypes to implement SysML elements but only to define stereotypes that inherit from SysML stereotypes or UML meta-classes.

Structuring Stereotypes For each language, we have defined stereotypes and respective customization elements to define the modeling elements along the profile outlined in Section 6.2 and additional elements to structure SysML4FMArch models and their elements systematically. Each stereotype definition is enhanced by a *customization* element of the same name that defines additional properties for the stereotype: The *abbreviation* defines the name of the element when it is created by the user [GJRR22b]. This name appears in the drop-down menu or in the selection department. The *category* defines a category in the drop-down menu that appears when the user right-clicks on an element with the respective stereotype in the containment tree in which the respective element is listed. This enables defining categories for elements of the language profile. The *disallowedRelationship*-list allows to prohibit relationships to respective model elements. The *hiddenOwnedTypes*-list defines stereotypes that shall be hidden from the user in the drop-down menu. To

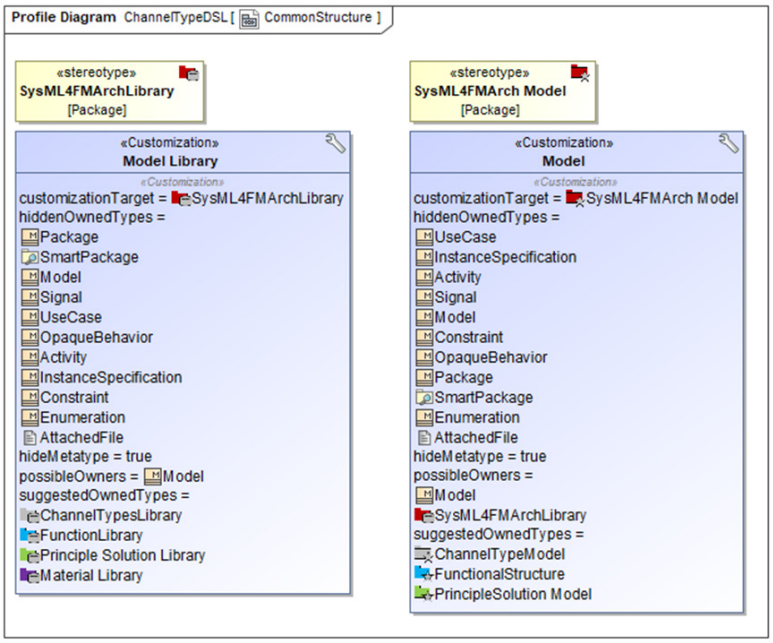


Figure 6.11: Stereotypes that provide the overall structure for models in SysML4FMArch.

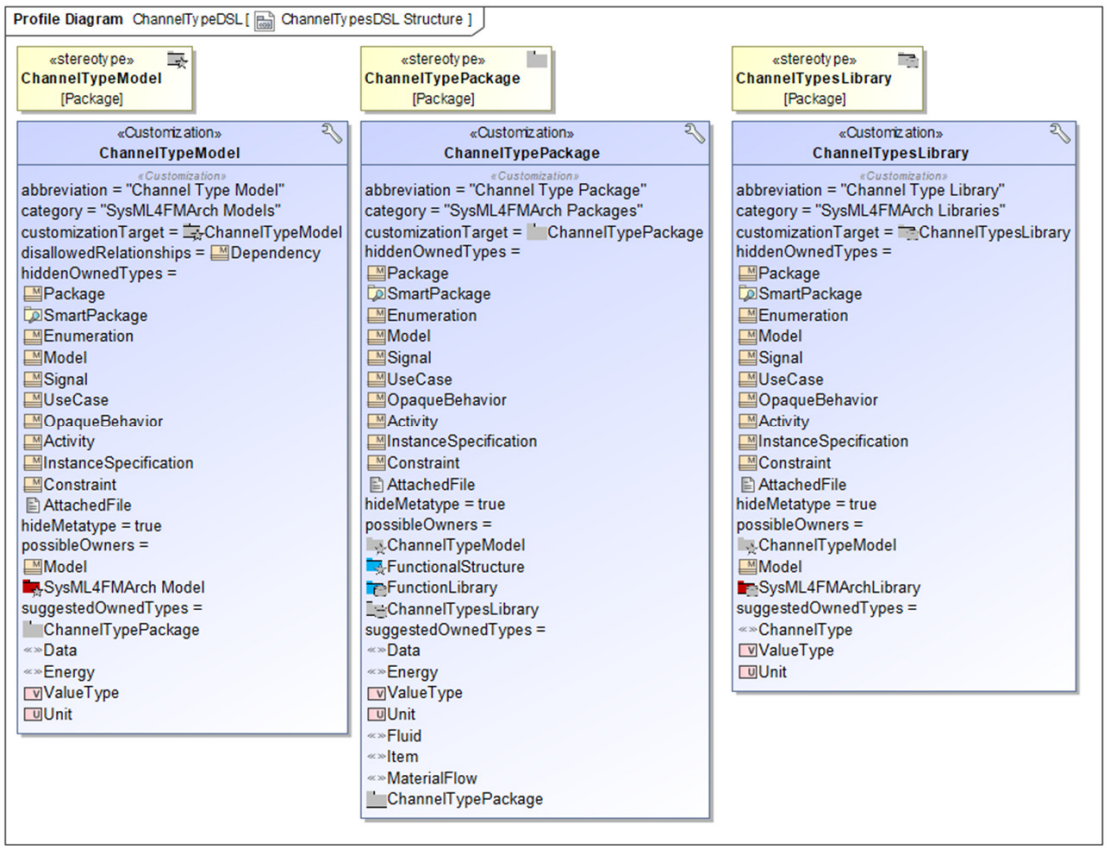


Figure 6.12: Stereotypes for structuring SysML4FMArch type definitions in MagicDraw.

prevent users from using the standard SysML elements instead of those elements that belong to SysML4FMArch we used this list to hide the standard SysML elements which would yield incorrect models. The *possibleOwners*-list defines the stereotypes of the elements that may contain the definition of the elements the customization targets. For SysML4FMArch, we have used this list, *e.g.,* to define the elements that may be contained in a *ChannelTypeModel* (*cf.* Figure 6.12). In MagicDraw, everything is owned by an instance of the meta-class *Model* which is, therefore, also in the *possibleOwners*-list. The *suggestedOwnedTypes*-list allows to customize the drop-down menu which appears with a right-click on an element with the respective stereotype in the containment tree.

Figure 6.11 shows the structural stereotypes that structure SysML4FMArch models and Figure 6.12 provides the modeling constructs for structuring type definitions in dedicated packages. All of these stereotypes implement the package stereotype which is a UML 2 [Man17] meta-class. A *Model* is the overall structural element for a system specification. *ModelLibraries* enable storing functional system models for reuse. All elements that are part of a functional system model in the SysML4FMArch implementation in MagicDraw will be contained in a *Model* or *ModelLibrary*. As part of our project, we have implemented parts of the Koller catalog [KK98] in SysML4FMArch, whose elements are structured in a *ModelLibrary*. The automotive cooling system which is our running example (*cf.* Section 1.3.3) is created as a *Model* that uses elements from the *ModelLibrary* that digitalizes parts of the Koller catalog. Figure 6.12 shows the implementation of the respective stereotypes and customizations in the SysML4FMArch implementation.

ChannelTypeModel's enable users to dedicate a set of type definitions for a specific purpose, *e.g.,* in a project or subdivision. Channel type definitions are structured in *ChannelTypeModels* which contain *ChannelTypePackages*. A *ChannelTypeLibrary* stores reusable definitions of cyber-physical types.

Stereotypes for Modeling Cyber-Physical Types Figure 6.13 shows the implemented stereotypes for defining cyber-physical types together with their respective customizations. The set of stereotypes follows the specification of SysML4FMArch presented in Figure 6.3. If one of the well-formedness rules Well-Formedness Rule 3, Well-Formedness Rule 1, Well-Formedness Rule 4, and Well-Formedness Rule 2 is hurt, the plugin produces a warning for the modeler which is implemented in the plugin using MagicDraw's Java API. The implementation of the language components of the function and solution building blocks are documented in Appendix E. To integrate the well-formedness rule Well-Formedness Rule 5, we have created SysML4FMArch-specific port types and hide SysML's standard ProxyPorts from the dropdown menu.
Figure E.8 shows the implementation in MagicDraw that prevents ambiguous port directions.

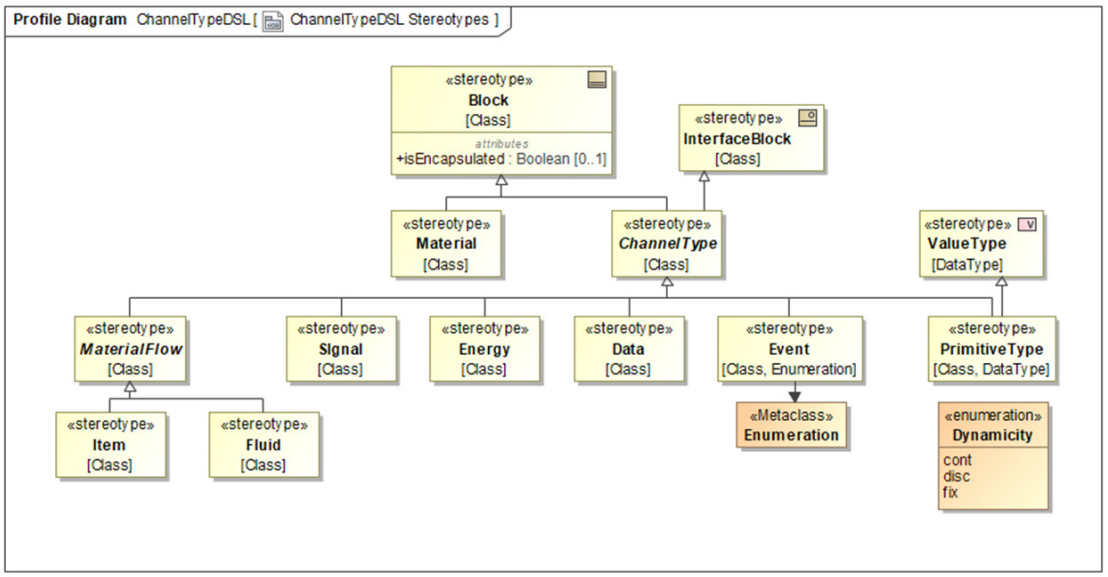


Figure 6.13: Stereotypes for modeling defining cyber-physical types in the SysML4FMArch MagicDraw-profile.

6.3.3 The Modeling Method of SysML4FMArch in MagicDraw

A motivation to digitalize the Koller catalog and support Koller's design methodology through systematic modeling techniques with tool support is to systematize the search for known solutions to recurring functions. The MagicDraw implementation of SysML4FMArch utilizes SysML's generalizations and redefinition mechanisms to fulfill this modeling goal.

Figure 6.14 illustrates the modeling methodology for principle solutions on the example of the elementary function AFWME and a model of the *HydrodynamicPump* which is one possible principle solution. In SysML4FMArch, elementary functions hold two PartProperties whose types are *elementary effect* and *elementary geometry*. These are Blocks with respective stereotypes that are abstract. These serve as placeholders for the effect and geometry. *Principle effects* and *principle geometries* specialize the *elementary effect* and *elementary geometry* to indicate that they are suitable parts to create a principle solution for the elementary function. Using this modeling methodology enables a systematic and very easy selection process for (principle) solutions for (elementary) functions in MagicDraw, which Figure 6.15 illustrates. When redefining the MagicDraw-Attributes of a (principle) solution that are inherited from the (elementary) function, the user can choose from a list of the suitable alternatives that are lodged in the project or an imported modeling library. The modeler, thus,

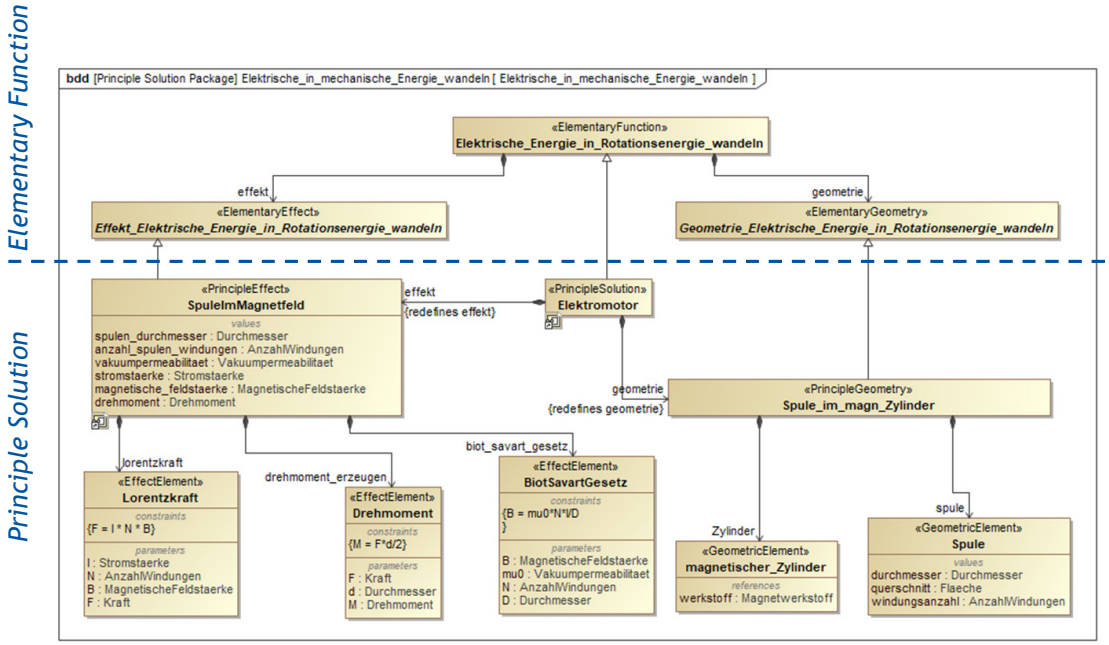


Figure 6.14: BDD that illustrates the modeling methodology for principle solutions on the example of the elementary function that converts electrical energy to rotational energy.

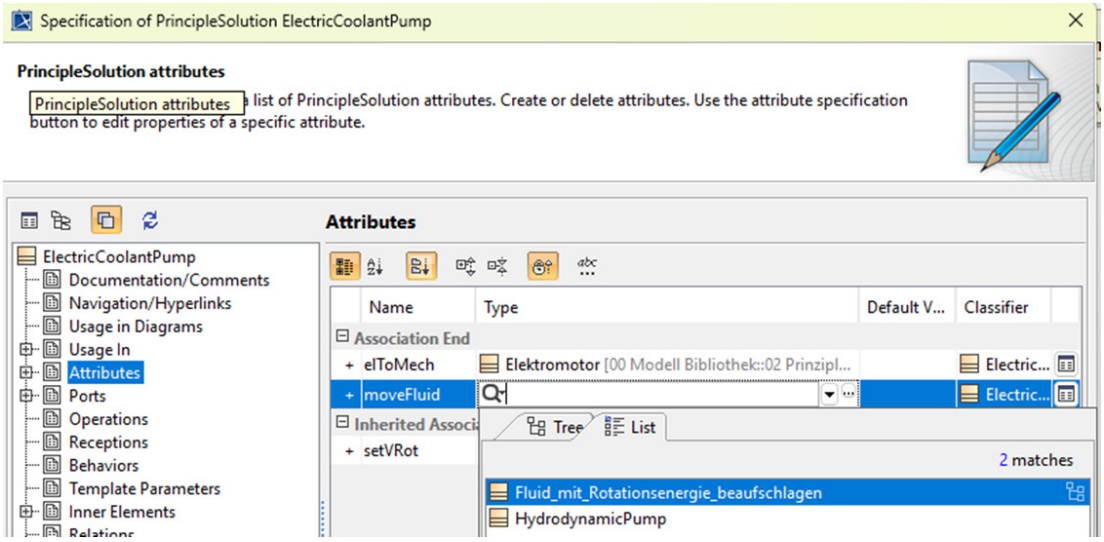


Figure 6.15: The redefinition mechanism implemented in MagicDraw enables the user to select from the list of lodged (principle) solutions to (elementary) functions.

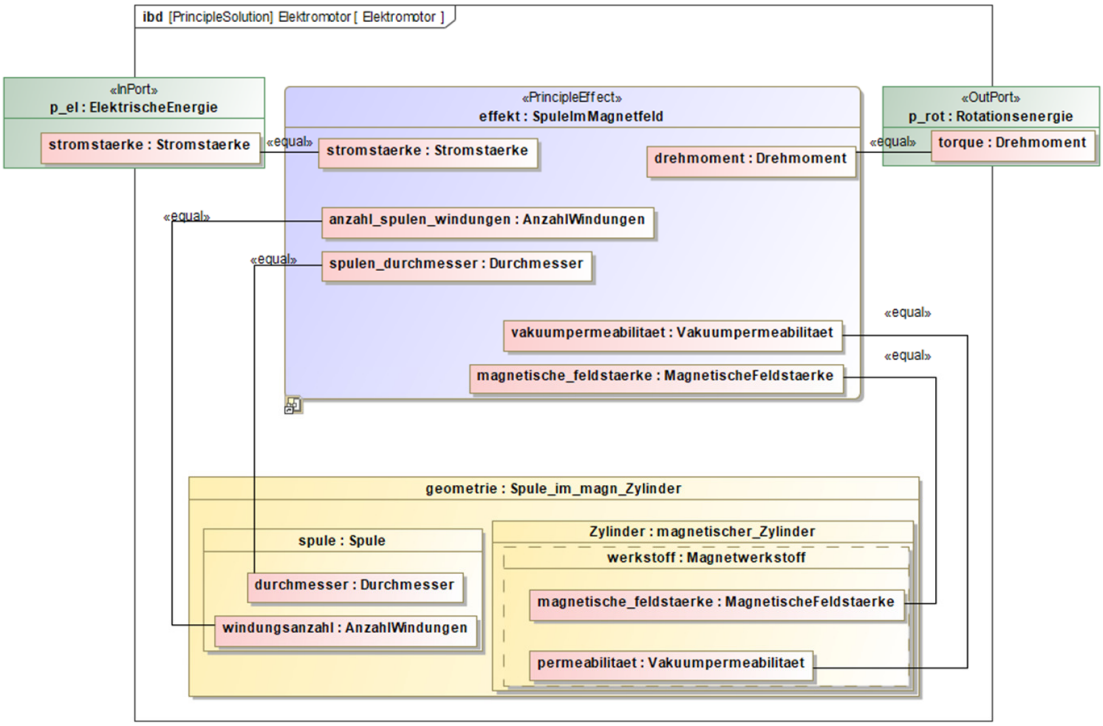


Figure 6.16: IBD of the «PrincipleSolution» that describes the electric engine. The example illustrates how to interlink the *principle effect* and the *principle geometry* in a principle solution.

does not have to infer from the model which effects or geometries are suitable once solutions for the function s/he is working on are lodged. If no solution is lodged, or the lodged solutions are ruled out, *e.g.,* due to requirements, the model user can create new solutions by creating other principle solutions that also inherit from the elementary function.

Once the modeler has chosen a principle effect and a principle geometry, s/he creates an IBD and links the respective parameters using BindingConnectors. The BindingConnectors mean the equality of the values of the ValueProperties at all times and, thereby, represent the dependency of the physical phenomenon described by the *principle effect* and the geometric appearance of the system quantified by the *principle geometry*. Figure 6.16 shows the IBD of the «PrincipleSolution» that describes an electric engine.

The methodology works analogue for solutions, *i.e.,* the model user creates a solution that specializes an architecture and redefines the PartProperties that represent the sub-functions of the functional structure. In the redefinition process, s/he can choose

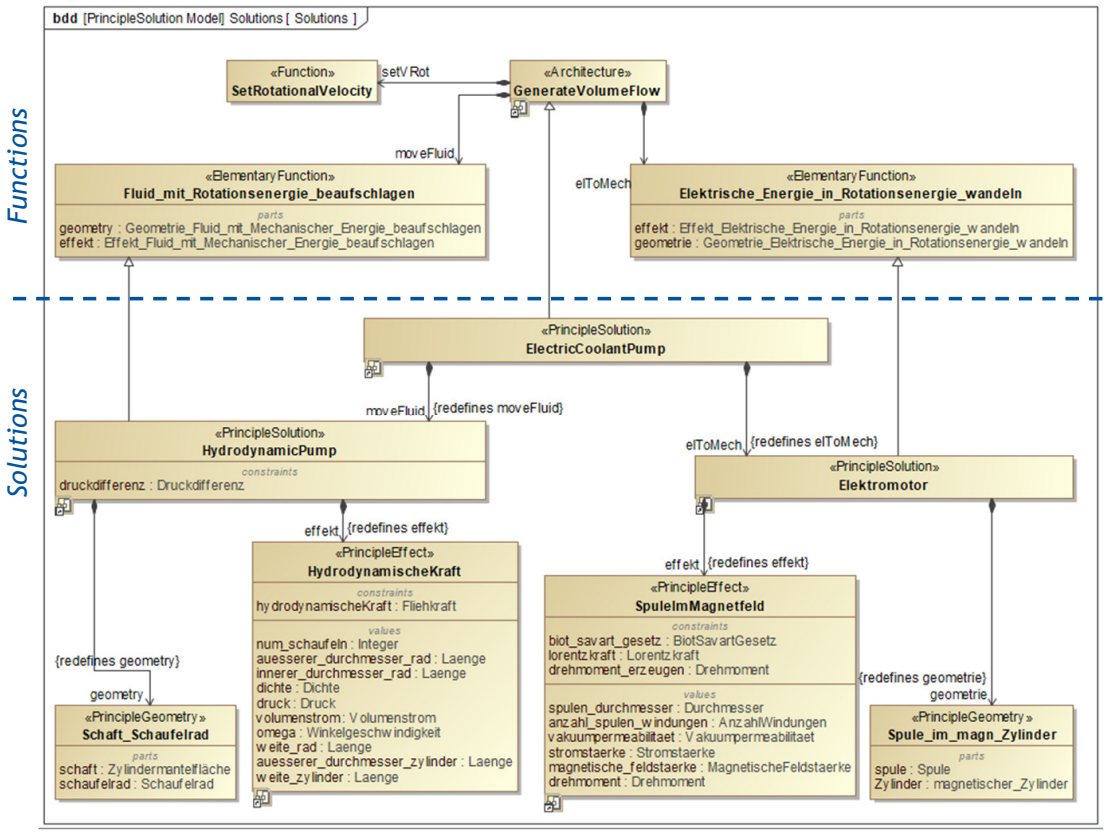


Figure 6.17: BDD that illustrates the modeling methodology for solutions on the running example from Section 1.3.3. So far, the component `setVRot` does not yet have a solution.

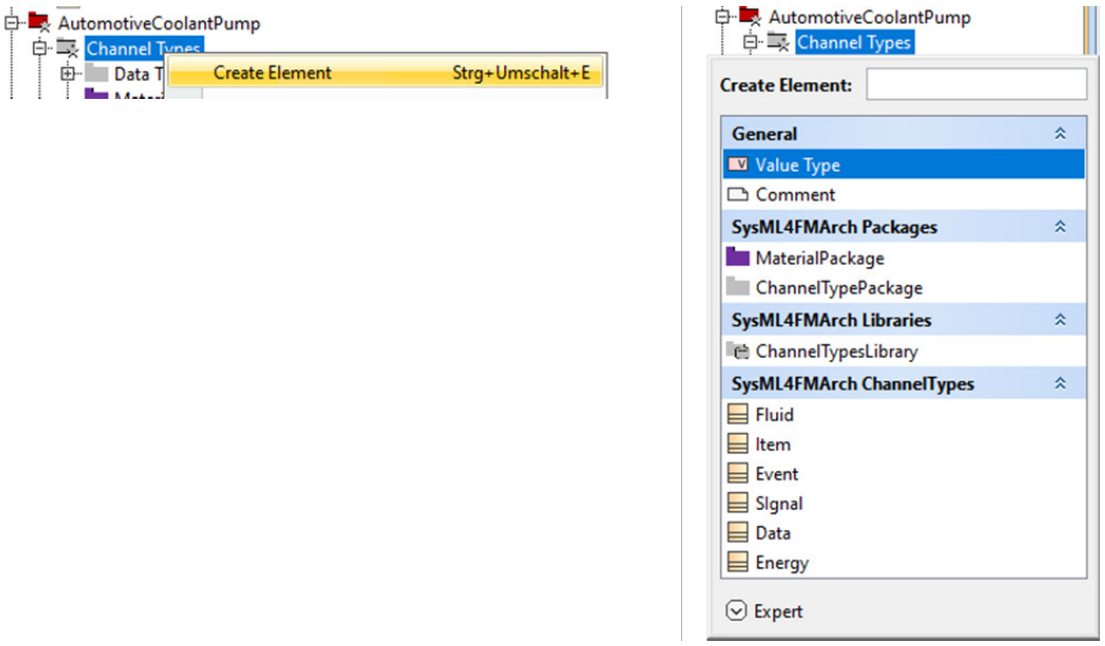


Figure 6.18: Example of the drop-down menu for creating elements of a *ChannelTypeModel* as specified by the customization shown in Figure 6.12.

known solutions for each sub-function from a drop-down menu that lists only appropriate solutions. Figure 6.17 illustrates how solutions are modeled in MagicDraw.

6.3.4 User Experience Features of SysML4FMArch in MagicDraw

The implementation of SysML4FMArch as a MagicDraw profile offers certain features to improve the user experience. Mainly these features focus on structuring the available modeling elements to make it as intuitive as possible for the model user to use the elements correctly. The customizations for the stereotypes of SysML4FMArch include properties to customize the user experience. For SysML4FMArch, we used the properties *category*, *suggestedOwnedTypes*, and *hiddenOwnedTypes*. These properties customize the drop-down menu for creating elements to an element in the containment tree. The drop-down menu appears upon a right mouse click on the element in the containment tree, and selecting "Create Element" Figure 6.18 shows the customized drop-down menu for *ChannelTypeModels*. The *category* allows structuring the modeling elements displayed in the drop-down menu. The *suggestedOwnedTypes*-list enables to define those modeling elements that shall appear in the drop-down menu, while the *hiddenOwnedTypes*-list enables hiding elements from MagicDraws standard

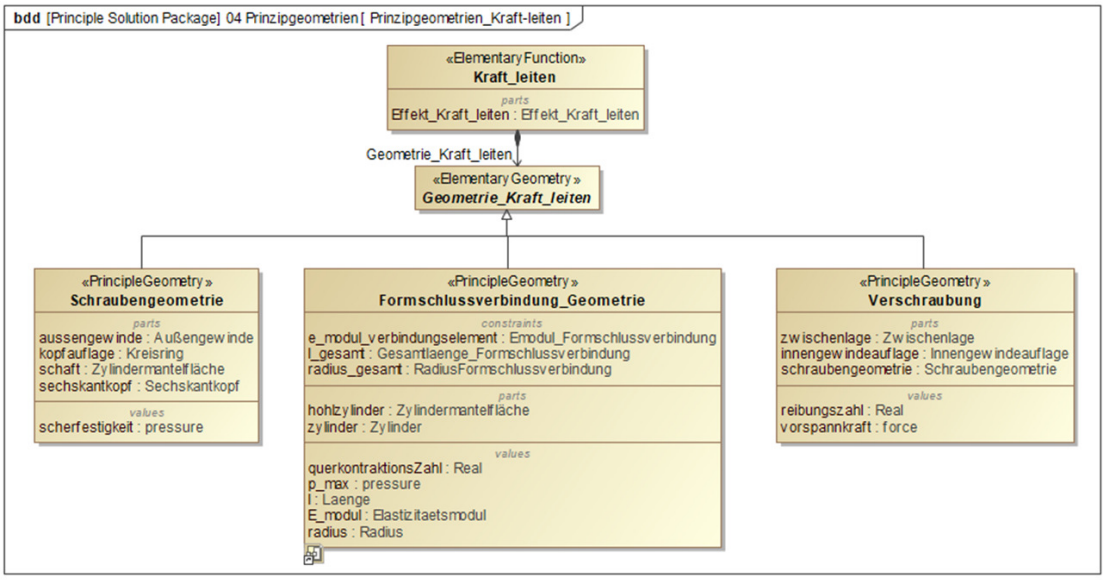


Figure 6.19: Example of the principle geometries offered for creating principle solutions to the elementary function "conduct force" in the model library provided with the MagicDraw SysML4FMArch plugin.

SysML profile from the user to prevent their usage.

6.4 A Digital SysML4FMArch Design Catalog in MagicDraw

Along with the conceptual basis and the implementation of SysML4FMArch presented in this chapter, we have digitalized parts of the Koller design catalog [KK98] in MagicDraw with SysML4FMArch. The digital design catalog is set up as a *ModelLibrary* (*cf.* Figure 6.11) that includes a *ChannelTypeLibrary*, a *FunctionLibrary*, a *SolutionLibrary*, and a *MaterialLibrary* (see Appendix E for the definition of these SysML4FMArch elements in MagicDraw). The model library is included in the plugin. Figure 6.14 shows an example of the elementary function "connect fluid with rotational energy" that illustrates the general structure and relations of the elements contained in the library. So far, the library includes 41 elementary functions that are modeled using the methodology presented in Section 6.3.3. Further, the library contains four complete principle solutions for the elementary functions convert electrical to mechanical energy, apply rotational energy to a fluid, conduct rotational energy, and increase or decrease temperature. Principle solutions consist of a principle effect and a principle geometry, which consist of effect elements and geometric elements, respectively. The implementation of SysML4FMArch in MagicDraw has introduced this structure to

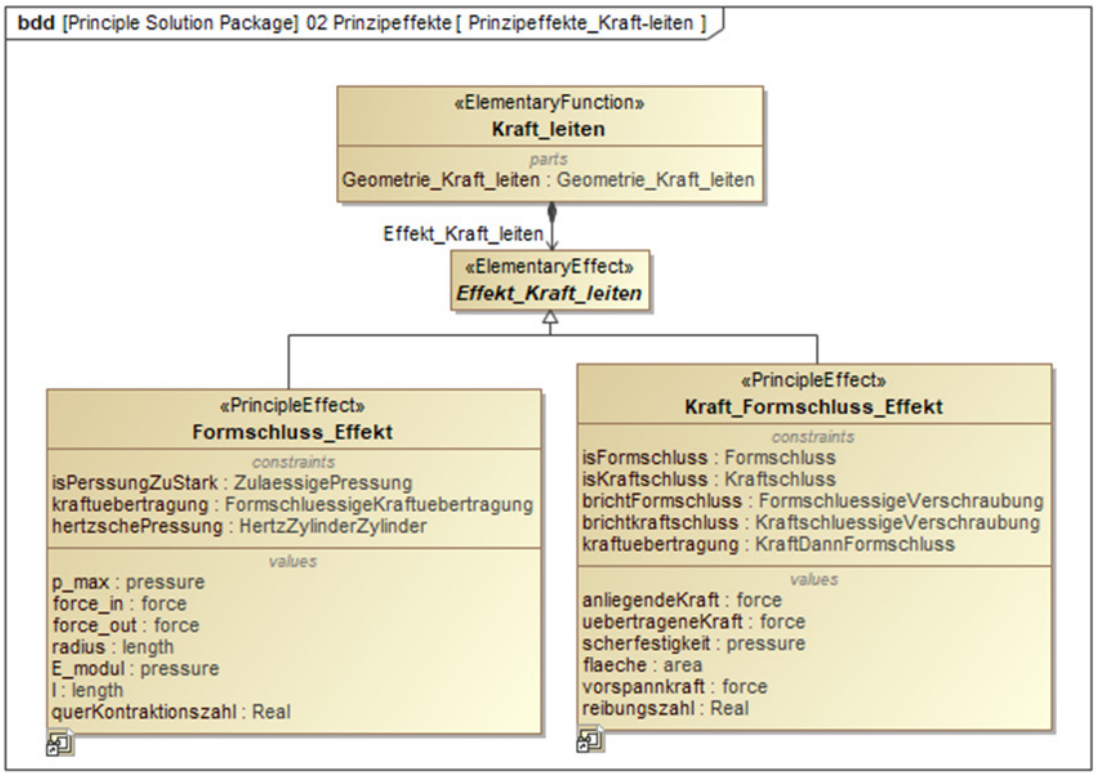


Figure 6.20: Example of the principle effects offered for creating principle solutions to the elementary function "conduct force" in the model library provided with the MagicDraw SysML4FMArch plugin.

enable storing physical effects and active surfaces as reusable building blocks of principle effects and principle geometries for each elementary function. In general, one principle geometry or principle effect can represent the geometry or effect of more than one principle solution which is why we have implemented SysML4FMArch such that these elements can also be stored as reusable building blocks for principle solutions. The library, so far, comprises nine principle geometries and six principle effects. For example, Figure 6.19 and Figure 6.20 show the principle geometries and principle effects offered in the library for building principle solutions to the elementary function "conduct force" represented by the SysML4FMArch element `Kraft_leiten`. Concerning the building blocks of principle effects and principle geometries, the library offers 32 effect elements and 25 geometric elements. Concerning types, the library comprises three data types (the project dealt mainly with mechanical systems that do not process much data, therefore, there are only three data types), 12 energy types,

seven material types, and 44 physical quantities.

