Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Innovations in Model-based Software And Systems Engineering

Katrin Hölldobler[a] Judith Michael[a] Jan Oliver Ringert[b]
Bernhard Rumpe[a] Andreas Wortmann[a]

a. Software Engineering, RWTH Aachen University, Germany

b. Department of Informatics, University of Leicester, UK

Abstract

Engineering software and software intensive systems has become increasingly complex over the last decades. In the ongoing digitalization of all aspects of our lives in almost every domain, including, e.g., mechanical engineering, electrical engineering, medicine, entertainment, or jurisdiction, software is not only used to enable low-level controls of machines, but also to understand system conditions and optimizations potentials.

To remain in control of all these heterogeneous systems of systems, a precise, but abstract understanding of these systems is necessary. To this end, models in their various forms are an important prerequisite to gain this understanding.

In this article, we summarize research activities focusing on the development and use of models in software and systems engineering. This research has been carried out by the working group of Bernhard Rumpe, which started 25 years ago in Munich, continued in Braunschweig, and since 10 years carries on at RWTH Aachen University.

Keywords Model, Modeling, Software Engineering, Language Workbench

1 Introduction

Development of a system consists of understanding what the system should do, decomposing the problem into smaller problems until they can be solved individually and composed into an overall solution. An important part of this process is decomposition, which leads to an architecture for the individual components. Quality can be obtained by powerful analyses or intensive forms of testing.

In each of these activities an understanding of the system is needed. Understanding the system is generally and always done by using *models*. Sometimes these models remain implicit as *mental models*, but often it is desired or required that models are made explicit for documentation, communication and analysis of the desired properties of the system.

Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, Andreas Wortmann. *Innovations in Model-based Software And Systems Engineering*. In Journal of Object Technology, vol. 18, no. 1, 2019, pages 1:1–60. doi:10.5381/jot.2019.18.1.r1

The roles of modeling in software and systems development are manifold and can vary greatly dependent on the chosen development process. To gain a better understanding of the manifold forms of models and their use in a development process, we discuss in the following series of research topics the use of *explicit models* denoted in *artifacts* of a more precisely, potentially even formally defined *modeling language*. We also discuss techniques to efficiently design new domain-specific languages (DSLs) using a language workbench, such as *MontiCore* [HR17]. Techniques such as language extension, aggregation and composition are the foundation of the *Software Language Engineering* discipline.

For a precise *analysis* of the correctness and quality of a model it is necessary that the modeling language itself has a precise meaning [HR04]. We therefore also discuss how to define semantics for languages, such as UML [Rum16, Rum17], SysML and derivations [RRW14, BHH+17]. Semantics for a DSL (in the sense of "meaning") completes the DSL definition process.

Knowing the semantics of a DSL (or any modeling language) allows to derive algorithms and finally tools that enable developers to use models in the development process. This includes of course *code generation* and potentially more sophisticated *model and code synthesis* techniques, but also the generation of *automated tests*, *consistency checking* between models, *metrics* to determine the quality of models, and constructive *refactoring* and *transformation* steps to *evolve* models. Models can also be used to describe *variability*, e.g., using delta-models to describe the difference to a base variant as an additive feature.

Agile software and systems development based on models is not a contradiction. With the right *tools* to automate development tasks, such as analysis, transformation and generation, an efficient, evolutionary process that uses models as primary artifacts becomes feasible. The last years, however, have shown, that it is still a long way to go. In particular, the (meta-)development of comfortable and usable *development tools* is an important prerequisite to further improve efficiency and predictability of the quality of a development project.

This of course holds for software development, but will more and more also hold for systems development, where hardware is designed in a model-based way and is quality ensured, e.g., through virtual integration as well as 3D-printer based prototyping.

Many software modeling languages used for software modeling are either describing component structure (and thus architecture), data structure (and thus informational content) or behavior (and thus state based reaction on incoming events). Formal methods have provided a *digital theory of informatics* in the last decades, which includes automata in its various forms, Petri nets [Rei92], temporal logics [Pnu77, Lam94], CSP [Hoa78], CCS [Mil78], π-calculus [Mil99] and various related approaches.

All other engineering and also scientific disciplines mainly use *differentiation/integration calculus* to describe continuous processes. Therefore, there is still a challenging problem to be solved: the digital theory of informatics is not well integrated with calculus that is needed to describe continuous processes of the physical world.

While the following sections describe steps towards a better integration of models into software and systems development techniques, there is also still a lot more to do.

As a consequence, the mission of the Software Engineering Department at RWTH Aachen University was and still 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.

The remainder of this paper presents our activities in different areas of software engineering. These activities are structured into three overarching topics: 1. From agile methods to language engineering (cf. Section 2), 2. the application of model-based engineering techniques to different domains (covered in Section 3), and 3. the tools and languages developed using the language workbench MontiCore (cf. Section 4). Thereafter in Section 5, we briefly summarize our research activities, identify open challenges, and formulate further research goals.

2 Topics From Agile Methods to Language Engineering

As observed in other engineering disciplines, also the efficiency of software engineering must be assisted by powerful models and tools. These must allow for an agile development process with effective iterations. Central research topics in model-based software development therefore integrate agile methods, code generation, evolution and re-factoring. In addition, model-based software development also faces the tricky question of what is a "good modeling language". The following subsections address semantics of UML and DSLs and behavior modeling based on automata. On the meta-level these subsections also address the engineering of languages with suitable semantics.

2.1 Agile Model-Based Software Engineering

Today, many developers and project managers assume that the use of models in software development leads to heavy-weight, tedious development processes. On the one hand, they believe that sooner or later, models are outdated, are not being co-evolved with software changes, contain bugs, and are no longer helpful. On the other hand, agility means to concentrate on the program as a core artifact without much extra documentation. Agility enables efficient evolution, correction and extension. As such, it seems to conflict with modeling.

We believe, and have shown, that using an executable, abstract and multi-view modeling language for modeling, designing, and programming still allows to use an agile development process (see the research hypotheses in [Rum04]). Such modeling languages, often consisting of one or more DSLs, are used as a central notation in the development process (see Figure 1). DSLs or the UML serve as a central notation for software development. A DSL can be used for programming, testing, modeling, analysis, transformation and documentation.

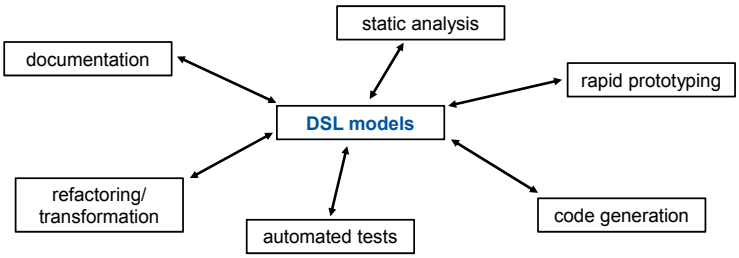

Figure 1 – DSL models as central artifacts: They can be used for many purposes.

We found that modeling will be used in development projects much more, when its benefits become evident early. This means constructive generation or synthesis of code from the models needs to be among the first steps of a model-based development process. All other interesting techniques, such as test synthesis or high level analysis techniques seem to come second. As a consequence, executability of modeling languages is a desired feature.

Execution of UML and DSLs

The question, whether UML should be executable, is discussed in [Rum02]. We found this a promising approach for larger subsets of UML, but also identified a number of challenges. We therefore started our research agenda to solve these challenges in order to make model-based software engineering (MBSE) truly successful when employed for agile software development. We explored in detail, how UML fits for that purpose. Not only the deficiencies of existing UML tools but also the UML language itself need to be adapted to fit the needs of an agile software development process, e.g., in [Rum03] we discuss how modeling of tests helps to increase reuse and efficiency and in [GKRS06] the integration of models and ordinary programming code.

Moreover, in [Rum12], [Rum11], [Rum16], and [Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The UML/P embodies class, object, sequence diagrams, Statecharts, and OCL in combination with Java to model code as well as tests as sketched in Figure 2. Moreover, these publications include a detailed discussions on how to use the UML/P for code generation, testing and on how to refactor structural models such as class diagrams, as well as behavioral models such as Statecharts. Additionally, forms of language integration, e.g., using object diagrams within OCL to describe desired or unwanted object structures, are presented there as well.

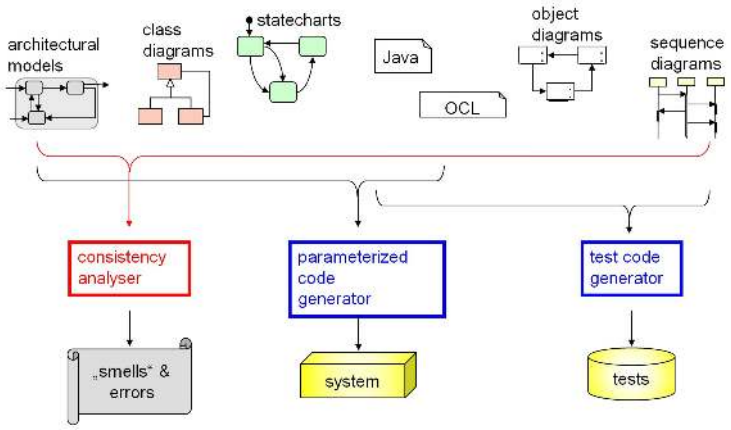

Figure 2 – UML/P used for code and test generation.

In the last decade, we implemented a language workbench called MontiCore (see Section 4.1) that was first described in [GKR+06]. On top of MontiCore, we realized most of the language components of the UML/P [Sch12]. This work includes a precise definition of textual languages, type checks, checks for context conditions within and between UML sub-languages and a framework for the implementation of code generators.

Specific Concepts Assisting Agile Development

Agile development processes require quite a lot of specific activities, techniques and concepts. Our research includes a general discussion of how to manage and evolve models [LRSS10] and a precise definition for model and modeling language composition [HKR+09]. Compositionality is particularly important and must be designed carefully as it allows for incremental analyses and code generation, thus being much more agile than today's modeling tools. We also discussed in detail what refactoring means and how it looks like in various modeling and programming languages [PR03]. UML/P is implemented to support modeling without redundancies even on different levels of abstraction, which enhances refactoring and evolution techniques on models. To better understand the effect of an evolved design, we discuss the need for semantic differencing in [MRR10].

When models are the central notation, model quality becomes an important issue. Therefore, we have described a set of general requirements for model quality in [FHR08]. We distinguished between internal and external quality. External quality refers to the correctness and completeness of a model with respect to the original that it describes, while internal quality refers to the model presentation and thus plays the same role as coding guidelines for programming languages.

We also know that, even when using the UML/P, there is additional effort necessary to adapt the tooling infrastructure to the project specific needs. This becomes more pressing, when a domain specific language is specifically designed for a project. We discuss the additional roles and activities necessary in a DSL-based software development project in [KRV06]. However, it may be that not all of the participants in a project need to be physically at the same place and fully available anymore [JWCR18].

We expect that the use of models at runtime will soon become an agile and efficient development technique. It allows developers to defer design decisions to runtime adaptation and configuration of systems. However, reliability then becomes an issue. In [CEG+14] we have therefore discussed how to improve reliability while retaining adaptability. In [KMA+16] we have also introduced a classification of forms of reuse of modelled software components.

2.2 Generative Software Engineering (GSE)

We believe that *modeling will only become an integral part of the process in many industrial projects*, if *automatic derivation of executable code* and *smooth integration with handwritten code* is a standard feature of its tooling. In Section 2.1 we clarify that generating software is an important capability for a tooling infrastructure that successfully assists modeling in the development process. We therefore examined various aspects of generation. For instance, in [Rum12, Rum11, Rum16, Rum17] we define the language family UML/P (a simplified and semantically sound derivate of the UML) which is designed specifically for product and test code generation from class diagrams, object diagrams, Statecharts and sequence diagrams as shown in Figure 2.

We developed several generators, e.g., a flexible, modular and reusable *generator for the UML/P* [Sch12], a generator for generating simulation components out of the architectural analysis and design language MontiArc [Hab16] (used for the cloud as well as cyber-physical systems (CPS), such as cars or robotics [HRR12], see Section 4.2) or the generator MontiGEM [AMN+19] for enterprise information systems (see Section 4.4). All of them are based on the MontiCore language workbench [KRV10, GKR+06]. With MontiCore we are able to easily define extensions of

languages as well as new combinations and thus are able to reuse the defined UML/P sub-languages and generation techniques in various applied projects.

Tooling and especially generators will only be successful in practical projects, if they have an appropriate impact on the development process, i.e., *development processes need to be adapted* or completely reshaped according to the availability of a generator. In [KRV06], we discuss additional roles necessary in a model-based software development project (while other roles either vanish or their workload can greatly be reduced).

Working with *generated and handwritten code* leads to challenges related with continuous repetitive generation (the generation gap problem). In [GKRS06, GHK+15a, GHK+15b] we discuss mechanisms to keep generated and handwritten code separated, while integrating them in the product and enabling the repetitive generation (which is much more valuable than one-shot generation).

For various purposes, including preparation of a model for generation, it is helpful to define *model transformations*. We are able to create transformation languages in concrete syntax, that reuse the underlying language concepts [Wei12, HRW15, Höl18]. Even more important, we describe how to systematically derive a transformation language in concrete syntax. Since then, we have applied this technique successfully for several UML sub-languages and DSLs [HHRW15, AHRW17a, Höl18] (see Section 2.7).

Sometimes *executability* can be a disadvantageous characteristics for a modeling language, especially when people start modeling concrete algorithms instead of abstract properties. We therefore discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] as well as the advantages and perils of using modeling languages for programming in [Rum02].

2.3 Unified Modeling Language (UML)

Challenges for the standardization of the UML, such as problems for defining a semantics for the entire UML, have been well-known for a long time [KER99]. Thus, many of our contributions build on the UML variant UML/P which is suitable for programming. UML/P is described in [Rum16] and [Rum17] and implemented in a first version in [Sch12] (see Section 4.3).

