Shepherding Model Evolution in Model-Driven Development

Arvid Butting[1] Steffen Hillemacher[1] Bernhard Rumpe[1]
David Schmalzing[1] Andreas Wortmann[1]

Abstract: Model-driven development (MDD) for cyber-physical systems leverages sophisticated tool chains to translate models into programming language artifacts. As minuscule model changes can entail severe changes to generated artifacts and their integration with handcrafted artifacts, evolving models can produce unforeseen challenges. Reporting outdated handcrafted artifacts, generated artifacts with missing handcrafted artifacts, and modified artifacts, supports developers in comprehending the effect of model changes and can guide evolving related programming language artifacts. Current work on artifact modeling in MDD processes supports modeling of relations between artifacts only, which prevents expressing that artifacts are generated from model parts contained by artifacts. We present an artifact model that supports this granularity. Based on such an artifact model and artifact data extractors we present a method to determine the impact of model evolution on generated and handcrafted code. This method is realized with the MontiCore language workbench and enables developers to identify the effect of model changes. Ultimately, it facilitates model evolution in MDD for cyber-physical systems.

Keywords: Model-Driven Development; Impact Analysis; Artifact Model

1 Introduction

Applying model-driven development during the development of heterogeneous cyber-physical systems (CPS), e. g. as part of collaborative systems, is one possibility to control and manage the complexity of such systems. Model-driven development (MDD) [Völ+13] lifts models to primary development artifacts. These models can be more abstract, better comprehensible, and better suitable to automated processing than general-purpose programming language (GPL) artifacts. To create software from models, these usually are translated using model-to-model or model-to-text transformations. In both cases, MDD tool chains can translate one model to an arbitrary number of artifacts of different kinds (e. g. GPL artifacts, documentation, configurations).

Where the intended system interacts with GPL software, such as drivers for sensors or actuators of cyber-physical systems, the abstraction of models usually requires bridging the gap between system specification and system implementation [FR07] manually by integrating

[1] Software Engineering, RWTH Aachen University, Aachen, Germany. <lastname>@se-rwth.de
This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

handcrafted GPL artifacts with GPL artifacts generated from the models [Gre+15]. Evolving models can invalidate the relations between handcrafted and generated artifacts, e. g. handcrafted artifacts may reference missing formerly generated artifacts derived from model parts that have been removed during evolution or vice verse. Consequently, model evolution can produce comprehensive target system evolution efforts and gaining an overview of its effects can be complicated.

We propose to support developers in model evolution by automatically providing reports of artifacts relevant to evolution with respect to the specific MDD tool chain. To this end, we propose a method that shepherds model evolution through reporting impacted generated and handcrafted artifacts to the developer, hence reducing the effort of comprehending the evolution effects. To achieve this, we (1) explicate the relations between various kinds of artifacts of an MDD tool chain through an artifact model; (2) derive instances of this model from the file system automatically; and (3) compare these for code generated from models before and after evolution to derive impacted artifacts. This paper consequently contributes a method to prepare MDD tool chains and derive sets of affected artifacts based on which the developers can comprehend required changes to the generated system more efficiently. This method is realized with the MontiCore language workbench and illustrated through evolution of a MontiArc software architecture model. To this end, Sec. 2 describes preliminaries, before Sec. 3 exemplifies the benefits of the presented method. Sec. 4 introduces the method and illustrates its application. Sec. 5 discusses it and debates related work. Sec. 6 concludes.

2 Preliminaries

Our method to explicate artifact relations in MDD processes relies on the artifact model [But+17] and is realized with the MontiCore language workbench [Hab+15; KRV10]. Throughout the paper we use models of the MontiArc [HRR12; Rin+15] to illustrates its application. This section introduces all three.

2.1 The Artifact Model

Our method to explicate artifact relations in MDD processes relies on the artifact model [But+17] and is realized with the MontiCore language workbench [Hab+15; KRV10]. Throughout the paper we use models of MontiArc [HRR12; Rin+15] to illustrates its application. This section introduces all three.

