Improving Reuse in Architecture Modeling with Higher-Order Components

Kai Adam[1], Bernhard Rumpe[1], Andreas Wortmann[1]

Abstract: Modern architecture description languages (ADLs) combine the benefits of component-based software engineering and model-driven development. Conceiving software component models as black-boxes entails challenges to their reuse when they must be customized to the new application context. Thus, reuse in ADLs usually is limited to situations with perfect fit between software architecture and already available components, requires fixing reuse options apriori, or entails the generic complexities of, for instance, delta modeling [HKR+11]. We present a concept of higher-order components for component and connector (C&C) ADLs that enables modeling components with component-valued parameters to enable injecting customized subcomponents where necessary. This concept relies on metamodel element adaptation and can be retrofitted into many C&C ADLs, which is demonstrated with MontiArcAutomaton architecture modeling infrastructure. Higher-order components are more flexible than 150% [GKPR08] models and less challenging than delta modeling. They enable to customize components prior to reuse easily and hence C&C architecture modeling with facilitate off-the-shelf components.

Keywords: Model-Driven Development; Architecture Description Languages; Component and Connector Architectures; Component Reuse

1 Introduction

The ultimate vision of component-based software engineering (CBSE) is to compose complex systems from off-the-shelf, black-box components. Yet customizing third-party components can be necessary to prevent engineering slightly different components from scratch, fixing reuse options apriori, or learning delta modeling languages [HKR+11]. Furthermore, many approaches to CBSE employ general programming languages (GPLs), which confront developers with the "accidental complexities" [FR07] of their idiosyncrasies. Model-driven development (MDD) [VSB+13] lifts models to primary development artifacts to abstract from both. These models conform to modeling languages and can be translated into GPL artifacts automatically. Component and connector (C&C) architecture description languages (ADLs) [MT00] are modeling languages to describe software architectures. Although the abstraction of C&C ADLs enables many useful features, it usually is still expected to reuse complete components in a black-box fashion, which at best can be parametrized with data-type-valued arguments.

We present an approach to parametrize hierarchical C&C models with component-type-valued arguments to enable the injection of subcomponents into black-box components. These higher-order component parameters are tied to component-types and enable a postponed customization for such models by passing instances of the (sub-)type without changing the component model (*i.e.,* the choice of subtypes of a subcomponent is left open).

[1] RWTH Aachen, Software Engineering, Aachen, Germany, http://www.se-rwth.de

[ARW17] K. Adam, B. Rumpe, A. Wortmann:
Improving Reuse in Architecture Modeling with Higher-Order Components.
In: Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIII (MBEES'17).
https://www.inf.uni-hamburg.de/en/inst/ab/swk/research/events/mbees2017.html, 2017.
www.se-rwth.de/publications/

This approach relies on introducing adapters to the ADLs' metamodels and introducing few well-formedness rules. Employing adaptation enables to reuse the remaining parts of existing infrastructure (existing well-formedness rules, model transformations) without modification and hence is a worthwhile extension to many C&C ADLs.

The contribution of this paper is a concept for higher-order components with component-valued parameters. Specifically, we present the concept relative to metamodel elements common to many C&C ADLs, show how it can be retrofitted into such languages, and present its realization for the MontiArcAutomaton ADL. In the following, Sec. 2 describes necessary preliminaries before Sec. 3 illustrates the benefits of this approach by a motivating example. Sec. 4 presents the concept on metamodel elements common to many C&C ADLs and Sec. 5 describes retrofitting it into MontiArcAutomaton ADL. Afterwards, Sec. 6 highlights related work and discusses challenges. Finally, Sec. 7 concludes.

2 Preliminaries

Architecture description languages combine the benefits of CBSE (encapsulation, reuse) and MDD (abstraction, comprehensibility) to enable describing complex software systems as topologies of interacting component models. Science and industry have produced over 120 ADLs [MLM+13] that employ various architectural styles and provide different features. Component and connector ADLs [MT00] are a specific architectural style that focuses on describing component communication through explicitly modeled connectors between the interface elements (*e.g.,* ports) of component models. Usually, the components can be composed hierarchically by creating configurations of subcomponent topologies and many C&C ADLs moreover support to configure subcomponents with data type arguments (such as offsets, identifiers, *etc.*). Other popular architectural styles are multitier architectures or client-server architectures. With C&C ADLs being modeling languages, we follow [CvCR15] in assuming that these can be defined in terms of concrete syntax (the ADL's words), abstract syntax (the structure of its sentences), static semantics (its well-formedness rules), and dynamic semantics (its behavior). We do not impose restrictions on the language definition constituents, *i.e.,* the concrete syntax may be graphical or textual, the abstract syntax may be defined by grammars or metamodels, *etc.*