Defining variants of a modeling language in a systematic way is useful for adapting the language to domain or project specific needs. Thus, *semantic variation points* of the UML are first discussed in [GR11]. It discusses formal semantics for UML [BHP+98] and describes 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 [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. These concepts are also applied 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].

Other important questions are 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]. Figure 2 demonstrates the principal forms of uses of UML models in agile development projects. The exemplary diagrams, namely object diagrams and sequence diagrams, are used

for test case definition, whereas the more "complete" diagrams are used for code generation (see also Section 2.2).

The idea of *adaption and extension* of the UML in order to better suit the needs of specific domains or settings, is another important aspect. [PFR02] describes product line annotations for UML. More general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

To use UML not only for analysis but also for programming has an impact on the development process. 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].

2.4 Domain-Specific Languages (DSLs)

The ability to abstract is one of the basic cognitive abilities of the human being. Both science and philosophy use models to understand and describe the concepts and phenomena in their fields. Engineering disciplines use models to describe the systems they intend to design. All human beings use models, but only informatics defines and studies the set of valid models, namely the *modeling language* explicitly. This is made necessary because computer scientists use models not only to communicate among each other, but also with computers.

Informatics, therefore, is very much about languages. We use universally applicable modeling languages to describe problems and problem contexts. We employ general-purpose programming languages (GPLs) to implement solutions. We specify properties, architect and design solutions. And we define tests, as well as an increasing number of application specific languages and DSLs tailored for a concrete target area.

A DSL is always constructed with a *particular domain* in mind. Examples include HTML for websites, Matlab for numerical computation, or SQL for relational database management. In each instance, the DSL trades some of the expressiveness of GPLs in order to allow for more concise models in the target domain.

As software systems have become essential components of nearly all innovative products, increasingly many non-ICT experts now find themselves working with these systems. Furthermore, complexity of software-based systems is increasing. While modeling languages such as UML provide a high level of abstraction to deal with complexity, these languages are usually still too technical (hence UML profiles are useful, as discussed in [GHK+07] and [PFR02]). DSLs address both of these problems. *Non-ICT experts benefit from DSLs* by being able to transfer already familiar language concepts to the new application. Experienced users benefit by having a smaller mental gap between the software system and the associated real world models.

The main drawback of domain specific languages currently is still their *challenging creation process*. Not only does the creation of a computer language necessitate the fundamentals, such as a carefully defined grammar and corresponding translation programs. Productive usage of a language also requires extensive tool support. Generative software engineering techniques are at the center of attention for attempts to meet these challenges. In [SRVK10] we discuss the state of the art and current efforts to develop languages through meta-modeling.

Figure 3 depicts the architecture of a typical DSL processing tool in a model-based software engineering process. DSLs and the models expressed with them are becoming first-class elements of the software engineering process. In order to support this

development, research was and is necessary focusing on new, effective, and efficient ways of creating DSLs and corresponding tool support. The processing of a model is relatively similar to classical compiler architecture it consists of a front end handling the input, an internal validation and transformation part and a back and to produce the desired results. Like in classical compiler construction parts of this infrastructure is generated using a meta-tool, in this case MontiCore.

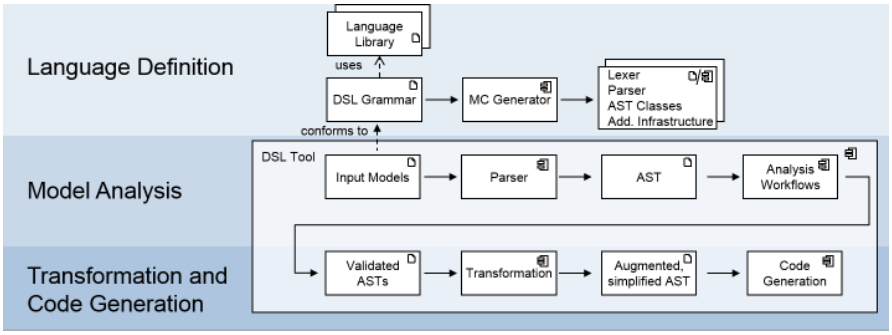

Figure 3 – The role of DSLs in a model-based software engineering process.

DSL Definition

DSLs have to be designed carefully to meet their respective requirements. The core design of a DSL consists of a desired *concrete and abstract syntax* [CFJ+16]. We examine the relations between concrete and abstract syntax and propose a language definition format in [KRV07b] and [KRV10], which allows the combined definition of concrete and abstract syntax.

Our experience shows that guidelines for the creation of DSLs tremendously improve their quality. They target and enable suitability, reuse, conciseness, and usability. In [FHR08], we discuss *metrics and potential guidelines*, that help to achieve high quality models and extend this into a *collection of design guidelines for DSLs* in [KKP+09].

Another important aspect is how to *define the semantics* of DSLs. Variability in syntax and semantics for DSLs in general and UML in particular has been discussed in [GR11]. For an extensive discussion on semantics we refer the reader to Section 2.6, Semantics of Modeling Languages.

Composition of DSLs

Modularity is a key concept in software development and the enabler for efficient reuse. We investigated the application of modularity to the development of DSLs in [GKR+07], [KRV08] and [Völ11]. Modularity has been successfully applied in various areas of the DSL development process, such as concrete and abstract syntax, context conditions, and symbol table structures and has been implemented in our language workbench MontiCore [HR17].

We can *compose independently developed languages* into *integrated families of DSLs*, which allows us to describe a system from various viewpoints using these different DSLs. The language family UML/P, defined in [Sch12], serves as an example of this technique. We can *reuse existing languages* by embedding them as sub-languages, e.g., Java's expression language can be used for various purposes within a modeling DSL. Consequently, we have integrated both, Java statements and expressions, into UML/P. We are further investigating the decomposition of generators and modular composition

of generated code. Another important aspect for composition is *inheritance and adaption of existing concepts*. As described in [KRV08, HRW18], we can inherit from existing languages and adapt certain language concepts. An often used example is to extend an action language by new forms of actions. These concrete techniques are summarized in the broader discussion on the so called *"global" integration of domain specific modeling languages and techniques in a conceptual model* [CBCR15], which is published in [CCF+15].

DSL Tooling

As previously mentioned, the usability of a language depends on the availability of powerful tooling. We have implemented the *MontiCore DSL workbench* as a realization of all the aforementioned concepts regarding DSLs. It is available as a stand alone tool as well as a collection of Eclipse plugins. It also creates stand alone tools as well as tailored Eclipse-based plugins for the defined DSLs [KRV07a]. We generate editors with syntax highlighting, syntactic and semantic content assist and auto completion, graphical outlines, error reporting, hyperlinks, etc., just from the DSL definition. More details about the MontiCore DSL workbench can be found in Section 4.1, in [GKR+06, KRV08, KRV10, HR17] as well as on the MontiCore Website[1].

Moreover, there is a strong need for evolution and management of models [LRSS10], especially for comfortable transformation languages. Therefore, [Wei12, HRW15, Höl18] present a tool that creates an infrastructure for transformations that are specifically dedicated to an underlying DSL. The generated *transformation language* is quite understandable for domain experts and comes with an engine dedicated to transform models of this DSL.

2.5 Software Language Engineering (SLE)

Identifying or engineering appropriate languages for the various activities in software and systems development is one of the most important issues in software engineering. Even programming languages are still subject to improvement. For many other activities, such as architectural design, behavioral modeling, and data structure specifications, we use the general purpose UML [Rum16, Rum17]. Nevertheless, UML and its tooling still are much less elaborate and hence subject to extensive syntactic, semantic, and methodical improvement.

In various domains, however, it is more appropriate to employ DSLs to enable non-software developers specifying properties, configuring their systems, etc. in terms of established domain concepts and corresponding language elements. DSLs have already achieved a significant degree of penetration in industry [HWR14]. With the upcoming age of digitalization, we thus expect DSLs to grow even stronger and therefore also involve increasing effort in their efficient engineering, integration and composition.

Design of a DSL is a complex task, because, on one hand, it needs to be precise enough for being processed by a computer and, on the other hand, comprehensible by humans. Monolithic design of a language can already benefit from methods for language engineering in the small including design guidelines and tooling. The MontiCore language workbench [HR17] is such a tool to assist development of languages. It provides, e.g., techniques for an integrated definition of concrete and abstract syntax of a language [KRV07b, Kra10], but is much more a framework for compositional language design [KRV10, HRW18].

[1] `http://www.monticore.de/`

Language Engineering in the Large

To efficiently engineer languages in the large, we need all the help that we can get. As software languages are software too, it is not surprising that the following techniques largely discussed in [CFJ+16] help:

- Elaborate tooling to assist language development.
- Reuse of tools, e.g., for parsing and for parameterizable pretty printing.
- Reuse of language components.
- Decomposition of the language to be designed in smaller components.
- Refinement and adaptation of existing languages.
- Automatic derivation of new languages from existing ones.

To improve understanding of language engineering, we have defined the terms *language* and *language components* in [CBCR15, BEK+18] and how to capitalize on this from a global perspective in [CCF+15, CKM+18]. Additionally, we discuss the possibilities and the challenges using *metamodels for language definition* [SRVK10], identifying, for instance, the need for metamodel merging and inference, as well as assistance for their evolution .

Language and Tool Composition

Divide and conquer is one of the core concepts for managing large and complex tasks. Language design therefore needs to be decomposed along several dimensions: First, we want to decompose the language in language components [BEK+18]. Some of these components, for example the basic language for full qualified names, constants, expressions, or imperative statements, should be designed in a reusable form.

In a second dimension, we want to decompose the tooling along the activities (front-end: model processing, context conditions, internal transformations, backend: printing) and decompose each of these activities along the individual language components. MontiCore 3 [HR17], e.g., is able to decompose the front-end language processing along the decomposition of the language itself [KRV10, Völ11, KRV08, HMSNRW16, MSN17]. MontiCore also assists modular decomposition of the backend code generation based on different targets and different sublanguages [RRRW15, BBC+18] (cf. Section 2.9).

Language Derivation

Language derivation is, to our believe, a promising technique to *develop new languages* for a specific purpose that are relying on existing basic languages [HHK+13, HHK+15a, HRW15, GLRR15, BDL+18, BJRW18]. Formally, a language derivation is a mapping D, that maps a base language B into another language $D(B)$. This mapping produces new languages, not models. To automatically derive such new languages $D(B)$ or, at least, assist such derivation with tools, the base language B itself has to be modeled explicitly, for instance as a metamodel or as a grammar together with its well-formedness rules in a reasonably explicit form. Thus, language derivation can be partially understood as model transformation on a metalanguage. We, so far, successfully conceived three language derivation techniques.

Transformation languages in concrete syntax Instead of using a fully generic transformation language that is applicable to a base language B, we automatically derive a transformation language *T(B)* that merges elements of the concrete syntax of B with generic - and thus reusable - elements for defining transformations on B models. The result is a comprehensible and easy applicable transformation language that modelers find familiar, because it systematically reuses the syntax of the base language B. Automatic derivation of such transformation languages using concrete syntax of the base language is described in [HRW15, Wei12, Höl18].

As the language derivation operator T is applicable to any language, we have successfully applied it to, e.g., class diagrams, object diagrams, MontiArc, Automata. The operator T not only derives the new languages *T(B)*, but the tool infrastructure behind T also generates the transformation engine necessary to execute transformations defined in *T(B)* (which finally transform models of the base language B).

Tagging languages A tagging model is used in the context of a base model M and adds additional information in form of tags to the elements defined in M. This, for example, can be used to add technology-specific information or advice on how code generation, model merging and other algorithmic transformations have to handle the tagged elements. Tags can, for example, instruct a persistence generator, whose data model classes are mapped into single transportable DAOs or add security restrictions to data objects. For activity diagrams, tags can describe, where to find the appropriate activity implementation, etc.

Tagging models share the idea of UML's stereotypes, but are not part of the base model. Instead the separate tagging model references the base model. This has the advantages (1) that the base model can be reused without technology specific pollution, (2) several different tag models are possible for the same base model in different technological spaces (e.g., iPhone, Android or Windows clients), and (3) a tag model can also be reused for different base models.

A tagging language is the language of the tagging models and thus is highly dependent on the base language that it tags (i.e., it must be aware of the modeling elements of the base language). [GLRR15] describes how to systematically derive tagging languages from a base language and how code for processing tagging models can be generated automatically.

This also rests on the concept of a tag definition language, which allows defining the possible form and values that tags may have, as well as which kind of model elements they can be applied to and therefore acts as type definition for tags.

Delta languages Another way of deriving new languages from existing languages is described in [HHK+15a] and [HHK+13], where a base language B is used to derive a delta language *Delta(B)*. The delta language *Delta(B)* enables to explicitly describe differences between a base model of B and the model variant (also of B). This helps to define system variability in a bottom-up fashion. A delta model describes which model elements are added, modified, or deleted on the base model. Thus delta approach is popular for the management of Variability and Software Product Lines (SPL) (see Section 2.10). Again the delta operator transforms a base language B into a language *Delta(B)* allowing to describe delta models. Each delta model can be applied individually and therefore n deltas amount to 2^n variants (modulo application dependencies and orders). Delta language techniques are *specifically suited for architectural languages*, such as MontiArc to add and modify components as well as channels, but also have been *applied to Simulink in an industrial context.*

2.6 Semantics of Modeling Languages

To deal with model analysis, synthesis, evolution, definition of views, and abstraction based on models, we need a *sound semantic foundation of the meaning of the models*. We also need a proper semantics when applying a given language to *new domains*, such as monitoring energy consumption or modeling flight safety rules for the European air traffic (see Section 3). We do this regularly with our language workbench MontiCore [KRV10, HR17].

The Meaning of Semantics and its Principles

Over the years we have developed a clear understanding of what the semantics of a model and a modeling language is. For example in [HR04] we discussed different forms of semantics and what they can be used for. We, in particular, distinguish between "meaning" that can be attached to any kind of modeling language and an often used narrow interpretation, that uses "semantic" synonymously to behavior of a program.