6.5 Discussion and Related Work

Discussion. SysML is a general-purpose modeling language that is defined as a UML profile. Creating SysML models is intuitive for those who are familiar with object-orientation, which is mostly not the case for experts whose background is not related to software engineering or computer science. Therefore, it is very valid to ask the question of whether SysML is appropriate for modeling the functions considered in the engineering of mechanical systems and further, suited as a (basis for) a modeling language that unifies the domain and enables interdisciplinary collaboration. We encoded the meta-model presented in Section 6.1 as a SysML profile because SysML is fairly known in the automotive domain [DGH+19, KMS+18] and since there exist modeling tools with integrated model-processing. This also enabled to utilize the commercial tool MagicDraw for the implementation which comes with an implementation of SysML and extensive tooling to create custom graphical DSLs. The design of SysML4FMArch and its implementation in MagicDraw that uses the redefinition mechanism to select elements from the library that digitalizes the Koller catalog has simplified the process of creating models of principle solutions significantly. However, creating principle solutions and the respective elements it is composed of is very effortful. Further, there exist extensions and plugins that provide, *e.g.,* model execution engines. These have been used to automate dimensioning and testing using SysML4FMArch models in the design process [HJZ+21, HHB+22, ZRJ+22]. On the downside, the graphical notation of SysML that is also intended by MagicDraw may hinder the manageability of larger models with many attributes. Recently, the OMG announced and published a prototype of SysML v2 which comes with significant changes, among them, a textual variant of the language which may facilitate overcoming this issue. Further, OCL provides a formal and well-known modeling language for interface assertions and can be integrated with SysML [Man19] in tools such as MagicDraw[7]. This facilitates reusing existing tools that implement formal analyses based on the Focus semantics. While the work presented in this dissertation emerged, SysML v2 was not yet available, therefore SysML4FMArch is a profile of SysML 1.6.

Related Work. The formal theory of Focus was put into an MDE methodology named SPES [BBD+21]. Research conducted around this approach has also produced a SysML profile [GJRR22b] that implements concepts very similar to those presented here. Even more interesting, the SPES research group researches software language engineering for application in systems engineering with a focus on graphical

[7] https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/

languages [GJRR22a, GKR+21, FRR09]. As mentioned earlier, SPES is conceptually similar to the methodology and modeling techniques presented in this dissertation. For our purposes, we could unfortunately not reuse the SPES-profile as it was not available at the time and also, the sources are not available. Further, SPES does not specifically address the needs imposed by integrating mechanical design theory which is one of the main contributions of this work. For example, SPES does not provide mechanisms to describe or even consider geometry. SPES is based on the discrete Focus theory [BS01] but may integrate with our hybrid semantics which is also based on Focus. Another tool that implements the discrete Focus semantics is AutoFOCUS3 [AVT+15b] which is, however, tailored for distributed, embedded software systems. The theory introduced in [Alu15] is tailored for designing and verifying control systems. The UPPAAL-tool [BLL+96] provides a means to employ this technique in practice. With a strong focus on control engineering, the technique considers system components to be functional, and also regards geometric or materialistic properties of the system and its components. However, these appear only in the form of variables in the specification of the functional component. The geometric product architecture is not linked or regarded in the system specification and dimensioning is not possible. Similarly, in [Pto14] provides expressive means to specify a system in terms of its functions. In particular, the technique enables the utilization of multiple specification paradigms in one specification. Engineers can specify (functional) components using a specification paradigm that fits the component best. However, the technique does not offer to integrate geometric parts of the system that is later found in the physical product. Modeling languages based on UML or SysML have emerged in the mechanical engineering domain, *e.g.,* MechatronicUML [BGT04], SysML4Modelica [BvLK15], or SysML4- Mechatronics [KV13], in the field of production systems engineering [FHK+15], and in the context of Industry 4.0 [WBCW20], *e.g.,* UML4IoT [TC16]. Neither of these languages enables relating (elementary) functions and (principle) solutions of mechanical systems. The FAS-method [WLRW15, LW14], extended for mechanical engineering by FAS4M [MKG+15] promotes modeling functional architectures for system design and both define respective SysML profiles. As introduced in [MAK15, Moe15] these techniques use trace links to underly SysML elements with informal sketches of geometric components. The focus of these contributions lies on the connection between requirements and functions. 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.,* [KK98], 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 [WS09, GDP+10, EGZ12, ZAMM12]. In particular, the approaches in [GDP+10, EGZ12] do not consider the functional structures of [KK98, BG21] and do not systematically establish consistency between function and

principle geometry in a model-driven approach. Currently, precise modeling languages to support development processes based on mechanical design methodology [KK98, Kol14] do not exist. Explicit modeling techniques for a model-driven and functional approach to mechanical engineering do not support principles prevalent in software engineering such as abstraction, automation, composition, refinement, and separation of concerns.

Chapter 7

Evaluation

7.1 A Functional Model of an Audio Entertainment System

To evaluate whether the presented modeling technique is suitable to provide a functional and domain-independent model of a CPS which is part of the overall research question of this dissertation (*cf.* Section 1.2), Such systems are interesting in our context since modern implementations include cloud streaming services in which the music data is stored digitally and transmitted to the speaker wirelessly. HiFi audio systems, on the other hand, use the means of mechanics to play music.

7.1.1 Audio Entertainment Systems

In principle, audio entertainment systems transform music data into sound waves. Sound waves are oscillations of pressure that travel through space. Thereby, they transport energy in the form of compression [FR76], and the value of the pressure carries information. Sound waves are therefore analog signals. Since energy is a conserved quantity, the energy that is transported in the form of compression by the sound wave must be given as input to the system at some point. Typically, audio entertainment systems take energy in the form of electricity as input, which also comes in handy when transforming audio data into respective sound waves. The data is either stored in digital formats such as *e.g.,* mp3 or CDs, or analog, *e.g.,* on vinyl records or cassettes. It depends on the point of view whether or not to count storing the music as a part of the system. Here, we will exclude storing the data from the system of interest.

Audio Data To understand what audio data encodes, it is helpful to understand how sound is recorded, because playing music reverts this process. We, therefore, summarize the principles presented in [Wei12]. To record sound, a microphone is placed in the space through which the sound waves travel to capture the sound waves to be recorded. Since the sound waves carry compression energy, the changes in pressure cause a membrane within the microphone to start moving. The membrane is made from a material that conducts electricity and placed within a magnetic field. The movement of the membrane is translated to an electrical voltage by the law of conduction (if an

electrical conductor moves within a magnetic field, the Lorentz force induces a current in the conductor) which thereby encodes the changes in pressure caused by the sound wave. Depending on the type of microphone, the membrane is placed in a magnetic field, or such that the membrane forms a terminal of a capacitor. In both cases, the value of the electric voltage at the membrane changes proportionally with the pressure of the sound wave. Audio data encodes these changes in the electric voltage over the time of the recording. Loudspeakers revert this process. Typically they utilize changes in the electric voltage to cause a change of current in an electrical coil that is placed movable inside of a magnetic field. By the law of induction [HMS16], this results in a Lorentz force on the charge in the coil, which becomes apparent as a movement of the coil. The moving coil is coupled to a membrane which will start moving along the coil and thereby change the volume of the environment. These changes in volume indicate an exchange of energy between the membrane and the environment in the form of compression energy whose intensive quantity is the pressure [FR76]. If the pressure swings with audible frequencies, it carries sound information. Because the physical processes in a loudspeaker employ the same physical laws as the physical processes that allow the recording only in reverse order, the resulting pressure is again proportional to the voltage, and the resulting sound, therefore, echoes the recorded sound information. Table 7.1 shows a CPCD that models compression energy and sound signals as cyber-physical types. The definitions of the two energy types follow [FR76] and we consider sound information to be carried by pressure. This interpretation abstracts from the form in which the voltage information is provided which could be in the form of a record or a digital format such as mp3. As discussed multiple times throughout this dissertation, the exchange of energy in the form of electrical energy is bound to the quantities current and voltage which is reflected by the respective attributes of the class `ElectricalEnergy` in Table 7.1. Compression energy is a form of energy that is bound to the quantities pressure and volume flow rate [FR76], where Pressure $\stackrel{\text{def}}{=} \mathbb{R}(\text{Pa})$ and VolumeFlowRate $\stackrel{\text{def}}{=} \mathbb{R}(\text{m}^3\,\text{s}^{-1})$. Sound is continuous information that is conveyed by changes in pressure. The class `Sound` represents this signal-type in Table 7.1.

CPCD

«energy» ElectricalEnergy
Voltage v
Current i

«energy» Compression
Pressure p
VolumeFlowRate dV

«data» Sound
Compression c

Table 7.1: CPCD of the types used for modeling the audio entertainment system.

Audio Systems A system that plays music, takes audio data as input, and creates a sound wave, such that the pressure at each point in time is proportional to the value of the voltage encoded in the audio data. Therefore, Table 7.2 specifies the CPF to take a voltage signal as input and to produce a sound signal with a pressure that is proportional to the value of the voltage, once a Boolean button press event initiates turning the music playing on or off. The proportionality factor is represented by the parameter n in the specification. Further, the system that realizes the specified CPF can be switched on and off via a button press.

CPF

PlayAudioData($\mathbb{R}_+$ δ))

in «signal» $\mathbb{R}$(V) voltage, «event» $\mathbb{B}$ buttonPress out «signal» Sound sound
$\forall t \in \mathbb{R}_+ : \exists n(t) \in \mathrm{C\,m^{-3}}$: $lt(\text{buttonPress}, t) \Rightarrow \text{sound}(t+\delta).c.p = n(t) \cdot \text{voltage}(t)$ $\neg lt(\text{buttonPress}, t) \Rightarrow \text{sound}(t+\delta).c.p = 0\text{Pa}$ where $lt(c, t^*) = c(\max\{0 \leq t \leq t^* \mid c(t) \neq \xi\})$

Table 7.2: Specification of the overall function of the audio system

Following mechanical design theory [BG21], we have to decompose the overall CPF in Table 7.2 into smaller sub-functions. A possible result is displayed in Figure 7.1 In the formal methodology proposed in Section 4.2, this decomposition should yield a refinement or refactoring. The decomposition already requires making certain design decisions. Here we decompose the function into three components: (1) a function that supplies electrical energy upon a button press, (2) a function that amplifies the signal that represents the audio data, and (3) a function that transforms the incoming electrical energy to a sound wave such that the pressure of the sound wave is proportional to the incoming voltage. The functionality of providing energy upon a button press has been discussed in the context of modeling energy sources as elementary functions specified in Koller's design catalog [KK98] in Section 5.2.1. Here, we reuse the elementary function defined in Table 5.18 for electrical energy with slight adaptations: The function produces a fixed power upon a button press and, therefore, does not have an energy-input port typed of type ElectricalEnergy. The function is specified in Table 7.3. The respective paragraph in Section 5.2.1 provides possibilities for alternative specifications.

Voltage provided by the energy source needs to be adjusted to get as close to the incoming voltage information provided via the signal coming into the function on the input port `voltage`. A CPF that increases or decreases an incoming voltage until it

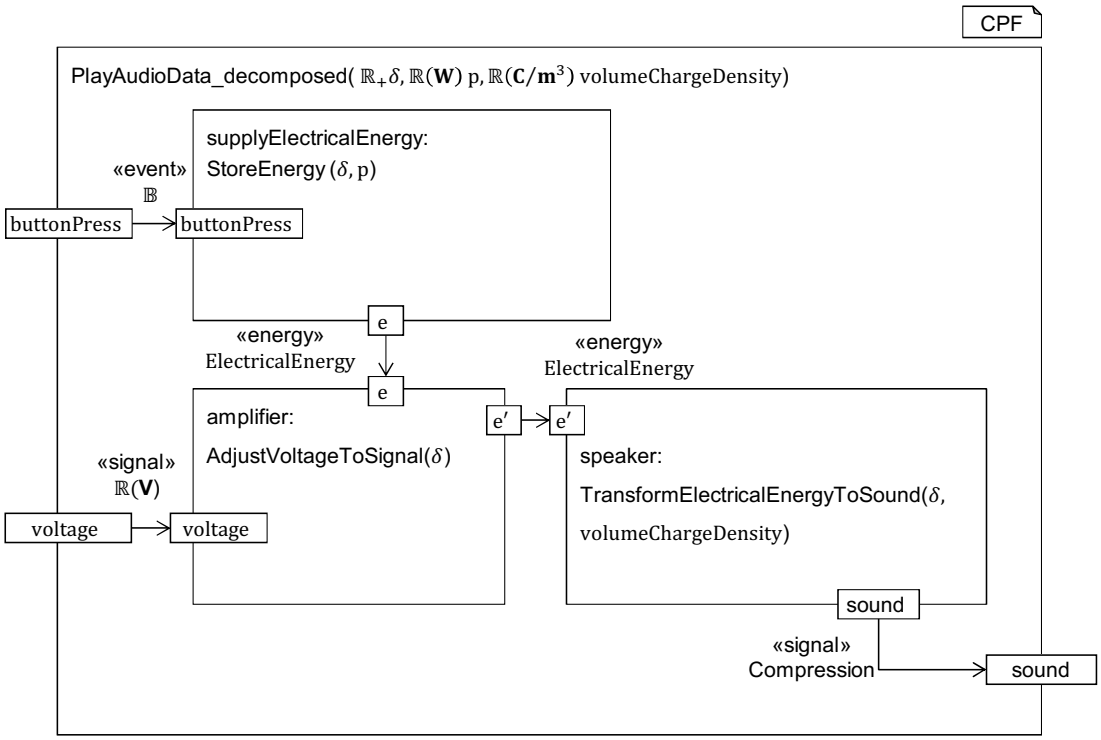


Figure 7.1: Compositional black box specification of the CPF `PlayAudioData`.

CPF

StoreEnergy($\mathbb{R}$ δ, W p)

in «event» buttonPress out «energy» ElectricalEnergy e
$\forall t \in \mathbb{R}_+$: $lt(\text{buttonPress}, t) \Rightarrow \text{pow}(e)(t+\delta) = \text{p}$ $\neg lt(\text{buttonPress}, t) \Rightarrow \text{pow}(e)(t+\delta) = 0\text{W}$ where $lt(c, t^*) = c(\max\{0 \leq t \leq t^* \mid c(t) \neq \xi\})$

Table 7.3: Specification of energy storage that provides a fixed amount of power upon a button press.

reaches a value specified by an incoming signal models an electrical control system. The outgoing voltage is gradually increased/decreased to reach the specified value after some time delay δ. The CPF `AdjustVoltageToSignal` in Table 7.4 models this with a linear course (the value of the voltage of the outgoing electrical energy at time $t + \delta$ is determined by the difference quotient between the voltage signal incoming on the port v at time t, and the voltage of the incoming electrical energy on port e at time t). Of course, it is possible to specify other courses of the voltage to reach the value of the signal. The system then converts the electrical energy that is bound to the

CPF

AdjustVoltageToSignal($\mathbb{R}$ δ)

in «signal» $\mathbb{R}$(V) voltage, «energy» ElectricalEnergy e out «energy» ElectricalEnergy e'
$\forall t \in \mathbb{R}_+$: $e'(t+\delta).v = \text{voltage}(t) \cdot e(t).v$

Table 7.4: Specification of the CPF that adjusts an incoming voltage to a value specified by an incoming signal or data message.

adjusted voltage to compression energy which is the audio signal that the listener's ear will receive. Table 7.5 specifies the transformation of the incoming electrical energy to compression energy carrying the audio information such that the pressure of the sound wave is proportional to the incoming voltage. The proportionality factor is the so called volume charge density which is a parameter of the function. Here, it becomes apparent that the specified system reverts the recording process by specifying the relation between the voltage and the pressure.

CPF

TransformVoltageToSound($\mathbb{R}(\mathrm{C\,m^{-3}})$ volumeChargeDensity, $\mathbb{R}$ δ)
in «energy» ElectricalEnergy e' out «signal» Sound sound
$\mathrm{pow}(\mathrm{sound})(t) \leq \mathrm{pow}(e')(t)$ $\mathrm{sound}(t+\delta).c.p = \mathrm{volumeChargeDensity} \cdot e'(t).v$

Table 7.5: This CPF is an energy transformer that transforms electrical energy to compression energy. The pressure which is a component of the compression energy encodes the audio information in the format understood by the human ear.

Check Refinement Following the functional MDE methodology proposed in Section 4.2, the decomposed specification in Figure 7.1 should yield a refinement or refactoring of the specification in Table 7.2. To verify that PlayAudioData_decomposed is a refinement of PlayAudioData, we need to show the following:
Let $\delta > 0$, $p \in \mathbb{R}(\mathrm{W}), vCD \in \mathbb{R}(\mathrm{C\,m^{-3}})$ be parameters of the decomposed function PlayAudioData_decomposed(δ, p, vCD). Further, let $f \in [\![\text{PlayAudioData\_decomposed}]\!]$ be a TSPF in the behavior of that CPF. We need to show that there exists $\delta' > 0$ such that $f \in [\![\text{PlayAudioData}(\delta')]\!]$. That is, for all times $t \in \mathbb{R}_+$ and all input channel histories $x \in \overrightarrow{\{\text{buttonPress}, \text{voltage}\}} \stackrel{\text{def}}{=} \vec{I}$ there exists an output channel history $y \in \overrightarrow{\{\text{sound}\}}$ such that $f(x) = y$ and the predicate in the specification of PlayAudioData in Table 7.2 holds.
To this effect, let $x \in \vec{I}$ and $t \in \mathbb{R}$. The first case is that $lt(x(\text{buttonPress}), t)$. Then, by the definition of StoreEnergy(δ), we have that

$$\mathrm{pow}(e(t+\delta)) = p \;\Leftrightarrow\; e(t+\delta).v \;\cdot\; e(t+\delta).i = p \tag{7.1}$$

By the definition of AdjustVoltageToSignal(δ) it holds that

$$e'(t+2\delta) = x(\text{voltage})(t) \;\cdot\; e(t+\delta).v \tag{7.2}$$

Further, by the definition of TransformVoltageToSound we have that

$$\mathrm{sound}(t+3\delta).c.p = vCD \;\cdot\; e'(t+2\delta).v \tag{7.3}$$

Plugging in Equation 7.2 and Equation 7.3 and setting $\delta' = 3\delta$, we get that

$$\mathrm{sound}(t+3\delta).c.p = \frac{vCD \;\cdot\; e(t+\delta).i}{p} \;\cdot\; x(\text{voltage})(t) \tag{7.4}$$

Setting $n(t) = \frac{vCD \cdot e(t+\delta).i}{p} \cdot 1$ V in the definition of PlayAudioData in Table 7.2, we get that $\text{sound}(t+\delta').c.p = n(t) \cdot x(\text{voltage})(t)$. Assuming that $lt(x(\text{buttonPress}), t)$ for $t \in \mathbb{R}$, we get that $e(t+\delta) = 0$ which implies by a very similar chain of equations that $\text{sound}(t+\delta').c.p = 0\text{Pa}$. It follows that there exists $y \in \vec{O}$ such that $f(x) = y$ and the predicate from Table 7.2 holds. Therefore, $f \in [\![\text{PlayAudioData}(\delta')]\!]$.

7.1.2 Discussion

The specification of the audio system in Table 7.2 abstracts from the technical details of an implementation of an audio system and specifies the CPF to PlayAudioData from a customer's point of view. The decomposed version in Figure 7.1 considers certain technical details, *e.g.,* that the electrical energy provided by a source needs to be adjusted to the signal encoded in the audio data before it can be converted to sound. Or, that the electrical switch requires a force to switch on or off the supply of electrical energy. However, this decomposed specification is still very high-level and abstracts, *e.g.,* from the format in which the audio data is provided. The format dictates whether the input is a discrete or a dense stream which has implications on the interface assertions of the other functions. By this example, we show that it is possible to abstract from the domain that provides an implementation because analog data formats such as vinyl records would require mechanical implementations of the sound-generating functions, whereas digital formats require software implementations of these functions. Independent of the implementation domain, however, the CPF defined by an audio entertainment system remains the same. Starting from our model, a development process would continue to make design decisions, *e.g.,* about which data format the system takes as input which will have implications on the definition and decompositions of the other functions.
Considering the functional point of view which abstracts from technical implementations, one may argue that modeling the audio system from a functional point of view must consider audio data as the information about which value the pressure had at each point in time during the recording. This is a question of how the interface of the CPF is defined. Audio data often encodes a voltage which is due to the process in which it is obtained. Here, we do not abstract from the existing standards of audio data, and consider the CPF to be implemented by an audio entertainment system, to decode the voltage information stored in the audio data. It is, of course, possible to abstract from the fact that audio data encodes a voltage that is proportional to the sound it records. Such a specification would leave open how the sound information is encoded in an incoming signal. However, as audio data *always* encodes a voltage in practice [Wei12], we consider the given specification more realistic for the evaluation of our approach.
Another way to model the information is to model the voltage input as a data type. The stream at the corresponding channel of Figure 7.1 would then be declared as

«data». This would change the definition of AdjustVoltageToSound as it would process a discrete stream instead of a dense hybrid stream.

7.2 Modeling an Automotive Electric Coolant Pump in SysML4FMArch

This section details the application of SysML4FMArch in an interdisciplinary industrial project where it was used to model the cooling system for an automotive combustion engine drive train (*cf.* Section 1.3.3). To address the research question RQ4 "How can these functional specifications facilitate dimensioning and testing to support agile development of CPSs?" We have conceived concepts and modeling techniques that enable describing dimensioning and testing procedures in a SysML model. There exist tools such as the CAMEO Systems modeler[1] which is a derivative of MagicDraw for systems engineering that comes with functionalities to link elements in these models with simulations or CAD models and automatically execute these models as specified by the SysML model. Section 7.3.1 and Section 7.3.3 provide the conceptual basis for functional dimensioning and testing in SysML. This section presents and completes the SysML4FMArch models of the electric coolant pump from [DRW+20], a subsystem of the cooling system together with models for the (automatic) dimensioning and testing of the principle solutions.

7.2.1 Channel Types

The models in Figure 7.2 and Figure 7.3 define all channel types used in the models of the electric coolant pump: The energy types `ElectricalEnergy`, and `RotMechEnergy`, the material flow `Fluid` and the signal `ControlSignal`. The first two represent two types of energy. To model the difference we utilized the essential idea of Bond Graphs [GB07]: A flow of energy results from the simultaneous intervention of two independent physical quantities often referred to as (generalized) flow and (generalized) effort [MC12]. The power, *i.e.,* the amount of energy transferred per time, is given by the product of these two physical quantities. Depending on the type of energy, these physical quantities vary. The channel types modeled in Figure 7.2 illustrate this: The attributes of the flow properties of electrical energy, for example, are of type current which is the flow variable of the electrical energy domain, and of type voltage which is the effort variable of this energetic domain. To this effect, the channel types representing electrical and rotational mechanical energy each hold three attributes: `ElectricalEnergy` represents electrical energy in terms of a current (flow) and a voltage (effort). The value of electrical power is given as the product of the values of these two quantities, which is modeled by the constraint property `power`

[1] https://www.3ds.com/products-services/catia/products/no-magic/cameo-systems-modeler/

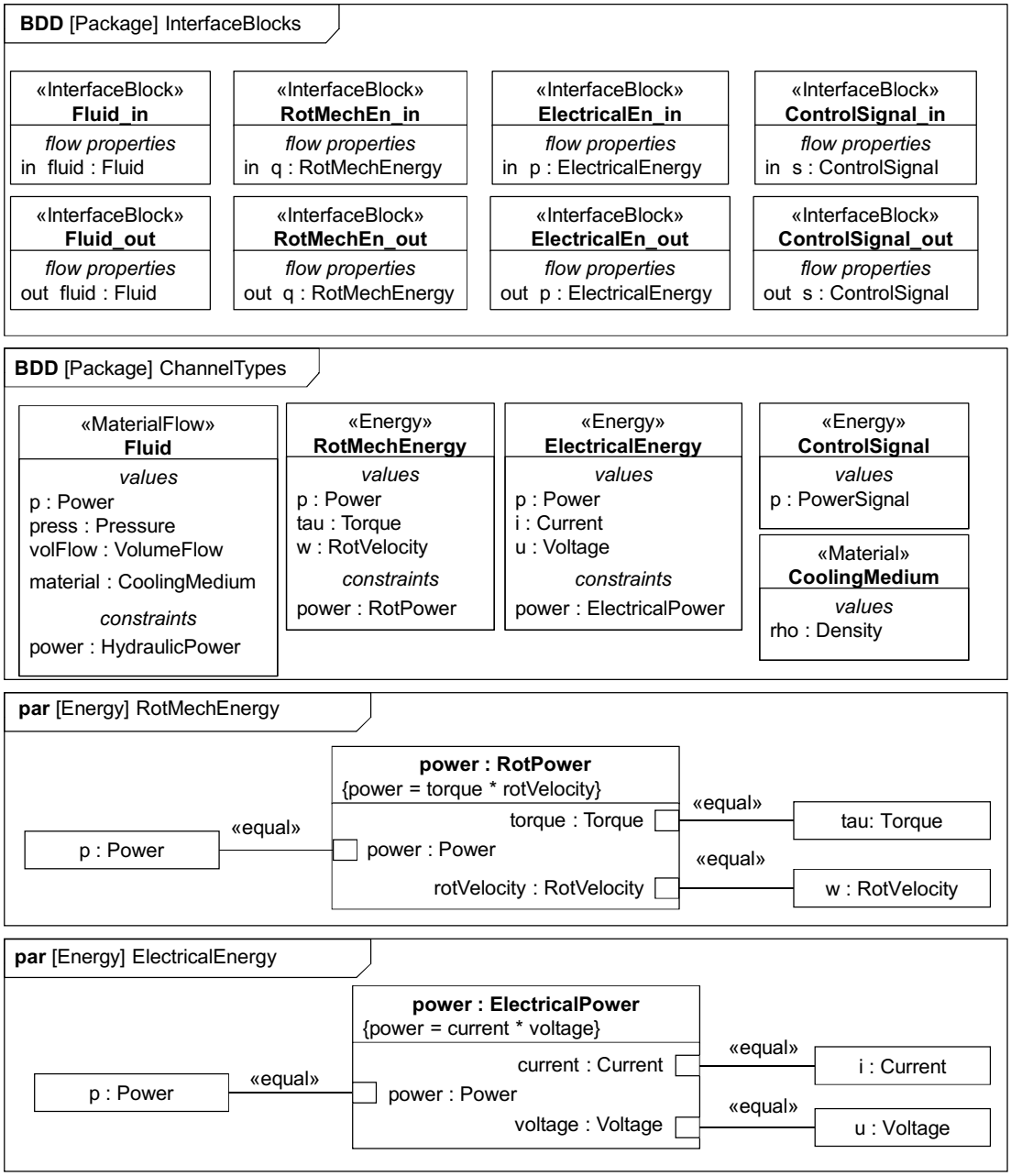


Figure 7.2: Channel types used in Figure 1.5 for modeling the automotive coolant pump in SysML4FMArch [DRW+20]. Top: Interface Blocks for typing the ProxyPorts of functional interfaces. Middle and bottom: BDD containing the type definitions for the flow properties of the interface blocks together with their internal structure: Power calculates as the product of two other attributes [GB07].

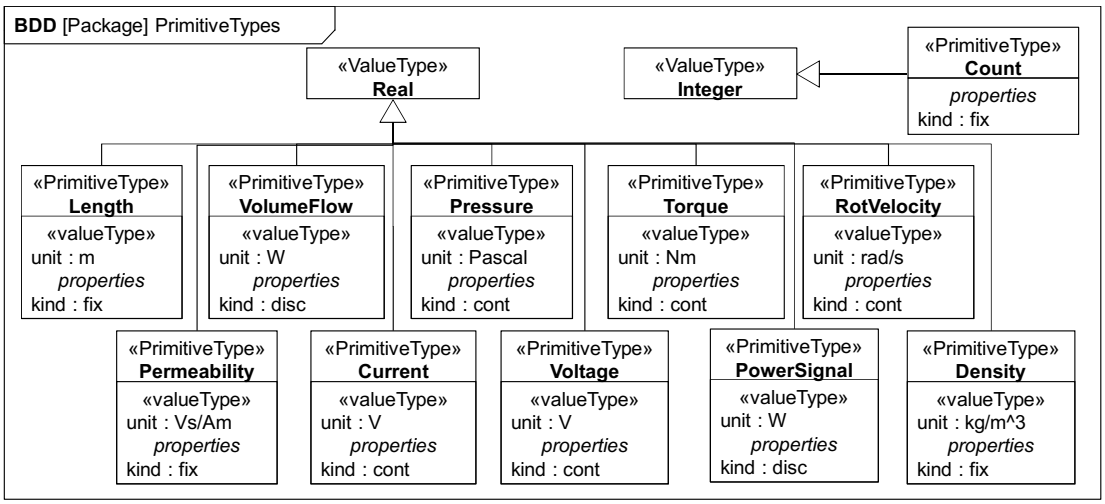


Figure 7.3: Value types used for typing the attributes of the channel types used in the automotive coolant pump specification [DRW+20].

of type `ElectricalPower` which relates the effort and flow attributes correspondingly. Similarly, `RotMechEn` represents rotational mechanical energy in terms of rotational velocity (flow) and torque (effort). Again, the constraint property `power` of type `RotPower` relates these attributes and the p-attribute of type `Power` to represent their mathematical relationship.