MontiArcAutomaton is an extensible architecture modeling infrastructure comprising the MontiArcAutomaton ADL, exchangeable model-to-model transformations, and a powerful code generation framework [RRRW15]. It is built around the MontiArcAutomaton C&C ADL and has been applied to teaching [RRW13] and industry [HMR+15]. The MontiArcAutomaton ADL describes software architectures as hierarchies of interacting components. These components exchange messages through the unidirectional connectors between their interfaces of directed, typed ports only. It distinguishes *component types* from *component instances* and supports component configuration parameters to pass data type arguments to components at their instantiation (similar to constructors in many object-oriented languages). These component parameters may use generic type parameters similar to Java. Component types are either atomic or composed. Atomic component types are related to a behavior description, either in form of embedded automata or behavior

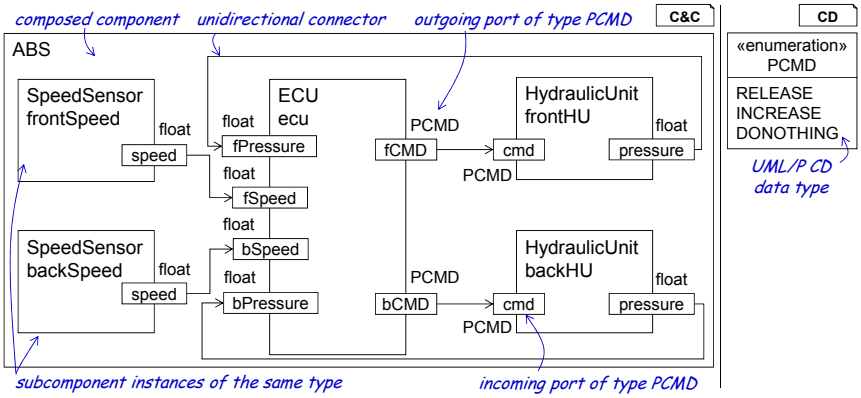

Fig. 1: Top-level MontiArcAutomaton component type of an ABS implementation for motorcycles with five subcomponents and the related UML/P class diagram data type `PCMD`.

implementations in the general purpose language (GPL) the components are translated to. The behavior of composed components instead emerges from the behavior and interconnection of their subcomponent instances. Moreover, MontiArcAutomaton supports component type inheritance, where sub-types inherit the super-types ports and configuration parameters. Retaining interface compatibility, this inheritance enables component instances of a sub-type to be used where ever its super-type is required. MontiArcAutomaton architectures operate in the context of UML/P class diagram [Rum16] types, *i.e.,* the data types of ports are classes. Fig. 1 illustrates the modeling elements of MontiArcAutomaton with an anti-lock braking system (ABS) for motorcycles. The top-level component `ABS` contains a speed sensor and a hydraulic unit per wheel as well as a single electronic control unit (ECU). Speed sensors are represented by the subcomponent instances `frontSpeed` and `backSpeed` of component type `SpeedSensor`. Hydraulic units are represented by instances `frontHU` and `backHU` of component type `HydraulicUnit`. Speed sensors report the current speed of the front wheel (or back wheel) to the ECU via connectors between their `float` ports `speed` to ECU ports `fSpeed` and `bSpeed`. The ECU also receives the hydraulic pressures of both wheels. Based on these, it controls adapting the hydraulic pressure by sending pressure commands (`PCMD`s) to the hydraulic units.

3 Example: Motorcycle ABS Architecture

The motorcycle ABS architecture depicted in Fig. 1 comprises two instances of component type `HydraulicUnit`. The structure of this component type is as depicted in Fig. 2: It receives a pressure command, which its subcomponent instance `ctrl` of type `Controller` translates into increasing pump pressure (via `PumpActuator`) and opening a valve to decrease pressure (via `ValveActuator`) for a fixed amount of time. Both are atomic components that interact with the environment. The `PressureSensor` also interacts with the environment and emits the sensed hydraulic pressure via port `pressure`.