Each modeling language, whether it be *UML or a DSL deserves a semantics*, even if the language itself is for modeling structure, such as Class Diagrams or Architecture Description Languages. Furthermore, modeling languages are *not necessarily executable* and as their main *purpose is abstraction* from implementation details, they are usually not fully determined, but exhibit forms of *underspecification*. We discuss a very general framework for semantics definition in [HR04]. At the core, we use a denotational semantics, which is basically a mapping M from source language L (syntax) into a target language respectively a target domain S (semantic domain). In Figure 4 we see a combination of functions, where the first simplifies the syntax language by mapping redundant concepts to their simplest form (less concepts used, but usually more complex models).

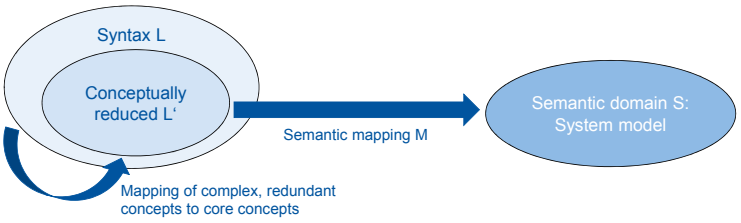

Figure 4 – Composed semantic mapping: (1) Syntax reduction, (2) translation to the semantic domain

While many attempts of defining semantics only give examples on how mapping M looks like, we advocate an explicit and precise definition of M to be able to analyze or compare the semantics of models. For example, refinement and evolution of models rely on such explicit denotational semantics.

System Model as Semantic Domain

To define a semantic domain we use a mathematical theory, that allows us to explicitly specify the desired properties of the target system, we are aiming at. We call the developed theory *System Model*. Its first version is explicitly defined in [RKB95] and [BHP+98] (including work from [GKR96], [KRB96] and [RK96]).

The *System Model* for the full UML, however, became a rather large mathematical theory, that captures object-oriented communication (method calls, dynamic lookup,

inheritance, object identity) as well as distributed systems at various levels as states and state machines. We therefore developed the full system model for the UML in [BCGR09b] and discuss the rationale for it in [BCGR09a]. See also [BCR07a] and [BCR07b] for more detailed versions and [CGR08] for an application on class diagrams. Figure 5 shows the hierarchy of the mathematical model.

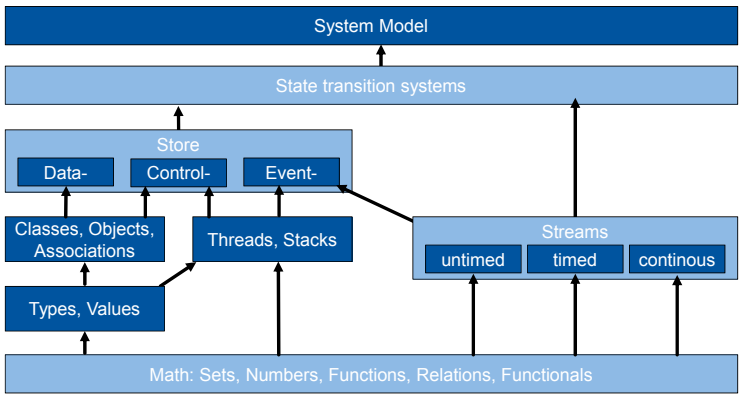

Figure 5 – Structure of the semantic domain: The system model

The system model and its variants are used for a variety of tool embeddings of the semantic domain. We provide a structured approach for system model-based modeling language semantics definitions in [GRR09]. [MRR10] explains the case for semantic model differencing as opposed to syntactic comparison. In [MRR11a] (based on [MRR11b]) we encoded a part of the semantics, big enough to handle differences of activity diagrams based on their semantics, and in [MRR11e] we compare class and object diagrams based on their semantics.

In [BR07] we have defined a much simpler mathematical model for distributed systems based on black-box behaviors of components, hierarchical decomposition, but also the sound mathematical theory of streams for refinement and composition. While this semantic model is useful for distributed real-time systems, such as Cloud, Internet of Things or CPS, it does not exhibit concepts of objects and classes.

We also discussed a *meta-modeling approach* [EFLR99]. As nothing is as mighty and comfortable as mathematical theories, one needs to carefully design the semantics in particular if a concept of the language does not have a direct representation in the semantics domain. Using a meta-model to describe the semantics is appealing, because the syntactic domain L is meta-modeled anyway, but also demanding, because both the semantic domain S and the mapping M need to be encoded using meta-modeling instead of mathematical concepts. We learned, that meta-modeling is limited, e.g., in its expressiveness as well as due to finiteness.

Semantics of UML and Object-Orientation

In the early days, when modeling technology was still in its infancy, it was of interest to precisely understand objects, classes, inheritance, their interactions and also how modeling technologies, like the upcoming UML, describe those. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called Sequence Diagram, and a complete description of object interactions, which obviously needs additional mechanisms, e.g., a sequential, parallel or iterative composition of Sequence Diagrams. Furthermore, [BGH+98] discusses the relationships between system, a view and a complete model in the context of the UML.

Abstraction, Underspecification and Executability

A modeling language is only a good language, if it allows to abstract from implementation details. *Abstraction* however often means that its models are not fully determining the original, but exhibit *underspecification*. Underspecification is regarded as freedom of the developer or even of the implementation to choose the best solution with respect to the given constraining specification. It is an intrinsic property of a good modeling language to allow underspecification.

As a consequence a semantic mapping of an (underspecified) model into a single running program cannot be correct or useful in order to capture the semantics adequate. To tackle *underspecification*, we use a set-based mapping. This means a single model is mapped to a set of possible implementations all of which fulfill the constraints given by the model. This approach has several advantages:

1. Each element in the semantics can be an *executable implementation*, we just do not know, which of them will be the final implementation.
2. Given two models, the *semantics of composition* is defined as intersection: these are exactly the systems that implement both models. This approach is based on "loose semantics", where an implementation is allowed to do everything that has not explicitly been forbidden by the specification.
3. A model is *consistent* exactly when it has a nonempty semantics.
4. *Refinement* of a model on the syntactic level maps to set inclusion on the semantics.

Using sets of executable systems in the semantic mapping combines the denotational approach with an operational approach that is perfectly suited for semantics for modeling languages.

Semantic Variation Points

In the standardization of the UML the contributors had some challenges to agree on the meaning of quite a few modeling concepts. To some extent this is due to political reasons (tool vendors try to push their already implemented solution), but to a large extent this is also due to the attempt of the UML to describe phenomena in various real world and application domains as well as software/technical domains. As it is a bad idea to capture different phenomena with the same syntactical concept, the UML standard introduces the *semantic variation point* without describing precisely what it means and how to describe it.

In [GR11], [CGR09] we have discussed the general requirements for a framework to describe semantic and syntactic variations of a modeling language. We also introduced a mechanism to describe variations (1) of the syntax, (2) of the semantic domain, and (3) of the semantic mapping using feature trees for class diagrams and for object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. Feature trees are a perfect concept to capture variation points and denotational semantics based on a system model allowing to explicitly describe the effect of the variant.

In [Rum17] (or its German version [Rum12]) we have embodied the semantics in a variety of *code and test case generation, refactoring and evolution techniques* to make UML semantics amenable to developers without exposing the formalism behind. Additionally, [LRSS10] discusses evolution and related issues in greater detail.

Streams and Automata as Semantic Foundation

It is also noteworthy that we have used the mathematical concept of *streams* (e.g., [BS01, RR11] also discussed in [RW18]) and various extensions including *automata* [Rum96] as semantic basis for the kind of systems, we have in focus: distributed, asynchronously communicating agents, which can be regarded as active objects. We illustrate our state machine formalism and its stream-based composition in Section 2.13.

2.7 Evolution & Transformation of Models

Models are central artifacts in model-driven software development (MDD). However, software *changes over time* and so do models. Many of the new requirements imposed by stakeholders, technology adaptations, or bug and performance improvements do not only affect the implementation, but also require an evolution, refinement or refactoring of the models describing the system. When models play a central role in the development process, it is therefore necessary to provide a well-founded, methodologically sound and tool-based assistance for evolving models according to changing needs. [CFJ+16] discusses several of the following.

Evolution

Agile methods, such as XP or Scrum, rely to a large extent on the ability to evolve the system due to changing requirements, architectural improvements and incremental functional extensions. While agile methods use code as their central artifacts, a model-driven method concentrates on modeling artifacts. In [Rum04] and [Rum12] we describe an agile, model-based method that relies on iterated and fully automatic generation of larger parts of the code as well as tests from models, which in turn enables us to apply evolutionary techniques directly on the various kinds of models, e.g., the UML. We argue that combining automatic and repeatable code generation with tool-assistance for model transformation allows to combine agile and model-based development concepts for a new and effective kind of development process.

An overview on current technologies for evolving models within a language and across languages is given in [LRSS10]. We refined this with a focus on evolving architecture descriptions for critical software-intensive systems [MMR10].

Refinement

Refinement is a specialized form of transformation of models that adds informational details, while all conclusions a developer could derive from the abstract model still hold. Such an addition may for example be the structural refinement of the state space of state machine, but also the reduction of underspecification expressed as alternative behaviors. Stepwise refinement is therefore an important development technique as it prevents unwanted surprises when abstract models are implemented.

In [PR94] we developed a precise understanding of automaton refinement that is especially useful for software development, as it uses a loose semantics approach, where no implicit assumptions are made that need to be invalidated in the refinement steps. In [KPR97] we applied this refinement concept to feature specifications.

Finally, we developed a powerful set of refinement rules for pipe-and-filter architectures in [PR99]. Its rules allow us to refactor the internal structure of an architecture, while retaining respectively refining the externally promised behavior. We speak of "glass box" refinement as opposed to "black box" refinement, where only the external visible behavior is taken to consideration, and "hierarchical decomposition",

where a black box behavior is decomposed into an (forthwith immutable) decomposed architecture.

As performing a refinement step is error-prone, we present a language independent and fully automated method to *repair failed model refinements* [KR18]. This is possible by identifying syntactic changes that does not lead to refined models.

Refactoring of models

Refactoring aims to improve the internal structure while preserving its observable behavior and became prominent with agile development. In [PR01] we traced back refactoring of programs to related techniques, e.g., known from math or theorem provers. In [PR03] we have discussed the existing refactoring techniques for specifications and models. We, e.g., found a number of well-defined refactoring techniques for state machines, logic formula, or data models that come from formal methods, but have not yet found their application in software development. In [Rum12] we therefore discuss refactoring techniques for various UML diagrams in detail. Additionally, libraries of reusable refactorings for class diagrams and MontiArc models were developed [Höl18].

If a model refactoring is actually a refinement, then dependent artifacts are not affected at all. However, it may be that a refactoring does have effect on related artifacts. In [MRR11a] we discuss a technique to identify semantic differences for UML's activity diagrams. It can be used to understand the effects of a refactoring resp. evolutionary change.

It is important to understand semantic differences between refactoring steps on data structures by exhibiting concrete data sets (object structures) as a witness of semantic differences. Thus, we provide a mapping of UML class diagrams to Alloy [MRR11c].

Understanding model differences

While syntactic differences of models are relatively easy to understand, it is an interesting question what the semantic differences between two given models are, where one evolved from the other, and what their clear semantics is. In [MRR10] we discussed the necessity for this and since then have defined a number of semantic-based approaches for this (see Section 2.6). We also applied *compatibility checking of evolved models* on Simulink, e.g., in [RSW+15], where we identified in which system context a component can be replaced safely by another version and still achieve the same functionality.

Delta transformations to describe software variability

Software product line engineering is most effective if planned already on the modeling level. For this purpose, we developed the delta approach for modeling. Each delta describes a coherent set of changes on a model. A set of deltas applicable to a base model thus describes a model variant (see also Variability in Section 2.10).

We successfully applied delta modeling for software architectures [HRRS11] and extended this into a hierarchical approach in [HRR+11]. Second, we discussed in [HRRS12], how to evolve a complete product line architecture, by merging deltas or extracting sub-deltas, etc., which allows us to keep a product line up to date and free of undesired waste. Third, based on the experience we gained from applying the delta approach to one particular language, we developed an approach to systematically derive delta languages from any modeling language in [HHK+13] and [HRW15].

Model transformation language development

As we do deal with transformations on models in various forms, we are very much interested in defining these transformations in an effective and easily understandable form. Today's approaches are focussing on the abstract syntax of a modeling language, which a typical developer should not be aware of at all. We heavily demand better transformation languages. Thus, in [Wei12, HRW15, Höl18] we present a technique that derives a transformation language from a given base language. Such a transformation language reuses larger parts of the concrete syntax of the base language and enriches it by patterns and control structures for transformations. We have successfully applied this engine on several UML sub-languages and DSLs [HRW15, HHRW15, AHRW17a, AHRW17b, Höl18].

2.8 Modeling Software Architecture

Distributed interactive systems have become more and more important in the last decades. It is becoming the standard case that a system is either logically or physically distributed. Typically such systems consist of subsystems and components like

- sensors, control units, and actuators in cyber-physical machines,
- high performance computing nodes and micro-services, and
- big data storage nodes.

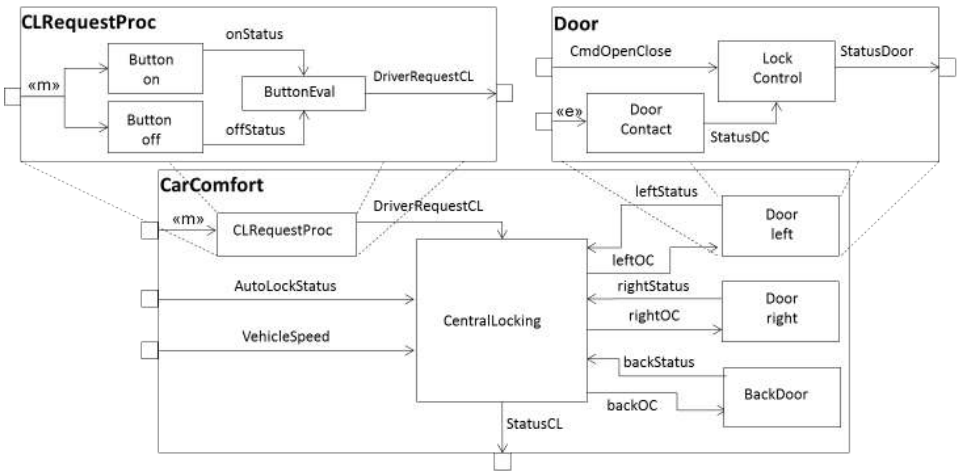

Figure 6 – Excerpt of a component and connector model of a car locking device

The logical or physical architecture of a hierarchically decomposed system can be modeled like the excerpt of a car locking device in Figure 6. The main paradigm for communication in distributed systems is asynchronous message passing between actors.