According to [Tay08], a physical quantity is the product of a number and a unit. SysML offers an integration of this notion with ValueTypes [Man19] based on the SI. Most of the attributes in the electrical coolant pump have real numbered values and therefore their «Attribute»-types specialize the value type `Real`. The SI-unit of `Power` is Watt (W). The attributes of `Fluid` are of types `VolumeFlow` measured in $\mathrm{m}^3\,\mathrm{s}^{-1}$ and `Pressure` measured in Pascal (Pa). Both represent quantities whose values change continuously. The attributes of `ElectricalEnergy` are of types `Current` specified in Ampere (A) and `Voltage` specified in Volt (V). `RotMechEnergy` holds attributes of type `Torque` in Newton metre ($\mathrm{N\,m}$) and `RotVelocity` in radiant per second ($\mathrm{rad\,s}^{-1}$). All of these types specify physical quantities whose values change continuously over time, and therefore their `kind` is specified as `cont`. The type `PowerSignal`, on the other hand, is an attribute of the «Signal» `ControlSignal`, which models the exchange of information through a channel. In this case, the information tells by which factor the power currently generated by the pump has to be adjusted. Attributes of this type, thus, change their values discretely. The types `Length` holds the SI-unit meter (m), and `Density` with the SI-unit kilogram per metres cubed ($\mathrm{kg\,m}^{-3}$) and types attributes whose values are fixed during runtime,

such as *e.g.,* dimensions or the density of a material. Similarly, `Count` specializes the `Integer` value type and also types attributes whose values are fixed during runtime.

7.2.2 Architecture of the Electric Coolant Pump

The coolant pump's main functionality is to keep the cooling medium flowing which is necessary for convection to occur, which causes the cooling medium to absorb heat from the engine's cylinders. The IBD of the «Architecture» `GenerateVolumeFlow` in Figure 1.5 shows the functional architecture that specifies this functionality as a composition of three functions. The architecture has three ProxyPorts that model the 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 6.3 shows the InterfaceBlocks for typing the ProxyPorts representing the functional flows of fluid. As its type indicates, the flow `flowControl` represents an incoming signal flow (changing its value discretely at runtime) telling how much the pump's power needs to be adjusted, in order to drive the flowing fluid to generate enough hydraulic power to absorb enough heat from the engine. The latter is modeled as «Function», *i.e.,* so far it is unknown or irrelevant whether the function that adjusts an incoming flow of electrical energy and outputs the adjusted flow of electrical energy on `pEl_out`, is further decomposed and is simply used as a black-box component. The interface assertion specifies the behavior of this function in a way that is similar to an example from [Bro12]: The function outputs rotational power, given by the product of the outgoing torque and rotational velocity, as a multiple of the incoming rotational power. The factor by which the power is increased is the last value received on the «Signal»-port. The power increase takes some time which the formula captures by the delay δ. In case no message is received on the port from time $t - \delta$ to t, the function regulates the electrical power to be zero.
Section 5.1.1 describes how to model the «ElementaryFunction» `TransformElEnToMechEn` which transforms the incoming electrical energy at `p_el` into mechanical energy at `p_mech_out` including its interface assertion (*cf.* Figure 6.5). The specification defines functional behaviors that transform the incoming electrical power to mechanical power but allows losses which, as part of a refinement, could become part of the interface.
The function `ApplyFluidWithMechEn` impinges this mechanical energy `p_mech` upon the incoming fluid `cm_in` and resulting in the outgoing flow `fluid_out`. The parametric diagram at the bottom of Figure 7.4 shows the interface assertion and the relationships in the internal structure of the physical function: The formula specifies the functional behavior in terms of power balance and utilizes the idea from bond graphs explained above [GB07]. In this case, hydraulic power is calculated as the product of pressure difference at the input and the output interface and the volume flow rate of the flowing fluid. To regard the law of mass conservation, the volume flow

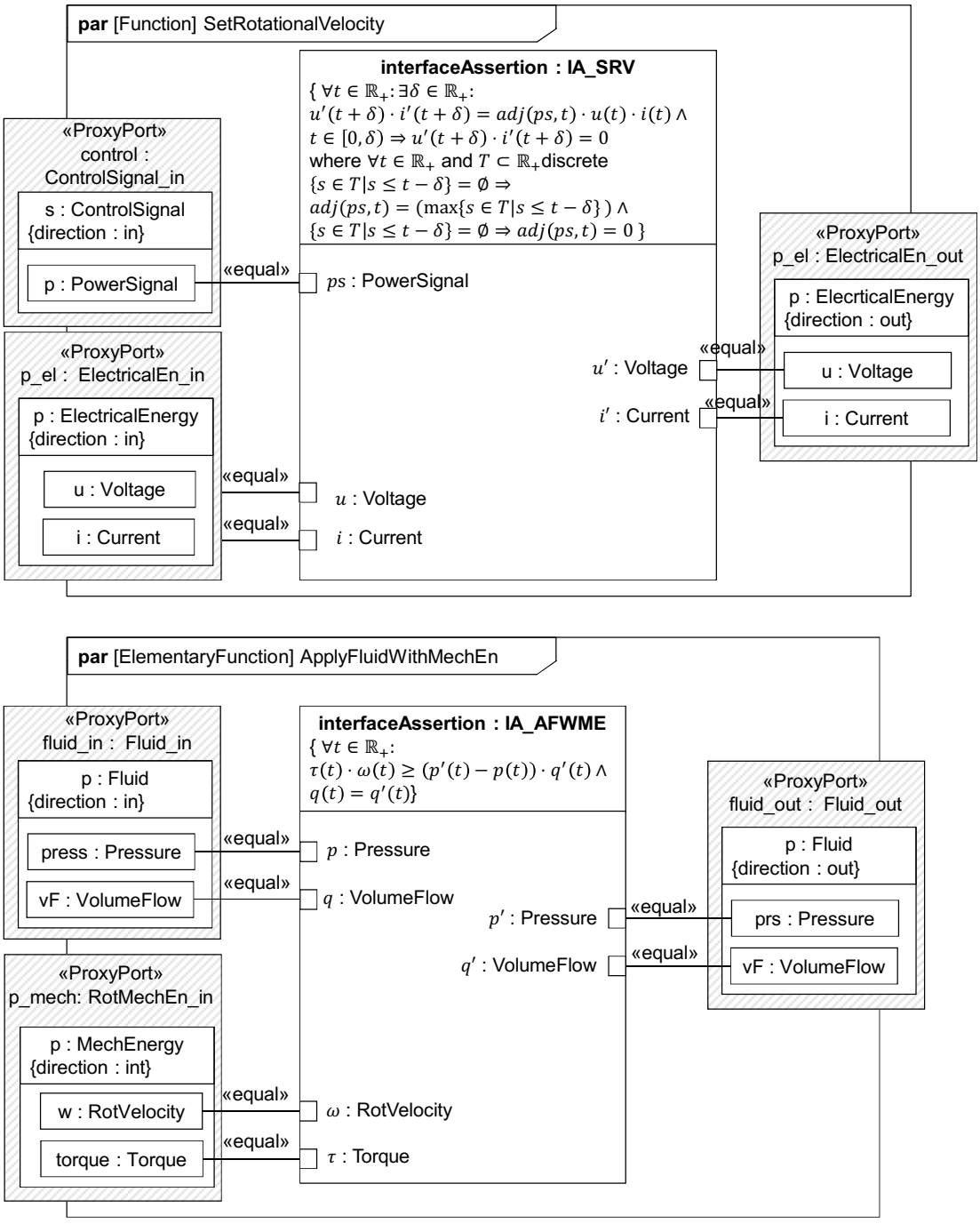


Figure 7.4: IBD of `ApplyFluidWithMechEn` and `SetRotationalVelocity` with interface assertions.

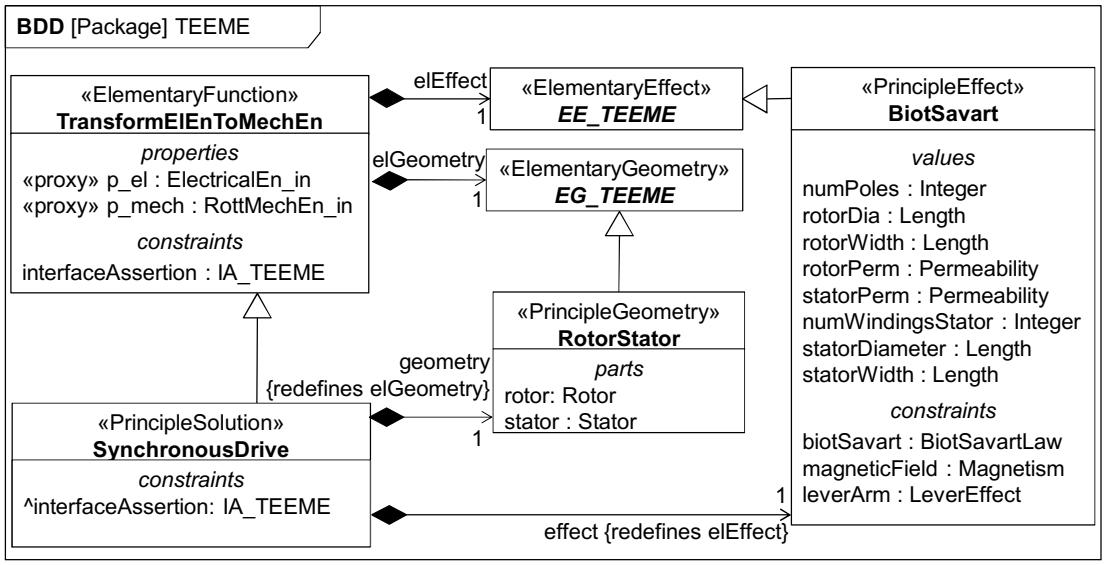


Figure 7.5: Model of a possible principle solution to realize the elementary function `TransformElEnToMechEn`. The BDD shows the structural relations of the `TransformElEnToMechEn` and `SynchronousDriving` and their respective parts.

rate remains constant. The resulting hydraulic power is less than or equal to the incoming rotational mechanical power which is given by the product of torque and rotational velocity to allow specifying energetic losses in principle solutions to this function. The inequality enables including energetic losses in this specification through a refinement step [Bro18].

7.2.3 Solution-Models

The «Architecture» `GenerateVolumeFlow` comprises two «ElementaryFunctions» for which [KK98] lists physical effects suitable to realize these functionalities. The solution of the architecture considered here is an electric coolant pump. Therein, an electric motor drives a paddlewheel which is placed within a cylinder through which the cooling medium flows. The paddlewheel is part of the principle geometry in the principle solution to `ApplyFluidWithMechEn` which accelerates the fluid through hydrodynamics. Figure 6.8 shows the model of this principle solution in SysML4FMArch which was explained previously in Section 6.2.3. This section presents a model of a hydrodynamic pump as a solution to the overall function `GenerateVolumeFlow`.

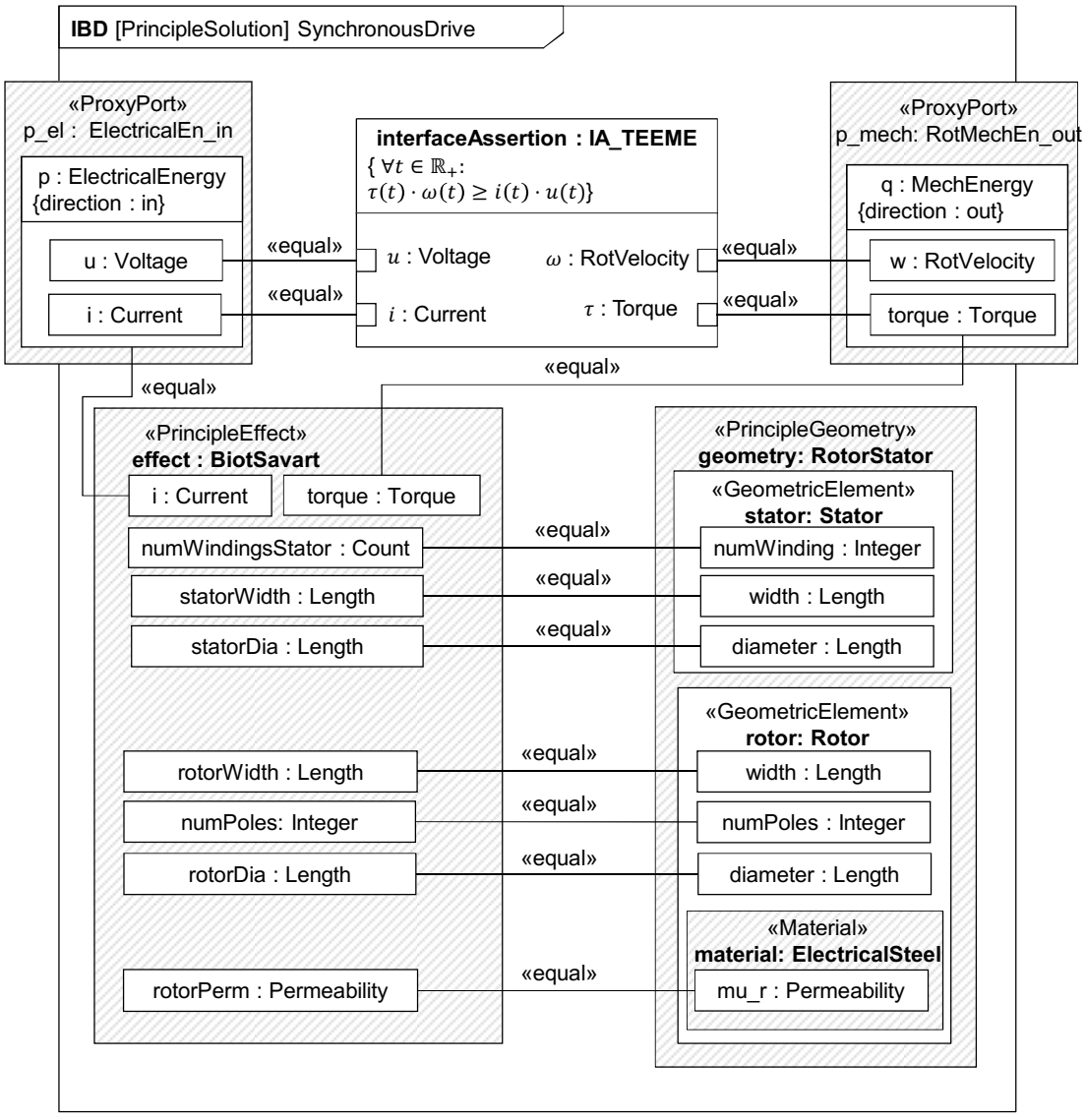


Figure 7.6: IBD
of the principle solution `SynchronousDrive`.

Principle Solution to Transform Electrical To Mechanical Energy: Figure 7.5 to 7.7 shows models of a principle solution to `TransformEl- EnToMechEn`, which transforms an incoming electrical energy to a rotational mechanical energy. The BDD at the top left shows the structural relations of the SysML4FMArch elements: Therein, `TransformElEnToMechEn` contains an abstract «ElementaryEffect» (`EE_TEEME`)

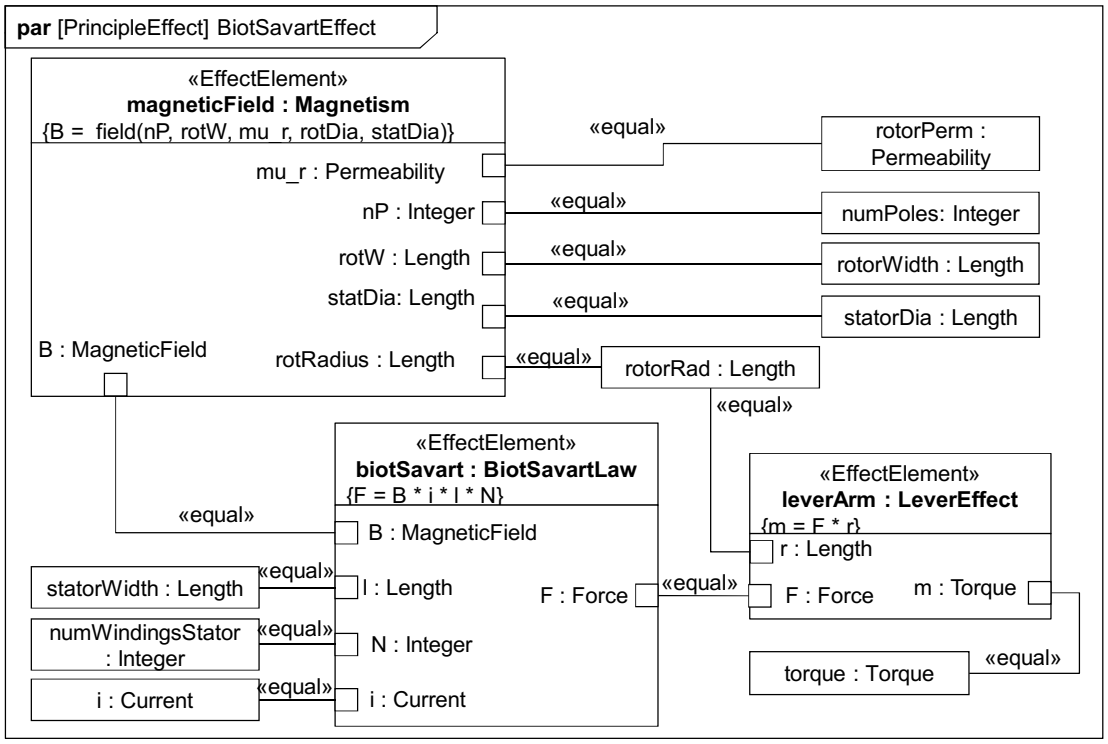


Figure 7.7: Parametric diagram showing the internals of the principle effect used by `SynchronousDrive`.

and an abstract «ElementaryGeometry» (`EG_TEEME`). The «PrincipleEffect» `BiotSavart` specializes the elementary effect and is therefore suitable to implement the function's behavior. The effect models the following physical phenomenon: An electromagnetic coil (stator) is positioned within a magnetic field B. The magnetic field is created by a permanent magnet (rotor), which 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 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 is the number of windings of the stator. If losses are not considered, electrical input power is equal to mechanical output power (see, *e.g.,* [HD19] for details). The parametric diagram in Figure 7.7 shows a SysML4FMArch-model of the Biot-Savart-Effect [HD19] which is

the interaction of the physical laws described by the Req. (1) to (3): 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 [JKPB12], 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 Req. (1). The «EffectElement» `BiotSavartLaw` models Req. (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 Req. (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`.

The principle solution `SynchronousDrive` modeled in Figure 7.5 (re)uses `BiotSavart`: By specializing `TransformElEnToMechEn` (*cf.* Figure 1.5), the «PrincipleSolution»`Syn- chronousDriving` inherits the interface of the «ElementaryFunction» and the interface assertion. Further, `SynchronousDrive` specifies the «PrincipleEffect» `BiotSavart` as its effect and the «PrincipleGeometry» `RotorStator` representing a geometry comprising a rotor and a stator. The geometric elements hold attributes with `fix`-dynamicity only (see Figure 7.2), meaning they do not change their values at system runtime. 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. Similar to the principle solution `HydrodynamicPump` in Figure 6.9, the IBD in Figure 7.6 shows how `SynchronousDrive` refines the interface assertion of `TransformElEnToMechEn`.

Solution to Generate a Volume Flow The models of the principle solutions introduced above can be used to model a solution to `GenerateVolumeFlow` whose internal structure is modeled in Figure 1.5. A solution to `GenerateVolumeFlow` is a «Solution»-block that specializes the «Architecture» `GenerateVolumeFlow`, which the BDD at the top of Figure 7.8 shows. The solution inherits all the functional subcomponents and the interface from the function. Since `HydrodynamicPump` and `SynchronousDriving` specialize `ApplyFluidWithMechEn` and `TransformElEn- ToMechEn`, respectively, they represent possible principle solutions for implementing these elementary functions. The latter provide types for the PartProperties `moveFluid` and `elToMech` of `GenerateVolumeFlow`, as shown in Figure 1.5. The «Solution» `ElectricalCoolantPump` inherits the interface and the PartProperties of `Generate- VolumeFlow`. By redefining the part properties to part properties of types `Transform- ElEnToMechEn` to `SynchronousDriving` and `ApplyFluidWithMechEnergy` to `Hydro- dynamicPump`, the solution integrates these «PrincipleSolutions» and forms a model of the solution to the entire architecture.

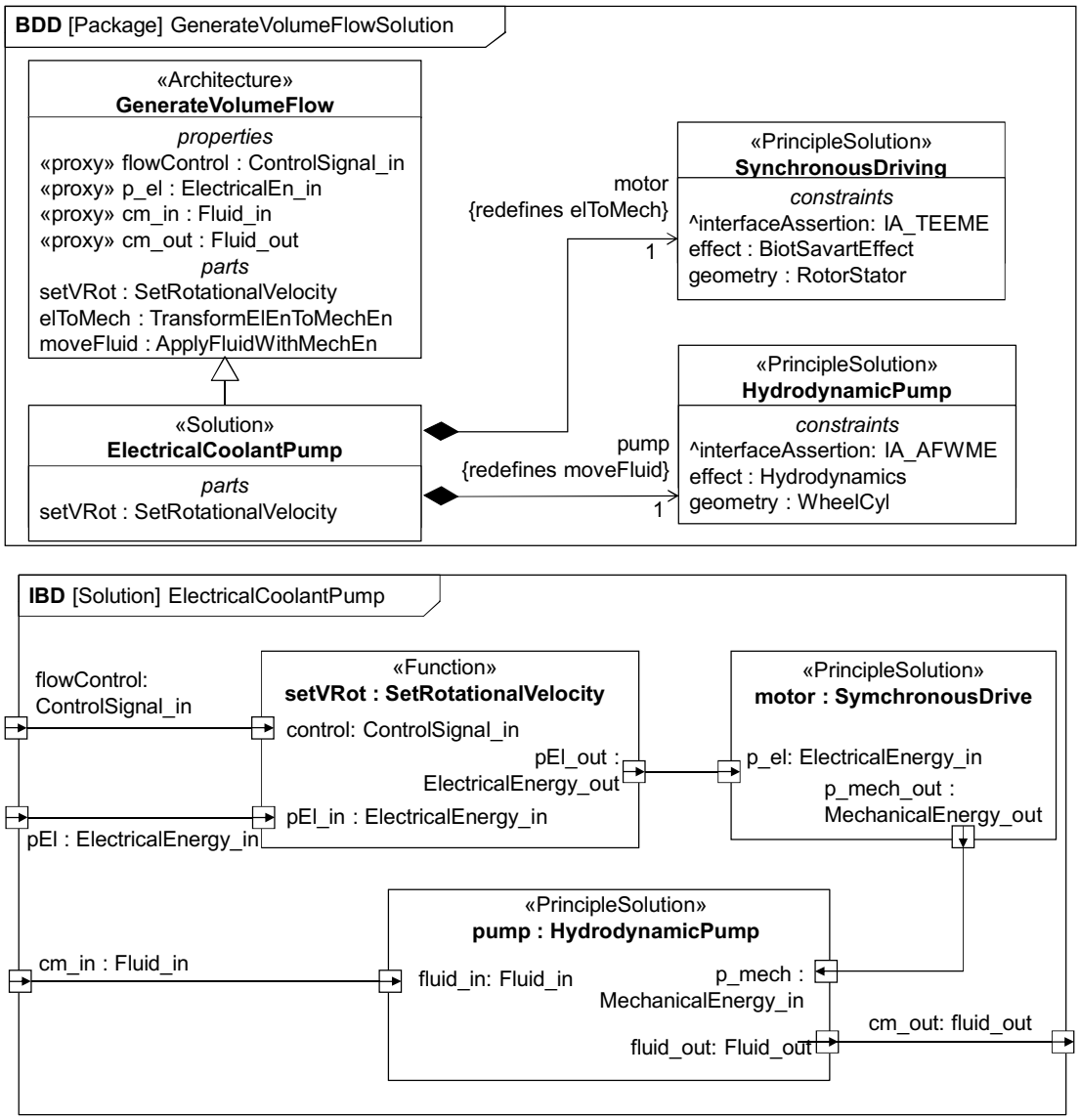


Figure 7.8: A solution to the composed architecture `GenerateVolumeFlow` (re-)uses principle solutions of its elementary functions.

The connectors between the ports are also inherited by the solution from the function. In this case, the solution models an electric coolant pump that consists of an electric engine and a paddle wheel. The function `setVRot` is implemented by a controller which we have not specified here, because the project from which these models

emerged was concerned with the engineering of the mechanical parts of this system. However, the function is specified by an interface assertion, and not providing a solution to this function emphasizes the principle of underspecification from the functional development paradigm.

7.3 Dimensioning and Testing an Automotive Electrical Coolant Pump

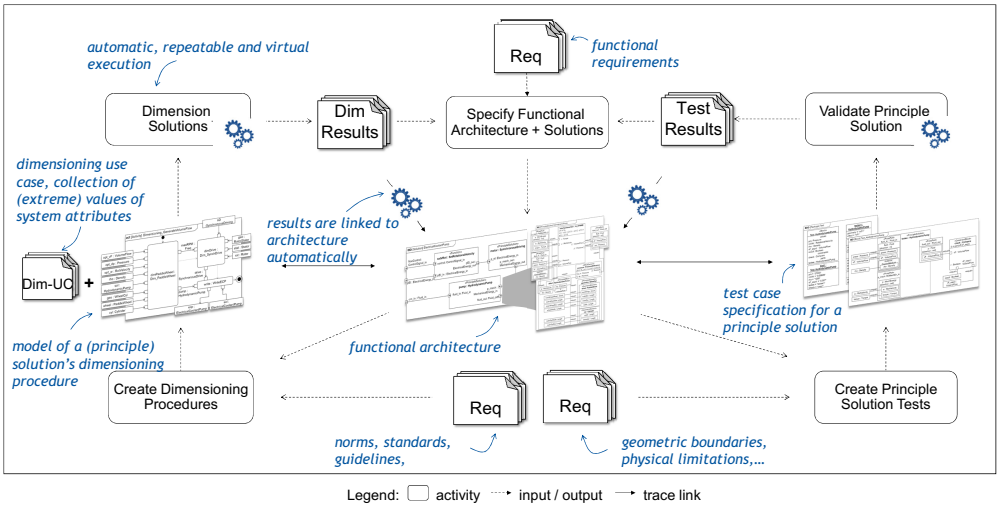


Figure 7.9: Integration of dimensioning and testing in our MDE approach. The models are in SysML4FMArch and detailed in Section 7.2 and 7.3

Modeling a system in terms of its functions and principle solutions enables bridging the gap between the physical product architecture and functional requirements [DRW+20]. In turn, this forms the basis for implementing automated support for developing solutions and holistically assuring the quality of the final product. When it comes to quality assurance and efficient solution development, dimensioning and testing are two important activities. Both are high-effort tasks that require the collaboration of experts from various fields of mechanical engineering. During these activities, simulations allow the approximation of the system's behavior and to make design decisions based on their predictions. Because this dissertation deals with integrating mechanical design into a functional model-driven approach to systems engineering, one research question is RQ 4 "How can these functional specifications facilitate dimensioning and testing to support the agile development of CPSs?". This section addresses this question and details a concept for dimensioning and testing (principle) solutions to support making design decisions based on objective criteria at any stage during the development, in

particular, early on when detailed models of the system's geometry, behavior, or even prototypes are unavailable. Figure 7.9 shows the conceptual activities, their inputs, outputs, and how these are linked. The basis for dimensioning and testing is a functional description of the system under development and the technical principles used to implement its functions. Specifying the functional architecture formalizes the functional requirements posed upon the system under development as Figure 7.9 shows. Therefore, engineers consult the functional architecture and link models describing dimensioning procedures or test cases for principle solutions to respective architectural elements. Integrating the results of the dimensioning and testing activities contributes to the iterative refinement of the functional architecture.
We show how the functional architecture of the coolant pump can be enhanced with models describing dimensioning and testing procedures that can be executed automatically by tools such as MagicDraw with the respective extensions which is outlined, *e.g.,* in [HJZ+21, HHB+22, ZRJ+22]. Doing so enables validating principle solutions and identifying misleading design decisions, *i.e.,* unsuitable (principle) solutions, early on, when exchanging them is still possible. Functions and therefore (principle) solutions interact via their interface only. This makes the architecture modular and varying (principle) solutions straightforward. This section illustrates how to model dimensioning procedures and tests for principle solutions in SysML4FMArch which is the technique applied for the approaches to automate these processes in [HJZ+21, HHB+22, ZRJ+22].

7.3.1 Dimensioning

Dimensioning is a task in mechanical design [BG21, GBP+21b] that is time-consuming and effortful. In general, the dimensioning process is concerned with finding optimal values for geometric and materialistic attributes of the system but also includes planning, controlling, and monitoring what leads to design decisions [BG21, GBP+21b]. So far, the process is mostly driven by the long-standing experience of involved experts who apply norms, standards, and design guidelines which standardize procedures to find optimal values for the geometric dimensions of the system under development. Through the physical interaction of a system's geometric components as well as that of the system with its environment, the optimality of attribute valuations, *i.e.,* values for sets of attributes, depends not only on other attributes of the component or system but also on those of other components or the environment. In particular, these dependencies arise due to the physical behavior of components interacting with other components or the environment [Isl04]. Finding optimal values requires algebraically rearranging the equations describing the physical behavior of a function. Since these equations are mostly differential, this can only be approximated. Mathematically speaking, dimensioning corresponds to solving a (possibly very hard to solve) optimization problem, including constraints that may even contradict each other.

Therefore, norms, standards, and design guidelines specify procedures that prescribe the dimensioning of the physical components, surfaces, and their material [uM04]. Experts implement these procedures in the form of simulation or calculation models. Monitoring and re-executing the procedures, simulations, and calculations lead to design decisions, and how these procedures evolve is therefore necessary.

Early development stages involve deciding upon the technical principles with which to implement system functionalities. This corresponds to evaluating the (principle) solutions within the functional architecture with respect to system requirements, limitations, and boundaries. Doing so, however, often requires a concretized concept of the product's shape [BG21, GBP+21b], *i.e.,* finding values for the attributes of the geometric elements in the (principle) solution. Therefore, supporting design decisions that regard choosing a principle solution based on objective criteria needs dimensioning procedures. Since principle solutions describe technical principles, there exist norms, standards, or guidelines that describe such procedures. For example, one part of a coolant pump in the cooling system of an automotive is a paddlewheel which rotates and thereby creates a pressure difference within the cooling medium. This pressure difference causes the cooling medium to flow, *i.e.,* a volume flow rate that is greater than zero. The dimensions of the paddlewheel determine among others how fast the cooling medium flows and therefore, the maximal volume flow rate of the cooling medium. The latter determines how much heat can transfer from the engine into the cooling medium. The paddlewheel has optimal dimensions, if, for an engine of a specified performance, it pumps enough cooling medium such that the engine remains at operating temperature regardless of the load the engine operates on. A possible dimensioning procedure is given in [Wes16].