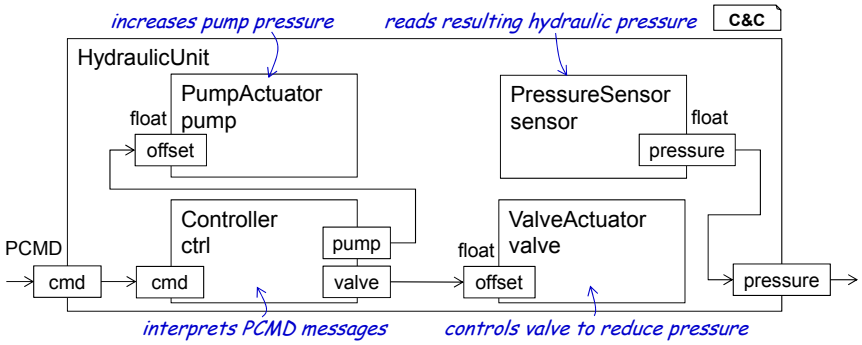

Fig. 2: The composed component type `HydraulicUnit` comprises four subcomponent instances to adjust hydraulic pressure.

Following the vision of CBSE, the company implementing the `ABS` software architecture for their line of sportbikes bought the component type `HydraulicUnit` off-the-shelf as a black-box. When using the architecture for their line of choppers, they found that the choppers' weight imposes manipulating pump and valve differently. If the source of the component types `HydraulicUnit` and `Controller` are provided, the company could customize both manually to their requirements. This, however, usually is error-prone and costly. Instead, they decide to replace the instances of `HydraulicUnit` by instances of `ExtensibleHydraulicUnit` as depicted in Fig. 3. This component type already implements our concept of component-valued parameters and yields a single configuration parameter of component type `Controller` and of name `ctrl`. Furthermore, `ExtensibleHydraulicUnit` features only three static subcomponent instances `pump`, `sensor`, and `valve`. The subcomponent instance `ctrl` of type `Controller` is *derived* from the higher-order parameter of the same name. This flexibility enables the company to easily reuse the component type `ExtensibleHydraulicUnit` with both lines of motorcycles: at the level of the `ABS` component type, only proper sub-types of `Controller` for the different pump and valve manipulations must be instantiated and passed to the two instances of `ExtensibleHydraulicUnit`. This omits the need for error-prone and costly white-box reusing partial architectures with extensible components.

4 Higher-Order Component Parameters

As illustrated in the example, we aim to improve component reuse with C&C ADLs by enabling the parametrization of component types with component instances. To this effect, we aim to enable passing subcomponent instances as configuration arguments to other subcomponent instances and have the latter interpret these as 'normal', *i.e.,* statically declared, subcomponent instances. Exploiting component type inheritance, this enables to create appropriate component types that inherit from the types mentioned in the potential containing components' parameters and pass their instances as arguments. To enable such interpretation, we must enable interpretation of component types as data types and enable interpretation of configuration parameters as subcomponent instances.

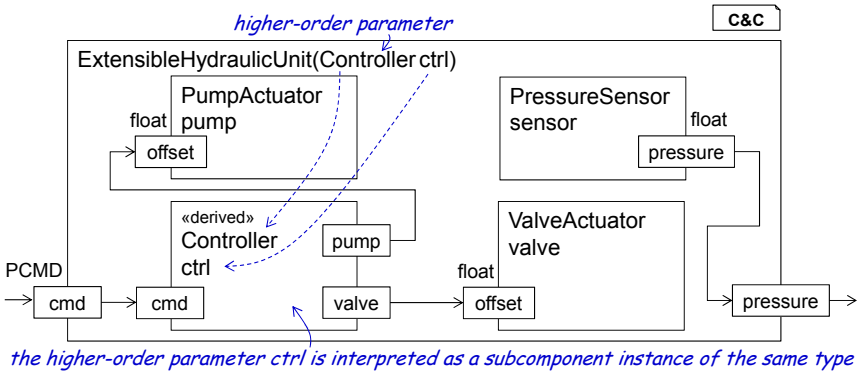

Fig. 3: An extensible variant of `HydraulicUnit` featuring a higher-order parameter that accepts component type arguments and interprets these as subcomponent instances.

We propose to augment the metamodel of C&C ADLs with adapters between component types and data types and between parameters and subcomponent instances as depicted in Fig. 4. The figure depicts an excerpt of a typical C&C ADL metamodel, where the classes `ComponentType`, `Connector`, `Port`, `DataType`, and `SCInstance` represent elements common to many C&C ADLs (whether component types require and provide interfaces or yield interfaces of typed ports is irrelevant to the proposed extension). The new metamodel elements are adapters between `ComponentType` and `DataType` (to enable interpretation of component types as data types) and between `Parameter` and `SCInstance` (to enable interpretation of parameters as subcomponent instances).