Messages can be

- values measured by sensors and discrete event signals,
- event signals, e.g., messages on a bus, streams of telephone or video data, method invocation, or

- complex data structures passed between software services.

Some challenges in the design and implementation of these systems are the development of an appropriate architectural decomposition of the system and fitting component interfaces suitable for property analysis, effective realization, and reuse of components under variability considerations. We have made a number of contributions to this field from more theoretical considerations up to a concrete tooling infrastructure called MontiArc.

In the following, we give an overview over the works of our group on the foundations of modeling software architecture, concepts of software architecture variability, and their evolution.

Foundations, MontiArc, and Code Generation

Our approach is formally sound and defined using streams [BS01], state machines [Rum96], components [BS01, Rum96, Rin14, Hab16], as well as expressive forms of composition and refinement.

A *theoretical foundation* of a model-based development in terms of an integrated, homogeneous, but modular construction kit for architectural models is described in [BR07]. Mathematical foundations are given for modeling of interfaces, building architectures through composition and decomposition, layering architectures as well as hierarchical decomposition, and implementation of components using state machines. Especially the refinement [PR99] of hierarchy, interfaces, and behavior is discussed as well as abstraction mechanisms for the integration of abstract viewpoints. The presented theory consists of a set of theorems and provides a basis for architectural modeling without sticking to a concrete syntax of a modeling language.

The *architecture description language MontiArc* (cf. Section 4.2) has been developed for modeling distributed interactive systems based on this theory. It comprises active components, their interfaces (ports), the communication infrastructure between the components, and a hierarchic decomposition. It is realized as a textual MontiCore language that features and editor, analyses, simulation, and code generation implemented in Java.

Software Architecture Variability

The *inherent variability software systems* have to be considered and modeled by appropriate means *during all phases* of the development but especially in the architectural design. In the following we present a brief overview of variability in the MontiArc language family. For a more detailed discussion on variability see Section 2.10 where much variability research was applied to and experimentally verified using MontiArc.

We explored a variability mechanism based on MontiArc that allows specifying component variants fully integrated at any level of the component hierarchy [HRR+11]. Here variation points may have hierarchical dependencies. Associated variants define how this variability can be realized in component configurations. As a general drawback of this approach, systems are restricted to the set of predefined variations and cannot be extended. This approach is not additive.

We thus explored delta-modeling as an *additive approach* to variability design. This allows for incremental introduction and development of software product lines, before a full variability model is established (reengineered). The main idea is to represent any system by a core system and a set of deltas that specifies modifications [HRRS11]. A delta-language is defined describing how to add, remove, or modify architectural

elements. The concrete realization of Δ-MontiArc using the DSL development framework MontiCore (see Section 4.1) is described in [HKR+11]. The developed language allows the modular modeling of variable software architectures and supports proactive, reactive as well as extractive product line development. As a next step, we explored in [HRRS12] how to evolve a complete delta-based product line, e.g., by merging or splitting deltas.

Runtime variability was discussed in a separate line of work: we have analyzed dynamic reconfiguration of software architectures [HKR+16] as a means for achieving.

Specification and Evolution

A methodological approach to *close the gap* between the *requirements architecture and the logical architecture of a distributed system* realized in a function net is described in [GHK+07, GHK+08a]. It supports the tracing of requirements to the logical software architecture by modeling the logical realization of a feature that is given in a requirement in a dedicated feature view. This allows us to break down complexity into manageable tasks and to reuse features and their modular realization in the next product generation. [GKPR08] extends this modeling approach to model variants of an architecture.

We have defined a precise verification technique that allows developers to decompose logical architectures into smaller pieces of functionality, e.g., individual features in [MRR13] and [Rin14], and to verify their consistency against a complete architecture in [MRR14]. Our hypothesis is that with this technique, developers will be able to decompose requirements into features and compose their implementation late in the development process. We have documented an evaluation our specification and verification techniques in an experience report [MRRW16]. These concepts are now successfully integrated into automotive development processes [DGH+19].

An overview and a detailed discussion on the challenges of co-evolution of architectural system descriptions and the system implementation is given in[MMR10]. Architectural descriptions of a system deal with multiple views of a system including both its functional and nonfunctional aspects. Especially, critical aspects of a system should be reflected in its architecture. The description must also be accurately and traceably linked to the software's implementation so that any change of the architecture is reflected directly in the implementation, and vice versa. We provide powerful analyses of software architecture behavior evolution in [BKRW19]. Otherwise, the architecture description will rapidly become obsolete as the software evolves to accommodate changes. One way for understanding evolution based on semantic differences is presented in [BKRW17b].

2.9 Compositionality & Modularity of Models

Divide and conquer as well as abstraction are the most fundamental strategies to manage *complexity*. Complex (software) systems become manageable when divided into modules (horizontally, vertically and/or hierarchically). Modules encapsulate internal details and give us an abstract interface for their usage. Composing these modules as "black boxes" allows us to construct complex systems.

MBSE uses models to *reduce complexity of the system under development*. Nevertheless, it has reached a point, where the models themselves are becoming rather complex. This clearly rises the need for suitable mechanisms for *modularity within and between models*. A modular approach for MBSE cannot only help us mastering

complexity, but is also a key enabler for model-based engineering of heterogeneous software systems as discussed in [HKR+09]. In [BR07] we have described such a set of compositional modeling concepts, perfectly suited for modular development of interacting systems.

A *compositional approach* has to take into account several levels of the entire MBSE process, starting with the respective modeling language in use, the models themselves and, eventually, any generated software components. We have examined various aspects of model composition in [HKR+07], describing a mathematical view on what model composition should be. It defines the mechanisms of encapsulation, and referencing through externally visible interfaces.

In [KRV10] and [KRV08], we examine modularity and composition for the definition of DSLs. As DSLs are becoming more and more popular, the efficient reuse of DSL fragments (i.e., language components) is vital to achieve an efficient development process. But aside from the language definition, the accompanying infrastructure needs to be modular as well (as described in [KRV07b]). Infrastructure such as validation or editor functionality should be reusable if parts of the underlying DSL are reused, e.g., as part of another language. [Völ11] provides the underlying technology for compositional language development, which we, e.g., applied to Robotics control [RRRW15] and nowadays investigate under the umbrella term of Software Language Engineering (see Section 2.5). Based on the experiences in language design, we also have developed a set of guidelines to estimate the quality of a DSL in [KKP+09]. We have summarized our approach to composition and the challenges that need to be solved in [CBCR15], which describes a conceptual model of the compositional, so called "globalized" use of domain specific languages, which we published together with related topics in [CCF+15].

As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional, e.g., technical information for model elements in extra documents and thus facilitates reuse of the original model in different contexts with individual tag sets, but also of tags on different models. It furthermore allows to type the tags.

2.10 Variability & Software Product Lines (SPL)

Many products exist in various variants, e.g., cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. In most cases, product variants are created through software variability. On a software level, variants are managed in a SPL that captures product commonalities as well as differences. SPLs have many benefits: they decrease development time of new product variants, decrease time to market, lead to better software quality, improve reuse, and reduce bug fix time [GHK+08a, GRJA12, HKM+13, HHK+15a, BEK+19].

Feature diagrams are a popular mechanism to 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]. Feature diagrams suffer from the need to first decompose the problem space and understand possible features in order to build the feature diagram before being able to apply it. In [GHK+08a] and [GKPR08] we also speak of a 150% model. This normally enforces a product line definition phase in which the requirements and features need to be collected which creates additional costs. Among others we discuss decreasing these costs 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 base variant. A set of

commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [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]. In [HHK+15a] we have generalized this approach to investigate, how to synthesize a delta modeling language based on a given modeling language. Thus deltas can generally be applied to almost any language.

On a related line of research, we also have studied variability of modeling languages. For this purpose we defined a systematic way to define variants of modeling languages [CGR09]. We applied this research, e.g., in the form of semantic language refinement on state charts in [GR11]. In [FPR02] we discussed how to apply annotation to the UML to describe product variation points. Current work continues this line of research to support the definition and development of 150% language families for textual and generative modeling languages [BEK+18, BEK+19].

2.11 Cyber-Physical Systems (CPS)

CPS are software controlled, collaborating physical machines [Lee08, KRS12]. This new term arises mainly due to the increased ability of computers to sense their environment and to interact with their contexts in various ways. As consequence, CPS are usually designed as distributed networks of interacting nodes and physical devices (machines) that carry out certain tasks. Often some of these devices are mobile (robots or autonomous cars, but also smart phones, airplanes and drones) and interaction with humans is essential. CPS are therefore complex in several dimensions: they embody characteristics of physical, networked, computational-intensive, and of human-interactive systems. Furthermore, they typically cannot be developed as monolithic systems, but need to be developed as open, composable, evolving, and scalable architectures (see also Section 2.8).

Nowadays, CPS are found in many domains, including aerospace, automotive, energy, healthcare, manufacturing, and robotics. Many distributed CPS use a virtual communication network mapped to the internet or telecommunication infrastructure.

The *complexity and heterogeneity* of CPS introduces a wide conceptual gap between problem and solution domains. Model-driven engineering of such systems can decrease this gap by using models as abstractions and thus facilitate a more efficient development of robust CPS [RW18].

CPS Application Domains

For the aviation domain, we have developed a modeling language [ZPK+11] that allows to specify flight conditions including trajectories, status of the airplanes and their devices, weather conditions, and pilot capabilities. This modeling language allows EuroControl to operationalize correct flight behavior as well as specify and detect *interesting events*.

As long term interest, we intensively do research on how to improve the engineering for distributed automotive systems as well. For example [HRR12, KRRvW17], outline our proposal for an architecture centric development approach.

Automotive is a highly innovative CPS subdomain. We discuss in [GRJA12] what an OEMs needs to understand about costs arising from requirements complexities and from cross-platform dependencies in their automotive development projects. Transforming a set of individual projects with similar requirements and technology

into a product line for a central part of an automotive system is discussed in [HRRW12]. Another important aspect are current and future processes and tools for development of *autonomous driving cars*. We discuss this in [BR12a] based on our experiences in building such a car and using sophisticated simulation techniques for the context of autonomous robots (cars). Moreover, fully automatic simulation of the cyber-physical contexts of cars and fully automatic checking of the robots behavior leads to an highly efficient development process with high quality results [BBR07]. Optimized code-generators [KRSvW18] and domain specific code generation [AHRW17b] are key for CPS. Moreover, we have extended our work from individual CPS to *product lines of CPS* [RSW+15, KRR+16, RRS+16].

Robotics is another highly innovative CPS subdomain. It is characterized by an inherent heterogeneity of involved domains, platforms, and increasing set of challenges. Engineering of robotics applications requires composition and interaction of complex, distributed systems as well. We developed a component and connector architecture description language suitable for the specific challenges in robotics [RRW13, Wor16] as well as in [RRW14]. This language serves from requirements modeling [RRW12] to the complete development of CPS software [RRSW17].

Smart and energy efficient buildings embody large amounts of IT technology. There is a multitude of networked systems and sensors to continuously control the building's *behavior*. We have built the Energy Navigator [KPR12, FPPR12] to be able to model the specifications of such buildings in order to control the measured actual data against the desired specification, e.g to save energy. In [KLPR12] we discuss how such a specification approach improves development quality in the energy subdomain of CPS.

2.12 Model-Driven Systems Engineering (MDSysE)

Systems engineering is the interdisciplinary engineering and management that focuses on the design and management of complex CPS over their life cycles. Systems engineering can and should be applied to a great variety of fields from automotive, to avionics and robotics.

We have a long tradition on contributing to systems engineering in automotive [FNDR98, GHK+08b], which recently culminated in developing a new comprehensive model-driven development process for automotive software function testing with the BMW Group [KMS+18, DGH+19]. In this, we leverage SysML to enable the vertical flow down from requirements to implementations that was well-received by the software developers.

Moreover, we recently started intensifying our research efforts towards a model-driven systems engineering that leverages methods and concepts from software engineering to make the systematic engineering of CPS more efficient. To this end, we conducted a systematic mapping study on modeling for Industry 4.0 that uncovered a gap between the communities, concepts, and modeling techniques of automation engineering and software engineering [WCB17]. To facilitate modeling 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].

2.13 State-Based Modeling with Automata

Today, we see that many informatics theories are based on state machines in various forms including Petri Nets or temporal logics. Software engineering is particularly

interested in using state machines for modeling systems. Nonetheless, we believe that a sound and precise integration of the digital theory (automata) of informatics with control theory (calculus) used by almost all other engineering and science disciplines is one of the most interesting challenges that we experience at the moment. Cyber-physical systems (CPS) (see Section 2.11) urgently require such an integrated theory.

Our contributions to state-based modeling can be split into three parts:

1. Understanding how to model object-oriented and distributed software using state machines resp. Statecharts;
2. Understanding refinement and composition on state machines; and
3. Applying state machines for modeling of systems.

State Machines as Semantics for Object-Oriented Distributed Software

A practically usable language for state-based modeling must be different from the pure theory because a concrete modeling notation, for example, allows us to denote finitely many (typically very few) states only, while the theory normally has an infinite state space.

In early publications, such as [GKR96], we have discussed how a system model can describe object-oriented systems. Built on this experience, a complete semantic model has been created for object-oriented systems in [BCR07b]. Objects, inheritance, states, method calls, stack, distribution, time as well as synchronous and asynchronous communication are completely defined and encoded into state machines. The theory is, therefore, suitable as semantic model for any kind of discrete systems. Hence, [BCGR09b] describes a condensed version of this system model and [BCGR09a] discusses design decisions, how to use the system model for denotational semantics – and taming the complexity of the system model.