For the dimensioning of a composed solution, such procedures are executed in an order which is currently implicit knowledge. The idea of Figure 7.9 is to complement (principle) solutions within the functional architecture with models of these procedures to make the knowledge about how to dimension (principle) solutions explicit and reusable. As Figure 7.9 shows on the left-hand side, norms, standards, and guidelines as well as system requirements, boundaries, and limitations together with the functional architecture form the input for defining a dimensioning procedure. In this activity, the engineer models prescribed dimensioning procedures for (principle) solutions to (elementary) functions in the functional architecture. A range of modeling techniques exists for this purpose, *e.g.,* ADs known in UML and SysML or BPMN. In these models, a task or activity corresponds to executing a simulation in a respective domain tool, such as *e.g.,* MATLAB[2] or AmeSim[3], that determines values of geometric or materialistic attributes of a (principle) solution. Flows that connect tasks or activities model the order and information exchange between these tasks, *i.e.,* the

[2]https://www.mathworks.com/products/matlab.html
[3]https://www.plm.automation.siemens.com/global/en/prod ucts/simcenter/simcenter-amesim.html

propagation of the results from a simulation or calculation to the next simulation. Libraries that digitize design catalogs [KK98] in the future should include such models.
To ensure the proper functioning of the system at all times, values for geometric attributes are typically determined such that the requirements are met for a set of possibly extreme use cases, *e.g.,* the system acting under maximal load. Such a use case fixes the values of a set of system attributes to provide a set of input values for a dimensioning procedure. The dimensioning procedure then determines values for the geometric attributes by executing (multiple) simulations or calculations. In the example of the paddlewheel, one such use case is a situation when the engine produces a maximal amount of heat, *e.g.,* when operating at maximal load. The maximal load yields a value for the engine's performance which is input to the dimensioning procedure. Section 7.3 provides details on this procedure.
Figure 7.9 illustrates how this integrates in a functional MDE approach: The "Dimension Solutions" activity takes a model of the dimensioning procedure and a dimensioning use case as input and calls the simulations and calculations in the specified order and propagates the results of each step along the information flow defined in the model of the procedure. Workflow engines such as the one integrated in CSM[4] allow to link tasks or activities in the workflow model with simulation models and enable to automate this activity entirely. Through respective trace links, the results of the dimensioning procedure, *i.e.,* the values for geometric attributes in the functional architecture are then accessible from the functional architecture and available for testing on which Section 7.3.3 provides further details.
The chosen dimensioning procedures are often specific to a principle solution and imply further requirements on the system. Complementing models of (principle) solutions with a description of their dimensioning procedures enables to include these restrictions in a reusable way which enhances efficiency when deciding upon technical principles to implement system functionalities. Executing such a procedure to obtain values for the attributes of the principle solution's geometry such that system restrictions or requirements are met optimally forms the foundation for upcoming geometric design activities. Automating the dimensioning of principle solutions, *e.g.,* by using appropriate tools to make their models executable, enables the identification of misleading design decisions continuously and early on, when changes are still possible.

7.3.2 Dimensioning Procedures in SysML4FMArch

Validating a (principle) solution with respect to system requirements requires executing a principle solution using simulation models. These models take values for the geometric attributes of the principle geometry as input that need to be determined by a dimensioning procedure. This reflects that validating a technical principle requires

[4] https://www.nomagic.com/products/cameo-systems-modeler

a concept of the product's shape [GBP+21b]. Dimensioning procedures include executing multiple simulations or calculations to obtain approximate (intermediate) values of physical quantities or other system attributes. These simulations or calculations do not emulate the physical behavior of the solution but represent an approximate rearrangement of the (differential) equations that describe this behavior. Thus, engineers define the simulations or calculations themselves as well as the order in which to execute them guided by norms, standards, or guidelines. The inputs and results of each step need to be propagated between the different tools that execute the simulation or calculation. The idea is, to complement models of principle solutions with models of these procedures, to enhance efficiency and reuse. Here, we have used SysML ADs because respective tools implement mechanisms to link activities with simulation or calculation procedures created with appropriate tools. This section outlines how we have modeled dimensioning procedures for (principle) solutions of the automotive coolant pump as SysML ADs.

Modeling Dimensioning Procedures for Principle Solutions A dimensioning procedure takes a set of attributes as input and outputs an instance of a principle solution, such that the instances of geometric elements contained in the instance of the principle solution have values calculated by the dimensioning procedure. The activities of the ADs model either a simulation or calculation to obtain values for (intermediate) attributes or writing them to an instance of the considered principle solution. The activity's parameters define the inputs and outputs of the simulation or calculation it performs. The object flow in the AD represents the propagation of attribute values to and from a simulation or calculation activity and the control flow models their execution order as prescribed in a standard, norm, or guideline. Figure 7.10 shows the dimensioning procedure for the paddlewheel described in [Wes16], which takes values of the following attributes as input:

- a value of the volume flow rate (`opt_vF`) at which the cooling medium has to flow to absorb enough heat,
- a value for the pressure difference (`opt_dp`), the pump has to generate for the cooling medium to flow at this rate,
- the necessary rotational velocity (`opt_w`) of the paddle wheel to generate this pressure difference, and
- the density (`rho`) of the cooling medium.

The parameters of the activity `CalculatePaddleWheel` which performs the calculation described in [Wes16], represent these inputs. The input values for `opt_vF`, `opt_dp` and `opt_w` define a dimensioning use case. The density of the cooling

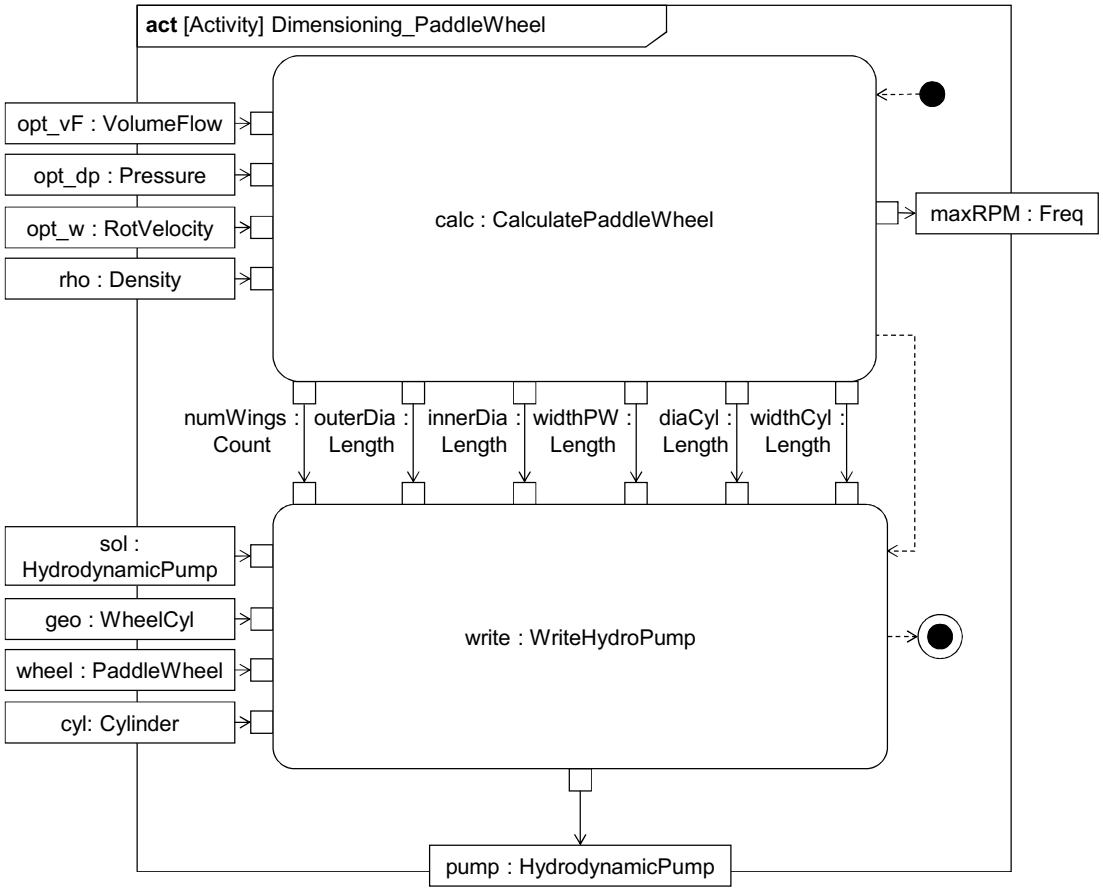


Figure 7.10: Activity Diagram modeling the dimensioning procedure for the principle solution `HydrodynamicPump` to AFWME.

medium depends on the type of material that enters the pump. From these, the procedure calculates the necessary dimensions of the paddle wheel, i.e., the necessary number of wings (`numWings`), the paddle wheel's outer and inner diameter (`outerDia`, `innerDia`) and its width (`widthPW`) as well as the diameter of the cylinder, the paddle wheel is mounted in (`diaCyl`) and its width (`widthCyl`). The procedure also calculates the value of `maxRPM` which characterizes the performance of an electric coolant pump with this paddle wheel which is needed for dimensioning the electrical coolant pump described below and therefore an output parameter of this activity. The activity `WriteHydroPump` assigns these values to the attributes of a `PaddleWheel`- and a `Cylinder`-instance and sets them the geometric elements of the `WheelCyl`-instance which is the instance of the `geometry`-attribute of the output

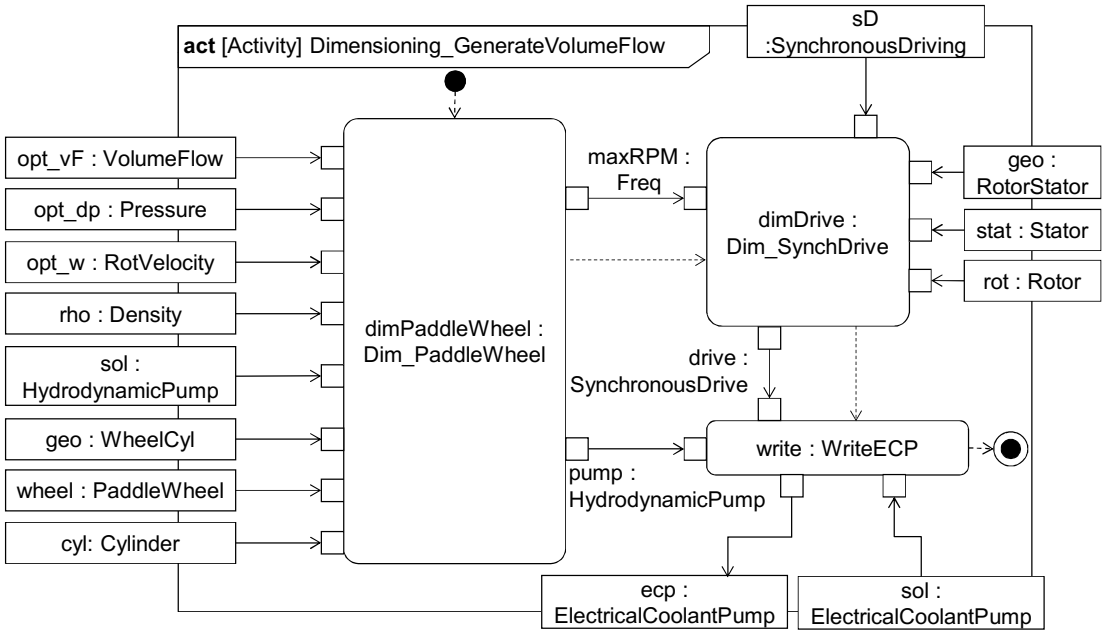


Figure 7.11: Activity Diagram modeling the dimensioning procedure for the solution `GenerateVolumeFlow`.

`HydodynamicPump`-instance. Figure 1.5 shows the definitions of these elements. The instance of the principle solution `HydrodynamicPump` contains the results of the dimensioning.

Modeling Dimensioning Procedures for Composed Solutions A dimensioning procedure for composed solutions can reuse other dimensioning procedures. Figure 7.11 shows the dimensioning procedure for `ElectricalCoolantPump` (*cf.* Figure 7.8) which reuses the dimensioning procedure for the paddle wheel described previously and one for `Syn- chronousDrive` (*cf.* Figure 7.5). The procedure to do the latter takes the maximal frequency at which the paddle wheel may turn as input and calculates values to the geometric elements of the principle geometry of `SynchronousDriving`. The activity `WriteECP` creates an instance of the composed solution `ElectricalCoolantPump` that contains instances of principle solutions for the two elementary functions with dimensioned principle geometries. The built-in execution engine of CSM makes the ADs described above executable. CSM can integrate domain tools, such as MATLAB, and activities can link to models created using these tools. The execution engine triggers the execution of these models when linked to an activity with the values of the activity's parameters. Additionally, the execution engine

provides the functionality to retrieve the results of the computations to propagate them for further processing. In our case study, we utilized activities to write the calculated values directly to CAD models of the principle geometries.

7.3.3 Testing Principle Solutions in SysML4FMArch

The functional architecture decomposes a complex system functionality down to atomic physical functions. Engineers refine the latter to principle solutions such that their composition yields a solution to the composed function. The functional architecture is thereby modular in the sense that each function can be developed and *tested* individually. This reflects the idea of a component in the C&C paradigm, which encapsulates a functionality as a self-contained building block [Kus21]. Validating a principle solution in this way regarding the context of a specific system, requires to find values for the geometric attributes that are part of its principle geometry. The dimensioning procedures sketched in the previous section serve this purpose. Here, we outline a concept for testing principle solutions in a functional architecture that meets the demands of [Rum17]. Figure 7.9 shows that the testing activity includes specifying test cases, provided as models, for the principle solutions included in a functional architecture. Given appropriate tools, these models can be executed automatically to enhance agility and efficiency during the development process.
We consider a test to comprise a set of test data for the inputs, a (model of) an implementation of the system and its environment as a functional specification. The latter may be provided in the form of an oracle which is a component which decides, given the input test data together with the output produced by the (model of) the implementation, whether the specification is met or not. In our setting, the input data and the expected results are histories [Bro18] of values for the input and output channels of the respective principle solution under test. System requirements provide these values. A (model of) an implementation is an executable instance of a principle solution, in which the values for the principle geometry have been determined previously by a dimensioning procedure. In the context of a system's functional architecture, simulations serve as a means to emulate a physical effect, *i.e.,* a behavior of a physical function. To validate a principle solution (*cf.* Figure 7.9), a test driver hands the input values to the simulation model emulating the physical effect of a principle solution under test executes the simulation, and retrieves the simulation output. Comparing the outputs of the testee to the expected outputs specified in the test case yields the test result. Existing tools enable engineers to specify simulations for testing principle solutions. The automatic execution of these test cases is crucial for agile MDE [Rum17]. The test results provide an objective basis to decide whether a principle solution is applicable in the context of a specific system and therefore lead to refinements of the functional architecture. The modularity of the functional architecture allows for the exchange of principle solutions once they have been

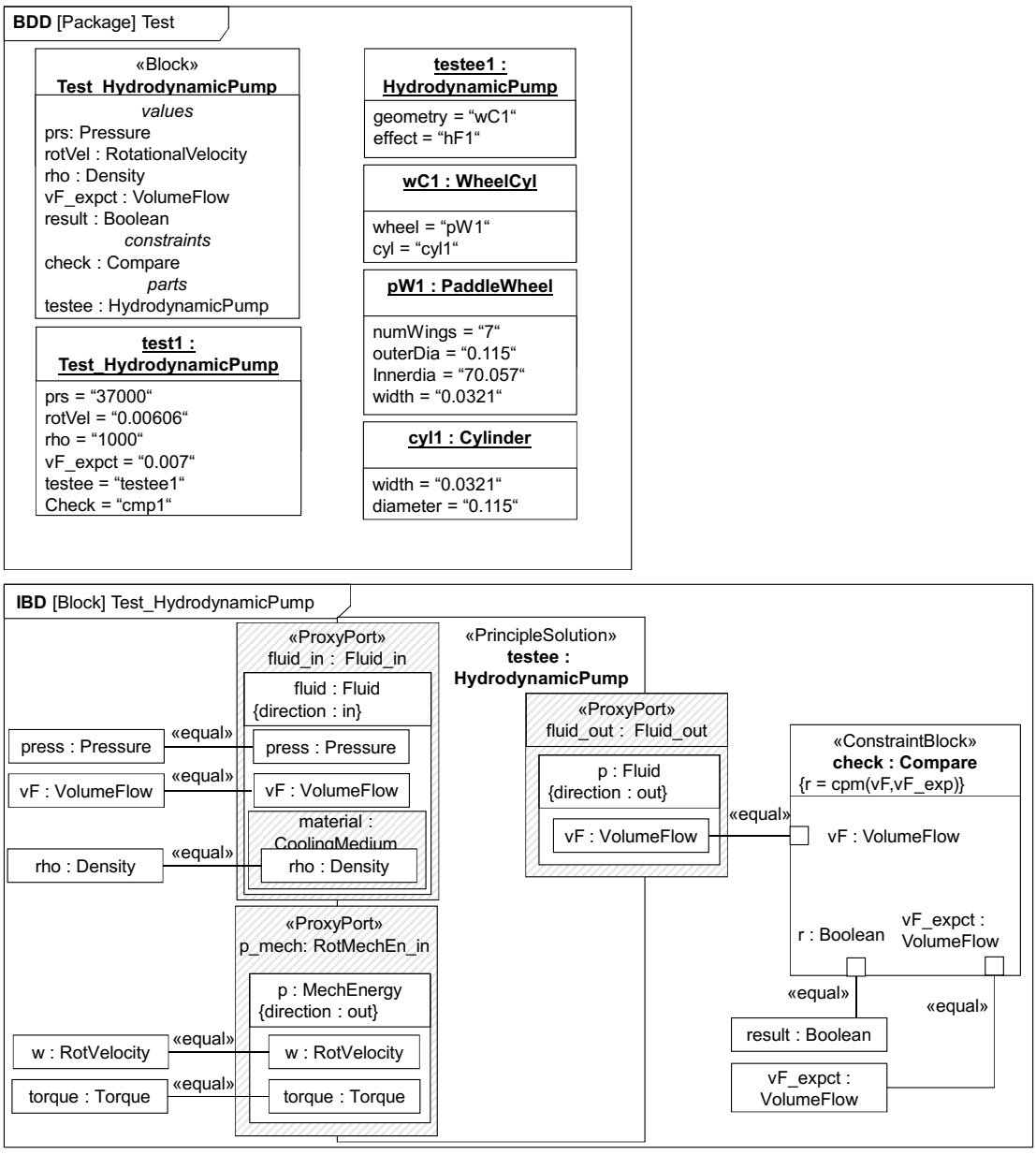


Figure 7.12: Test-specification for `HydrodynamicPump` (*cf.* Figure 6.9). A test case is an instance of `Test_HydrodynamicPump` which CSM can execute automatically.

identified as unsuited by a test and all tests can be re-executed automatically for another principle solution which allows their systematic variation.

7.3.4 Modeling Tests in SysML4FMArch

The functional architecture specifies the system as a network of functions that interact by exchanging flows of energy, material, or data via their interfaces only. The constituents of a principle solution hold attributes for which paragraph 7.3.2 has outlined how to model and execute dimensioning procedures that find values for the geometric attributes of principle solutions. The next step after executing a dimensioning procedure as outlined in paragraph 7.3.2is to validate whether the dimensioned principle solution instance describes a technical principle that is suited to meet system requirements. This section details how we have applied this concept to test principle solutions in the electric coolant pump modeled in SysML4FMArch.
A test is modeled as a block that specifies input data and expected outcomes through ValueProperties with respective types for the attributes of the FlowProperties of the interface blocks typing a port of the principle solution. The expected output does not need to be specified for all ValueProperties of outgoing ports. A test case executes a principle solution under test. To this effect, the test block holds a PartProperty typed by the principle solution. The principle effect in this testee links to a simulation model that emulates the physical behavior that the test driver, in our case provided by CSM, can execute. A ConstraintProperty specifies comparing the output of the testee to the values of the expected output ValueProperties. A test case is an instance of such a block, whose ValueProperties hold numeric values. To illustrate this, consider the BDD at the top of Figure 7.12. The block `Test_HydrodynamicPump` specifies test cases for validating that the fluid leaving the principle solution `HydrodynamicPump` flows at an appropriate volume flow rate. The attributes `prs`, `rotVel`, and `rho` specify the input data for these test cases. The expected volume flow rate, *i.e.,* the attribute `vF_expct` specifies the expected outcome. The attribute `result` is a Boolean representing the success or failure of a test case and will be set according to the constraint property `check`. The testee, in this case, is an instance of the principle solution `HydrodynamicPump`. The IBD at the bottom of Figure 7.12 shows the internal structure of the tests. As detailed in Section 6.2.3, the effect element `CentrifugalForce` which models the behavior of the principle solution `HydrodynamicPump` links to an executable simulation model, here, created in MATLAB. The constraint block `Compare` links to another MATLAB model that sets the value of `result` to `true` iff the result, *i.e.,* volume flow `vF` of the outgoing fluid is greater than the expected value `vF_expct`. A concrete test case is given by the instance `test1`: In the project, from which SysML4FMArch emerged, a pump wheel turning with a rotational velocity of $0.00606\mathrm{m}^3\,\mathrm{s}^{-1}$ was required to accelerate an incoming fluid of density $1000\mathrm{kg}\,\mathrm{m}^{-3}$ with pressure 37kPa to flow with a rotational

velocity of at least 0.007 $m^3 s^{-1}$. The instance `testee1` of `Test_HydrodynamicPump` represents this test case. The attribute testee is instantiated with the output of the dimensioning procedure modeled in Figure 7.11, which CSM can execute. After execution, the value `result` of `testee1` tells, if the test succeeded or failed.

7.3.5 Discussion

The systematic relation between functions and solutions in the SysML4FMArch-models enabled the utilization of a functional architecture model for virtual dimensioning and testing, which, in practice, is mostly driven by experience. For software engineers, the process of dimensioning is counter-intuitive because software systems simply do not have geometric dimensions and compilers have long automated the process of building a software system from code including many optimizations. Considering the broad literature on testing in SE, functional testing is well-integrated and understood in this domain. The presented modeling techniques and concepts for dimensioning and testing aim to apply these ideas from SE to capture dimensioning and testing from a holistic systems engineering point of view that enables systematic automation of these processes. To evaluate the concept of dimensioning and testing of principle solutions we have applied them for dimensioning the paddlewheel and evaluated whether the obtained principle solution meets system requirements. The respective models have been presented in this section. We have modeled the automotive cooling system in the MagicDraw implementation of SysML4FMArch presented in Section 6.3 and enhanced the set of models by the models of the dimensioning and testing procedures presented in Section 7.3 The MagicDraw derivative Cameo Systems Modeler enabled the execution of the latter automatically. In the project, automating the dimensioning and testing procedures allowed for systematic variation of principle solutions and to make design decisions based on objective criteria rather than experience and implicit knowledge at early development stages.
Up to now, engineers translate requirements directly and heuristically into fully designed components which is a prerequisite for validation [PJHB19, ABJ19, GJW+18, BJKS16, WJD18]. Creating these models is complex and time-consuming and therefore validating principle solutions is postponed to later development stages, when changes are expensive. Recent approaches that aim to formalize a notion of principle solutions, either do not integrate testing or dimensioning or do not describe the principle solutions in a way that enables automatic dimensioning. The approaches presented in Section 7.3 enabled us to validate a technical principle described by a principle solution based on a geometry that solely comprises active surfaces. These are less detailed than the elaborate geometric models used for validation in practice, but allow evaluating principle solution as implementations of elementary functions at early development stages. Automated

validation procedures can support making design decisions based on objective criteria. Further, the approach enables the execution of tests much more frequently as they are automated which supports agile development. The described approaches for virtual dimensioning and testing, therefore, make product development in mechanical engineering much more efficient and enable the validation of requirements virtually and early in the development process.

7.3.6 Related Work

Validation and Verification (V&V) Formal methods are on their way to becoming a standard in the V&V of software-intensive (sub-)systems in CPSs. They employ mathematical theories to obtain proof that the system fulfills a certain property and thereby make extensive testing dispensible [KPRR20b]. The survey presented in [WLBF09] analyzes the utilization of formal methods employed in industry for rigorous verification and validation of systems. Formal system specification techniques such as FOCUS [BS01, Bro10] as well as [Alu15] or [Pto14] provide the formal basis to develop and implement such methods. Verification tools utilize theorem proving based on such theories to obtain formal proofs that a specified system fulfills a certain property [KPRR20b, KPRR20a, SHT12].

Model-Based Testing MBT [Rum17] is a technique that utilizes models describing the behavior of a system under development to generate executable test cases [DJK+99] and thereby strongly appeals to the model-driven paradigm. Since test cases in MBT are executed automatically, agile development becomes possible. There exist various techniques that extract test cases from system behavior models, where most of them rely on a specific modeling language: The techniques proposed in [WYY+04, KKBK07] utilize ADs to generate test cases based on extracting paths. This enables assuring specific test coverages. Procedures that generate test cases from UML or SysML SCs are presented in [SSK11, OA99, OLAA03]. The methods rely on a transformation of SCs to extended finite state machines proposed in [KHBC99] or [SSK11]. Similarly, the techniques proposed in [MAD11] and [SMM10] transform a combination of ADs with Sequence Diagrams [Rum16] or SCs, respectively into a formalism that allows extracting test cases based on formal rules. As such, these methods employ a mathematically well-understood formalism to interpret the model to generate test cases, which is also often employed by formal methods. Based on a systematic survey [KMS+18], the methodology presented in [DGH+18, DGH+19] formalizes functional system requirements in the form of UML/P [Rum16] SCs and ADs to improve testing in the automotive domain by generating test cases from these models automatically. All of these techniques stem from the SE domain and focus on testing software or software- intensive systems and require a model of the system's behavior. In our setting, such techniques could be applied, while a solution

architecture with linked simulation methods enables to *execute* the system. In our approach, workflow models are employed to specify the execution logic of a dimensioning process, *i.e.,* the order in which to execute simulations or calculations to obtain values for geometric system attributes.

Executable Workflow Models Process and workflow modeling is a very broad research field in SE and particularly MDE. Standards specifying such modeling languages are, *e.g.,* BPMN [OMG13, vdAtHW03] or ADs which are part of the UML [Man15] and the SysML [Man19]. When it comes to business process modeling, BPMN and UML ADs often compete [RvdAtHW06, BO10], however, BPMN has become the de-facto standard in this area [CT12]. A comparison of the two with respect to business process modeling is given in [RvdAtHW06]. Graphical and textual implementations of (variants) of both these languages exist, *e.g.,* [UTF10] proposes a BPMN language and [BR02] an extension of UML ADs for modeling workflows in production systems engineering. Systems engineers most often use SysML ADs for modeling system behavior [Man19] and research has proposed frameworks and formalisms to interpret these models, *e.g.,* [OAMD12, JDB09]. An overview of the approaches and techniques for simulating SysML ADs is given in [NKT+15]. In our case study, we have used CSM [Cas17] which includes an engine for simulating SysML ADs. In contrast to the approaches that utilize SysML ADs for modeling system behavior, we utilize them to model dimensioning workflows, which, in particular, allows for formally capturing guidelines, norms, and standards on such procedures and thereby enhance the reuse of principle solutions. As SysML4FMArch is based on SysML, using SysML ADs, among others, to link dimensioning workflows to principle solution models easily and also to reuse existing tooling that already came with a built-in execution engine for these models.

Chapter 8

Conclusion and Future Work

Innovations of our modern systems are driven by functionalities and features. Software, mechanical or electrical (sub-)systems alone cannot realize these functions adequately. Therefore, CPSs have emerged as complex systems-of-systems that are capable to meet the expectations of the various stakeholders through the interaction of these diverse systems. With this trend, the complexity of engineering innovative modern systems has increased significantly. Very often, in the systems engineering practice, experts from software, mechanical, and electrical engineering work in separate teams. These teams are often arranged following the geometric decomposition of the system under development. The engineers receive a set of textual requirements, make up an idea of the realization in their minds, possibly throughout many meetings and start to create models of this conceived implementation using established tools such as CAD, simulation models, code, circuit diagrams, *etc.*. This gives rise to a problem-implementation gap between the textual requirements and the models of the implementation: It is unclear how and which parts of the implementation realize which of the functional requirements.

In this dissertation we have defined a functional development paradigm which puts CPFs, which drive innovations of modern system, into focus. To make this paradigm applicable, we have conceived a formal modeling technique based on the theory of TSPFs over continuous time domains together with a methodology that allows specifying the CPS's functions. Unlike any other theory, TSPFs are compositional with respect to refinement, *i.e.,* refining one component in a network of TSPFs yields a refined version of the composed function. Since development teams are distributed across different sites, and since reuse of existing, known and tested solutions is of crucial importance, each component can therefore be developed or used on its own by connecting the interface. Further, the theory allows to apply underspecification in a controlled manner to deal with lack of information, uncertainty, *etc.*. in the models. We have showed that the modeling technique can be applied to formalize the informal specifications from Koller's design catalog [KK98], and presented an example from an industrial project that involved experts from industry and academia that uses these specifications. Thereby, we have showed that functional models of the system under development created with our methodology are understood and narrow the gap

between requirements and models of the implementation for functions in the mechanical engineering domain. Because we use a theory from software engineering, the example shows that the models are understood equally in the domains of mechanical and software engineering.
With the audio entertainment system example we have showed that the modeling technique is suitable to create functional models that abstract from the implementation domain. The audio entertainment system can be implemented mechanically, *e.g.,* as a phonograph, or digitally with on-demand streaming services. The CPF specification of a CPS, therefore, provides an abstraction in between the requirements and models of the system's implementation.
Defining the systems engineering methodology is impossible because systems, companies, projects, *etc.* are as diverse as the subsystems of the system itself. Therefore, we abstain from predefining abstraction layers and modeling languages, but provide a formal methodology in the sense of [Rum96]. Chapter 6 and the evaluation provide concrete examples of modeling languages and applications to signpost the idea. Methodologies that do define layers, such as, *e.g.,* SMARDT [DRW+20], may utilize the proposed methodology to define the relations among the models of each layer formally to enable automated consistency checks among the layers.
The idea presented in this dissertation is still in its infancy. Thinking in terms of functions instead of geometric components as an intermediate step between requirements and the implementation requires extensive training of the experts. Certainly, the presented perspectives on methodology and modeling languages need further evaluation in practice. In the future, the contributions can be taken as a basis to enhance the application of functional verification, *e.g.,* as proposed in [KPR+22, KMP+21, KPRR21] in the verification of functional requirements. Deriving test cases automatically from specifications, *e.g.,* combined with the methods proposed in [DRW+20], is certainly an interesting point to enhance the efficiency of testing in the context of validation.
The functional development paradigm together with the presented modeling technique provide the initial basis to establish functional systems thinking in the systems engineering community together with ways to utilize this way of thinking to overcome the complexity of engineering CPSs.