The new classes are introduced on the metamodel level solely and, hence, do not yield representations in the concrete syntax, *i.e.,* it is impossible to create instances of the elements with models. Both adapters must be instantiated whenever a component configuration parameter that is of a component type is found. This enables using the arguments passed to a composed component type expecting higher-order arguments as subcomponent instances for all purposes. It furthermore enables to pass higher-order arguments (*i.e.,* the subcomponent instances to be) down through multiple levels of the architectures hierarchy.

While this extension is minimally invasive to the metamodel at hand, it enables using higher-order parameters in unintended ways. For instance, this would also enable to use higher-order parameters as the data types of ports, which could be interpreted as sending component instances between components, and should be prevented for now. Moreover, it should be prevented to pass higher-order arguments to atomic subcomponent instances (which cannot use these as intended) and to define higher-order arguments with dimensions (*i.e.,* passing arrays of subcomponent instances). Although more elaborate changes to the underlying metamodel might prevent such behavior (*i.e.,* introducing a new parameter type with a specific concrete syntax keyword and adapting to this type), we propose to formulate such restrictions as part of the static semantics (*i.e.,* the well-formedness) rules. This prevents introducing more technical concepts into the metamodel. Consequently, we propose to add the following well-formedness rules to the ADL under extension: (1) Prevent

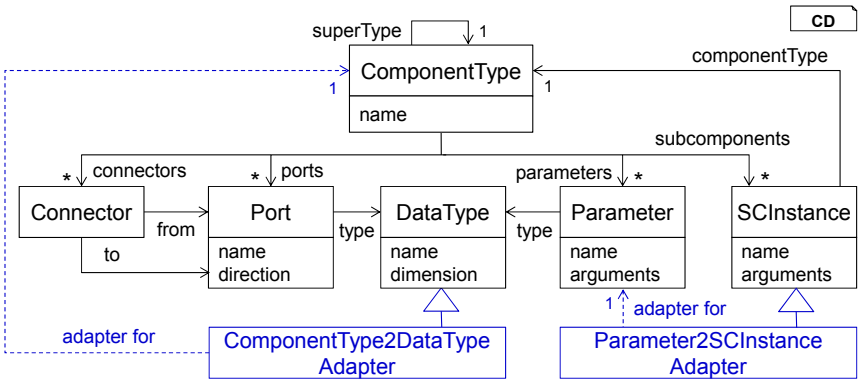

Fig. 4: An excerpt of a metamodel of common C&C modeling elements augmented with adapters to enable interpretation of component types as data types and of parameters as subcomponent instances.

higher-order component parameters from being used as port data types. (2) Prevent cycles of nested higher-order component parameters. (3) Prevent passing higher-order component parameters atomic subcomponents. (4) Prevent defining higher-order arguments with dimensions.

Other well-formedness rules might apply depending on the constraints of the ADL to be extended. For MontiArcAutomaton, this for instance includes preventing use of higher-order parameters as generic type arguments and that (in contrast to existing MontiArcAutomaton well-formedness rules) the subcomponent instances derived from higher-order parameters must not be connected when they are passed down to another subcomponent instance. The dynamic semantics of derived subcomponents follows from the dynamic semantics of subcomponent instances, *i.e.,* as they appear to be subcomponent instances to the operational or translational dynamic semantics of the ADL, they will be treated as such by interpreters or code generators.

5 Retrofitting Higher-Order Components into MontiArcAutomaton

MontiArcAutomaton is a C&C ADL built on top of the language workbench MontiCore [KRV10, HLM+15]. As such, its textual concrete syntax and its abstract syntax are defined as context-free grammars (CFGs) from which MontiCore generates model processing infrastructure (*e.g.,* parser, abstract syntax classes), a well-formedness checking framework for Java context conditions, and a template-based code generation framework for implementation of translational dynamic semantics. From the abstract syntax, MontiCore partially derives the metamodel of a language in form of the so-called *symbols*. These are abstractions of the technical abstract syntax and are used for further model processing after the textual models have been parsed into the abstract syntax tree. MontiCore's symbol table infrastructure also enables resolving names with model elements kinds to symbols for lazy calculation of model properties. Well-formedness checking and code generation process are based on symbols. Consequently, all classes of the MontiArcAutomaton meta-