Refinement and Refactoring of Statemachines

Starting with [PR94], we investigated how to use state machines to describe the abstract behavior of superclasses and refine it in subclasses. While the description in [PR94] was rather informal, we have formalized the refinement relation in [RK96] by mapping a state machine to a set of possible component behaviors based on the streams of the Focus (see Section 2.8) theory. In [Rum96], constructive transformation rules for refining automata behavior are provided and proven correct. This theory is applied to features in [KPR97], where a feature is a sub-automaton that adapts the original behavior in a refining form, precisely clarifying where feature interaction is allowed or harmful.

It became apparent that a state machine either serves as an implementation, where the described behavior is partial and can only be extended but not adapted, or that a state machine describes a specification, where the behavior is constrained to a possible, underspecified set of reactions, promised to the external users of a state machine. Here, refinement always means reduction of underspecification, telling more behavioral details to the external user. This is constructively achieved, e.g., by removing transitions that have alternatives or adding new behavior (transitions), if previously no transition was given at all.

Specification languages are particularly strong if only explicitly given statements and no implicit additional assumptions hold (such as: implicit ignoring of messages, if they cannot be processed by a transition) as detailed in [Rum96, Rum16]. The

concept of chaos completion should be used to define semantics of incomplete state machines. This is much better suited for behavioral refinements than the concept of ignoring messages or error handling in cases where no explicit transition is given. The main disadvantage of "implicit ignoring" is that you never know whether the specifier intended this as desired behavior or just did not care – which is a big difference when aiming to refine the specifier's model.

Our State Machine Formalism: I/O^ω Automata

[Rum96] describes an I/O^ω-automaton as $(S, M_{in}, M_{out}, \delta, I)$ consisting of:

- states S,
- input messages M_{in},
- output messages M_{out},
- transition relation $\delta \subseteq S \times M_{in} \times S \times M_{out}^\omega$,
- initial states I,

where $M_{out}^\omega = M_{out}^* \cup M_{out}^\infty$ is the set of all finite and infinite words over M_{out}.

The transition relation δ is non-deterministic and incomplete. Each transition has one single input message from M_{in} but an arbitrary long sequence of output messages from M_{out}. Nondeterminism is handled as underspecification allowing the implementation (or the developer) to choose. Incompleteness is also understood as underspecification allowing arbitrary (chaotic) behavior, assuming that a later implementation or code generator will choose a meaningful implementation, but a specifier does not have to decide upfront. Fairness of choice for transitions is not assumed (but possible), as it is counterproductive to refinement by deciding on one alternative during the implementation process.

Most interestingly, describing transitions in δ with input and corresponding output leads to a much more abstract form of state machines, which can actually be used in the modeling process. First there are no (explicit) intermediate states necessary that would distribute a sequence of output messages in individual transitions (which is the case in classic I/O-automata [LS89], where a transition has exactly one input or output message). Second our I/O^ω automata preserve the causal relation between input and output on the transitions (whereas I/O automata distribute this over many transitions). We believe I/O^ω automata are therefore suited as a human modeling language and are thus used in a syntactically enriched, comfortable form as Statecharts in [Rum16] and [Rum17].

Composition of State Machines

One state machine describes one component. In a distributed system, many state machines are necessary to describe collaborating components. The overall behavior of the component collaboration must then be derivable from the knowledge about the form of composition (architecture describing communication channels) and the specified behavior (state machines) of the components. [GR95] describes how timed state machines are composed.

This technique is embedded in the composition and behavioral specifications concepts of Focus using streams and state machines in a compact overview article [BR07]. Most important, refinement of a component behavior by definition leads to

a refinement of the composed system. This is a very important property, which is unfortunately not present in many other approaches, where system integration is a nightmare when components evolve.

Unfortunately, the untimed, event driven version of state machines that is very well suited for refinement and abstract specification has no composition in general. Further investigation is necessary.

Usage of Automata-based Specification

All our knowledge about state machines is being embedded in the model-based development method for the UML in [Rum16] and [Rum17]. Furthermore, we applied it to robotics (cf. Section 3.1) with the MontiArcAutomaton infrastructure (see Section 4.2), a modeling language combining state machines and an architectural description language in [THR+13] as well as in building management systems in [FLP+11].

2.14 Model-Based Assistance and Information Services (MBAIS)

The aim of information services and systems is to turn data into useful information. 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.

Figure 7 shows the main components of such an assistive system according to the architectural patterns for model centered architecture solutions [MMR+17] . Information about human behavior is collected via sensors or smart devices and processed in an observation engine, which stores the data and models. The behavior engine compares and connects the current behavior step with already existing models of behavior. The support engine decides which next step should be provided as behavior support for the assisted person. This support can be provided as step-by-step information multi-modal on different kinds of devices.

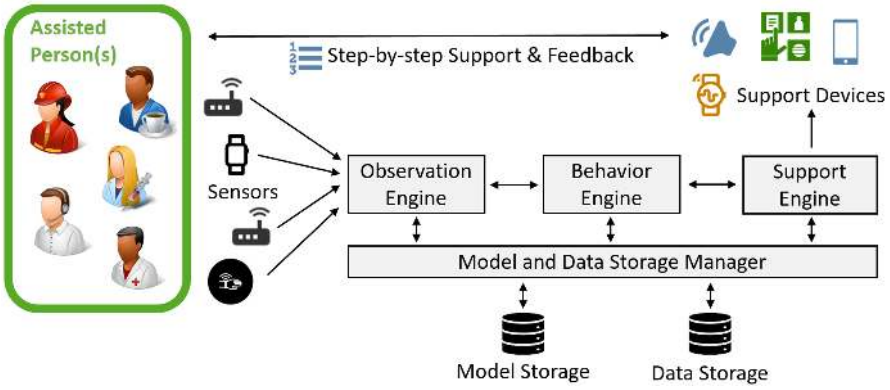

Figure 7 – Main concepts of systems for human assistance

For both, information and assistive services and systems, the application of agile, model-based and generative methods (see Section 2.1 and Section 2.2) fastens the development process, enables a quick response to requirements changes in a user-centered engineering process, and ensures consistency-by-design.

Our current work on assistive systems is based on former work within the Human Behavior Monitoring and Support (HBMS) project [MGSM13, MSS+18] in which a domain specific language [MM13] and domain specific modeling method for assistive systems [MM15] were developed.

Current work extends these approaches: to enable useful support, it is important to know more than just the behavior of a person. Context-aware systems need detailed information about the task context (including temporal information), the personal and social context, the environmental context, as well as the spatial context. We have investigated the modeling of these contexts primary for the active assisted living and smart home domain [MS17]. Recent research discusses the context model for user-centered privacy-driven systems in the IoT domain including special aspects for the use in combination with process mining systems [MKM+19].

The mark-up of online manuals for non-smart devices [SM18] as well as websites [SM19] is one further step to provide human-centered assistance. Using these approaches reduces system set-up time and improves flexibility for changes by automatically integrating device and application functionality into supporting systems.

Due to the General Data Protection Regulation (GDPR) organizations are obliged to consider privacy throughout the complete development process. Our work suggests solutions for privacy-aware environments for cloud services [ELR+17] as well as privacy preserving information systems demonstrated on an IoT manufacturing use case [MKM+19].

3 Applying Model-based Engineering in a Particular Domain

The results discussed above are transferred into practical applications in various areas including Robotics, Automotive, Energy Management, Cloud Computing, Enterprise Information Systems and the Internet of Things.

3.1 Modeling Robotics Architectures and Tasks

Robotics can be considered a special field of CPS that is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. Engineering 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.

Our research in model-driven software engineering for robotics on one hand focuses on software architectures (cf. Section 2.8) to structure reusable units of behavior. On the other hand, we concentrate on DSLs (cf. Section 2.4) for robotic product assembly tasks in industrial contexts as well as planned and unplanned logistic tasks. We apply this to indoor robots interacting with humans as well as to industrial robots and to autonomous cars.

Modeling Robotic Application Architectures and Behavior

Describing both, a robot's software architecture and its behavior in integrated models, yields many advantages to cope with complexity: the models are platform independent, can be decomposed to be developed independently by experts of the respective fields, are highly reusable and may be subjected to formal analysis.

In [RRW12] we have introduced the architecture and behavior modeling language and framework MontiArcAutomaton which provides an integrated, platform independent structure and behavior modeling language family with an extensible code generation framework. MontiArcAutomaton's central concept is encapsulation and decomposition known from component & connector architecture description languages (ADLs). This concept applied to the modeling language, the code generation process and the target runtime to bridge the gap of platform specific and independent implementations along well designed interfaces. This facilitates the reuse of robot applications and makes their development more efficient.

MontiArcAutomaton extends the ADL MontiArc (see Section 4.2) and integrates various component behavior modeling languages implemented using MontiCore as well as code generation for the ROS robot operating system [QGC+09]. The integration of automata and tables to model component behavior are described in [RRW13]. The integration capabilities of MontiArc have been extended and generalized in [RRRW15, AHRW17b]. For interested readers, the MontiArcAutomaton website[2] provides further information on the MontiArcAutomaton framework.

Capability-Based Robotics Architectures

Although the costs for robotics hardware and software decrease, deploying a successful robotics application still requires tremendous effort. While various challenging issues for service robotics have been solved to a degree where their deployment is possible, integration of the corresponding hardware components and software components requires intensive collaboration of domain experts, robotics experts, and software experts. Model-driven software engineering can facilitate development and integration of robotics software components while retaining a proper separation between domain expert concerns and software expert concerns.

To foster the separation of concerns in engineering service robotics applications, we conceived the iserveU modeling framework to describe domains, actors, goals, and tasks of service robotics applications [ABH+16]. From these declarative modeling languages [ABH+17], we generate parts of a MontiArcAutomaton architecture including component structure and behavior. In this architecture, goals and tasks are translated into models of the planning domain description language (PDDL) [MGH+98] and solved online using the Metric-FF [HN01] planner [ABK+17]. Through our work, domain experts can focus on describing the domain and its properties and robotics experts can focus on implementing actors with their properties and actions.

Modeling Assembly Robotics Tasks

The importance of flexible automated manufacturing grows continuously as products become increasingly individualized. Flexible assembly processes with compliant robot arms are still hard to be developed due to many uncertainties caused – among others – by object tolerances, position uncertainties and tolerances from external and internal sensors. Thus, only domain experts are able to program such compliant robot arms. The reusability of these programs depends on each individual expert and tools allowing to reuse and the compose models at different levels of detail are missing.

In cooperation with the DLR Institute on Robotics and Mechatronics we have introduced the LightRocks (Light Weight Robot Coding for Skills) framework in [THR+13, BRS+15] which allows robotics experts and laymen to model robotic assembly tasks on different levels of abstraction, namely: assembly tasks, skills, and elemental actions.

[2] http://monticore.de/robotics/montiarcautomaton/

Robotics experts provide a domain model of the assembly environment and elemental actions which reference this model. Factory floor workers combine these to skills and task to implement assembly processes provided by experts. This allows a separation of concerns, increases reuse and enables flexible production.

3.2 Model-based Autonomic Driving & Intelligent Driver Assistance

Development of software for automotive systems has become increasingly complex in the past years. Sophisticated driver assistance, infotainment and Car2X-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. While we are carrying out in numerous projects in the Automotive domain, here we concentrate on three aspects: Autonomic driving, modeling of functional and logical architectures and on variability. To understand all these features in [GRJA12] we describe a requirements management that connects with features in all phases of the development process helps to handle complex development tasks and thus stabilizes the development of automotive systems.

Modeling logical architecture: function nets

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07] and [GHK+08a]. Here, feature views modeled as a function net are used to implement the mapping between feature-related requirements and the complete logical architecture of a car.

In a more elaborate version, we have helped a larger car manufacturer to design their company specific method, *SMaRDT*, that injects model-based software development for the logical architecture of a car and connects it with the requirements and the technical implementation. Furthermore, we have added automatic testing techniques to ensure model quality from the beginning in [DGH+19] and [KMS+18].

Variability of car software

Automotive functions that may be derived from a feature view are often developed in Matlab/Simulink. As variability needs also to be handled in development artifacts, we extended Matlab/Simulink with Delta-Modeling techniques (see also Section 2.10). A core Simulink model represents the base variant that is transformed to another variant by applying deltas to it. A delta contains modifications that add, remove or modify existing model elements. This way, features of an automotive system may be developed modularly without mixing up variability and functionality in development artifacts [HKM+13]. New delta models that derive new variants may be added bottom-up without the need for a fully elaborated feature model.

In practice, product lines often origin from a single variant that is copied and altered to derive a new variant. In [HRRW12], we provide means to extract a well defined Software Product Line from a set of copy and paste variants. This way, further variant development is alleviated, as new variants directly reuse common elements of the product line.

Ways to identify potential variants of components for potential product lines are to use similarity analysis on interfaces [KRR+16], or to execute tests to identify similar behavior [RRS+16]. And a third approach is described in [RSW+15] that uses logical and model checking techniques to identify commonalities and differences of

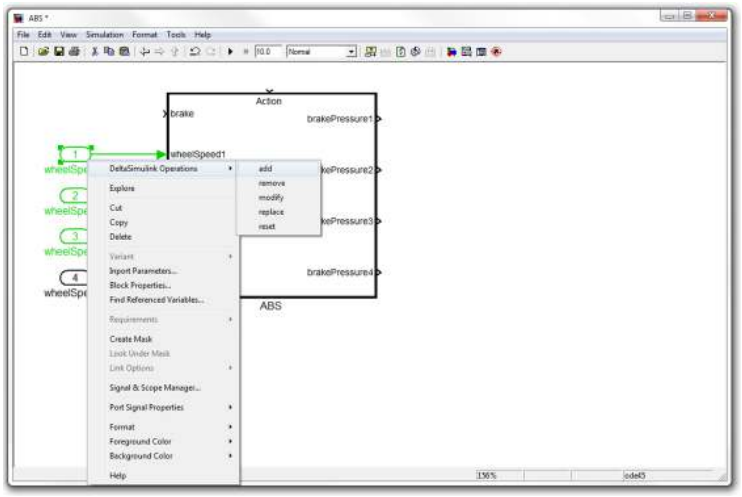

Figure 8 – Screenshot of the Delta-Simulink tool

two Simulink models describing the same control device in different variants. All these techniques allow us to understand incompatibilities or identify the portion of compatibility of two components respectively their models.