Bibliography

[ABH+21] Moussa Amrani, Dominique Blouin, Robert Heinrich, Arend Rensink, Hans Vangheluwe, and Andreas Wortmann. Multi-paradigm modelling for cyber–physical systems: a descriptive framework. *Software and Systems Modeling*, 20(3), 2021. 4.3

[ABJ19] Faysal Andary, Joerg Berroth, and Georg Jacobs. An energy-based load distribution approach for the application of gear mesh stiffness on elastic bodies. *Journal of Mechanical Design*, 141(9), 2019. 7.3.5

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1), 1995. 3.4.2, 3.5, 3.7, E.3

[Alu15] Rajeev Alur. *Principles of cyber-physical systems.* 2015. 1.1.2, 2.2.2, 3.4.2, 3.4.2, 5, 5.3.3, 6.5, 7.3.6

[Arf85] George Arfken. *Mathematical Methods for Physicists.* Academic Press, 1985. 2.2.1, 2.2.1

[AVT+15a] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. In *Joint proceedings of ACES-MB 2015–Model-based Architecting of Cyber-physical and Embedded Systems*, 2015. 1.1.2, 6.1.2

[AVT+15b] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. In *ACES-MB&WUCOR@MoDELS 2015*, CEUR Workshop Proceedings, pages 19–26. CEUR-WS.org, 2015. 6.5

[BBD+21] Birthe Böhm, Wolfgang Böhm, Marian Daun, Alexander Hayward, Sieglinde Kranz, Nikolaus Regnat, Sebastian Schröck, Ingo Stierand, Andreas Vogelsang, Jan Vollmar, Sebastian Voss, Thorsten Weyer, and Andreas Wortmann. Engineering of Collaborative Embedded Systems.

In *Model-Based Engineering of Collaborative Embedded Systems*, pages 15–48. Springer, January 2021. 4.3, 4.3, 6.5

[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. 5.3.3

[BDD+93] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - an Introduction to FOCUS, 1993. 3.1.1, 3.3.3, 3.4.3, C.1, C.1, C.2, C.3, C.4, C.5, C.1

[BG21] Beate Bender and Kilian Gericke. *Pahl/Beitz Konstruktionslehre*. Springer Vieweg, Berlin, Heidelberg, 2021. 1.1.2, 1.1.2, 1.3.5, 3.2.5, 4.1.1, 4.3, 5, 5.1, 5.1, 5.1.1, 5.1.1, 5.1.2, 5.1.2, 5.1.2, 5.1.2, 5.1.2, 5.1.4, 5.6, 5.26, 5.3, 5.3.3, 6.1.1, 6.1.2, 6.1.3, 6.2.1, 6.2.2, 6.5, 7.1.1, 7.3.1, E.3, E.3, E.3

[BGK+09] Manfred Broy, Mario Gleirscher, Peter Kluge, Wolfgang Krenzer, Stefano Merenda, and Doris Wild. Automotive architecture framework: Towards a holistic and standardised system architecture description. Technical report, Technische Universität München, 2009. 1.1.1

[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. 6.5

[BJKS16] Joerg Berroth, Georg Jacobs, Tobias Kroll, and Ralf Schelenz. Investigation on pitch system loads by means of an integral multi body simulation approach. *Journal of Physics: Conference Series*, 753, 2016. 7.3.5

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*. Springer Berlin Heidelberg, 1996. 6.5

[BM97] John Bell and Machover Moshe. *A Course in Mathematical Logic*. Elsevier Science and Technology, 1997. 1.3.1, 4.2.2

[BO10] Dominik Birkmeier and Sven Overhage. Is bpmn really first choice in joint architecture development? an empirical study on the usability of

bpmn and uml activity diagrams for business users. In George T. Heineman, Jan Kofron, and Frantisek Plasil, editors, *Research into Practice – Reality and Gaps*, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 7.3.6

[BR02] R. M. Bastos and D. D. A. Ruiz. Extending UML activity diagram for workflow modeling in production systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, 2002. 7.3.6

[Bro97] Manfred Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44(6), 1997. 3.1.1

[Bro01] Manfred Broy. Refinement of time. *Theoretical Computer Science*, 253(1):3 – 26, 2001. 3.1, 3.1, E.3

[Bro07] Manfred Broy. Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *Innovations Syst Softw Eng*, 3, 2007. 6.1.2

[Bro10] Manfred Broy. A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 53(10), 2010. 1.1.2, 1.1.2, 1.2, 1.3.1, 1.3.2, 1.5, 2.2.4, 3, 3.2, 3.4, 3.4.3, 5.7, 7.3.6, E.3

[Bro12] Manfred Broy. *System Behaviour Models with Discrete and Dense Time*, pages 3–25. Springer Berlin Heidelberg, 2012. 1.1.2, 1.2, 1.3.1, 1.5, 2.7, 3, 3.1, 3.1.1, 3.1, 3.1.1, 3.2, 3.10, 3.3.1, 3.11, 3.12, 3.3.4, 3.17, 3.2, 3.3.4, 3.3.4, 3.18, 3.4.1, 3.4.1, 3.3, 3.4.2, 3.4.2, 3.4.2, 4.2, 5.1.1, 5.3.3, 7.2.2, C, E.3

[Bro13] Manfred Broy. Engineering cyber-physical systems: Challenges and foundations. In Marc Aiguier, Yves Caseau, Daniel Krob, and Antoine Rauzy, editors, *Complex Systems Design & Management*, pages 1–13. Springer Berlin Heidelberg, 2013. 1.1.2

[Bro18] Manfred Broy. On Architecture Specification. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, pages 19–39. Springer International Publishing, 2018. 5.3.3, 6.2.2, 7.2.2, 7.3.3

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer-Verlag, 2001. 1.1.2, 1.3.4, 2.1, 2.2.4, 3.4, 3.4, 4.3, 5, 5.1.1, 5.3.3, 6.1.2, 6.5, 7.3.6

[BS08] John T. Boardman and Brian J. Sauser. Systems thinking: Coping with 21st century problems. 2008. 1.1

[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. 5.3.3

[BvLK15] 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. 6.5

[Cam14] F. Campagne. *The MPS Language Workbench: Volume I.* The MPS Language Workbench. 2014. 3.4.2, 3.4.2

[Cas17] Olivier Casse. *SysML in Action with Cameo Systems Modeler.* Elsevier, 2017. 7.3.6

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools.* Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016. 1.1.1

[CKY05] Chun-Hsien Chen, Li Pheng Khoo, and Wei Yan. Pdcs—a product definition and customisation system for product concept development. *Expert Systems with Applications*, 28(3), 2005. 1.1

[CT12] Michele Chinosi and Alberto Trombetta. BPMN: An introduction to the standard. *Comput. Stand. Interfaces*, 34(1):124–134, 2012. 7.3.6

[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. 4.3, 4.3, 7.3.6

[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. 1.1, 1.1.1, 1.1.2, 6.5, 7.3.6

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology*, 20:1–24, November 2021. 3.3.4

[DJK+99] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294. ACM, 1999. 7.3.6

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019. 1.3.2, 3.3.4, 3.3.4, 4.2

[DR08] Wolfgang Dahmen and Arnold Reusken. *Splinefunktionen*. Springer Berlin Heidelberg, 2008. 5.1.2

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020. 7, 1.1.1, 1.1.2, 1.3, 1.3.3, 1.3.5, 1.5, 1.4, 1.5, 4.3, 5, 5.1, 6, 6.1, 6.1.2, 6.1.3, 6.2, 6.3, 6.4, 6.2.1, 6.5, 6.6, 6.2.3, 7.2, 7.2, 7.3, 7.3, 8, E.3

[EE2] Electrical Engineering. `https://en.wikipedia.org/wiki/Electrical_engineering`. Accessed: 2023-06-27. E.3

[EGZ12] Martin Eigner, Torsten Gilz, and Radoslav Zafirov. Proposal for functional product description as part of a PLM solution in interdisciplinary product development. In *Proceedings of the DESIGN 2012, the 12th International Design Conference*, 2012. 6.5

[Esc20] Jost-Hinrich Eschenburg. *Was ist Geometrie?*, pages 1–5. Springer Fachmedien Wiesbaden, Wiesbaden, 2020. 1.1.1, 5.2.1

[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. 6.5

[Fis01] Gerd Fischer. *Analytische Geometrie*. Vieweg + Teubner Verlag Wiesbaden, 2001. B

[Fis13] Gerd Fischer. *Lehrbuch der Algebra*. Springer Spektrum Wiesbaden, 2013. 5.1.2, 5.2.1, B.3, B.4

[FLD04] H.-J Franke, S Löffler, and M 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. 5.3.3

[Foe10] Otto Foerster. *Analysis II*. Vieweg+Teubner, 2010. B.5

[Foe16] Otto Foerster. *Analysis I*. Springer Spektrum, Wiesbaden, 2016. 1.3.1

[Fow10] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010. 6.3.1

[FR76] Gottfried Falk and Wolfgang Ruppel. *Energieformen*. Springer Berlin Heidelberg, 1976. 2.2.1, 2.2.2, 2.2.2, 2.2.2, 2.2.2, 2.2.2, 2.2.2, 2.1, 2.2.3, 2.3, 3.2.1, 3.2.3, 5.1.4, 5.2.1, 5.2.1, 5.2, 5.33, 5.3.1, 5.34, 5.3.2, 5.38, 7.1.1, 7.1.1, E.3

[FR83] Gottfried Falk and Wolfgang Ruppel. *Mechanik, Relativität, Gravitation - die Physik des Naturwissenschaftlers*. Springer Berlin Heidelberg, 1983. 2.2.1

[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. 1.1.1, 1.1.1, 3.3.3, 4, 6.1.2

[FRR09] Florian Fieber, Nikolaus Regnat, and Bernhard Rumpe. Assessing usability of model driven development in industrial projects. In T. Bailey, R. Vogel, and J. Mansell, editors, *4th European Workshop on "From code centric to model centric software engineering: Practices, Implications and ROI" (C2M)*, University of Twente, NL-Enschede, June 24 2009. CTIT Workshop Proceedings, Enschede. 6.5

[GA09] Karl-Heinrich Grote and Erik K. Antonsson. *Springer Handbook of Mechanical Engineering*. Springer, Berlin, 2009. 5.1.2

[GB07] P. J. Gawthrop and G. P. Bevan. Bond-graph modeling. *IEEE Control Systems Magazine*, 27(2):24–45, 2007. 2.2.2, 2.2.2, 6.2.2, 7.2.1, 7.2, 7.2.2, E.3

[GBG18] Karl-Heinrich Grote, Beate Bender, and Dietmar Göhlich, editors. *Dubble: Taschenbuch für den Maschinenbau*. Springer Vieweg Berlin, Heidelberg, 2018. 3.4.3, 5.3.1

[GBP+21a] Kilian Gericke, Beate Bender, Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Der Produktentwicklungsprozess*, pages 57–93. 2021. 4

[GBP+21b] Kilian Gericke, Beate Bender, Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Funktionen und deren Strukturen*. Springer Berlin Heidelberg, 2021. 7.3.1, 7.3.2

[GDP+10] Jürgen Gausemeier, Rafal Dorociak, Sebastian Pook, Alexander Nyßen, and Axel Terfloth. Computer-aided cross-domain modeling of mechatronic systems. In *Proceedings of the DESIGN 2010, the 11th International Design Conference*, 05 2010. 6.5

[GHJV95] Erich. Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995. 5.1.4, 6.1.2

[GJRR22a] Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. Design Guidelines for Improving User Experience in Industrial Domain-Specific Modelling Languages. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Association for Computing Machinery, October 2022. 6.5

[GJRR22b] Rohit Gupta, Nico Jansen, Nikolaus Regnat, and Bernhard Rumpe. Implementation of the SpesML Workbench in MagicDraw. In *Modellierung 2022 Satellite Events*, pages 61–76. Gesellschaft für Informatik, June 2022. 4.3, 6.3.1, 6.3.2, 6.5

[GJW+18] Reza Golafshan, Georg Jacobs, Matthias Wegerhoff, Pascal Drichel, and Joerg Berroth. Investigation on the Effects of Structural Dynamics on Rolling Bearing Fault Diagnosis by Means of Multibody Simulation. *International Journal of Rotating Machinery*, 2018:1–18, 2018. 7.3.5

[GKR+21] Rohit Gupta, Sieglinde Kranz, Nikolaus Regnat, Bernhard Rumpe, and Andreas Wortmann. Towards a Systematic Engineering of Industrial Domain-Specific Languages. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SE&IP)*, pages 49–56. IEEE, May 2021. 6.3.1, 6.10, 6.5, E.3

[Gro00] IEEE Architecture Working Group. IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000. 5.3.3

[GRSS11] H. Giese, B. Rumpe, B. Schätz, and J. Sztipanovits. Science and engineering of cyber-physical systems (dagstuhl seminar 11441). *Dagstuhl Reports*, 1(11), 2011. 1.1, 1.1.1

[GS96] Peter Gawthrop and Lorcan Smith. *Metamodelling: For Bond Graphs and Dynamic Systems*. 1996. 2.2.2

[Gü10] Johann Friedrich Gülich. *Centrifugal Pumps*. Springer, Berlin, Heidelberg, 2010. 1.3.3

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016. 1.4, 1.3.4, 6.1.2, E.3

[HD19] Austin Hughes and Bill Drury. *Electric motors and drives: fundamentals, types and applications*. Newnes, 2019. 7.2.3

[HHB+22] Christian Habermehl, Gregor Hoepfner, Joerg Berroth, Stephan Neumann, and Georg Jacobs. Optimization Workflows for Linking Model-Based Systems Engineering (MBSE) and Multidisciplinary Analysis and Optimization (MDAO). *Applied Sciences*, 12(11), 2022. 6.5, 7.3

[HJZ+21] Gregor Hoepfner, Georg Jacobs, Thilo Zerwas, Imke Drave, Joerg Berroth, Christian Guist, Bernhard Rumpe, and Jens Kohl. Model-Based Design Workflows for Cyber-Physical Systems Applied to an Electric-Mechanical Coolant Pump. In Georg Jacobs and Sebastian Stein, editors, *IOP Conference Series: Materials Science and Engineering*, volume 1097:012004. IOP Publishing, Feburary 2021. 7, 1.4, 3.2, 5.1.2, 6.1, 6.5, 7.3

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of

Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007. 1.3.2, 1.3.2

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021. 1.3.4, D.2

[HMS16] Ekbert Hering, Rolf Martin, and Martin Stohrer. *Physik für Ingenieure*. Sprinter Vieweg, Berlin, Heidelberg, 2016. 7.1.1

[HNZ+23] Gregor Hoepfner, Imke Nachmann, Thilo Zerwas, Joerg K. Berroth, Jens Kohl, Christian Guist, Bernhard Rumpe, and Georg Jacobs. Towards a Holistic and Functional Model-Based Design Method for Mechatronic Cyber-Physical Systems. *Journal of Computing and Information Science in Engineering (JCISE)*, 23(5), 2023. 7, 1.4, 4.3, 4.3, 5

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004. 1.1.1, 1.4, 4.1, 6.3.1

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, volume 2010-01 of *Informatik-Bericht*, pages 13 – 22. fortiss GmbH, Germany, 2010. 1.3.4

[HRR12a] A. Haber, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, 2012. 3.4.3

[HRR12b] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012. 1.3.4

[IJZK23] Lukas Irnich, Georg Jacobs, Thilo Zerwas, and Christian Konrad. Combining and evaluating function-oriented solutions in model-based systems engineering. *Forschung im Ingenieurwesen*, 87, 2023. 5.1.2, 5.1.4, 6.1.2

[Isl04] M.N. Islam. Functional dimensioning and tolerancing software for concurrent engineering applications. *Computers in Industry*, 54(2):169–190, 2004. 7.3.1

[JDB09] Y. Jarraya, M. Debbabi, and J. Bentahar. On the meaning of sysml activity diagrams. In *2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 95–105, 2009. 7.3.6

[JDL+17] Soheil Jafari, Julian F Dunne, Mostafa Langari, Zhiyin Yang, Jean-Pierre Pirault, Chris A Long, and Jisjoe Thalackottore Jose. A review of evaporative cooling system concepts for engine thermal management in motor vehicles. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 231(8):1126–1143, 2017. 1.3.3

[JKB+21] Georg Jacobs, Christian Konrad, Joerg Karl Berroth, Thilo Zerwas, Gregor Höpfner, and Kathrin Spütz. *Function-Oriented Model-Based Product Development*. 2021. 6.2.2, 6.2.3

[JKB+22] Georg Jacobs, Christian Konrad, Joerg Berroth, Thilo Zerwas, Gregor Höpfner, and Kathrin Spütz. *Function-Oriented Model-Based Product Development*. Springer International Publishing, 2022. 5.1

[JKPB12] Thomas Johnson, Aleksandr Kerzhner, Christiaan J.J. Paredis, and Roger Burkhart. Integrating models and simulations of continuous dynamics into SysML. *Journal of Computing and Information Science in Engineering*, 12(1), 2012. 1.3.5, 6.2.3, 7.2.3

[Kau21] Oliver Kautz. *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Aachener Informatik-Berichte, Software Engineering, Band 46. Shaker Verlag, April 2021. 1.4, 4.1

[KHBC99] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. *IEE Proceedings - Software*, page 187, 1999. 7.3.6

[KHSE15] Michael Krappel, Claire Heidecker, Simon Streng, and Alfred Elsäßer. Electrical 48v coolant pump for highest thermal management requirements. In Michael Bargende, Hans-Christian Reuss, and Jochen Wiedemann, editors, *15. Internationales Stuttgarter Symposium*, pages 1219–1234. Springer Fachmedien Wiesbaden, 2015. 1.3.3

[KK98] Rudolf Koller and Norbert Kastrup. *Prinziplösungen zur Konstruktion technischer Produkte*. Springer, Berlin, Heidelberg, 1998. 1.2, 1.5, 2.2.1, 2.2.2, 2.3, 4.3, 5, 5.1, 5.1.1, 5.1.2, 5.1.2, 5.1.2, 5.1.3, 5.2, 5.1.4, 5.1.4, 5.1.4, 5.1.4, 5.1.4, 5.2, 5.2, 5.2.1, 5.2.1, 5.2, 5.2.1, 5.2.1, 5.2.1, 5.2.1, 5.2.2, 5.2.2, 5.2.2, 5.2.3, 5.2.3, 5.3, 6.1.1, 6.1.2, 6.1.3, 6.2.2, 6.2.2, 6.2.3, 6.3.2, 6.3.2, 6.4, 6.5, 7.1.1, 7.2.3, 7.3.1, 8, E.3

[KKBK07] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyoung Ko. Test cases generation from uml activity diagrams. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, pages 556–561. IEEE, 2007. 7.3.6

[Kle52] *Introduction to Metamathematics*. Van Nostrand, 1952. 3.1.1, C.1

[KM03] Hans-Joachim Kowalski and Gerhard O. Michler. *Lineare Algebra*. De Gruyter, 2003. 2.2.1, B, B.1, B.2, B.6, B.7, B.8, B.9, B.10, B, B.12, B.13, B.14, 1, B, B.15, B.16

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021. 8

[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. 6.5, 7.3.6

[Kol85] Rudolf Koller. *Konstruktionslehre für den Maschinenbau: Grundlagen des methodischen Konstruierens*. Springer,Berlin,Heidelberg, 1985. 1.1.2, 3.4.3, 3.3, 5, 5.1.1, 5.1.1, 5.1.2, 5.1.2, 5.1.2, 5.2, 5.1.2, 5.1.3, 5.2, 5.1.4, 5.1.4, 5.5, 5.2.1, 5.2.1, 6.1.2, E.3, E.3

[Kol98] Rudolf Koller. *Konstruktionslehre für den Maschinenbau: Grundlagen zur Neu- und Weiterentwicklung technischer Produkte mit Beispielen*. Springer,Berlin,Heidelberg, 1998. 3.1, 3.4.3, 4.1.1, 4.3, 5, 5.1, 5.1, 5.1.1, 5.1.1, 5.1.2, 5.1.2, 5.1.2, 5.1.3, 5.1.4, 5.1.4, 5.2, 5.2, 5.2.1, 5.2.1, 5.2.1, 5.2.1, 5.2.2, 5.2.2, 5.2.2, 5.2.2, 5.2.3, 5.3, 6.2.1, E.3, E.3

[Kol14] Rudolf Koller. *Konstruktionslehre für den Maschinenbau - Grundlagen zur Neu- und Weiterentwicklung technischer Produkte mit Beispielen.* Springer, Berlin, 4 edition, 2014. 6.5

[KPR+22] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Correct and Sustainable Development Using Model-based Engineering and Formal Methods. In *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. IEEE, September 2022. 8

[KPRR20a] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems. In Regina Hebig and Robert Heinrich, editors, *Combined Proceedings of the Workshops at Software Engineering 2020*, volume 2581. CEUR Workshop Proceedings, February 2020. 7.3.6

[KPRR20b] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety. In *AIAA Scitech 2020 Forum*. American Institute of Aeronautics and Astronautics, January 2020. 7.3.6

[KPRR21] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams. In Sebastian Götz, Lukas Linsbauer, Ina Schaefer, and Andreas Wortmann, editors, *Proceedings of the Software Engineering 2021 Satellite Events*, volume 2814. CEUR, February 2021. 8

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020. 4.2.2

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021. 1.3.4, 2.1.1, 2.2.1, 3.4.3, 7.3.3, D.2, D.2

[KV13] K. Kernschmidt and B. Vogel-Heuser. 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)*, 2013. 6.5

[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)*, 2008. 1.1

[Lei21] Jan Marco Leitmeister. *Einführung in die Wirtschaftsinformatik*. Springer Gabler Berlin, Heidelberg, 2021. 4.1

[LRSS23] Achim Lindt, Bernhard Rumpe, Max Stachon, and Sebastian Stüber. CDMerge: Semantically Sound Merging of Class Diagrams for Software Component Integration. *Journal of Object Technology*, 22(2), 2023. 4.2

[LW14] Jesko G. Lamm and Tim Weilkiens. Method for Deriving Functional Architectures from Use Cases. *Systems Engineering*, 17(2), 2014. 6.5

[MAD11] Monalisha Khandai, Arup Abhinna Acharya, and Durga Prasad Mohapatra. Test case generation for concurrent system using uml combinational diagram. *International Journal of Computer Science and Information Technologies*, pages 1172–1181, 2011. 7.3.6

[MAK15] G. Moeser, A. Albers, and S. Kümpel. 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)*, pages 50–55, 2015. 6.5

[Man15] Object Management Group. OMG Unified Modeling Language (OMG UML) Version 2.5, 2015. 1.3.5, 7.3.6

[Man17] Object Management Group. OMG Unified Modeling Language (OMG UML), 2017. 1.3.5, 6.3.1, 6.3.2, 6.3.2

[Man19] Object Management Group. OMG Systems Modeling Language (OMG SysML) Version 1.6, 2019. 1.3, 1.3.5, 6.3, 6.2.1, 6.2.1, 6.2.1, 6.2.2, 6.5, 7.2.1, 7.3.6, E.3

[MAT16] MATLAB & SIMULINK. Mathworks Inc. Simulinc User's Guide. Technical Report 2016b, 2016. 6.1.2

[MC12] Behrooz Mashadi and David Crolla. *Appendix: An Introduction to Bond Graph Modelling*, pages 511–528. John Wiley & Sons, Ltd, 2012. 7.2.1

[ME2] Mechanical Engineering. https://en.wikipedia.org/wiki/Mechanical_engineering. Accessed: 2023-06-27. 5

[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. 5.3.3

[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*, pages 69–78. Carl Hanser Verlag GmbH & Co. KG, 2015. 6.5

[MNN+22] Judith Michael, Imke Nachmann, Lukas Netz, Bernhard Rumpe, and Sebastian Stüber. Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines. In *Modellierung 2022*, pages 33–48. Gesellschaft für Informatik, June 2022. 5.1

[Mod22] OpenModelica User's Guide. online, 2022. 6.1.2

[Moe15] Georg Moeser. Example on "usage of free sketches in mbse". "raising the applicability of model-based systems engineering for mechanical engineers". Technical report, Karlsruher Institut für Technologie (KIT), 2015. 6.5

[Nar85] Louis Narens. *Abstract Measurement Theory*. MIT Press, 1985. D.2

[NKT+15] M. Nikolaidou, G. Kapos, A. Tsadimas, V. Dalakas, and D. Anagnostopoulos. Simulating sysml models: Overview and challenges. In *2015 10th System of Systems Engineering Conference (SoSE)*, 2015. 7.3.6

[NRSS22] Imke Nachmann, Bernhard Rumpe, Max Stachon, and Sebastian Stüber. Open-World Loose Semantics of Class Diagrams as Basis for Semantic Differences. In *Modellierung 2022*, pages 111–127. Gesellschaft für Informatik, June 2022. 2.1.1, 3.3.4

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *UML 99 — The Unified Modeling Language: Beyond the Standard Second International Conference Fort Collins, CO, USA, October 28–30, 1999 Proceedings*, pages 416–429. Springer Berlin Heidelberg, 1999. 7.3.6

[OAMD12] Samir Ouchani, Otmane Ait Mohamed, and Mourad Debbabi. Efficient probabilistic abstraction for sysml activity diagrams. In *Software Engineering and Formal Methods*. Springer Berlin Heidelberg, 2012. 7.3.6

[OLAA03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, pages 25–53, 2003. 7.3.6

[OMG13] OMG. Business Process Model and Notation (BPMN), Version 2.0.2. Specification, Object Management Group, 2013. 7.3.6

[PBFG07] Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Engineering Design*. Springer, London, 2007. 2.2.3, 5.1, 5.1.1

[PJHB19] Gerwin Pasch, Georg Jacobs, Gregor Höpfner, and Joerg Karl Berroth. Multi-Domain Simulation for the Assessment of the NVH Behaviour of a Tractor with Hydrostatic-Mechanical Power Split Transmission. In *77th International Conference on Agricultural Engineering / VDI-Wissensforum ; Supporters: VDI Max-Eyth Society for Agricultural Engineering*, Land.Technik AgEng 2019 : Hannover, 2019. 7.3.5

[Pto14] Claudius Ptolemaeus. System Design, Modeling, and Simulation using Ptolemy II, 2014. 1.1, 1.1.2, 3.4.2, 3.4.2, 5, 5.3.3, 6.5, 7.3.6

[Pum10] Sulzer Pumps. *Centrifugal Pump Handbook*. Elsevier, 2010. 6.2.3

[Rod91] Wolf G. Rodenacker. *Methodisches Konstruieren: Grundlagen, Methodik, praktische Beispiele*. Springer,Berlin,Heidelberg, 1991. 5, 5.1, 5.1.1, 5.1.2, 5.3.3

[Rot94] Karlheinz Roth. *Konstruieren mit Konstruktionskatalogen - Band I: Konstruktionslehre*. Springer, Berlin, 2 edition, 1994. 5.1, 5.1.2, 5.3.3

[Rot96] Karlheinz Roth. *Konstruieren mit Konstruktionskatalogen - Band III: Verbindungen und Verschlüsse - Lösungsfindung*. Springer, Berlin, 2 edition, 1996. 5.1.2, 5.3.3

[Rot00] Karlheinz Roth. *Konstruieren mit Konstruktionskatalogen: Band 1: onstruktionslehre*. Springer Berlin Heidelberg, 2000. 5.1.2, 5.1.2, 5.3, E.3

[Rot01] Karlheinz Roth. *Konstruieren mit Konstruktionskatalogen: Band 2: Kataloge*. Springer Berlin Heidelberg, 2001. 5, 5.1, 5.1.3

[Rot02] Karlheinz Roth. *Design catalogues and their usage*. Springer London, 2002. 5.1, 5.1.1

[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. 5.3.3

[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. 1.1.2, 1.3.4, 1.5, 3, 3.1, 3.1, 3.2.2, 3.2.2, 3.2.4, 3.2.6, E.3

[RRS23] Jan Oliver Ringert, Bernhard Rumpe, and Max Stachon. On implementing open world semantic differencing for class diagrams. *Journal of Object Technology*, 22(2), 2023. 2.1.1

[RS16] Stefan Roth and Achim Stahl. *Temperatur*. 2016. 6.1.1

[Ruc17] Anne Ruckpaul. *Synthese-getriebene Analyse technischer Systeme in der Produktentwicklung – Ein Beitrag zum Messen und Verstehen von Analyseprozessen während der Konstruktion unter Einsatz von Eye Tracking*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2017. 5.1.1

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996. 1.1.2, 1.1, 1.4, 1.5, 1.6, 1.3.2, 1.3.2, 1.3.2, 1.9, 2.1, 3.1.1, 3.1.1, 3.1.1, 3.1.1, 3.3.3, 3.16, 4, 4.1, 4.2, 4.2.2, 8, C

[Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013. 1.3.2, 1.3.2

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. 1.3.4, 3.3.4, 5.3, 6.1.1, 7.3.6, E.3

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017. 6.1, 7.3.3, 7.3.6

[RvdAtHW06] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the suitability of uml 2.0 activity diagrams for business process modelling. In *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modelling - Volume 53*. Australian Computer Society, Inc., 2006. 7.3.6

[RYF15] David A. Rubenstein, Wei Yin, and Mary D. Frame. Chapter 1 - introduction. In David A. Rubenstein, Wei Yin, and Mary D. Frame, editors, *Biofluid Mechanics (Second Edition)*, pages 3–13. Academic Press, second edition edition, 2015. D.2

[SB91] Simon Saunders and Harvey R. Brown. *The Philosophy of Vacuum*. Oxford University Press, 1991. 2.2.3

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. 1.3.4

[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. 5.3.3

[SG90] D Scott and C Gunter. *Semantic Domains and Denotational Semantics*. Elsevier Science Publishers, 1990. 3.1.1, C

[Sha09] Mark F. Sharlow. Generalizing the algebra of physical quantities, 2009. D.2

[SHT12] Maria Spichkova, Florian Hölzl, and David Trachtenherz. Verified system development with the autofocus tool chain. *Electronic Proceedings in Theoretical Computer Science*, 86, 07 2012. 7.3.6

[SJZK23] Kathrin Spütz, Georg Jacobs, Thilo Zerwas, and Christian Konrad. Modeling language for the function-oriented development of mechatronic systems with motego. *Forschung im Ingenieurwesen*, 87(1), Mar 2023. 4.3, 4.3

[SMM10] Santosh Kumar Swain, Durga Prasad Mohapatra, and Rajib Mall. Test case generation based on state and activity models. *The Journal of Object Technology*, 2010. 7.3.6