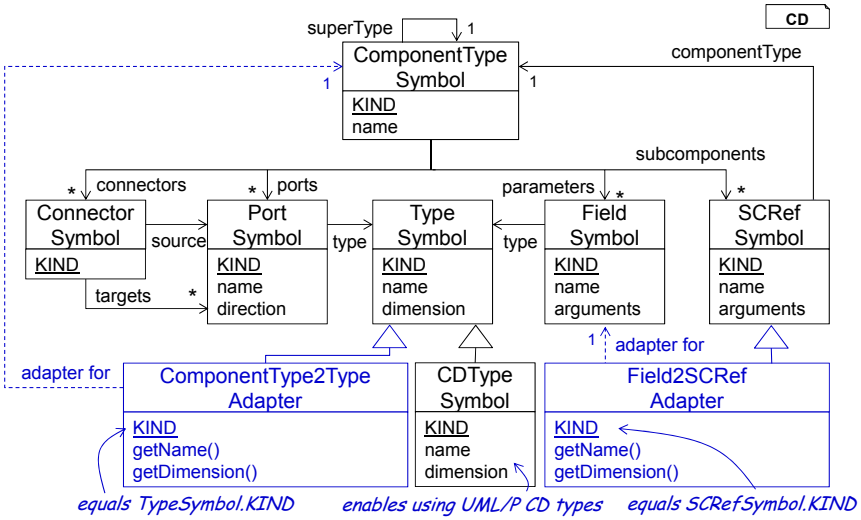

Fig. 5: Excerpt of MontiArcAutomaton metamodel extended with adapters required to use higher-order parameters.

model are subclasses of MontiCore's `Symbol` class, which yields a name and a kind. The latter is used for resolving and inheriting symbols must override this properly (*e.g.,* by introducing a component type kind).

Extending Abstract Syntax: The MontiArcAutomaton metamodel comprises the elements depicted in Fig. 5 (there are further elements, such as embedded component behavior models which are irrelevant to higher-order parameters and, hence, omitted). This includes a symbol for UML/P class diagram types with its own kind. For the new adapters to be resolved as data types (*i.e.,* `TypeSymbol`) or subcomponent instances (*i.e.,* `SCRefSymbol`), the adapters `KIND`s must equal their respective superclasses' `KIND`s. MontiArcAutomaton automatically creates instances these adapters when a name of the respective kind is resolved. For well-formedness checking and code generation, these adapters act as data types and subcomponent instances, respectively.

Extending Static Semantics: However, this metamodel extension does not ensure well-formedness of the models. To this effect, we extend MontiArcAutomaton's context conditions. Context conditions are Java classes that check well-formedness of specific model elements and MontiCore's context condition framework enables creating new context conditions via subclassing easily. These new classes must be registered with MontiArcAutomaton and are automatically applied to the parsed models afterwards. One of these new context conditions is depicted in Fig. 6 (left). The context condition `PortDoesNotUseComponentType` extends from the abstract MontiCore base class `ContextCondition` and overrides its `getKind()` method to return the `PortSymbol.KIND` as well as the `check()` method to perform the eponymous check. An excerpt of its implementation is depicted in Fig. 6 (right). The condition processes `PortSymbol`s and first looks up the passed port symbols' type via the MontiCore's symbol table (ll. 5-7). Then it

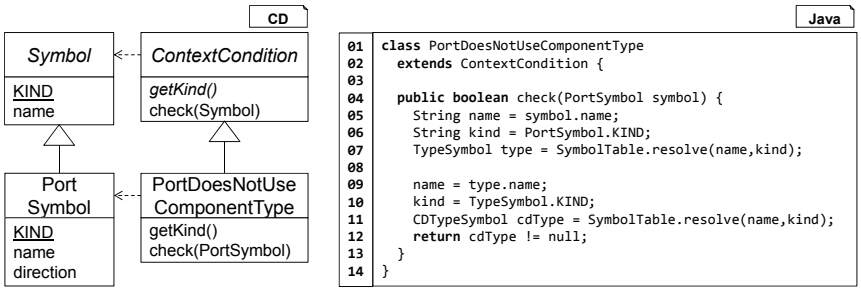

Fig. 6: Relations of the new `PortDoesNotUseComponentType` context condition (left) and an excerpt of its implementation (right).

checks whether this type exists as class diagram type (ll. 9-11). In this case – and as MontiArcAutomaton already prohibits ambiguous types – the context condition can assume that the type is defined as a class diagram and passes (*i.e.,* returns `true`). Otherwise, MontiArcAutomaton raises an error and ultimately aborts architectures processing. Similarly, the context conditions to prevent (1) cycles of nested higher-order component parameters, (2) passing higher-order component parameters to atomic subcomponents, (3) defining higher-order arguments with dimensions, and (4) using higher-order arguments for generic type parameters, are added to MontiArcAutomaton. Other context conditions, for instance ensuring the uniqueness of subcomponent names in a composed component type, adjust to the new source of subcomponents due to automatically resolving and including the adapters.