Autonomous driving

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b]), we have developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation (not only visualization) of the car within its surrounding: the city, pedestrians and especially other cars [BBR07]. Beside the simulation of a complete autonomic car with its sensors and actors, the simulation environment may also be used to test dedicated subsystems without any real hardware involved. By producing sensor input from the simulation and comparison of expected and actual behavior, subsystems may be automatically validated and thus developed in an agile way.

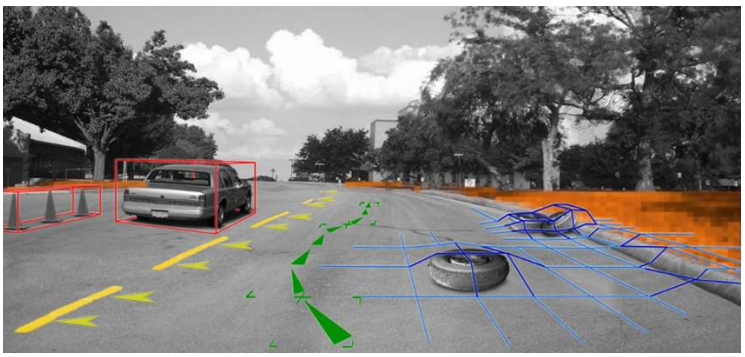

Figure 9 – How an autonomous car sees the world (taken from the Carolo project)

Driver Intelligence

From the viewpoint of software engineering, *intelligent driver assistance* and, in particular, *autonomic driving* is an interesting and demanding challenge because it includes the development of complex software embedded within a distributed, life-critical system (car) and the connection of heterogeneous, autonomic mobile devices (other cars, infrastructure, etc.) in one big distributed system.

We are involved in a number of projects with major European car manufacturers in which we transfer modern software development techniques to the car domain. This transfer is necessary as, with its increasing complexity, software becomes a demanding driver of the overall systems development process and not just an add-on.

In the *Carolo project*, we built Caroline [BBB+08] - a completely autonomous car - and participated in the Darpa Urban Challenge, where our car was driving autonomously in an urban area for hours. We successfully achieved the best place as newcomers (and best non-Americans). This resulted from a number of facts, including the rigorous application of *agile development methods*, such as XP and Scrum and a simulation for driving scenarios. In [BR12b] we describe the process driven by story cards as a form of use cases, a continuously integrated and running software up to a rigorous test, and simulation infrastructure, called Hesperia.

Figure 10 – Caroline arriving in the finish area after a successful race

In particular, we have developed a *rigorous test infrastructure* for intelligent, sensor-based functions through *fully-automatic simulation* (not only visualization!) of the car within its surrounding: the city, pedestrians and especially other cars [BBR07]. Our simulator is capable of running automatic back-to-back tests on the complete software system with no real hardware involved by producing sensoric input from the simulation and acting according to the steering output of the autonomic driving software. Every night and, when necessary for every version change, the tests are automatically executed.

This technique allows us a *dramatic speedup in development and evolution* of autonomous car functionality, and thus, enables us to develop software in an agile way [BR12a]. We have successfully shown that agile development of high-quality software is possible and very effective in the automotive domain. However, it remains a challenge to combine this innovative, modern way of agile, iterative systems develop-

ment with the current development standards, such as ISO 26262, in order to allow the OEMs to benefit both from efficiency and quality on one hand and legal issues on the other hand. As tooling infrastructure, we mainly used an IDE such as Eclipse and in particular the SSElab storage, versioning and management services [HKR12]. Without those, agile development would not have been possible.

In further projects, we have *evaluated and designed OEM specific architectures and processes*, on individual assistance functions and on the complete architecture. [BBH+13] contains metrics, e.g., to understand the cross-linkage complexity for software and functions. In [MMR10] we give an overview of the state-of-the-art in development and evolution on a more general level by considering *any kind of critical system* that relies on architectural descriptions.

In recent years, we also investigate in the *next steps of autonomy*, namely *cooperatively interacting autonomous vehicles*, allowing e.g., convoys with almost no distance to drive very energy efficient [FIK+18], as well as virtualization of development of safety algorithms, e.g., for the EuroNCAP and US NCAP scenarios discussed [BBH+15].

3.3 Models in Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Today housing, offices, shops and other buildings are responsible for 40 % of the overall energy consumption and 36% of the EU CO2 emissions. The EU 2020 Climate & Energy package sets three key objectives: (1) 20% reduction in EU greenhouse gas emissions, (2) Raising the share of EU energy consumption produced from renewable resources to 20% and (3) 20% improvement in the EU's energy efficiency compared to 1990.

Thus, the management of energy in buildings as well as in neighborhoods becomes equally important to efficiently use energy. Improvements in this field can be found at multiple scales: Smart Grids, Demand-Response Systems, Energy Efficient Neighborhoods, Energy Efficient Buildings, User awareness, Micro- and Mini Renewable Energy Sources, to name a few. While there has been a lot of research on increasing the efficiency of single devices and also of single buildings, there is a huge need for ICT based approaches within this field to integrate and combine the heterogeneous approaches. By such an integrated solution the efficiency can be raised even more.

Within several research projects we developed methodologies and solutions for integrating heterogeneous systems at different scales. Starting with single buildings we developed in collaboration with the Synavision GmbH and the Technical University Braunschweig the ICT tool Energy Navigator (see Figure 11).

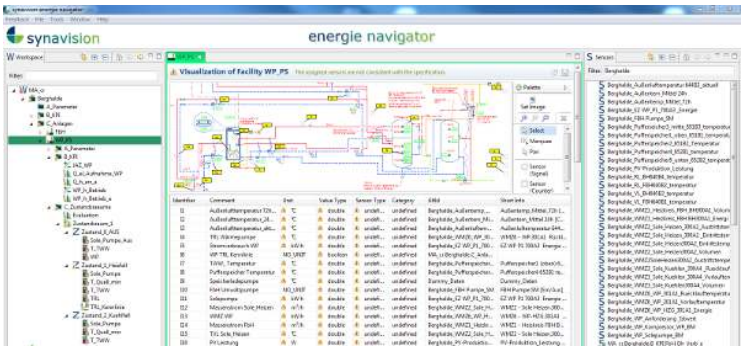

Figure 11 – Screenshot of the Energy Navigator software

During the design phase, the Energy Navigator's *Active Functional Specification (AFS)* [FPPR12, KPR12] is used for technical specification of building services already. Resulting from a lack of process integration the AFS can close the loop between modeling the structure and behavior of the building and its facilities, measuring operational data from sensors, matching model and operational data during analysis and reporting of the results (see Figure 12). The results can be reused to adapt the model or to find faults in the implementation.

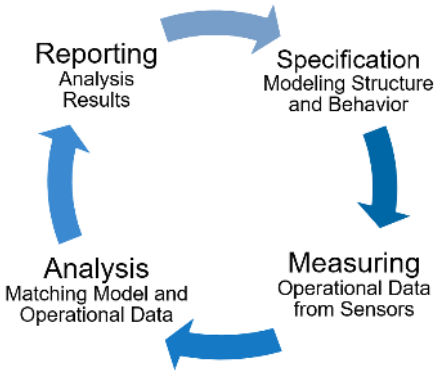

Figure 12 – The Energy Navigators' process loop

Within the Energy Navigator a DSL is used to enable the domain expert to express his specific domain knowledge via first class language concepts. These concepts include Rules, Functions, Characteristics, Metrics, Time Routines and States. Proposed by the DIN EN ISO 16484 a state based approach should be used to describe the functional behavior of facilities. We adapted the well known concept of state machines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Moving up the scale we investigated several existing approaches for energy efficient neighborhoods that aim at moving from a local, building specific optimum to a more global optimum. By efficiently using results of simulation and optimization calculated optimal set points for local consumption and generation can be utilized. Therefore information from several heterogeneous data sources, such as single sensor data, structural data, data on installed devices, geospatial data or weather data is needed. Based on existing approaches we developed a *Neighborhood Information Model* that follows a metamodel-based approached and utilized code generation techniques to automatically generate adapters between heterogeneous data models. Following this approach we are able to fully integrate the data sources on an abstract level and are still extensible at runtime.

Also Demand Response Systems are used to distribute energy more equally over time and enable a consumption during peak loads. We developed a secure high performance storage that is able to capture sensor data and DR signals.

3.4 Model-based Cloud Computing Applications

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. The development, integration, evolution,

operation and migration of web-based application and service architectures poses ever more and ever larger challenges to Software Engineering. In [KRR14] we discuss the paradigm of Cloud Computing that is arising out of a convergence of existing and new technologies in detail. 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.

Cloud-based systems pose a multitude of different challenges. The demand for seamless scalability with system load leads to highly distributed infrastructures and software architectures that can grow and shrink at runtime. The lack of standards, complemented by the variety of proprietary infrastructure and service providers, leads to a high degree of technological heterogeneity. High availability and interconnectivity with a multitude of clients leads to complex evolution and maintenance processes. These challenges come coupled with distinct requirements posed by the individual application domain. Application classes like Internet of Things as described in [HHK+14, HHK+15b], CPS described in [KRS12], 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. For example in [HHK+14] and [HHK+15b] we discuss how to handle privacy in the cloud in a trusted environment (see Figure 13).

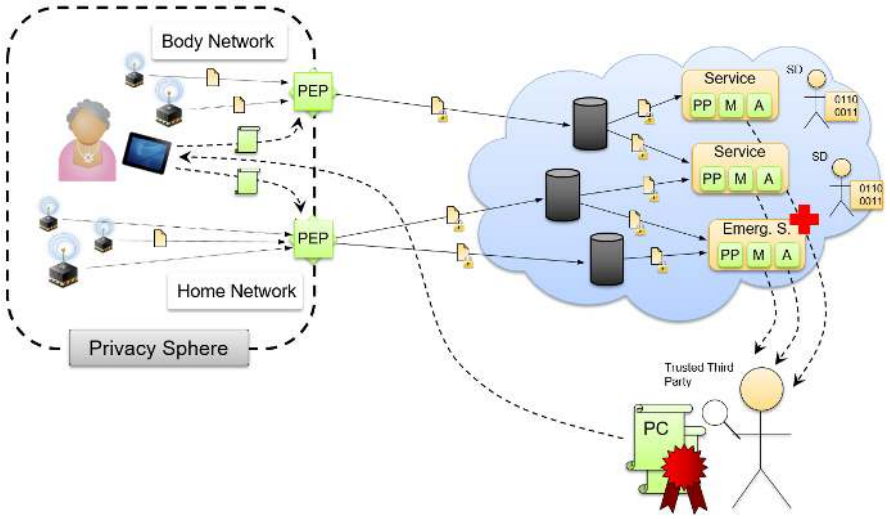

Figure 13 – Example for a trusted cloud environment

In our research [NPR13] we tackle these challenges by perusing a model-based, generative approach. The core of this approach are several 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. UML/P models, and class diagrams in particular, describe several other aspects of the system, such as its domain and data models, its interfaces and interactions, and its monitoring and scaling capabilities. Among other tools, code generators most prominently take these models as input and generate application-specific frameworks that implement big parts of the system's technical aspects and provide technology-agnostic, ease-to-use interfaces for the cloud-based

application's actual business logic.

We have applied these technologies to various cloud systems, cars, buildings, smart phones and smart pads and various other kinds of sensors. We built a rather successful and technologically sound framework for web-based software portals [HKR12] that we offer under sselab.de for general use. Another set of cloud systems helps to deal with energy management and is described in [FPPR12, KPR12]. It continuously monitors building operation systems to derive operational data and compare these to the building specification. We use cloud technologies to maintain data, dynamically execute calculations and host management services enabling reduction of building energy costs. Furthermore, we investigate the architecture of Cloud Services for the digital me in a privacy-aware environment [ELR+17]. We support developers with a model-driven and generative methodology supporting reuse of existing services, automated conversion between different data models, integration of ecosystems facilitating service composition, user data access control, and user data management.

We apply cloud technology also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now be developed easily.

3.5 Models in Enterprise Information System Development

Enterprise Information Systems (EIS) provide information to different user groups as main system goal. To be more precise, it is important to be able to create, read, update and delete data (CRUD). We use model-based methods to support these functionalities. Different types of models are used in the context of EIS: structural models to describe data structures, dynamic models to describe business processes, functional models to describe software functions or graphical user interface (GUI) models to describe graphical user interfaces. Usually, these models are developed using specific DSLs (cf. Section 2.4).

Using our experiences in the model-based generation of code with MontiCore [KRV10, HR17] (cf. Section 4.1), we developed several generators for data-centric applications. Figure 14 shows the main concepts for these approaches: Models from different DSLs are used as input for the generator in combination with predefined templates. As an output large parts of the information system are produced: the databases, data communication and infrastructure as well as the GUIs for different users and roles. Missing parts have to be added as handwritten code, such as application logic.

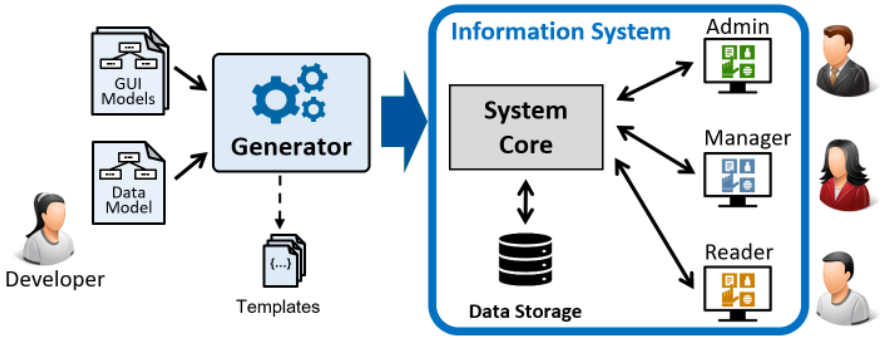

Figure 14 – Main concepts of information systems developed with mbse methods

The *MontiCore Data Explorer (MontiDEx)* code generator was used for the automatic generation of customizable, extensible data-centric business applications from

UML/P Class Diagrams [MSNRR15, Rot17]. It processes models to generate data-centric applications in Java and Java Swing. The department has developed further generators such as MontiEE [Loo17] or MontiWIS [Rei16].