[SRS99] Thomas Stauner, Bernhard Rumpe, and Peter Scholz. Hybrid System Model. Technical report, TUM, 1999. 1.1.2, 1.2, 1.3, 1.5, 3.1.1, 3.10, 3.3.1, 3.3.2, 3.1, 4.2.1, 4.2.1, C, C.1, E.3

[SSK11] Mahesh Shirole, Amit Suthar, and Rajeev Kumar. Generation of Improved Test Cases from UML State Diagram Using Genetic Algorithm. In *Proceedings of the 4th India Software Engineering Conference*, pages 125–134. ACM, 2011. 7.3.6

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. 1973. 1.3.2, 4.1

[Sta10] Michael E. Starzak. *The First Law of Thermodynamics*. Springer New York, 2010. 2.2.2

[Ste94] K. Stephan. Thermodynamics. In *Dubbel Handbook of Mechanical Engineering*. Springer London, 1994. 1.3.3

[Sti89] Klaus Stierstadt. *Physik der Materie*. Wiley-VCH Verlag, 1989. 2.2.1, 2.2.2, 2.2.3, 5.2.2

[Sti18] Klaus Stierstadt. *Thermodynamik für das Bachelorstudium*. Springer Spektrum Berlin, Heidelberg, 2018. 5.2.2

[SW21] Eugene Syriani and Manuel Wimmer. Guest editorial to the theme section on multi-paradigm modeling for cyber-physical systems. *Software and Systems Modeling*, 20(3), 2021. 4.3

[Swa00] P. M. Swamidass, editor. *Engineering design*. Springer, Boston, MA, 2000. 5.1

[Sze78] Peter Szekeres. he mathematical foundations of dimensional analysis and the question of fundamental units. *International Journal on Theoretical Physics*, 17, 1978. D.2

[Tar55] A Tarski. A lattice-theoretical fixpoint theorem and its application. In *Pacific Journal of Mathematics*, number 5, 1955. 3.1.1, 3.3.3, C

[Tay08] Barry N. Taylor. The International System of Units (SI), 2008. 2.2.1, 2.2.1, 7.2.1, D.1, D.1

[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. 6.5

[UE03] Karl Ulrich and Steven Eppinger. *Product Design and Development*. McGraw-Hill, New York, 3 edition, 2003. 5.1.1

[uM04] VDI-Fachbereich Produktentwicklung und Mechatronik. Systematic embodiment design of technical products. Standard, VDI-Gesellschaft für Produkt- und Prozessgestaltung, Berlin:Beuth, 2004. 7.3.1

[UTF10] Andreea Urzica, Claudiu Tanase, and Adina Magda Florea. Bridging the gap between business experts and software agents: BPMN to AUML transformation. *UPB Scientific Bulletin, Series C: Electrical Engineering*, 72, 2010. 7.3.6

[vdAtHW03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business Process Management: A Survey. In *Business Process Management*, 2003. 7.3.6

[VDI82] VDI. *VDI 2222 Blatt 2 - Konstruktionsmethodik - Erstellung und Anwendung von Konstruktionskatalogen.* Beuth Verlag, Berlin, 1982. 5.1

[VDI91] Konstruktionskataloge; lösung von bewegungsaufgaben mit getrieben; grundlagen. Standard, VDI-Gesellschaft für Produkt- und Prozessgestaltung, 1991. 5.1.2

[VDI97] VDI. *VDI 2222 Blatt 1 - Konstruktionsmethodik - Methodisches Entwickeln von Lösungsprinzipien.* Beuth Verlag, Berlin, 1997. 5.1

[VLM02] Hans Vangheluwe, Juan Lara, and Pieter Mosterman. An Introduction to Multi-Paradigm Modelling and Simulation. In *Proceedings of the AIS'2002 Conference*, 2002. 4.3

[Vog15] Andreas Vogelsang. Model-Based Requirements Engineering for Multifunctional Systems. Technical report, Technical University of Munich, Institute of Informatics, 2015. 4.3, 4.3, 4.3

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011. 1.3.2

[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. 6.5

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012. 7.1.1, 7.1.2

[Wes16] Wolfgang Wesche. *Radiale Kreiselpumpen - Berechnung und Konstruktion der hydrodynamischen Komponenten.* Springer Vieweg, 2016. 7.3.1, 7.3.2

[Whi68a] Hassler Whitney. The mathematics of physical quantities: Part i: Mathematical models for measurement. *American Mathematical Monthly*, 75:115–138, 1968. D.2

[Whi68b] Hassler Whitney. The mathematics of physical quantities: Part ii: Quantity structures and dimensional analysis. *American Mathematical Monthly*, 75:227–256, 1968. D.2

[Win93] G Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993. 3.1.1, C

[WJD18] Matthias Wegerhoff, Georg Jacobs, and Pascal Drichel. 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. 7.3.5

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009. 7.3.6

[WLRW15] Tim. Weilkiens, Jesko G. Lamm, Stephan Roth, and Markus Walker. *Model-based system architecture*. John Wiley & Sons, 2015. 6.5

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016. 1.3.4

[WS09] Stefan Wölkl and Kristina Shea. A Computational Product Model for Conceptual Design Using SysML. In *Proceedings of the ASME 2009 International Design Engineering Technical Conferences and Computers and Information Engineering Conference*, 2009. 6.5

[WYY+04] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *11th Asia-Pacific Software Engineering Conference*, pages 284–291, 2004. 7.3.6

[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&C²-A in SysML. *International Journal of Computer Science*, 34-39, 2012. 6.5

[ZJK+22] Thilo Zerwas, Georg Jacobs, Julia Kowalski, Stephan Husung, Detlef Gerhard, Bernhard Rumpe, Klaus Zeman, Seyedmohammad Vafaei, Florian König, and Gregor Höpfner. Model signatures for the integration of simulation models into system models. *Systems*, October 2022. 5.1

[ZJS+21] Thilo Zerwas, Georg Jacobs, Kathrin Spuetz, Gregor Hoepfner, Imke Drave, Joerg Berroth, Christian Guist, Christian Konrad, Bernhard Rumpe, and Jens Kohl. Mechanical Concept Development Using Principle Solution Models. In Georg Jacobs and Sebastian Stein, editors, *IOP Conference Series: Materials Science and Engineering*, volume 1097:012001. IOP Publishing, Feburary 2021. 7, 1.4, 5.1, 5.1.2, 6.1

[ZRJ+22] Yizhe Zhang, Julian Roeder, Georg Jacobs, Joerg Berroth, and Gregor Hoepfner. Virtual Testing Workflows Based on the Function-Oriented System Architecture in SysML: A Case Study in Wind Turbine Systems. *Wind*, 2(3), 2022. 5.1, 6.5, 7.3

Appendix A

Definitions of Physical Quantities

The following table lists the physical quantities used in this dissertation. The variable $n \in \{1, 2, 3\}$ indicates the dimensionality.

Physical Quantity	**Definition**
AngularVelocity_n	$\mathbb{R}^n(\text{rad}\,\text{s}^{-1})$
Capacity	$\mathbb{R}(\text{L}\,\text{h}^{-1})$
Current	$\mathbb{R}(\text{A})$
Energy	$\mathbb{R}(\text{J})$
Force_n	$\mathbb{R}^n(\text{N})$
Length	$\mathbb{R}(\text{m})$
Momentum	$\mathbb{R}^3(\text{N}\,\text{m})$
Position_n	$\mathcal{A}_n(\mathbb{R}(\text{m})) \cup \xi$
Power	$\mathbb{R}(\text{W})$
Pressure	$\mathbb{R}(\text{Pa})$
Speed_n	$\mathbb{R}^n(\text{m}\,\text{s}^{-1})$
Temperature	$\mathbb{R}(\text{K})$
EntropyStream	$\mathbb{R}(\text{J}\,\text{s}^{-1}\,\text{K})$
Torque_n	$\mathbb{R}^n(\text{N}\,\text{m})$
Velocity_n	$\mathbb{R}^n(\text{m}\,\text{s}^{-1})$
Voltage	$\mathbb{R}(\text{V})$
Volume	$\mathbb{R}(\text{m}^3)$
VolumeFlowRate	$\mathbb{R}(\text{m}^3\,\text{s}^{-1})$
VolumeFlowRate	$\mathbb{R}(\text{m}^3/(\text{As}))$
Work	$\mathbb{R}(\text{W}\,\text{h})$

Table A.1: Physical Quantities used throughout this dissertations.

Appendix B

Terms and Definitions from Algebra

This thesis uses basic terms and definitions from linear algebra. Group theory provides a mathematical tool for defining the physical types energy, and matter. Euclidean and affine geometry are established mathematical tools for describing physical phenomena that have to do with space. This section briefly summarizes the most important definitions given in [KM03], which also provides deeper insights on these matters for the interested reader.

Groups

Definition B.1 (Group [KM03])**.** *A group is a set U together with an operation $\cdot : U \times U \to U, (u, v) \mapsto uv$ such that*

1. *$(uv)w = u(vw)$ for all units $u, v, w \in U$,*
2. *there exists an element 1_U such that $eu = u$ for all units $u \in U$,*
3. *for all $u \in U$ there exists $u^{-1} \in U$ such that $uu^{-1} = 1_U$.*

The group is called an abelian group iff the operation is also commutative, i.e., $uv = vu$ for all $u, v \in U$.

Definition B.2 (Subgroup [KM03])**.** *A subset $U \subseteq G$ of a group G is a subgroup of G, denoted $U \leq G$ iff U together with the multiplication on G forms a group itself.*

Definition B.3 (Generating Set [Fis13])**.** *Let G be a group, and $U \subseteq G$ be a subset of G. The group generated by U is defined as*

$$\langle\ U\ \rangle \stackrel{\text{def}}{=} \{u_1 u_2 \dots u_n \mid n \geq 1, u_i \in U\ \vee\ u_i^{-1} \in U \forall i \in \{1, \dots, n\}\}.$$

The generated group is therefore the smallest group that contains the subset U.

Definition B.4 (Finitely Generated [Fis13])**.** *A group G is finitely generated iff there exists a subset $S \subset G$ such that $G = \langle S \rangle$.*

Geometry

Definition B.5 (Metric [Foe10]). *Let X be a set. A* metric *on X is a mapping*

$$|| \; ||_X : X \times X \to \mathbb{R}, (x, y) \mapsto ||x, y||_X,$$

such that the following conditions hold:

- $||x, y||_X = 0$ *iff* $x = y$,
- *for all* $x, y \in X$ *it holds* $||x, y||_X = ||y, x||_X$ *(symmetry)*
- *for all* $x, y, z \in X$ *it holds that* $||x, z||_X \leq ||x, y||_X + ||y, z||_X$ *(triangle inequality)*

A metric space *is a tuple* $(X, || \; ||_X)$ *where X is a set and* $|| \; ||_X$ *is a metric on X.*

Example B.1. *The p-dimensional vector space over $\mathbb{R}$ is a metric space with the metric*

$$||x, y||_{\mathbb{R}^p} = ||x - y||_p.$$

Definition B.6 (Multilinear Form [KM03]). *For $n > 0$, let $V_1, V_2, \ldots, V_n, W$ be vector spaces over a field K. A mapping*

$$\phi : V_1 \times V_2 \times \ldots \times V_n \to W$$

is a multilinear form iff

1. $\phi(v_1, \ldots, v_i + v_i', v_n) = \phi(v_1, \ldots, v_i, \ldots, v_n) + \phi(v_1, \ldots v_i', \ldots, v_n)$, *for all* $v_j \in V_j$, $j = 1, 2, \ldots, n$ *and* $v_i, v_i' \in V_i$, $i = 1, 2, \ldots n$, *and*
2. $\phi(v_i, \ldots, v_i k, \ldots, v_n) = \phi(v_1, \ldots, v_i, \ldots, v_n)k$ *for all* $k \in K$, $v_i \in V_i$, $i = 1, \ldots, n$.

A multilinear form with $n = 2$ is called a bilinear form.

Definition B.7 (Scalar Product [KM03]). *Let V be a vector space over the field K. A bilinear form β of V is called a scalar product of V, iff*

1. *β is symmetric: For all $v, w \in V$ it holds that $\beta(v, w) = \beta(w, v)$, and*
2. *β is positive definite: For all $v \in V$ that are not the zero-vector, it holds that $\beta(v, v) \geq 0$.*

Example B.2. *Let K be a field, and $* : K \to K$ be the multiplication of K. Further, let $v = (v_1, \ldots, v_n), w = (w_1, \ldots, w_n) \in K^n$ be vectors. The standard scalar product on K^n, is defined as*

$$v \cdot w \stackrel{\text{def}}{=} \sum_{i=1}^{n} vi * w_i$$

Definition B.8 (Euclidean Vector Space [KM03]). *A vector space V over the field of real numbers $\mathbb{R}$ together with a scalar product over V is called a Euclidean vector space.*

Definition B.9 (Affine Space [KM03]). *An affine space over a field K is a set $\mathcal{A}$ of so called* points *together with (in case $\mathcal{A} \neq \emptyset$) a K-vector space $V_\mathcal{A}$, and a mapping that assigns each ordered pair (p, q) of points in $\mathcal{A}$ a vector $\vec{pq} \in V_\mathcal{A}$ such that*

1. *for all points $p \in \mathcal{A}$ and all vectors $a \in V_\mathcal{A}$ there exists exactly one point $q \in \mathcal{A}$ with $a = \vec{pq}$,*
2. *for all $p, q, r \in \mathcal{A}$ it holds that $\vec{pq} + \vec{qr} = \vec{pr}$.*

The dimension of the affine space $\mathcal{A}$ is the dimension of the respective vectorspace $V_\mathcal{A}$.

Definition B.10 (Affine Subspace [KM03]). *A subset $\mathcal{U}$ of an affine space $\mathcal{A}$ is called an affine subspace of $\mathcal{A}$ iff either $\mathcal{A} = \emptyset$ or the set $V_\mathcal{U} = \{\vec{pq} \in V_\mathcal{A} \mid p, q \in \mathcal{U}\}$ is a subspace of the K vector space $V_\mathcal{A}$. We denote that $\mathcal{U}$ is an affine subspace of $\mathcal{A}$ by $\mathcal{U} \leq \mathcal{A}$.*

Lemma 1. *The intersection $D = \cap\{\mathcal{U} \mid \mathcal{U} \in S\}$ of a non-empty system S of affine subspaces of the affine space $\mathcal{A}$ is itself an affine subspace of $\mathcal{A}$. For $D = \emptyset$ it holds that $V_\mathcal{D} = \cap\{V_\mathcal{U} \mid \mathcal{U} \in S\}$.*

See [KM03] for the proof of this lemma.

Definition B.11. *Let $\mathcal{M}$ be a subset of an affine space $\mathcal{A}$. The smallest affine subspace of $\mathcal{A}$ that contains $\mathcal{M}$ is*

$$\langle \mathcal{M} \rangle = \bigcap \{\mathcal{U} \mid \mathcal{M} \subset \mathcal{U} \ \wedge \ \mathcal{U} \leq \mathcal{A}\}$$

This subspace is the space generated by $\mathcal{M}$.

Definition B.12 (Affine Span [KM03]). *Let S be a system of affine subspaces of the affine space $\mathcal{A}$. Then the affine span of S is defined as*

$$\vee\{\mathcal{U} \mid \mathcal{U} \in S\} \stackrel{\text{def}}{=} \langle \cup \mathcal{U} \mid \mathcal{U} \in S \rangle.$$

Definition B.13 (Coordinate System [KM03]). *Let $\mathcal{A}$ be an affine space. An $(n+1)$-tuple $(p_0, \ldots, p_n)$ of points $p_i \in \mathcal{A}$ is* independent *iff the vectors $p_0\vec{p}_1, \ldots, p_0\vec{p}_n \in V_\mathcal{A}$ are linearly independent. An ordered $(n+1)$-tuple of points $K = (p_0, \ldots, p_n)$ from $\mathcal{A}$ is called a* coordinate system*, iff the $n+1$ points p_i in K are independent, and $\mathcal{A} = \vee\{\{p_0\}, \ldots, \{p_n\}\}$.*

Lemma 2. *If $K = (p_0, \ldots, p_n)$ is a coordinate system of the affine space $\mathcal{A}$, then $\mathcal{A}$ is of dimension n. A tuple of points $p_0, \ldots, p_n$ form a coordinate system of $\mathcal{A}$ iff $\{p_0\vec{p}_1, \ldots, p_0\vec{p}_n\}$ is a basis of $V_\mathcal{A}$*

Definition B.14 (Cartesian Coordinate System [KM03]). *Let $\mathcal{A}$ be a real or complex affine space. Then, $\mathcal{A}$ is called a euclidean or unitary affine space iff there is a scalar product $|\cdot|$ defined on $V_\mathcal{A}$. A coordinate system $K = (p_0, \ldots, p_n)$ is called a Cartesian coordinate system iff $\{p_0\vec{p}_1, \ldots, p_0\vec{p}_n\}$ is an orthonormal basis of $V_\mathcal{A}$. In this case, the distance between two points $p, q \in \mathcal{A}$ is defined as*

$$\overline{pq} = ||\vec{pq}||_3 \stackrel{\text{def}}{=} \sqrt{\vec{pq} \cdot \vec{pq}}$$

and the cosine of the angle (p, q, r) with p as apex is defined as

$$\cos(p, q, r) = \cos(\vec{pq}, \vec{qr}) = \frac{\vec{pq} \cdot \vec{qr}}{|\vec{pq}| \cdot |\vec{qr}}$$

Theorem 1 (Change of Coordinates [KM03]). *Let $\mathcal{A}$, and $\mathcal{A}'$ be two affine spaces with respective Cartesian coordinate systems $k = (p_0, p_1, \ldots, p_n)$, and $k' = (p'_0, p'_1, \ldots, p'_r)$. Then, $B = \{\overrightarrow{p_0p_1}, \overrightarrow{p_0p_2}, \ldots, \overrightarrow{p_0p_n}\}$ is a basis of the vector space $V_\mathcal{A}$ associated with $\mathcal{A}$, and $B' = \{\overrightarrow{p'_0p'_1}, \overrightarrow{p'_0p'_2}, \ldots, \overrightarrow{p'_0p'_n}\}$ is a basis of the vector space $V_{\mathcal{A}'}$ associated to $V_{\mathcal{A}'}$. Further, let $\alpha : \mathcal{A} \to \mathcal{B}$ be an affine mapping such that $\hat{\alpha} : V_\mathcal{A} \to V_\mathcal{B} : v \mapsto T^B_{B'} \cdot v$, where $T^B_{B'}$ denotes the transformation matrix form basis B to B', i.e., for all $v' \in V_\mathcal{B}$ there exists $v \in V_{\mathcal{A}'}$ such that $v' = T^B_{B'} \cdot v$. With $t = (t_1, \ldots, t_r)$ being the coordinate vector of $\alpha(p_0)$ in the coordinate system k', it holds for all coordinate vectors $x = (x_1, \ldots, x_n) \in \mathcal{A}$ such that the image of x under α has the coordinate vector $x' = (x'_1, \ldots x'_r)$ that*

$$x' = t + T^B_{B'} \cdot x.$$

The proof of Theorem 1 is given in [KM03].

Definition B.15 (Affine Mapping [KM03]). *Let $\mathcal{A}, \mathcal{B}$ be affine spaces. A mapping $\alpha : \mathcal{A} \to \mathcal{B}$ is an affine mapping iff there exists a linear mapping $\hat{\alpha} : V_\mathcal{A} \to \mathcal{B}$ such that $\hat{\alpha}(\vec{pq}) = \overrightarrow{\alpha(p)\alpha(q)}$.*

Lemma 3. *Let $\alpha : \mathcal{A} \to \mathcal{B}, \beta : \mathcal{B} \to \mathcal{C}$ be affine mappings on the affine spaces $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$. Then:*

1. *α is injective/surjective iff $\hat{\alpha}$ is injective/surjective.*
2. *α is bijective iff it is injective and surjective. A bijective affine mapping is called an* affinity*. In this case, an inverse mapping α^-1 exists and is also an affine mapping.*
3. *for an affine subspace $\mathcal{U} \leq \mathcal{A}$ it holds that $\alpha(\mathcal{U})$ is an affine subspace of $\mathcal{B}$.*
4. *the composition $\alpha \circ \beta$ is an affine mapping with $\widehat{\alpha \circ \beta} = \hat{\alpha} \circ \hat{\beta}$.*

Definition B.16 (Congruence [KM03]). *Let $\mathcal{A}$ be a euclidean affine space. A congruence is an affinity $\alpha : \mathcal{A} \to \mathcal{A}$ such that α preserves the distance between every two points $p, q \in \mathcal{A}$, i.e., $\overline{\alpha(p)\alpha(q)} = \overline{pq}$ for all $p, q \in \mathcal{A}$.*

Example The n-dimensional vector space K^n over a field K can be seen as an affine space itself, when seeing it as a set of points and, at the same time, as a vector space [Fis01]. We denote this space by $\mathcal{A}_n(K)$ [Fis01]. The mapping that assigns each tuple of points a unique vector is given by

$$K^n \times K^n \to K^n, (p, q) \mapsto p - q.$$

For CPSs, in particular, when considering the physical domain, the Euclidean vector space $\mathbb{R}^n$ together with the standard scalar product and euclidean norm accurately describes the physical space mathematically. Therefore, throughout this work, we consider the euclidean affine space $\mathcal{A}_3(\mathbb{R})$ when modeling aspects that concern geometry or space. We denote the standard euclidean norm on $\mathbb{R}^3$ as

$$||\cdot||_3 : \mathbb{R}^3 \to \mathbb{R}^3, v \mapsto \sqrt{v \cdot v},$$

where $v \cdot v$ denotes the standard scalar product on $\mathbb{R}^3$.

Appendix C

A Complete Partial Order on The Set of Timed Streams

Chapter 3 introduces the semantic domain of CPFs as a version of the FOCUS theory where TSPFs operate on both discrete and dense streams and behaviors of CPFs are described by sets of functions. Essentially this theory synthesizes the ideas from [SRS99] that introduces FOCUS on hybrid streams (*cf.* Section 3.1) and describes behaviors as sets of TSPFs on these hybrid streams and of the ideas from [Bro12] which introduces TSPFs that operate on both discrete and dense streams but describes behaviors as set-based TSPFs.
In order to prove that composition is well-defined in this theory, we need a theoretical basis. Here, the well-formedness of composition requires that the composition of a set of behaviors

(1) is not empty, and

(2) yields a behavior again.

To do so, we utilize Scott's domain theory [SG90, Win93, SRS99] and define a CPO on the set of timed streams. Further, we show that this order can be extended to channel histories which enables to understand the sets of input and output channels of a CPF, respectively, as completely partially ordered sets. This CPO further enables to introduce a notion of monotonicity for CPFs. Similar to the theory of FOCUS on discrete streams, introduced in [Rum96], the Knaster-Tarski theorem [Tar55] provides a means to show the existence of least fixed points for monotone CPFs. This appendix introduces the mathematical basis.

C.1 Complete Partial Orders

The basics to apply Scott's domain theory in our setting is defined, among others, in [BDD+93].

Definition C.1 (Partial Order [BDD+93])**.** *A* partial order *is a pair* $(D, \sqsubseteq)$*, where* D *is a set, and* $\sqsubseteq \subset D \times D$ *is a relation that is*

(1) reflexive, *i.e., $d \sqsubseteq d$ holds for all $d \in D$,*

(2) antisymmetric, *i.e., if $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ hold, then $d_1 = d_2$ holds for all $d_1, d_2 \in D$, and*

(3) transitive, *i.e., if $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ then it follows that $d_1 \sqsubseteq d_3$ for all $d_1, d_2, d_3 \in D$.*

In a partially ordered set $(D, \sqsubseteq)$, an *upper bound* is an element that is greater than all other elements, *i.e.,* $u \in D$ is an upper bound, iff for all $d \in D$ it holds that $d \sqsubseteq u$. An upper bound u is a *least upper bound* iff there exists no other upper bound that is smaller, *i.e.,* $u \in D$ is called a least upper bound, iff for all upper bounds $v \in D$ it holds that $u \sqsubseteq v$.
A chain is a subset of a completely partially ordered set such that if the order becomes total when restricted to the subset [BDD+93].

Definition C.2 (Chain [BDD+93])**.** *Let $(D, \sqsubseteq)$ be a partial order. A* chain *is a subset $S \subseteq D$ of D that is non-empty such that for every two elements $d_1, d_2 \in S$ either $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$ holds.*

Knaster-Tarski's fixpoint theorem states the existence of fix points of monotone functions on CPOs, *i.e.,* a CPO such that the least upper bound of each chain and of the set itself are included.

Definition C.3 (Complete Partial Order [BDD+93])**.** *A partial order $(D, \sqsubseteq)$ is called a* complete partial order *iff*

(1) the set D has a least element, which is denoted $\bot$, and

(2) for all chains $S \subseteq D$ the least upper bound $\sqcup S$ exists in D.

Definition C.4 (Monotonicity [BDD+93])**.** *Let $(D, \sqsubseteq)$ and $(S, \sqsubseteq)$ be CPOs. A function $f : D \to S$ is called* monotonic *iff for all $d, d' \in D$ it holds that $d \sqsubseteq d'$ implies that $f(d) \sqsubseteq f(d')$.*

Section 1.3.1 provides Cauchy's definition of continuity for functions on metric spaces. The above, however, allows to provide another notion of continuity for functions on CPOs.

Definition C.5 (Continuity [BDD+93])**.** *Let $(D, \sqsubseteq)$ and $(S, \sqsubseteq)$ be CPOs and $f : D \to S$ be a monotonic function. Then, f is called continuous iff for all chains $D' \subseteq D$ it holds that the least upper bound of all images of the elements of D' is equal to the image of the least upper bound of D', i.e., $\sqcup\{f(d) \mid d \in D'\} = f(\sqcup D')$.*

While monotonicity implies that feedback composition is well-defined, *i.e.,* the behavior that results from feedback composition is non-empty and also defines a behavior in the sense of Definition 3.13, continuity implies that the behavior of a function is fully described by its behavior for finite inputs [BDD+93].
A result from Kleene [Kle52] allows to approximate these fixpoints: The least fixpoint of a function $f : D \to S$, as in Definition C.5, denoted $fix.f$, can be approximated by a finite chain of functions. That is $fix.f = \sqcup_{n\in\mathbb{N}} f^n(\bot)$, where

$$f^0(d) = \bot \forall d \in D$$
$$f^{n+1}(d) = f(f^n(d)) \forall d \in D$$

This chain models the computation process that takes place in feedback loops.
Another theory that guarantees the existence of fix points is by Banach which provides results for fix points for functions on metric spaces. These results are often used to prove well-formedness of feedback loops for TSPFs over hybrid streams, *i.e.,* streams in the form $\mathbb{R}_+ \to \mathcal{U}^\omega$, for example in [SRS99]. For our approach that considers behaviors over interfaces that include discrete and dense channel histories at the same time. In this setting, the discrete channels "slow down" the transformation performed by the CPF, which is accurate because they are processed by discrete transformations (this holds also for the processing of items). Therefore, Scott's domain theory is accurate to interpret the transformations of CPFs.

Appendix D

An Algebraic Interpretation of Units

This section establishes an algebraic notion of units and physical quantities which can serve as a formal semantics for modeling languages of cyber-physical types.

D.1 Unit Systems

We consider a unit system as a finitely generated abelian group. The units in the generating set represent the base units, while the others are the derived units in the sense of [Tay08].

Definition D.1 (Unit System)**.** *A unit system is a set of unit names U together with an operation $\cdot : U \times U \to U, (u, v)$ that forms a finitely generated abelian group. The units in the generating set are called the* base units *of the unit system.*

Appendix B introduces the concept of a group together with the relevant theorems and notation. The commutativity of the operation is required to make the multiplication of quantities with units compatible with the multiplication of numbers. Throughout the rest of this dissertation, we use the SI-unit system $U = SI$, standardized in [Tay08]. The SI-unit system distinguishes between base units and derived units. Every derived SI-unit can be expressed as the product of powers of the base units [Tay08] with respect to this multiplication.
Because derived units can be expressed by the base units, there exists an equivalence relation on the set of SI-units. The relation needs to be an equivalence relation to reflect the notion that every SI-unit can be expressed as the product of powers of the base units, and that there exist alternative notations of the same unit.

Lemma 4. *Let $u, v \in SI$ be SI-units. Then u and v are equivalent, denoted $u \equiv_{SI} v$ iff there exists a finite set of SI-units $B = \{b_1, \ldots, b_k\} \subseteq SI$ $(k \in \mathbb{N})$ and $i_1, \ldots, i_k \in \mathbb{Z}$ such that*

$$v = u b_1^{i_1}\ b_2^{i_2}\ \ldots\ b_k^{i_k}.$$

The relation $\equiv_{SI}$ defines an equivalence relation on SI.

Since this equivalence relation defines a partition on SI, we can use the base-unit representation of every SI-unit as representative for each class, and assume $\#\,\mathrm{unit}(m) = 1$ for all messages $m \in \mathcal{U}$ without loss of generality.

D.2 Physical Quantities

Based on that we define physical quantities as tuples of a number domain and a unit in the unit system and provide definitions for the the multiplication and addition of such physical quantities. Physical quantities are types that distinguish by their unit.

Definition D.2 (Physical Quantity)**.** *A physical quantity is a type* $P \subseteq \mathbb{C}^n \times SI \subseteq \mathcal{U}$ *for* $n \in \mathbb{N}$ *such that for all messages* $p \in P$ *it holds that* $\mathrm{unit}(p) = \mathrm{unit}(P) \in \wp(U) \setminus \emptyset$*. The set of all physical quantities is denoted* $\mathbb{D}^n[SI]$*.*

All messages of a physical quantity, therefore, have the same non-empty set of units. Because of Lemma 4, every unit $u \in SI$, defines a physical quantity as the type $\{pv \mid p \in \mathbb{R}^n, v \equiv_{SI} u\} \subseteq \mathcal{U}$. For every physical quantity P, the mapping $\mathrm{value} : P \to \mathbb{R}^n, pu \mapsto p$ returns the value of the messages in P.
Let $P, Q \in \mathbb{D}^n[SI]$ be physical quantities such that $\mathrm{unit}(P) = u, \mathrm{unit}(Q) = v$, and $p, q \in \mathbb{R}^n$ for $n \in \{1, 2, 3\}$ for all $pu \in P$ and $qv \in Q$. The product of the messages $pu \in P$ and $qv \in Q$ is defined as $pu \cdot qv = (p \cdot q)\ (uv) \in \mathbb{R}^n \times \{uv\}$, where $uv \in \langle SI \rangle$. Now, let $T \subseteq \mathbb{C}^n$ be a numeric type, $\mathrm{unit}(T) = \emptyset$, and $d \in T$ be a message. Then the product $pu \cdot d$ is defined as $(p \cdot d)u \in P_n$ yields the multiplication of the numeric value of the message pu with the number d. The addition of two messages $pu, qu \in P_n$ is defined via $pu + qu = (p + q)u \in P_n$ for all $pu, qu \in P_n$. Both operations can be extended to more than two messages generically. These notions are implemented in a reusable DSL in MontiCore as published in [Kus21].