Extending Dynamic Semantics: MontiArcAutomaton uses MontiCore's template-based code generation framework to translate software architectures into executable systems. To this effect, it processes `ComponentTypeSymbol` instances and related symbols and translates these into Java artifacts or Python artifacts. In the extended metamodel, the `Field2SCRefAdapter` acts as a subcomponent instance (represented by `SCRefSymbol`). Hence, whenever the code generator looks up subcomponent instances, the adapted higher-order parameters are returned as well. Thus, integrating these into code generation requires is effortless.

6 Related Work and Discussion

Many ADLs, such as AADL [FG12] or KOALA [VVKM00], support reusing complete components via importing as subcomponents only. Other foster reuse via abstract component types [MT00]. This enables reusing partly underspecified architectures, but requires introducing and connecting concrete components manually. Acme [GMW00] supports templates, which are syntactic constructs enabling parametrizing components with ports. While useful to connect a component according to its new operation context, changing the type of a port using templates might require a new component implementation to deal with the new message kinds, *i.e.,* it complicates reusing component behavior. KADL [PR06] supports component inheritance and generic parameters in component type definitions for

data types, port types, and component types. This enables to define and reuse architectural patterns in different contexts. Generic port types and generic component types introduce complexity similar to Acme templates. Compared to the powerful mechanisms of Acme, KADL, and ROOM [SGW94], our approach is more restricted on component type inheritance. This is a less powerful but an easy to use approach that prevents jeopardizing the communication integrity of the architecture model and reduces the complexity in reuse.

Our approach to improving reuse in ADLs with higher-order components parameters relies on adapting the ADL's metamodel and adding well-formedness rules to prevent side effects. This is less powerful than delta modeling [HKR+11], which enables comprehensive changes to components by adding or removing models parts. This can result in broken architectures, which higher-order components prevent. Also, this is more expressive than 150% modeling [GKPR08] (where every possible option is in the model), as it supports a postponed customization of higher-order components without introducing new modeling primitives. Although MontiArcAutomaton supports specifying components with generic type parameters (*i.e.,* design-time underspecification of configuration parameter types or port types), we explicitly prevent higher-order components with generic type parameters. While adaptation of component types to data types generally supports this, generic type parameters can be used as port types. Using component type valued ports also poses challenges on their integration into the type system (*e.g.,* the members a component type exposes) and the semantics of MontiArcAutomaton (component-typed ports could mean sending components throughout the architecture). These challenges are under investigation.

7 Conclusion

We presented a small extension for C&C ADLs to improve reuse with component-valued parameters. These parameters facilitate customizing black-box components by enabling to inject subcomponent instances. This approach is more flexible than 150% models but less challenging than delta modeling. The concept relies on adapting ADL metamodel elements and adding few new well-formedness rules, hence it requires only minor changes to the abstract syntax of ADLs to be retrofitted. Through adaptation, existing rules as well as model transformations (including code generation) can be applied without modification. We believe, this approach to component reuse is beneficial to model-driven development with ADLs and future extension will improve this further.

References

[CvCR15] 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*, pages 7–20. Springer, 2015.

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

[FR07] Robert France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering 2007 at ICSE.*, 2007.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Proceedings of Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, 2008.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. ACME: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, 68:47–68, 2000.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-Oriented Architectural Variability using MontiCore. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ACM, 2011.

[HLM+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, 580:45–66, 2015.

[HMR+15] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *International Conference on Intelligent Robots and Systems (IROS 2015)*, 2015.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2010.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 2013.

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

[PR06] Pascal Poizat and Jean-Claude Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. *Journal of Universal Computer Science*, 12(12):1741–1782, 2006.

[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.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Case Study on Model-Based Development of Robotic Systems using MontiArc with Embedded Automata. In Holger Giese, Michaela Huhn, Jan Philipps, and Bernhard Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, 2013.

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

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, April 1994.

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

[VVKM00] Rob Van Ommering, Frank Van Der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.