The most recent generator, *MontiGem* [AMN+19], was successfully applied in an application project for the financial controlling of the chairs of RWTH Aachen University, the MaCoCo project [ANV+18]. With MontiGem it is possible to generate large parts of data centric business applications: The data-structure and communication infrastructure, functions to access, retrieve and store the data, the GUIs, and parts of the access control (see also Section 4.4).

Enterprise Information Systems are currently facing new challenges: The General Data Protection Regulation (GDPR), in application since May 2018, marks a new era in data privacy. This regulations are also relevant for EIS dealing with private data. Thus, we investigate the architecture of Cloud Services for the Digital me in a Privacy-Aware Environment [ELR+17]. Further approaches go beyond architectural aspects: [MKM+19] discusses a privacy preserving data model and system architecture. The user-centered view on the system design allows to track who does what, when, why, where and how with personal data and can make this information available for users in an information portal.

3.6 Internet of Things (IoT) and Industry 4.0

In line with our mission, our research contributions to the Internet of Things and Industry 4.0 focus on modeling techniques to support and facilitate development of increasingly complex solutions. Early contributions include architecture modeling techniques and infrastructures for the efficient development of cloud-based systems [NPR13], secure distributed systems [HHRW15], and distributed robotics systems [RRRW15].

With the rise of Industry 4.0, we included modeling outside of informatics, e.g., in mechanical or electrical engineering, into our focus of research. To this end, we conducted a systematic mapping study on modeling in Industry 4.0 [WCB17] which uncovered that knowledge representation and discrete modeling of systems and processes demand for DSLs (see Section 2.4) usable by the automation and manufacturing experts.

In that study, we also identified products, resources, and processes as primary Industry 4.0 concerns that usually are related inflexibly. Hence, we conceived a multi-level framework for machining based on these concepts [BKL+18].

4 MontiCore Zoo of Model-based Tools and Languages

As one of the important results of practical software engineering research, tools or demonstrators arise. For quick definition of languages and rapid development of model-based tools, the language workbench MontiCore was designed. It allows to compositionally design model-based tools for various languages. Some of them are discussed in the following subsections to demonstrate their feasibility.

4.1 MontiCore: The Language Workbench for DSLs

MontiCore is a language workbench, which has been developed since 2004 [GKR+08, HR17]. We started its development because at that time the available tools for model management where often very poor in functionality and also not extensible, but

closed shops. In 2004 the first version of UML/P was published (and is now available as [Rum16, Rum17]) demonstrating that the conglomerate of languages – that the UML is made of – can be substantiated with useful transformation, refinement and refactoring techniques (cf. Section 2.7). We were mainly interested in model-based techniques for code and test code generation as well as flexible combination of language fragments, such as OCL within Statecharts or Class Diagrams for typing in Component and Connector Diagrams. However, hard coded modeling tools where not helpful in realizing these techniques. This original motivation for providing a flexible and adaptable toolset through MontiCore can also be found in the foundational PhD theses of MontiCore [Kra10, Völ11].

Later, it became apparent that UML will be complemented by several DSLs that will be connected to software development or execution in various ways. The definition of DSLs encounters the same difficulties as the definition of UML has faced, i.e., they are often built from scratch, reuse is difficult or not supported, and the same concepts get different syntactic shapes. Thus, combining DSLs is rather impossible. We therefore extended the focus of MontiCore to become a general language workbench that allows to define languages and language fragments and to derive as much as possible from an integrated, compact definition.

MontiCore provides sophisticated techniques to generate transformation languages and their transformation engines based on DSLs [HRW15, AHRW17a, RRW15, HHRW15, Wei12], we have explored tagging languages [Loo17, MRRW16, GLRR15], various forms of the UML and its derivatives [Sch12, Wor16, Hab16, Rei16, Rot17] and a variety DSLs. Despite MontiCore being an academic tool to explore modeling and meta-modeling techniques, after 14 years of development, it has reached an extraordinary strength and is thus increasingly used in industrial projects, like energy management [Pin14], as well as in scientific projects of entirely different nature, such as the simulation of urban scenarios for autonomous driving [Ber10] or human brain modeling [PBI+16]. MontiCore, however, does not primarily focus comfort, e.g., graphical editing, but advanced functionality for model-based analysis or synthesis of software intensive systems as well as efficient textual editing for experienced users.

4.2 MontiArc: Architectural Modelling With Semantics

MontiArc [HRR10, Hab16] is a framework for modeling and simulation of software architectures that has been developed for modeling distributed interactive systems using MontiCore (see Section 4.1). The domain of the architecture description language (ADL) MontiArc are information-flow architectures which describe the components of a (software) system and their message-based communication. Hence, MontiArc captures active components (agents, actors) of a logical or physical distribution, their interfaces (ports), the communication infrastructure between components, and hierarchical decomposition. A component is a unit which executes computations or stores data. It may have arbitrary complexity and size being a subsystem or a single function. A component has an explicitly defined interface via which it communicates with its environment. It is implemented as a textual language and comes with an eclipse-integrated editor.

On one hand MontiArc is an architecture description language that helps modeling distributed systems by supporting the user with context condition (well-formedness rules) checks and analyses. On the other hand MontiArc serves a simulation environment and a code generator to generate simulation components out of MontiArc models. MontiArc provides a timed, event-based simulation framework that can exe-

cute behavior implemented in Java and attached to MontiArc models in a declarative way so that analysis and validation of MontiArc models becomes possible [Hab16].

Because the language MontiArc is designed for extensibility, several sublanguages for behavior may be embedded directly within component definitions. As examples, MontiArc has been extended with automata to MontiArcAutomaton [RRW13, RRW14, Wor16], with replication message bundles to ClArc [NPR13], and with security annotations to MontiSecArc [HHRW15].

In [HRR10], an extension of MontiArc with Java is presented and later extended to MontiArcAutomaton, which becomes a full programming language with explicit control of architecture, data structures, and behavior [BKRW17a]. Details on the compositional nature and extensibility of MontiArcAutomaton are described in [RRRW15] and [BHH+17] respectively. This approach achieves a smooth integration of architectural design and programming. Supported by powerful code generation mechanism based on chains of model transformations [AHRW17b], we call our approach *architectural programming*.

4.3 UML/P: Executable Modeling with UML

The *UML/P* [Rum16, Rum17, Rum12, Rum11] is a realization [Sch12] of a subset of the modeling languages of the UML with particular focus on applicability for programming, i.e., modeling of software systems (the "P" in UML/P stands for "suitable for programming"). It comprises the following types of UML diagrams:

- *Class Diagrams* as basic structural modeling technique with solid yet configurable semantics [MRR11e, MRR11c],
- *Object Diagrams* to describe specific situations that are used for setups of tests or test verdicts as well as unwanted situations, etc. [MRR11d].
- *Statecharts* are the main technique to describe behavior. UML/P Statecharts can be used as abstract specifications, explicitly allowing underspecification in various forms as well as code-like detailed algorithms, where a direct mapping to code becomes feasible. UML/P Statecharts assist the large bandwidth between high-level abstract specifications and implementations and come with a powerful refinement calculus allowing for the refinement of specifications throughout the development process as well as from superclasses to specific implementations in subclasses.
- *Sequence Diagrams* describe interactions between objects in a specific situation and thus are generally exemplary. They can be used nicely for describing the set up and the execution of tests.
- *OCL* is a textual language in the spirit of predicate logic over object models, allowing to describe constraints of various forms. OCL/P is a specific variant of OCL that allows for writing constraints in a syntax inspired by Java [GJS05]. It is extended by various techniques for set and list comprehension known from functional languages like Haskell [HJW92].

These languages can be used individually as well as in a coherent language family that allows to connect models of different sublanguages into a well structured, accessible specification of complex systems. This constitutes a powerful way of describing software systems on various levels of abstraction from different viewpoints. In addition, it is

possible to use Java expressions to enrich the models with implementation details. As such, UML/P forms the foundation for applied Generative Software Engineering (see Section 2.2).

To shape UML/P as a modular, composable set of languages, we have intensively applied the techniques of software language engineering (see Section 2.5). As an example, OCL is not only a stand-alone constraint language, but it is also used for pre- and post-conditions in Statechart transitions, for state invariants, for desired properties to describe the verdict of tests, and for desired properties while inside a sequence diagram-governed test execution.

For the constructive part of UML/P Java expressions are also integrated in various UML/P sublanguages. This integration relies on solid foundations, actually allowing for intensive consistency checks within models of all sublanguages as indicated in Figure 2. This includes type checking of variables introduced and used in sublanguages as well as the mapping of symbols defined in one sublanguage to another sublanguage.

Because object diagrams have limited expressiveness on their own, we have integrated object diagrams into OCL/P to identify specific scenarios via diagrams and connecting them with powerful logic operators of OCL/P. The integration of object diagrams into OCL/P allows for describing the context of invariants and specifying situations that are allowed, forbidden, or alternatives. This yields a powerful yet partially diagrammatic logic specification technique.

We also have dealt with activity diagrams [MRR11a, MRR11b] as extension of the UML/P and out ADL MontiArc (see Section 4.2) completes UML/P with a notation for logically or physically distributed computation [BKRW17a].

Parts of UML/P are available as an Eclipse plugin featuring syntax-aware editors for the various types of diagrams. We have successfully applied UML/P for generatively creating various kinds of software systems. This spans from database schemas over web applications up to entire data management desktop applications. UML/P is under continuous extension with interesting new features.

4.4 MontiGem: Generating Enterprise Management Systems

MontiGem [AMN+19] is a specific generator for data-centric business applications. To a large extent, it uses standard models from UML/P as sources. Namely, Class Diagrams, OCL, and Statecharts are taken from UML. While the standard semantics of these modeling languages remains untouched, the generator produces a lot more functionality around these models, because it is well integrated into the target framework and target infrastructure which is used to execute the enterprise management system. The generator thus knows the backend technology stack, with an application core in Java, storage using a relational database and the frontend consisting of Typescript respectively JavaScript components based on Angular 6 and thus running in the browser.

The generator creates the data structure in the frontend and backend as well as the communication infrastructure to transport data in both directions. Furthermore, it generates the database tables as well as all necessary functionality to access, retrieve and store the data in the database.

As a highlight, the storage paradigm is based on the command pattern that allows to merge current changes much better and thus allows an optimistic update scheme.

As an extension, a DSL is used to describe the graphical layout of the user interface in a comfortable way. Again the GUI sub-language is well integrated with the class

diagram models allowing to directly describe what to visualize based on the storage structure in the database (see Figure 15 for some of the possible visualizations).

Figure 15 – GUI-components generated based on models

The internal architecture of the MontiGem generator includes the three typical main processes: reading, transformation and generation (see Figure 16), while generation produces a whole lot of resulting Java classes, Typescript and HTML files and related artifacts.

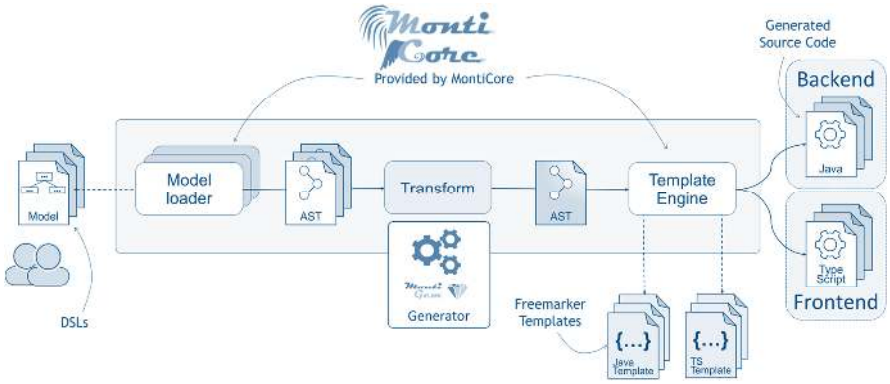

Figure 16 – Overview of the generator components

The whole generation is designed in a very extensible way: First, a generator target itself is written in a modular way allowing to reuse parts in a rather independent way. Secondly, templates are used allowing developers to add functionality in the systematic manner. They can, e.g., add additional methods to all generated classes of certain kind. Thirdly, the TOP mechanism created in MontiCore [HR17] is applied for all kinds of creative classes allowing to efficiently add handwritten code extensions to the generated classes, while fully retaining the ability to intentionally re-generate everything every time. For that purpose, all classes are also equipped with builders, which can be replaced using the TOP mechanism if required.

Thus, handwritten and generated code pieces are well integrated [GHK+15b]. This enables the continuous regeneration of the application with changing models and thus the co-evolution of the models and the handwritten additional functionality during the entire development process.

The input of the generator for such enterprise management systems can be expanded

to allow the tagging of existing models [GLRR15], e.g., for the definition of roles and rights, as well as model-based testing [DGH+18].

MontiGem builds on earlier versions. Together with MontiGem they are already in use for generating various applications, such as a library system, a Management Controlling Cockpit and a development artifact overview system. Current extensions adapt and extend MontiGem for mobile applications, further graphical representation components as well as the development of information portals in the Internet of Things domain.

5 Software Engineering at RWTH: "Modeling the World"

In this article we have summarized the research conducted and published by Bernhard Rumpe and his group at the TU Munich (1992-2003), at TU Braunschweig (2003-2008), and at the Software Engineering group at RWTH Aachen University (since 2009).

Over these years our work 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.