Physical Quantities as Metric Spaces The mapping value allows to interpret each physical quantity (equivalence class in the unit system) as a metric space (see Definition B.5), because the general p-norm provides a metric space on $P \in \mathbb{D}^n[SI]$ via

$$|| \; ||_3 : P \to \mathbb{R} \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

for all $u \in P$.

Discussion and Related Work on the Algebra of Physical Quantities Formalizing the notion of a unit system is not new. There exist many mathematical approaches that establish a mathematical interpretation of physical units [Nar85, Sze78, Whi68a, Whi68b, Sha09]. We utilize the simple algebraic

formalization of a unit system as an abelian group, and only distinguish between units and dimensions through Lemma 4. The term dimension refers to the actual physical quantity that is a measurable phenomenon in the real world, while the term unit refers to the measurements that can be applied to measure them [RYF15]. Other approaches utilize or establish unit systems as algebraic structures that, *e.g.,* also formalize the addition of physical quantities which is only possible between quantities of the same unit. Here, we have only defined the addition among physical quantities within the same equivalence class defined by Lemma 4. The implementation of an SI-unit modeling language provided in [Kus21] implements the multiplication and addition of physical quantities through context conditions [HKR21].

Appendix E

The SysML4FMArch MagicDraw Profile

This appendix provides the profile diagrams that show the implementation of SysML4FMArch in MagicDraw. The definitions of the stereotypes and the customizations that restrict their usage

E.1 Types

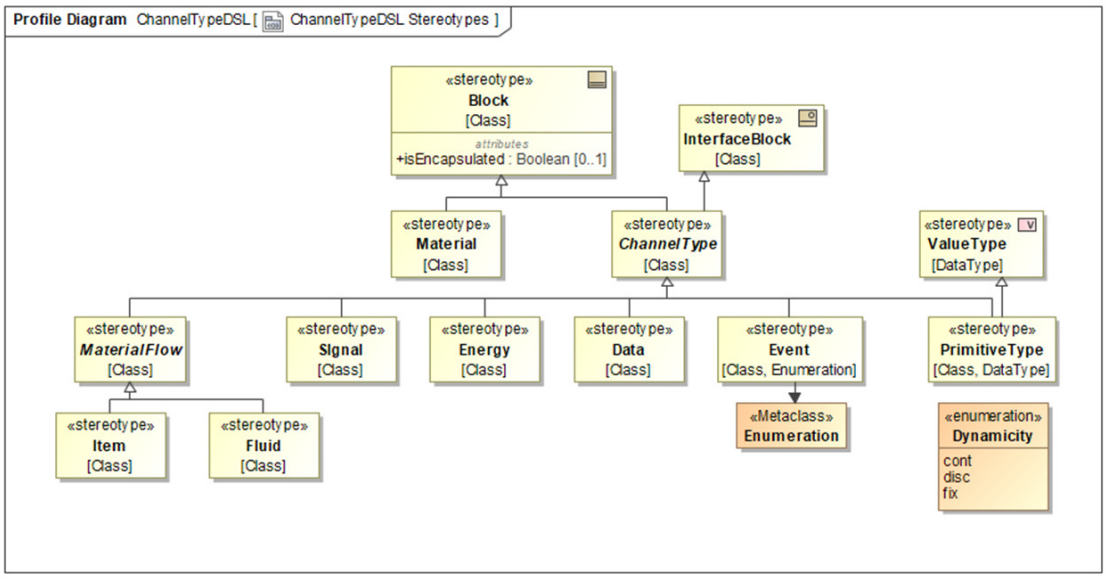


Figure E.1: Stereotypes that define the modeling elements to define cyber-physical types in SysML4FMArch.

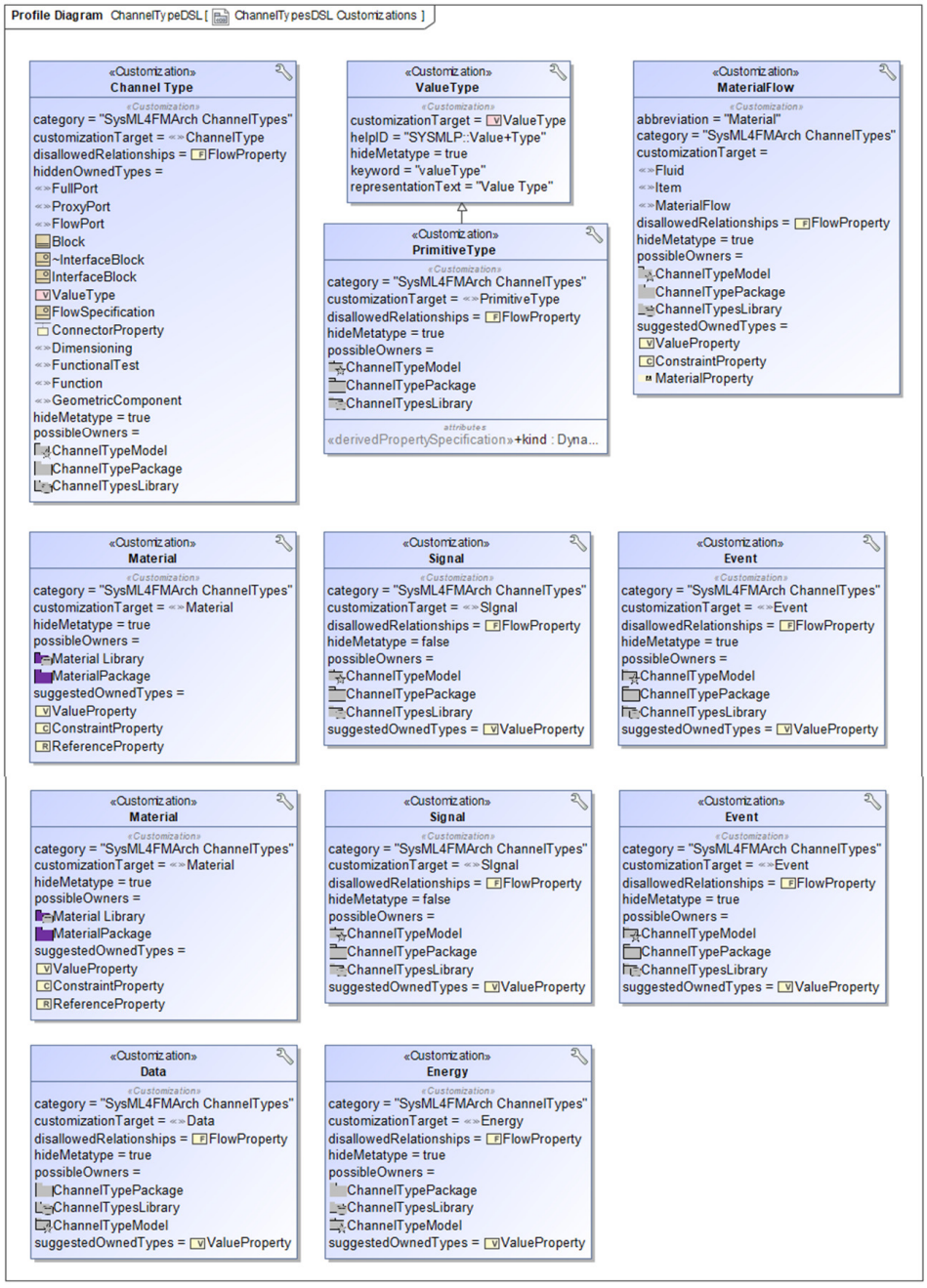


Figure E.2: Customizations for the stereotypes for modeling types in Figure E.1.

E.2 Functions

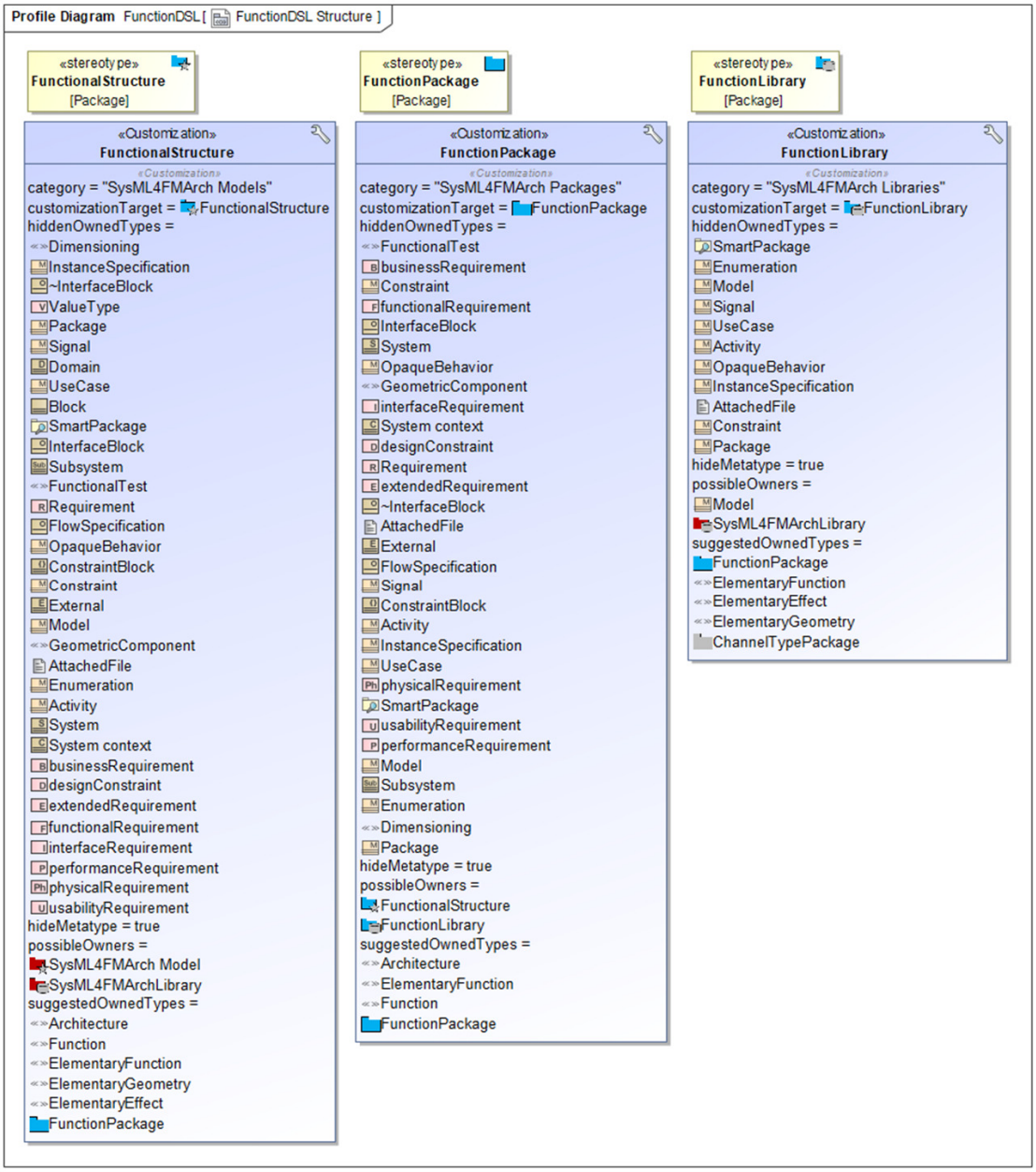


Figure E.3: SysML4FMArch stereotypes and customizations for structuring functional models in SysML4FMArch.

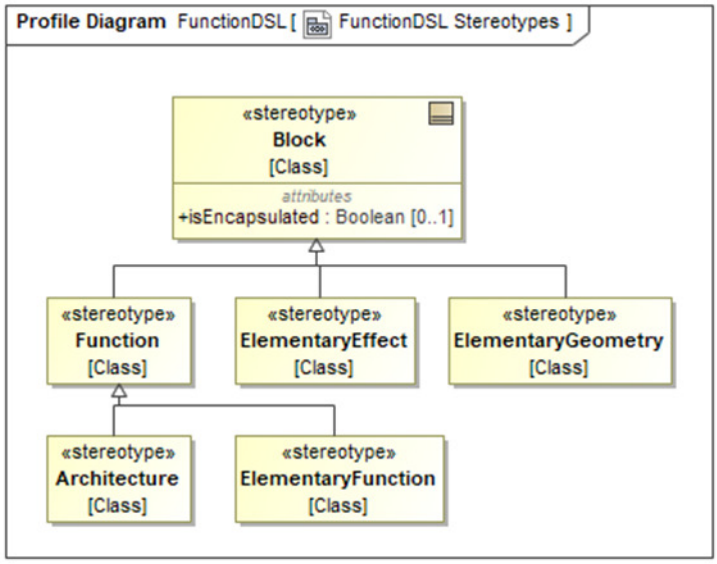


Figure E.4: Stereotypes that define the modeling elements to define CPFs in SysML4FMArch.

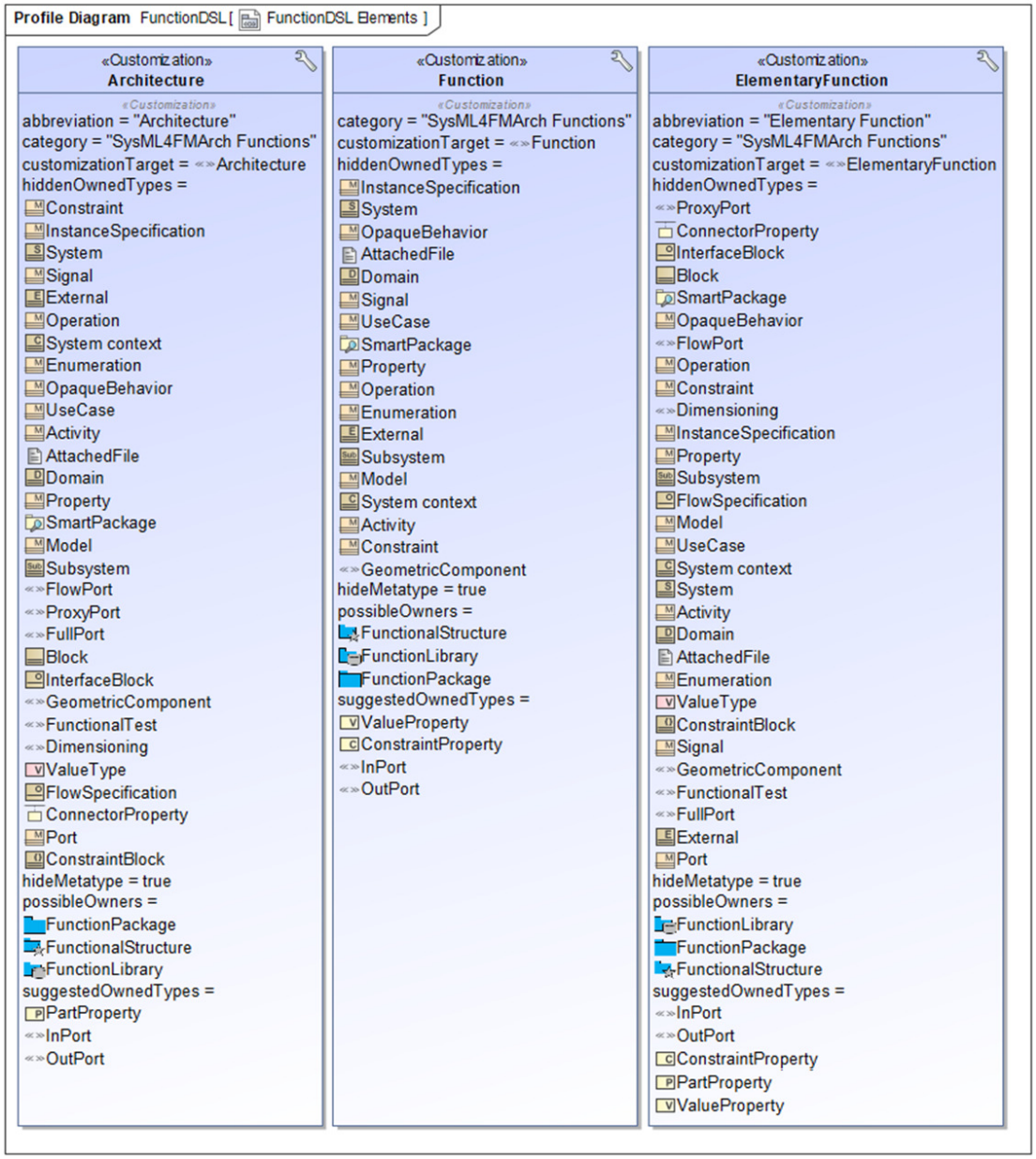


Figure E.5: Customizations for the stereotypes *Function*, *ElementaryFunction*, and *Architecture* in Figure E.4.

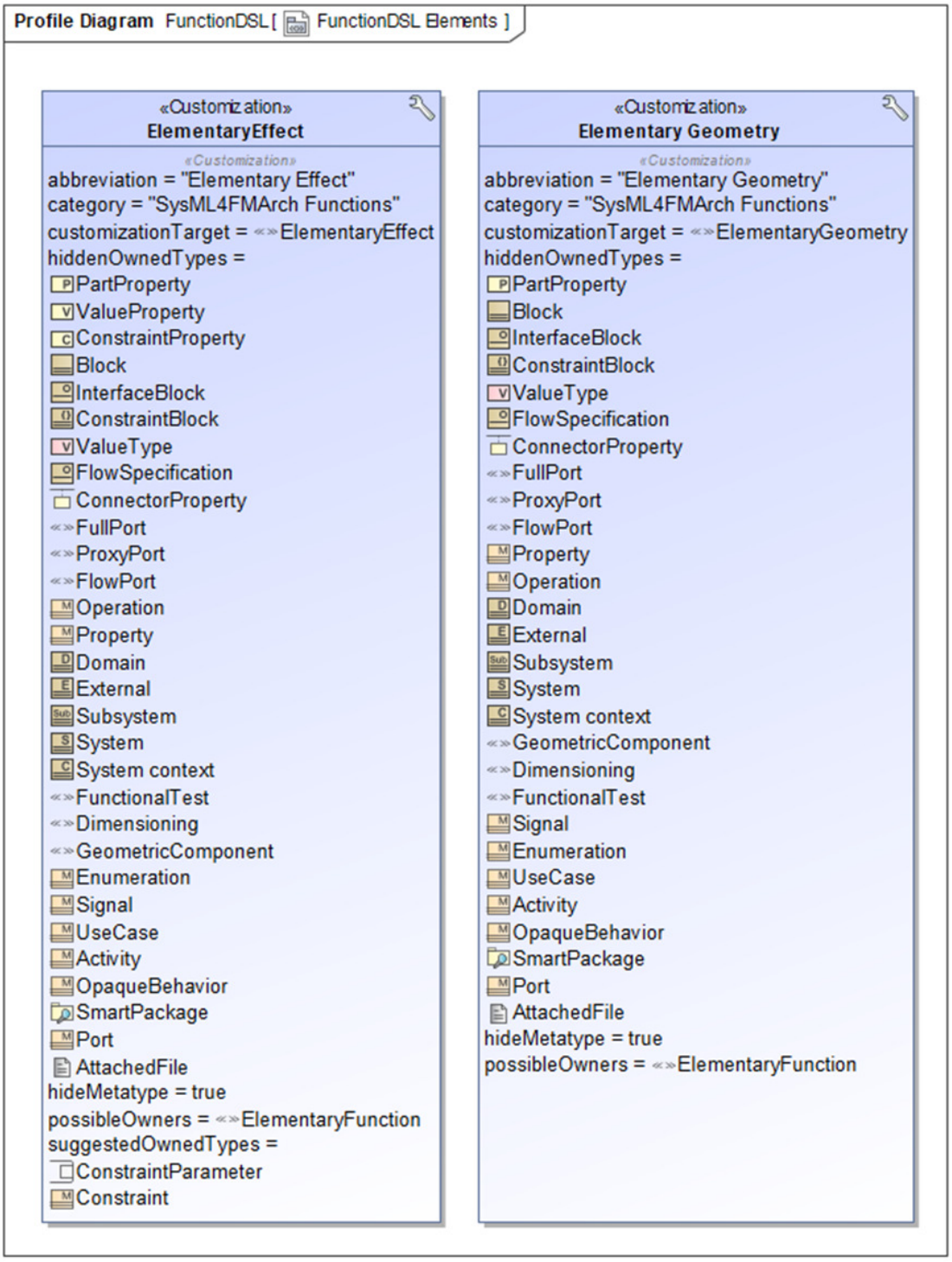


Figure E.6: Customizations for the stereotypes *ElementaryEffect* and *ElementaryGeometry* in Figure E.4.

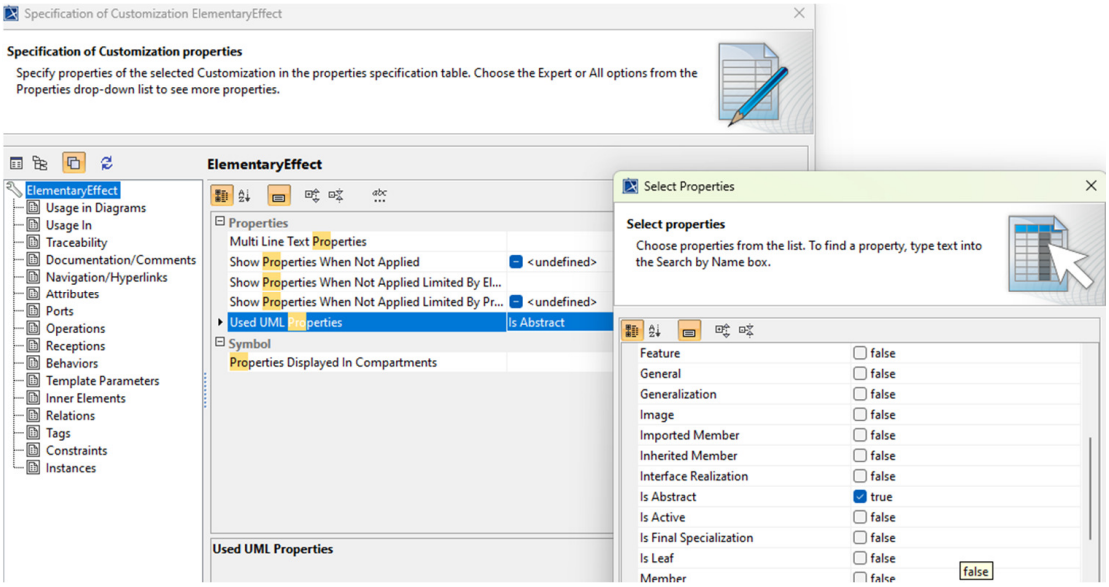


Figure E.7: Elementary effects and elementary geometries are abstract which is predefined in the customization.

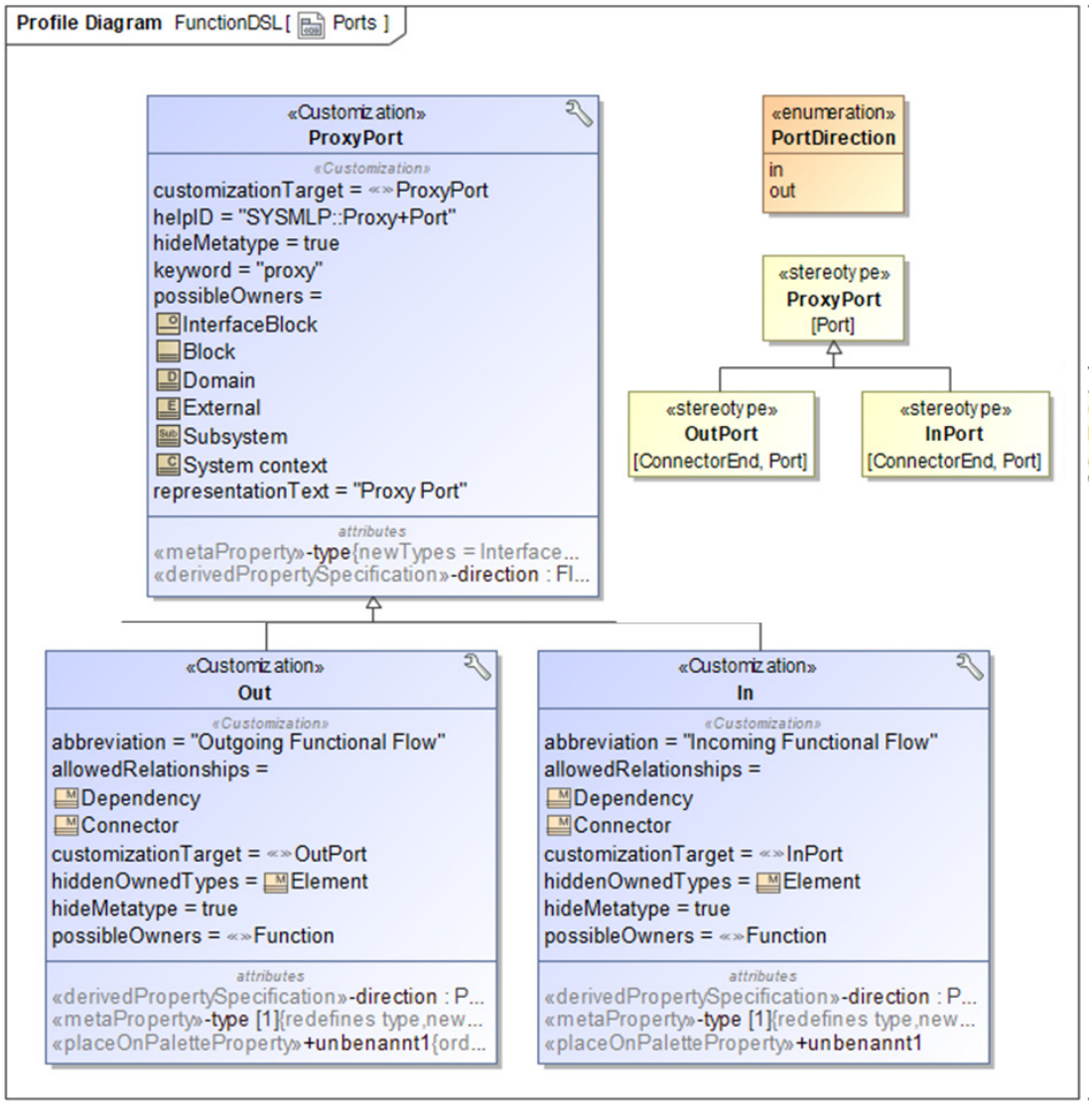


Figure E.8: To simplify modeling ports and their types, the MagicDraw implementation of SysML4FMArch offers special ports. This way modelers do not have to specify a channel type together with two InterfaceBlocks (*cf.* Figure 6.4). Because channel types are also InterfaceBlocks, it is sufficient to solely define the channel type. Distinguishing between *InPort* and *OutPort* allows to set the flow direction of the flow properties of the channel types and additionally prevents modelers from using ambiguous directions.

E.3 Solutions

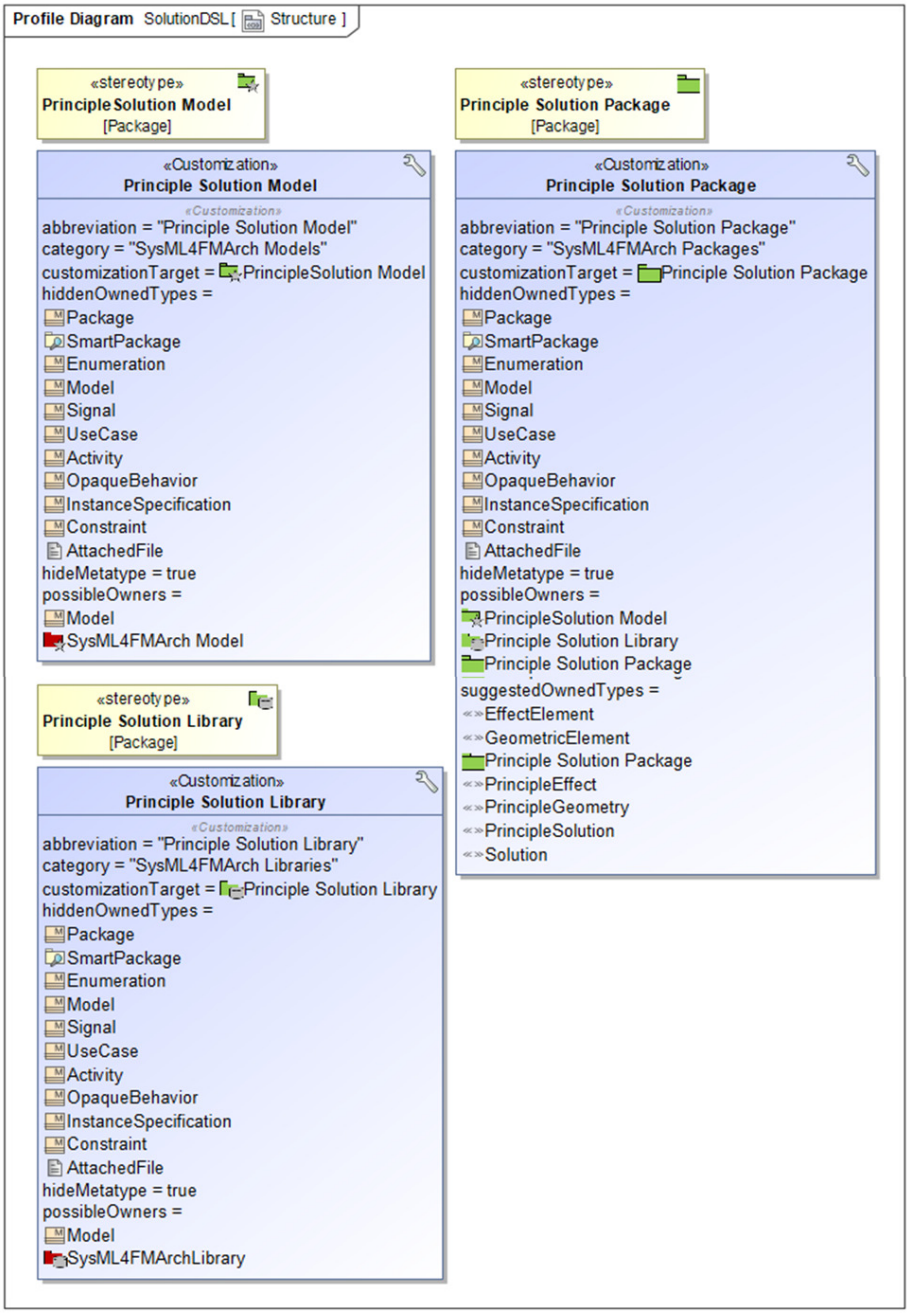


Figure E.9: SysML4FMArch stereotypes and customizations for structuring solution models in SysML4FMArch.