Typical MDD projects require a multitude of different artifacts that address the different domains' concerns and conform to different languages. Managing the complexity of these projects requires understanding the relations between artifacts, which entails understanding the relations between their languages as well as between the tools producing and processing

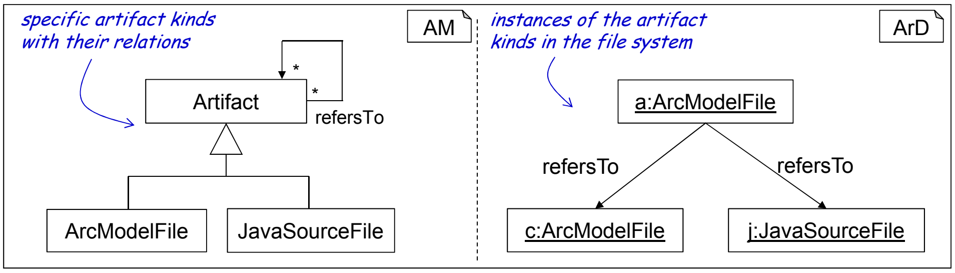

Fig. 1: Exemplary artifact model (left) with corresponding artifact data (right).

such artifacts. To this end, reifying this information as a first-level modeling concern in form of an explicit artifact model (AM) [But+17] helps to cope with its complexity. Such an AM precisely specifies the kinds of artifacts, tools, languages, and relations involved in an MDD project and thus represents the project in a structured and machine processable way. An AM describes all different situations in terms of present artifacts and relations that could arise during their lifetime. The current situation of the project can be inspected by automatically extracting artifact data (ArD) from the project conforming to the artifact models' entities and relations. This is displayed in Fig. 1 by example. The left side displays an excerpt of an AM realized as a class diagram. In the example AM an artifact can be a model file or a Java source file. Moreover, the association models that artifacts can refer to each other. The object diagram on the right side depicts the data corresponding to the artifact model ontologically, i. e. it represents an instance of the AM at a specific point in time. In this case, the model file `a` refers to another model file `c` as well as the Java source file `j`.

2.2 The MontiCore Language Workbench

MontiCore [Hab+15; KRV10] is a language workbench for the development of external, textual domain-specific languages (DSLs). A MontiCore grammar, which is a context-free grammar in an extended EBNF shape, describes the syntax of a DSL. From this, MontiCore generates language-processing infrastructure such as an abstract syntax data structure and a parser. For the definition of restrictions not expressible with context-free grammars, e. g. uniqueness of names, MontiCore supports to restrict the syntax of a language by adding well-formedness rules in form of Java context condition classes. Parsed models can be translated into target language artifacts by an employed template-based code generation engine based on FreeMarker[2]. The workflow of processing a model can be configured using Groovy scripts. Typically, the workflow starts with parsing the model to obtain an instance of the abstract syntax data structure, the abstract syntax tree (AST). Afterwards, the abstract syntax can be minimized to include only the language's essential structure. Further, context

[2] The FreeMarker Website: `https://freemarker.apache.org/`

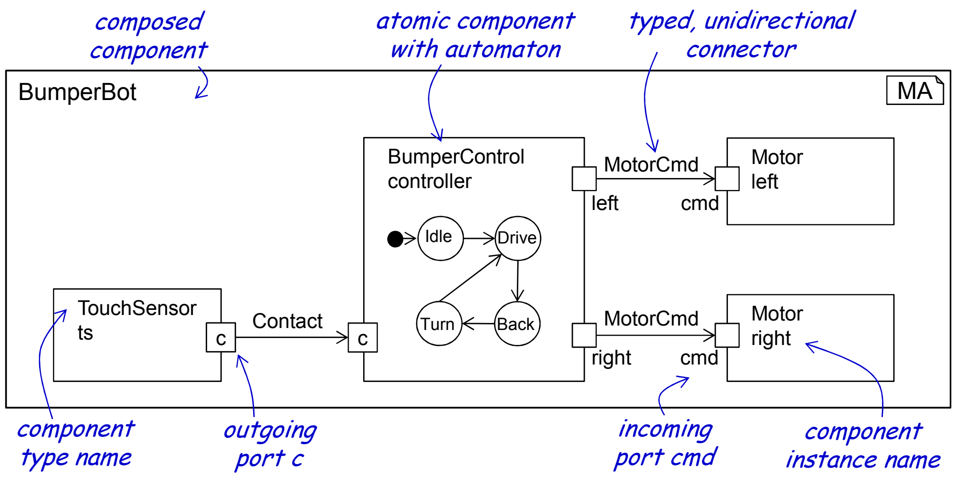

Fig. 2: Exemplary MontiArc software architecture of a CPS BumperBot.

condition are checked by a generated visitor infrastructure for the AST. The last step is to translate the parsed model into target language artifacts.