Models and therefore *modeling languages* have been chosen as the most important vehicle to take bold steps in this direction. Making models explicit and formally accessible in form of manageable artifacts is one subgoal, because this allows reusing, joint refactoring and evolving models. Combined with code synthesis and test case generation, this enables to achieve higher degrees of automation. Repeatable automation of analysis, transformation, and synthesis tasks is an important prerequisite for an efficient process. Efficiency and automation are furthermore prerequisites for agility.

Our ultimate goal is to have techniques and methods at hand that allow to model requirements for a system in a rather abstract form and use smart transformation tools with a high degree of automation during the process, such that the developers at least of standard products are liberated from architecture definition, fine design, test case invention, manual testing, reviewing various forms of the same information, and other potential tedious tasks as much as possible.

In the preceding sections, we have discussed various steps in this direction demonstrating that this goal on the one hand can be continuously approached, but also that the path towards it is relatively complex, because the optimal languages for abstract and compact design of software are still in the making, comfortable tools still need to be improved, and people need to be trained to be able to use powerful tools and models.

Modeling techniques mainly come into play, when the understanding of a domain has matured, the complexity and variability within the domain have increased due to various individual demands, and the domain has become a serious scientific or industrial sector. Standard business systems as well as web services and apps for mobile phones belong into this category. However, we also observe that the connection between physical systems and their controlling and managing software tightens and

a model-based systems engineering process emerges. This process allows developing mobility services (including cars, buses, bicycles), electrifying a smart home, efficiently assisting clinical surgery and caring, and much more. Still there are many more things to model in this world, to gain a better understanding and to improve it.

While currently, big data gains much attention, we assume that its potential can only be fully exploited when combined with pre-existing knowledge about the system under observation. This demands the combination of engineered models with data acquisition, and the data-driven refinement of such models will be one of the future's interesting challenges. Digital twins, for instance, are such models of products or systems that demand connection to data sources. Digital twins can also be used to describe production, business, and other complexities in the (cyber-physical) world. Because of the highly heterogeneous nature of the systems being described, we can safely assume that quite a number of general-purpose as well as domain-specific languages will be needed to enable the integrated development, harnessing, and evolution of digital twins. Language composition, aggregation, and evolution will help addressing these challenges as well as the continuous evolution of modelling mechanisms. It will be interesting to see how digital twins will benefit from using models at runtime, e.g., for enhancement, manipulation, but also for self-healing, self-adapting, and similar approaches. It may even be, that a strong connection between the digital twin and the raw physical system is the foundation for self-aware systems that will help to assist individual persons as well as humankind in various future challenges.

Acknowledgments

Figure 17 shows of the structure of our group consisting of a foundation firmly grounded on models, modeling, and methods that use models for software engineering. On top of these foundations, Figure 17 shows three of our currently most active application domains, where we are apply and refine modeling for software and systems development.

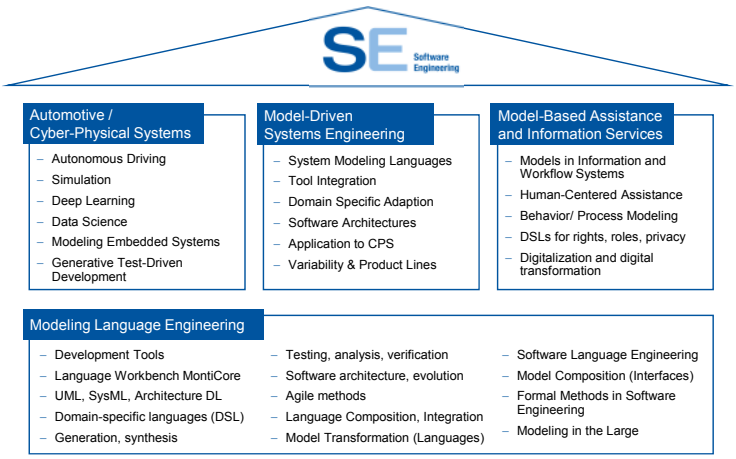

Figure 17 – Foundations and applications domains that define our research groups at the Software Engineering Department at RWTH Aachen University

Over the years many collaborators have been helping to elaborate our ideas. We would like to thank all current and former members of our group as well as all students and apprentices who helped us to elaborate our ideas. Namely, we would like to thank

Kai Adam, Professor Dr. Christian Berger, Vincent Bertram, Arvid Butting, Manuela Dalibor, Anabel Derlam, Imke Drave, Robert Eikermann, Timo Greifenberg, Dr. Hans Grönniger, Dr. Tim Gülke, Dr. Arne Haber, Guido Hansen, Olga Haubrich, Lars Hermerschmidt, Dr. Christoph Herrmann, Gabi Heuschen, Steffen Hillemacher, Nico Jansen, Oliver Kautz, Jörg Christian Kirchhof, Carsten Kolassa, Dr. Anne-Therese Körtgen, Thomas Kurpick, Evgeny Kusmenko, Dr. Holger Krahn, Dr. Stefan Kriebel, Stefan Kühnel, Achim Lindt, Dr. Markus Look, Professor Dr. Shahar Maoz, Matthias Markthaler, Dr. Dan Matheson, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Antonio Navarro Pérez, Nina Pichler, Dr. Claas Pinkernell, Dr. Dimitri Plotnikov, Deni Raco, Dr. Holger Rendel, Dr. Dirk Reiss, Dr. Daniel Retkowitz, Dr. Alexander Roth, Dr. Martin Schindler, David Schmalzing, Steffi Schrader, Dr. Frank Schroven, Dr. Christoph Schulze, Igor Shumeiko, Simon Varga, Dr. Steven Völkel, Dr. Ingo Weisemöller, and Michael von Wenckstern.

Special thanks goes to Marita Breuer, Sonja Müßigbrodt, and Galina Volkova, who help making our software tools a success and support us in administrative issues, and most important to Sylvia Gunder, who ensures that all financial and project activities supporting our department are running perfectly.

We also thank Manfred Nagl, who led the Department of Software Engineering from 1986 until 2009, when Bernhard Rumpe took over, and Horst Lichter, who runs his own Software Engineering research group allowing us to gain deeper insights into agility, DevOps, robust architectures and related research topics.

And finally, we would like to thank the new Editor-In-Chief team of the Journal Of Object Technology, namely Alfonso Pierantonio, Benoit Combemale, and Mark van den Brand for the invitation to provide this overview article of our last 25 years of research in Software Engineering challenges.

References

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse*. Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[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.

[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.

[AMN+19] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *Digital Ecosystems of the Future: Methods, Techniques and Applications (EMISA'19)*, LNI, pages 1–8, 2019. (in press).

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, volume 2097 of *CEUR Workshop Proceedings*, pages 75–79. CEUR-WS.org, May 2018.

[BBB+08] Christian Basarke, Christian Berger, Kai Berger, Karsten Cornelsen, Michael Doering, Jan Effertz, Thomas Form, Tim Gülke, Fabian Graefe, Peter Hecker, Kai Homeier, Felix Klose, Christian Lipski, Marcus Magnor, Johannes Morgenroth, Tobias Nothdurft, Sebastian Ohl, Fred W. Rauskolb, Bernhard Rumpe, Walter Schumacher, Jörn-Marten Wille, and Lars Wolf. Team CarOLO – Technical Paper. Informatik-Bericht 2008-07, Technische Universität Braunschweig, October 2008.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics*, 12, 2018.

[BBH+13] Christian Berger, Delf Block, Christian Hons, Stefan Kühnel, André Leschke, Bernhard Rumpe, and Thorsten Strutz. Meta-metrics for simulations in software engineering on the example of integral safety systems. In *Proceedings des 14. Braunschweiger Symposiums AAET 2013, Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel*, volume 14, pages 136–148, 2013.

[BBH+15] Christian Berger, Delf Block, Christian Hons, Stefan Kühnel, André Leschke, Dimitri Plotnikov, and Bernhard Rumpe. Simulations on Consumer Tests: A Systematic Evaluation Approach in an Industrial Case Study. *Intelligent Transportation Systems Magazine (ITSM)*, 7(4):24–36, October 2015.

[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.

[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.

[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.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BEK+18] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software*, 152:50–69, June 2019.

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+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.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[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.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *In 12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17b] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. *Journal of Systems and Software*, 149:437–461, March 2019.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[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.

[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.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling reusable, platform-independent robot assembly processes. *Proc. of the Sixth International Workshop on Domain-Specific Languages and Models for Robotic Systems*, 2015.

[BS01] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+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.

[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.

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

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[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.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019.

[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.

[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 *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[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.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FNDR98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. Bmw-room an object-oriented method for ascet. Technical report, SAE Technical Paper, 1998.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[FPR02] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML profile for framework architectures*. Addison Wesley object technology series. Addison-Wesley, 2002.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[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. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+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. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 112–132. Springer, 2015.

[GJS05] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 3rd edition edition, 2005.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, pages 67–81, 2006.

[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.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System model-based definition of modeling language semantics. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5522 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2009.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+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.

[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.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, volume 1463 of *CEUR Workshop Proceedings*, pages 18–23, 2015.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the Programming Language Haskell, A Non-strict Purely Functional Language. In *Sigplan Notices*, volume 27 of *ACM*. ACM Press, May 1992.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, volume 9839 of *LNCS*, pages 175–182. Springer, December 2016.

[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.

[HN01] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Höl18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[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.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, pages 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Computer Languages, Systems & Structures*, 54:386–405, 2018.

[HWR14] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, 2014.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software*, 35(6):40–47, 2018.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger 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.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, volume 9679 of *LNCS*, pages 122–137. Springer, June 2016.

[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.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, editors, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[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.

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

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[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.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[KRSvW18] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[Lee08] Edward A. Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 363–369. IEEE Computer Society, 2008.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[LS89] N. Lynch and E. Stark. A proof of the kahn principle for input/output automata. *Information and Computation*, 82:81–92, 1989.

[MGH+98] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl - the planning domain definition language. Technical report, Yale Center for Computational Vision and Control, 1998.

[MGSM13] Judith Michael, Andreas Grießer, Tina Strobl, and Heinrich C. Mayr. Cognitive modeling and support for ambient assistance. In Heinrich C. Mayr, Christian Kop, Steven Liddle, and Athula Ginige, editors, *Information Systems: Methods, Models, and Applications: Revised Selected Papers.*, volume 137 of *LNBIP*, Heidelberg, 2013. Springer.

[Mil78] Robin Milner. Synthesis of communicating behaviour. In J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978*, pages 71–83, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[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, *Information Systems Engineering in Responsible Information Systems*, volume 350 of *Lecture Notes in Business Information Processing*, pages 194–206. Springer Nature, 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In Wilfred Ng, Veda C. Storey, and Juan Trujillo, editors, *Conceptual Modeling - ER 2013*, volume 8217 of *Lecture Notes in Computer Science (LNCS)*, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. *Model Centered Architecture*, pages 85–104. Springer International Publishing, 2017.

[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.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[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.

[MRR11d] 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.

[MRR11e] 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.

[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.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, volume 1723 of *CEUR Workshop Proceedings*, pages 19–24, October 2016.

[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 International Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MSNRR15] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[MSS+18] Judith Michael, Claudia Steinberger, Vladimir A. Shekhovtsov, Fadi Al Machot, Suneth Ranasinghe, and Gert Morak. The hbms story - past and future of an active assistance approach. *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling*, 13:345–370, 2018.

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

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 93–108. Bonner Köllen Verlag, March 2016.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen*. Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[QGC+09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[Rei92] W. Reisig. Combining petri nets and other formal methods. In K. Jensen, editor, *Application and Theory of Petri Nets 1992*, pages 24–44, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

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

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[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.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[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.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

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

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[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.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

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

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SM18] Claudia Steinberger and Judith Michael. Towards cognitive assisted living 3.0. In *Advanced Technologies for Smarter Assisted Living solutions: Towards an open Smart Home infrastructure (SmarterAAL workshop) at Percom 2018*, pages 687–692. IEEE, 2018.

[SM19] Claudia Steinberger and Judith Michael. Using semantic markup to boost context awareness for assistive systems. In F. Chen, R.I. García-Betances, L. Chen, M.F. Cabrera-Umpiérrez, and C.D. Nugent, editors, *Smart Assisted Living - towards an open smart home infrastructure*, pages 1–21. Springer International Publishing, 2019. (in press).

[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.

[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.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[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.

About the authors

Katrin Hölldobler is PostDoc and a member of the SE group since 2012. She is leader of the Modeling Language Engineering (MLE) team. Her PhD thesis was

about domain specific model transformation and systematic transformation language engineering. Contact her at hoelldobler@se-rwth.de, or visit www.se-rwth.de.

Judith Michael is PostDoc in the SE group since early 2018 and leader of the Model-Based Assistance and Information Services (MBAIS) team. Her PhD thesis at the Alpen-Adria-Universität Klagenfurt was about cognitive modeling for assistive systems. Contact her at michael@se-rwth.de, or visit www.se-rwth.de.

Jan Oliver Ringert is a lecturer at the University of Leicester, UK. He was a PostDoc at Tel Aviv University, Israel from 2013 to 2018. He was a member of the SE group from 2008 to 2013. His PhD thesis was about analysis and synthesis of interactive component and connector systems. Contact him at jor4@le.ac.uk, or visit www.cs.le.ac.uk.

Bernhard Rumpe is heading the Software Engineering department at the RWTH Aachen University, Germany. Earlier he had positions at INRIA Rennes, Colorado State University, TU Braunschweig, Vanderbilt University, Nashville, and TU Munich.

His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain specific languages. He also helps to apply modeling, e.g. to autonomous cars, human brain simulation, BIM energy management, juristical contract digitalization, production automation, cloud, and many more.

He is author and editor of 34 books including "Agile Modeling with the UML" and "Engineering Modeling Languages: Turning Domain Knowledge into Tools". To contact him visit www.se-rwth.de.

Andreas Wortmann is tenured research associate and a member of the SE group since 2011. He leads the Model-Driven Systems Engineering (MDSE) team. His PhD thesis was about composing software languages for robotics software architectures. Contact him at wortmann@se-rwth.de, or visit www.se-rwth.de.