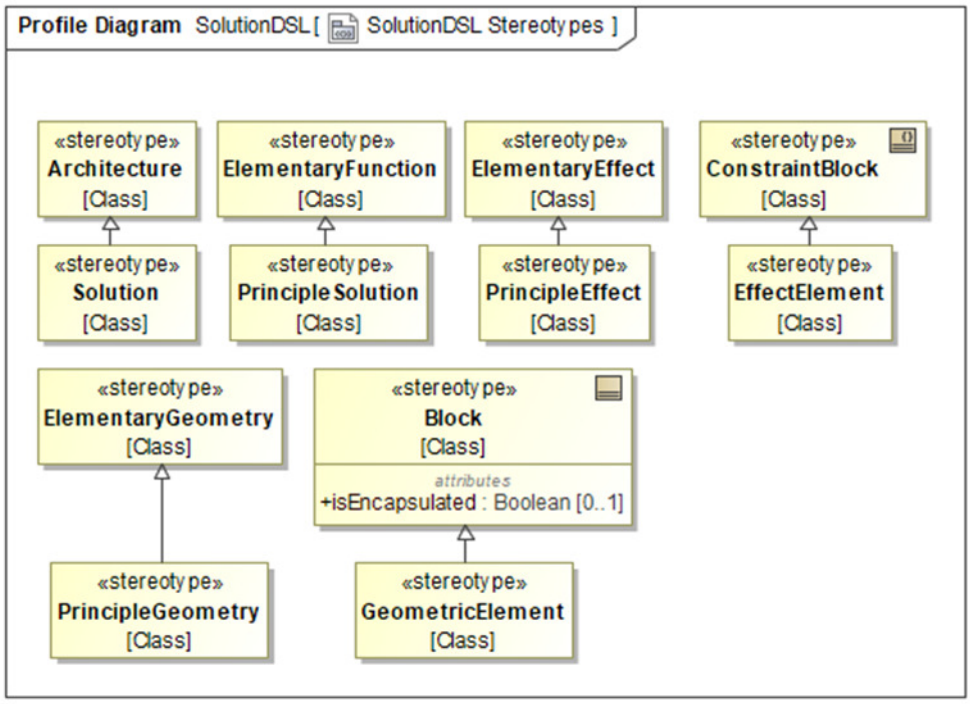


Figure E.10: Stereotypes that define the modeling elements to define solutions in SysML4FMArch.

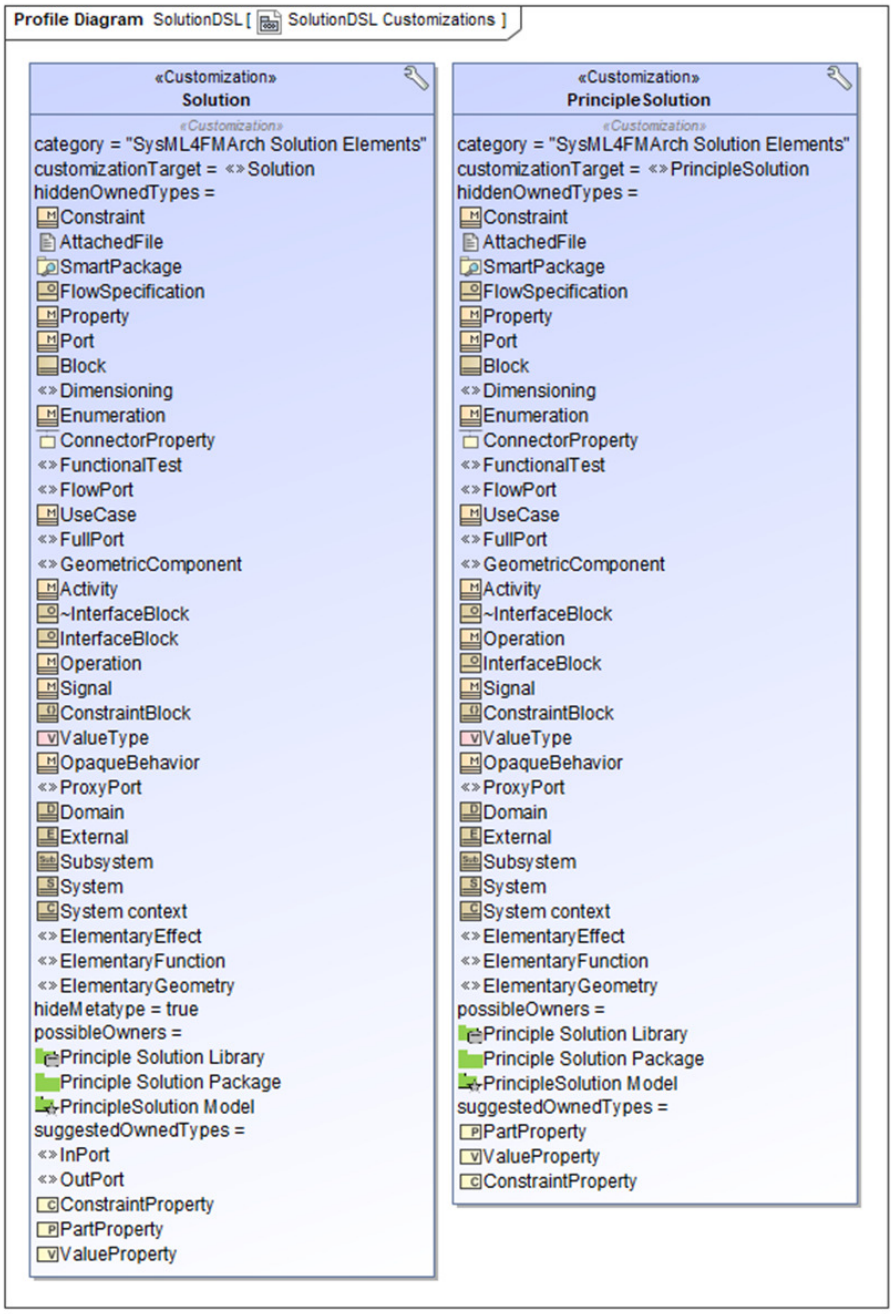


Figure E.11: Customizations for the stereotypes *Solution*, and *PrincipleSolution* in Figure E.10.

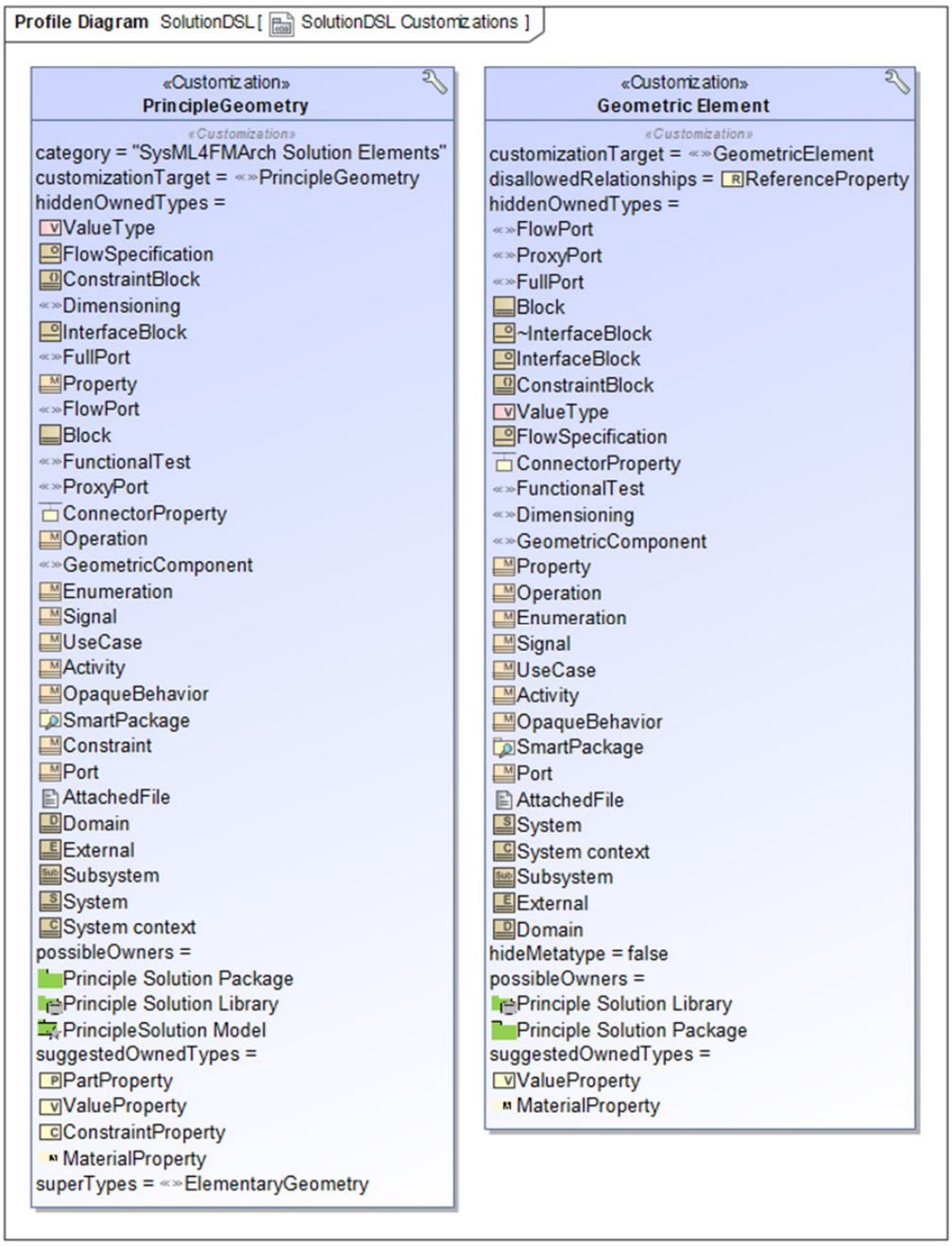


Figure E.12: Customizations for the stereotypes *PrincipleEffect* and *EffectElement* in Figure E.10.

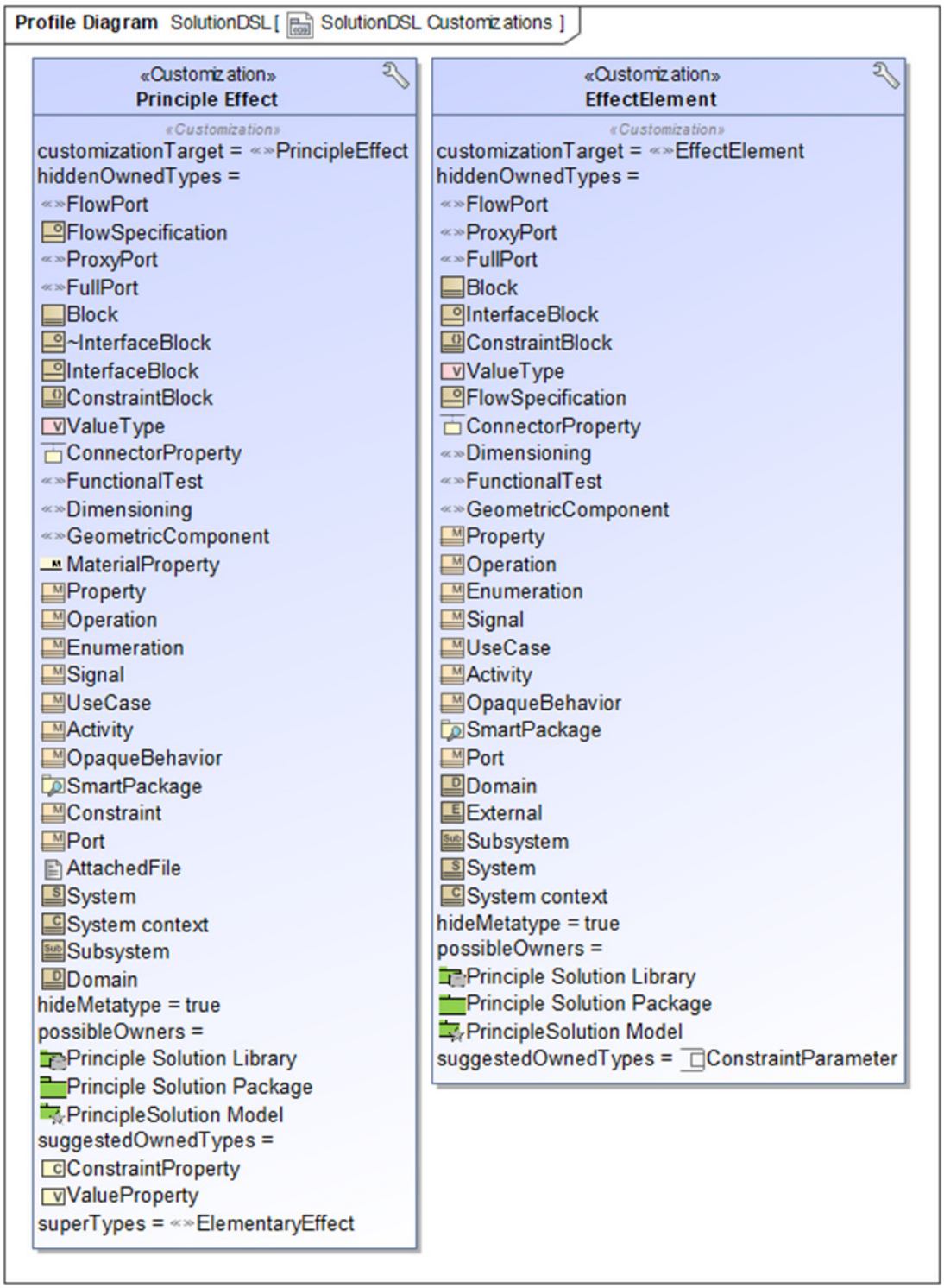


Figure E.13: Customizations for the stereotypes *PrincipleGeometry* and *GeometricElement* in Figure E.10.

Glossary

active surface the active surface describes the area at which or through which a physical phenomenon occurs [Kol85]. i, 86, 147

channel a channel is an identifier for a stream of messages [Bro01]. A channel always has a cyber-physical type that defines the image domain of the streams. i, 46, 238

class a class defines a cyber-physical type, and therefore always has a kind of type. Classes describe characteristics of the type through attributes and methods. See Chapter 2 for a detailed introduction. i

cyber-physical type a type is a set of messages. The kind of type defines whether the type describes energy, matter or data messages i, 21, 25, 123, 217, 237

data type data provides information about phenomena from the real world. A data type represents a finite or infinite set of discrete data messages. The set of all data types is denoted by $\mathcal{D}$. See Section 2.2.4 for details. i, 34

design catalog a design catalog is a collection of known and established solutions to design tasks or subfunctions [BG21] i, 91

design task *cf.* physical function. i, 79

effect carrier an effect carrier specifies the material or space to realize an elementary function. [Kol85] i, 82, 86, 118, 131

Electrical Engineering electrical engineering is"concerned with the study, design and application of equipment, devices and systems which use electricity, electronics and electromagnetism" [EE2] i

elementary function an elemenatry function is a function that is not further decomposed [Kol98, BG21]. It specifies a (set of) physical effects that realize the operation specified by the function. i, 81–100, 102–111, 153, 237, 243, 247–249

energy component The physical quantities that an exchange of energy in a specific form is bound to, are the *energy components* of that form of energy i, 28, 96

energy type when system components exchange energy, this energy appears in a specfic form. This form is fully described by an intensive and an extensive physical quantity. An energy-type therefore describes a form of energy. The set of all energy types is denoteed by $\mathcal{E}$: See Section 2.2.2 for details. i, 29

event an event is a predicate $e : TS \times \mathbb{R}_+ \to \mathbb{B}$. The event e occurs in the timed stream $s \in TS$ iff $e(s)$ holds. i, 34, 35, 238

event type an event type is a set of events. The set of all event types is denoted $\mathcal{E}v$. i, 35

functional view the functional view of a CPS describes the CPF that a CPS defines as a hierarchical strucutre. i, 238

geometric view the geometric view of a CPS describes the area in space that a CPS takes. i, 83, 238

interface an interface represents the signature of a CPF. Therefore, an interface comprises a set of typed channels via which the component that defines the CPF exchanges messages. Both, the functional view and the geometric view of a CPS share this interface. i

kind of type cyber-physical types categorize into energy-types, material and data types. The kind of a type indicates to which of these coategories a type belongs. i, 36, 118, 123, 237

material material is a form of matter. In this dissertation, the term refers to the types that define the physical entities that flow between two system components that exchange matter. The set of all materials is denoted $\mathcal{M}at$. Section 2.2.3 provides details on modeling material i, 33

mechanical engineering electrical engineering is"concerned with the study, design and application of equipment, devices and systems which use electricity, electronics and electromagnetism" [EE2] i, 116, 149, 150, 179

message a message is exchanged between cyber-physical components and describes an instantaneous interaction among these components. In this dissertation, messages must not be discrete (data) objects but also represent the instantaneous exchange of energy, matter, and signals. i, 29, 119, 237, 238

operation the operation is (the first) part in the textual description of a system's function considered in design methodology [Kol98, BG21]. It describes the

activity of the system under development that defines the specified function. An operation is a basic operation if it is not further divided into other operations. [Kol98] i, 80, 81, 85, 92

physical effect a ohysical effect is a physical phenomenon, physical occurrence, or process of a physical event and provides a causal relation between the inputs and outputs of a function. [Kol85, Kol98] i, 82, 83, 86, 147

physical function the general and desired relation between the input and output of a system to fulfill a task. [BG21] i, 79, 85, 86, 118, 123, 135, 237

physical law a physical law is a quantitative relation between physical quantities that involves material constants under certain circumstances. [Rot00] i, 82

physical principle a physical principle defines the effect and effect carrier to realize a function without defining their geometric shape. [Kol85] i, 82, 86

power data provides information about phenomena from the real world. A data type represents a finite or infinite set of discrete data messages. The set of all data types is denoted by $\mathcal{D}$. See Section 2.2.4 for details. i

primitive type a primitive type is a cyber-physical type from which all other cyber-physical types are constructed. i, 25

principle solution a principle solution defines a physical principle together with a geometric structure. [Kol85] i, 82, 83

signature the signature is a part of the definition of a cyber-physical component and describes the forms of interactions of the component with its environment. i, 238

Software Engineering software engineering means the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software i

Acronyms

AD Activity Diagram 170, 172, 174, 179, 180

ADL Architecture Description Language 13

AFWME Apply Fluid with Mechanical Energy 141, 173, 245

BDD Block Definition Diagram 15, 16, 126, 142, 144, 159, 163, 164, 166, 177, 243–245

BPMN Business Process Model and Notation 170, 180

C&C Component and Connector 14, 52, 61, 80, 120, 175

CAD Computer-Aided Design 75, 83, 135, 158, 181

CD Class Diagram 15, 18, 22, 23, 85, 243

CPCD Cyber-Physical Class Diagran 22, 23, 33, 42, 67–69, 152, 249

CPF Cyber-Physical Function 3–5, 8, 10, 13, 15, 18, 19, 25, 26, 35, 37, 39, 42, 45–50, 52–59, 61–64, 67–70, 72, 75, 77, 79, 81, 83, 84, 86, 88–95, 97–100, 102, 106–109, 111–116, 125, 137, 153–157, 181, 182, 213, 215, 224, 238, 243, 245–249

CPO Complete Partial Order 39–41, 213, 214

CPS Cyber-Physical System vii, xiii, 1, 3–7, 10, 11, 14, 18, 19, 21, 25, 26, 29, 32, 33, 35, 37–39, 41, 44–48, 50, 52, 54, 65–68, 70, 72, 73, 76–78, 82, 84, 116–118, 151, 158, 168, 179, 181, 182, 211, 238, 243

CSM Cameo Systems Modeler 171, 174, 177, 178, 180

DSL Domain-Specific Language 24, 25, 74, 123, 136, 137, 148, 218

HiFi High Fidelity 151

HIOS Hybrid I/O State Machine 52, 54, 56–59, 61, 64, 70–72, 86, 88, 89, 100–102, 104, 111, 121, 247, 248

IBD Internal Block Diagram 15–17, 74, 129, 135, 143, 161, 162, 164, 166, 177, 243–245

MBT Model-Based Testing 179

MDE Model-Driven Engineering 2–6, 9, 11, 14, 56, 65, 66, 68, 72, 75–77, 117, 118, 136, 148, 156, 168, 171, 175, 180, 243, 245

MLE Modeling Language Engineering 2, 136

OMG Object Management Group 148

SC Statechart 179

SE software engineering 178–180

SI International System of Units (Système International d'unités) 23–25, 27, 32, 33, 58, 61, 87, 105, 107, 160, 217–219

SMARDT Specification Method for Requirements, Design, and Test 67, 73, 74, 182

SPES Software Platform Embedded Systems 73, 74, 148, 149

SysML Systems Modeling Language xiii, 7, 15–17, 19, 74, 78, 83, 117, 122–127, 129, 133, 135, 137, 140, 141, 146, 148, 149, 158, 160, 170, 172, 179, 180, 243, 244

SysML4FMArch SysML for Functional Mechanical Architectures xv, 5, 7, 15, 17, 83, 117, 122–133, 135–141, 145–148, 158, 159, 161, 163–165, 167–169, 177, 178, 180, 221, 223, 224, 228, 231, 232, 244–246

TEE2ME Transform Electrical Energy to Mechanical Energy 129

TSPF Timed Stream-Processing Function 5, 6, 13, 14, 18, 37, 46–53, 58, 61, 70–73, 75, 76, 79, 115, 117–119, 122, 156, 181, 213, 215

UML Unified Modeling Language 9, 15, 85, 125, 135–137, 140, 148, 149, 170, 179, 180, 243

V&V Validation and Verification vii, 2, 4, 13

List of Figures

List of Tables

Related Interesting Work from the SE Group, RWTH Aachen

The following section gives an overview on related work done at the SE Group, RWTH Aachen. More details can be found on the website `https://www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:
Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question how digital and organizational techniques help to cope with physical distance of developers and [RRSW17] addresses how to teach agile modeling. Modeling will increasingly be used in development projects, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08, HR17] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, size, and number of the artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17]. To keep track of relevant structures, artifacts, and their relations in order to be able e.g. to evolve or adapt models and their implementing code, the *artifact model* [GHR17] was introduced. [BGRW18] explains its applicability in systems engineering based on MDSE projects.
An artifact model basically is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model therefore covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or

visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRvW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks. According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Smart semantic differencing operators have been defined for Activity Diagrams [MRR11a], Class Diagrams [MRR11d], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18].
We apply logic, knowledge representation and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests or find counterexamples using a theorem prover. And we have applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, HRR12] and based it on the core ideas of Broy's Focus theory [RR11, BR07]. Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams and Statecharts.
Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11a, KLPR12] and city quarters [GLPR15] to optimize the operation efficiency and prevent unneeded CO2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HR17]. In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKRS06, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12], we demonstrate how to systematically derive a transformation language in concrete syntax. [HMSNRW16] presents how to generate extensible and statically type-safe visitors. In [MSNRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLMSN+15b, HLMSN+15a]. To understand the implications of

executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] respectively [Rum12, Rum13] and is implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11e] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98], and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench
[GKR+06, KRV10, Kra10, GKR+08, HR17] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HR17] for easy development. New languages and tools can be defined in modular forms
[KRV08, GKR+07, Völ11, HLMSN+15b, HLMSN+15a, HRW18, BEK+18a, BEK+18b, BEK+19] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages in [KRV14]. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15] and the concern-oriented language development approach [CKM+18]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HRW18, BEK+19]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08, HLMSN+15b, HLMSN+15a, HMSNRW16, HR17, BEK+18a, BEK+18b, BEK+19] and the backend [RRRW15, MSNRR16, GMR+16, HR17, BEK+18b]. In [GHK+15b, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV14] describes the roles in software development using domain specific languages. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages [HRW18]. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets. The derivation of internal DSLs from grammars is discussd in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. In [RRW13a], we introduce a code generation framework for MontiArc. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [NPR13] and with the robotics domain [AHRW17a, AHRW17b]. [GHK+07] and [GHK+08a] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14b] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. Co-evolution of architecture is discussed in [MMR10] and modeling techniques to describe dynamic architectures are shown in [HRR98, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led

to the language workbench MontiCore [KRV10, HR17] that can even be used to develop modeling tools in a compositional form [HR17, HLMSN+15b, HLMSN+15a, HMSNRW16, MSNRR16, HRW18, BEK+18a, BEK+18b, BEK+19]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied for the development of CPS.

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12], and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance

[LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08a] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for compositional reuse [BEK+18b], and applied it as a semantic language refinement on Statecharts in [GR11].

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied for the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12], autonomous driving [BR12a, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. In [RRW13a], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition on contributing to systems engineering in automotive [GHK+08b], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19].

We leveraged SysML to enable the integrated flow from requirements to implementation to integration. To facilitate modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for machining based on these concepts [BKL+18]. Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20] and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14, RRW13a, RW18] as well as in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behaviour (2) based on information from previously stored and real-time monitored structural context and behaviour data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20] or UML/P based languages [MNRV19]. [MM15] describes a process how languages for assistive systems can be created.
We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view on the system design allows to track who does what, when, why, where and how with personal data, makes information about it available via information services and provides support using assistive services.

Modelling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15, HR17] that perfectly fit robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17a, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible to different robotics middleware platforms.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08a]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we introduce a framework for modeling the dynamic reconfiguration of component and connector architectures and apply it to the domain of cooperating vehicles. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of

building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App, and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

Model-Driven Engineering of Information Systems

Information Systems provide information to different user groups as main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17], we developed several generators for such data-centric information systems. *MontiGem* [AMN+20] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular and incremental, handwritten and generated code pieces are well integrated [GHK+15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017. E.3, E.3

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017. E.3, E.3

[AMN+20] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020. E.3

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007. E.3, E.3

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009. E.3, E.3, E.3

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009. E.3, E.3, E.3

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007. E.3, E.3

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007. E.3, E.3, E.3

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020. E.3, E.3, E.3

[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. E.3, E.3

[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. E.3, E.3, E.3

[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. E.3, E.3, E.3, E.3

[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. E.3, E.3, E.3

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich. E.3

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998. E.3, E.3

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017. E.3

[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. E.3

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017. E.3

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998. E.3, E.3

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018. E.3, E.3

[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. E.3

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017. E.3

[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. E.3

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007. E.3, E.3, E.3, E.3, E.3

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012. E.3, E.3

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012. E.3

[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. E.3, E.3

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015. E.3, E.3

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014. E.3

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008. E.3, E.3

[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. E.3, E.3

[CKM+18] Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures*, 54:139 – 155, 2018. E.3

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019. E.3

[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. E.3, E.3

[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. E.3, E.3

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020. E.3

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019. E.3

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020. E.3

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999. E.3, E.3

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line

Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019. E.3

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017. E.3

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998. E.3

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008. E.3, E.3, E.3

[FLP+11a] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011. E.3

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011. E.3, E.3

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012. E.3, E.3, E.3

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007. E.3, E.3, E.3

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008. E.3, E.3, E.3

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008. E.3

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß,

Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015. E.3, E.3, E.3

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015. E.3

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects.* Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017. E.3

[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. E.3

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996. E.3, E.3

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006. E.3, E.3, E.3

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007. E.3

[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. E.3, E.3, E.3

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67-81, 2006. E.3, E.3

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im*

Mobilen Internet, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015. E.3

[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. E.3, E.3, E.3

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020. E.3

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016. E.3, E.3, E.3, E.3

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995. E.3

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011. E.3, E.3, E.3, E.3

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012. E.3, E.3, E.3

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010. E.3, E.3

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013. E.3, E.3

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014. E.3

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015. E.3

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015. E.3

[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. E.3, E.3

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007. E.3

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009. E.3, E.3

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011. E.3

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012. E.3, E.3

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015. E.3, E.3, E.3, E.3

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015. E.3, E.3, E.3, E.3

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *The Journal of Object Technology*, 18(1):1–60, July 2019. E.3, E.3

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016. E.3, E.3, E.3

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004. E.3

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017. E.3, E.3, E.3, E.3, E.3, E.3, E.3

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998. E.3

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011. E.3, E.3

[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. E.3, E.3, E.3

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011. E.3

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012. E.3, E.3

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012. E.3, E.3

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015. E.3, E.3

[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. E.3, E.3, E.3

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018. E.3

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag. E.3

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009. E.3, E.3

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *The Journal of Object Technology*, 18(2):1–20, July 2019. The 15th European Conference on Modelling Foundations and Applications. E.3, E.3, E.3

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012. E.3, E.3, E.3

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020. E.3

[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. E.3, E.3

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019. E.3

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997. E.3, E.3

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012. E.3, E.3, E.3

[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. E.3

[KR18] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018. E.3

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010. E.3, E.3

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996. E.3

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing.* Springer, Schweiz, December 2014. E.3

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019. E.3

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017. E.3

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012. E.3, E.3

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006. E.3, E.3

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007. E.3, E.3

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007. E.3, E.3

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008. E.3, E.3, E.3

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010. E.3, E.3, E.3, E.3, E.3

[KRV14] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modeling Languages. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM‘ 06)*. CoRR arXiv, 2014. E.3, E.3

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering*, 27:119–151, April 2020. E.3

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011. E.3

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010. E.3, E.3, E.3, E.3

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019. E.3

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019. E.3

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013. E.3

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015. E.3

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010. E.3, E.3, E.3

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017. E.3

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019. E.3

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010. E.3

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011. E.3, E.3, E.3

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011. E.3, E.3

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011. E.3, E.3

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011. E.3

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011. E.3

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011. E.3, E.3

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011. E.3

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013. E.3

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014. E.3, E.3

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014. E.3

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020. E.3

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017. E.3

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016. E.3, E.3, E.3

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013. E.3, E.3

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002. E.3, E.3

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994. E.3, E.3

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999. E.3, E.3

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001. E.3

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003. E.3, E.3

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014. E.3

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996. E.3

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995. E.3

[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. E.3

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015. E.3, E.3, E.3

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017. E.3

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013. E.3, E.3, E.3, E.3

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013. E.3, E.3, E.3

[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. E.3, E.3, E.3

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015. E.3

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996. E.3

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002. E.3, E.3, E.3

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003. E.3, E.3

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004. E.3, E.3, E.3, E.3

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage.* Springer Berlin, September 2011. E.3

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012. E.3, E.3, E.3, E.3, E.3

[Rum13] Bernhard Rumpe. Towards Model and Language Composition. In Benoit Combemale, Walter Cazzola, and Robert Bertrand France, editors, *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, pages 4–7. ACM, 2013. E.3

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016. E.3, E.3, E.3

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017. E.3

[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. E.3, E.3, E.3, E.3, E.3, E.3

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012. E.3, E.3, E.3

[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. E.3

[SM18] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0 (Extended Abstract): Integration of non-smart resources into cognitive assistance systems. *EMISA Forum*, 38(1):35–36, Nov 2018. E.3

[SM20] Claudia Steinberger and Judith Michael. *Using Semantic Markup to Boost Context Awareness for Assistive Systems*, pages 227–246. Computer Communications and Networks. Springer International Publishing, 2020. E.3

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010. E.3, E.3, E.3

[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. E.3

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011. E.3, E.3, E.3

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012. E.3, E.3, E.3, E.3

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days.* EUROCONTROL, 2011. E.3, E.3