2.3 The MontiArc Architecture Description Language

MontiArc [HRR12; Rin+15] is a component & connector architecture description language (C&C ADL) realized with MontiCore. MontiArc enables describing component models with typed, directed communication interfaces (ports) and connectors connecting pairs of components via their ports. The data types sent and received via ports are defined in a class diagram model. The communication between components is realized as message exchange following semantics based of the FOCUS formalism [BS01]. MontiArc components may be hierarchically structured via introducing interconnected component instances as subcomponents. Components without subcomponents (atomic components) specify their behavior with embedded behavior models, such as automata. To this effect, MontiArc has an extension mechanism to include new behavior languages. MontiArc includes a code generator to translate the models into executable source code that can be deployed, e. g. to cyber-physical systems.

An exemplary MontiArc component model is depicted in Fig. 2. It models a robot that drives forward until it touches a wall, then drives backwards for a short period of time, turns around and proceeds to drive forward. The component `BumperBot` includes four subcomponent instances - each one of the types `TouchSensor` and `BumperControl` and two of type `Motor`. The component of type `TouchSensor`, e. g. contains a single outgoing port `c` of type `Contact`. The component behavior of the `controller` is defined by an embedded automaton. This automaton contains four states and four transitions. The

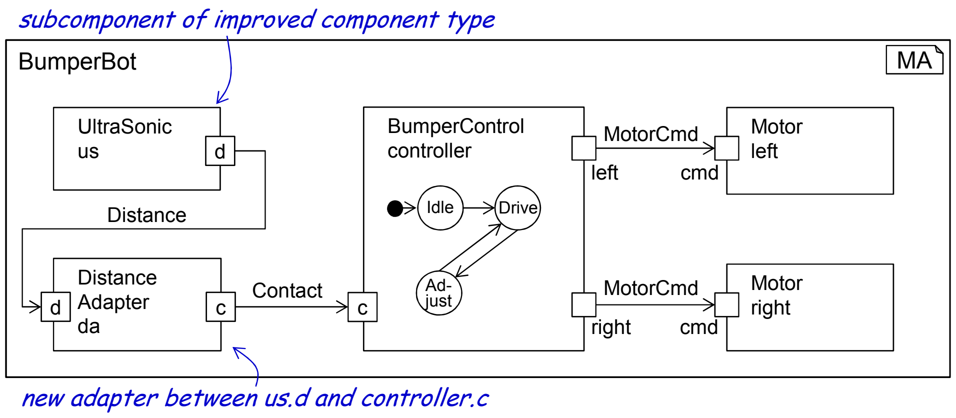

Fig. 3: Updated version of the `BumperBot` depicted in Fig. 2.

transition conditions are formulated against valuations of incoming ports and transition actions trigger messages on outgoing ports. From each component, the MontiArc Java code generator produces a Java class implementing the interface of the component (its name and ports), another Java class for the behavior (e. g. defined by an automaton model), a class capturing the input port valuations of a component and a class capturing the output port valuations. These classes interact with a run-time environment that realizes the message passing semantics and with classes generated from type definition class diagrams. The behavior of atomic components that do not contain a behavior model can be specified via handcrafted implementations. To this effect, MontiArc leverages the generation gap pattern [Gre+15] to integrate such handcrafted artifacts with generated code.

3 Example

Models and handcrafted artifacts of MDD projects evolve over time, which may be due to changing requirements, ongoing refinement, or refactoring. These artifacts are related to each other in different ways: for instance, artifacts can reference other artifacts (e. g. importing), artifacts can be created from other artifacts (e. g. code generation), or handcrafted artifacts impose requirements on other artifacts (e. g. tests for generated artifacts). All of these may change during model evolution.

Consider the evolution of the `BumperBot` architecture model depicted in Fig. 3, which has been changed as follows: (1) Due to a change in requirements, the system shall no longer recognize walls by contact, but instead detect walls via an ultrasonic sensor. To meet this new requirement, the subcomponent `TouchSensor` is exchanged with a new component `UltraSonic`. As the ultrasonic sensor outputs an Integer distance value and the `BumperControl` component expects a Boolean value indicating a contact, an additional component `DistanceAdapter` translates low distance values into a “contact”.

(2) The behavior automaton of the component `BumperControl` is modified by merging the states `Back` and `Turn` into the state `Adjust`. How these changes impact the effort of adjusting the handcrafted artifacts after evolution cannot be estimated with little effort as both model changes affect the generated artifacts, and hence, also to handcrafted code that interacts with the modified generated artifacts. For instance, artifacts generated from `TouchSensor` are not part of the project anymore, and therefore related tests are obsolete; the new components may not be tested at all and require additional efforts investigation; and the behavior automaton has changed, which might impact other artifacts as well. Making the artifact relations of a MDD tool chain explicit can support developers in understanding which artifacts are (potentially) affected by the model evolution and hence shepherd the evolution efforts.

4 Estimating Model Evolution Impact

This section presents our method to identify artifacts potentially affected by modifications of model artifacts and the preparation of an MDD tool chain for the collection of artifact data.

4.1 Reifying the Model-Driven Tool Chain

To apply our method in a concrete project, some preparing activities are necessary. These activities are depicted in Fig. 4 by an activity diagram. First, the artifact types and possible relations have to be identified. Artifact types can be distinguished based on file types, but further distinction based on project related purposes, e. g. separation into test and functionality implementation artifacts, is also conceivable. Relations can be different kinds of dependencies between generated or handcrafted source code artifacts, relations between model elements, or mappings between model elements and generated artifacts they contribute to. Sophisticated project knowledge helps to identify a project's relevant artifact types and relations. Once these are identified, the artifact model can be created. Afterwards, the artifact data extractors need to be developed. Extractors investigate the current state of project artifacts and create instances of artifact model types to produce artifact data conforming to the artifact model. They can be realized as data loggers capturing dynamic relations during the generation process, e. g. relations between model elements and generated artifacts they contribute to, or analyze the file system to gather static dependencies between artifacts. A part of the artifact model used in our running example is depicted in Fig. 5. Each MontiArc model is stored in its own artifact called `ArcModelFile` and consists of a number of MontiArc model elements. A MontiArc model element is described through its own artifact `ArcElement` and can be, among others, a component or automaton, which are represented in the artifact model by the artifacts `Component` and `FSMachine`, respectively. Java source files that are generated from the MontiArc tool chain can be represented in the artifact model by artifacts of type `JavaSourceFile`. A model element can contribute

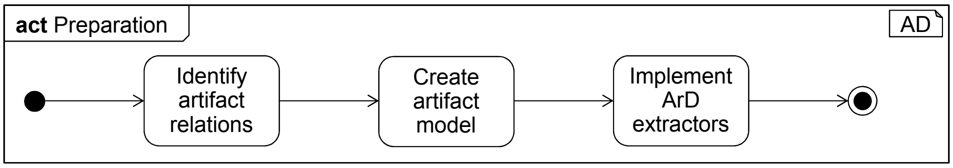

Fig. 4: Preparing activities to utilize an artifact model for impact analysis.

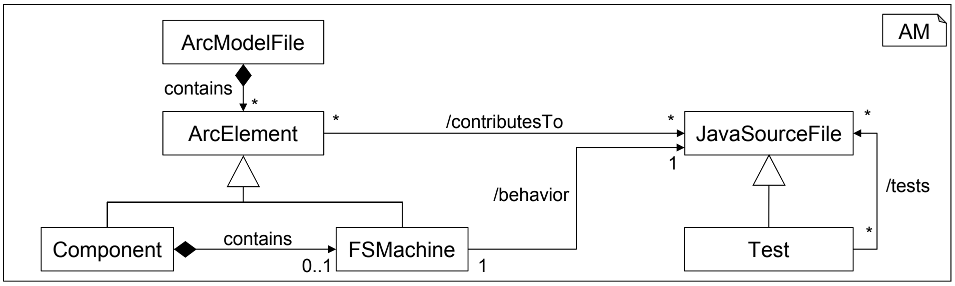

Fig. 5: Part of the artifact model used in our example.

to a number of Java source files and multiple model elements can contribute to the same Java source file. Java source files that test other Java source files are represented in the artifact model by the artifact `Test`. A test can test other Java source files while a Java source file can be tested by multiple test artifacts, which is indicated by the relation `tests`. In order to precisely define the derived associations `contributesTo`, `behavior` and `test`, we use the OCL/P [Rum11]. Based on the desired level of detail, the artifact type `JavaSourceFile` can be further divided into component, component input, component result, and component implementation artifacts. The first three are generated for each component, the latter has to be handcrafted if an atomic component does not contain an automaton.

4.2 Guiding Model Evolution

Once an artifact model and the required extractors have been created for an MDD project, they can be utilized to estimate the impact of changes in the project. The process we propose to identify differences between two versions of an MDD project is depicted in Fig. 6 by two activity diagrams, artifact differencing and model differencing. The result of the activities depicted in both diagrams are lists of artifacts that have to be considered by a developer when evolving the MDD project to its new version. These lists include generated artifacts that may exhibit new behavior, handcrafted artifacts that reference outdated generated artifacts, and new generated artifacts for which tests, documentation, or other handcrafted artifacts need to be developed.

The starting point of artifact differencing is some version `t` of the MDD project to be further developed, including all generated and handcrafted artifacts. Utilizing the previously

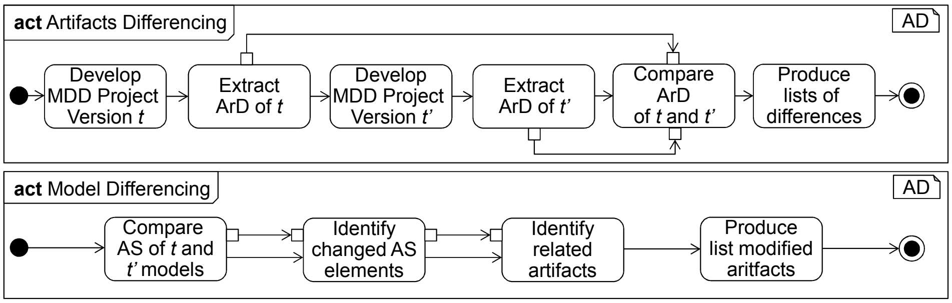
act Artifacts Differencing
AD
Develop MDD Project Version t
Extract ArD of t
Develop MDD Project Version t'
Extract ArD of t'
Compare ArD of t and t'
Produce lists of differences
act Model Differencing
AD
Compare AS of t and t' models
Identify changed AS elements
Identify related artifacts
Produce list modified aritfacts

Fig. 6: Activities involved in investigating the impact of changes.

implemented extractors, the current state of the artifacts under investigation has to be captured, in form of artifact data conforming to the used artifact model. This activity is usually automated. Once the artifact data has been extracted, the evolution to the next version `t'` can be initialized by adapting models, e. g. to new requirements and specifications, and by re-executing the generator for the modified models. The result of this step is an artifact landscape, again including models and generated artifacts according to the new version of the project, and (potentially outdated) handcrafted code artifacts. Utilizing the implemented extractors the artifact data of `t'` is extracted. The artifact data of the two versions can then be compared to identify differences between the two artifact landscapes. From this comparison we can identify (1) new artifacts for which handcrafted artifacts, such as functionality implementation code, tests, or documentation, do not yet exist, (2) handcrafted artifacts that reference generated artifacts no longer available, and (3) handcrafted code that interacts with generated code that differs in both versions. For model differencing we compare the abstract syntax (AS) of the models of the two project versions to identify model elements that have been modified, added, or deleted. Using the extracted artifact data we can identify related generated artifacts those model elements contribute to. As a result we obtain a list of generated artifacts that may have changed through modifications in the respective model. Handcrafted artifacts that reference these generated artifacts need to be reviewed by a developer and may need to be adapted to changes in the generated artifacts. Applying the presented processes to our example we can identify those artifacts that need be considered when evolving the `BumperBot` to its new version. By comparing the artifact data of the two versions, we can identify that the generated Java classes for the `TouchSensor` are missing in the new version and that an handcrafted artifact `TouchSensorImpl` previously referenced one of these artifacts. Further, we can identify that tests and documentation artifacts refer to this implementation artifact. We can therefore conclude that these artifacts are outdated and need to be modified or deleted. Comparing the artifact data of the two versions, we can also identify that new component artifacts for `UltraSonic` and `DistanceAdapter` were added to the project. Based on our project knowledge and the artifact model we can identify those artifacts, besides test and documentation, that should possibly be developed for these components, such as component implementation artifacts. Through the activities of model differencing, we can identify that the automaton of the component `BumperControl` has been modified. Since this model element contributes to the `BumperControlImpl` artifact, we can conclude that the behavior of this artifact may have changed and therefore artifacts with relations to the `BumperControlImpl` need to be reviewed and possibly modified.

5 Discussion and Related Work

In MDD projects, proper model management is crucial when working with large collections of models – even more, if impact analysis is performed on these. To improve the model management, [BJV04] introduces the notion of megamodels. Megamodels are still subject to ongoing research [Sal+16; Sim+15]. Under the assumption that everything is a model [Béz05], the notion of artifact models as proposed in this work is a megamodel, too. However, there is a difference between megamodels and the proposed artifact model, or artifact models in general, from our viewpoint. Most importantly, we require formal encoding of models and their relations. Moreover, the elements of megamodels represent models and the links represent relationships between models [Sal+16]. Using an AM, the focus is on the model-driven build process including a white-box view of the MDD tools, such as a code generator. As a consequence, using an AM allows for a detailed modeling of the insides of an artifact and its relationships based in these. In general, the quality of the proposed approach relies on the level of detail of the artifact model. The finer-grained the change can be analyzed, the more precise is the measurement of its impact. For example, if the artifact model of a MontiArc tool chain does not distinguish between generated classes that contain the input, the output, the behavior, or the structure of a component, the estimated effect of a change in a component model may not distinguish between those as well. Our approach considers syntactic changes in models only. Currently, syntactic changes that do not modify the behavior, e. g. through refactoring, are not distinguished between modifications that affect the behavior. Semantic differencing can be used to investigate this further.

6 Conclusion

We presented a method to facilitate shepherding model evolution for sophisticated MDD tool chains, such as in the development of CPS. This method relies on making the artifact kinds and their relations of a given MDD tool chain explicit as its artifact model. Based on this, the state of artifacts can be extracted from the file system and is accessible for analyses, such as identifying changed artifacts, missing handcrafted artifacts for generated artifacts, and vice verse. Automating this identification of artifacts potentially relevant to evolution supports developers in estimating the impact of evolution and guides their adjusting efforts, which ultimately facilities model evolution.

References

[Béz05] Jean Bézivin. "On the Unification Power of Models". In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. "On the Need for Megamodels". In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004.

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

[But+17] Arvid Butting et al. "Taming the Complexity of Model-Driven Systems Engineering Projects". In: *Grand Challenges in Modeling 2017 (GRAND'17)*. 2017.

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

[Gre+15] Timo Greifenberg et al. "Integration of Handwritten and Generated Object-Oriented Code". In: *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*. Vol. 580. CCIS. Springer, 2015, pp. 112–132.

[Hab+15] Arne Haber et al. "Composition of Heterogeneous Modeling Languages". In: *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*. Vol. 580. CCIS. Springer, 2015, pp. 45–66.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernard Rumpe. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Tech. rep. RWTH Aachen, 2012.

[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)*. Vol. 12. 2010, pp. 353–372.

[Rin+15] Jan Oliver Ringert et al. "Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems". In: *Journal of Software Engineering for Robotics (JOSER)* 6.1 (2015), pp. 33–57.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. 2nd. Xpert.press. Springer Berlin, Sept. 2011.

[Sal+16] Rick Salay et al. "Heterogeneous Megamodel Slicing for Model Evolution". In: *ME@ MODELS*. 2016, pp. 50–59.

[Sim+15] Jocelyn Simmonds et al. "A megamodel for Software Process Line modeling and evolution". In: *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE. 2015, pp. 406–415.

[Völ+13] Markus Völter et al. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. Wiley, 2013. ISBN: 9781118725764.