Arvid Butting, RWTH Aachen University
Andreas Wortmann, RWTH Aachen University

11

[BW21] A. Butting, A. Wortmann:
Language Engineering for Heterogeneous Collaborative Embedded Systems.
In: Model-Based Engineering of Collaborative Embedded Systems, pp. 239–253, Springer, Jan. 2021.
www.se-rwth.de/publications/

Language Engineering for Heterogeneous Collaborative Embedded Systems

At the core of model-driven development (MDD) of collaborative embedded systems (CESs) are models that realize the different participating stakeholders' views of the systems. For CESs, these views contain various models to represent requirements, logical functions, collaboration functions, and technical realizations. To enable automated processing, these models must conform to modeling languages. Domain-specific languages (DSLs) that leverage concepts and terminology established by the stakeholders are key to their success. The variety of domains in which CESs are applied has led to a magnitude of different DSLs. These are manually engineered, composed, and customized for different applications, a process which is costly and error-prone. We present an approach for engineering independent language components and composing these using systematic composition operators. To support structured reuse of language components, we further present a methodology for building up product lines of such language components. This fosters engineering of collaborative embedded systems with modeling techniques tailored to each application.

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_11

11.1 Introduction

Collaborative embedded systems

Engineering collaborative embedded systems (CESs) and collaborative system groups (CSGs) usually demands the cooperation of experts from various domains with different backgrounds, methods, and solution paradigms that contribute to different viewpoints (e.g., requirements, functional, logical, or technical viewpoints) of the system [Pohl et al. 2012].

The need to translate domain-specific solution concepts into software artifacts introduces a conceptual gap between the experts' problem domains and the solution domain of software engineering. This gap can give rise to accidental complexities [France and Rumpe 2007] due to the mismatch of solving problem domain challenges with solution domain (programming) concepts.

Model-driven development

Model-driven development (MDD) [France and Rumpe 2007] is a software engineering paradigm that lifts models to the primary development artifacts. In contrast to program code, which reifies concepts of the solution domain, models can leverage domain-specific concepts and terminology to express concepts of the problem domain, which facilitates contribution by domain experts. Models can also be more abstract and leave implementation details to smart software engineering tools (e.g., model transformations or code generators).

To enable models to be processed automatically, they must conform to explicit modeling languages [Hölldobler et al. 2018]. Engineering modeling languages is a challenging endeavor due to the multitude of formalisms and technologies involved, such as (i) grammars [Hölldobler and Rumpe 2017] or metamodels [Eysholdt et al. 2009] to define the languages' syntax, (ii) the Object Constraint Language (OCL) [Cabot and Gogolla 2012] or programming languages to define their well-formedness, and (iii) code generators [Kelly and Tolvanen 2008] or model transformations [Mens and van Gorp 2006] to realize their semantics (in the sense of meaning [Harel and Rumpe 2004]). As "software languages are software too" [Favre 2005], they are also subject to all the challenges typical to complex software as well. And similar to general software engineering, reuse is also the key to the efficient engineering of modeling languages. This holds especially for engineering collaborative embedded systems under the contribution of domain experts through viewpoints that are realized via domain-specific languages.

Software language engineering

Software language engineering (SLE) [Hölldobler et al. 2018] is a field of research that investigates the engineering, maintenance, evolution, and reuse of software languages. Research in SLE has produced a variety of solutions for reusing languages and language parts. However, the approaches for reusing complete (comprising realizations of syntax and semantics) language parts are missing, which severely hampers modeling for CESs and CSGs.

To address this, we present a method for modularizing modeling languages as language components, composing these, and ultimately building product lines of modeling languages to increase the reuse of languages beyond clone-and-own [Dubinsky et al. 2013].

Example 11-1: A family of architecture description languages

Consider a company that develops software for various kinds of CESs that operate in a smart factory. The company employs an architecture description language (ADL) [Medvidovic and Taylor 2000] to develop software component models for the software architecture of the CESs. The different kinds of CESs yield particularities regarding their software architecture. For some systems, it should be possible to perform dynamic reconfiguration of their software architecture based on mode automata [Butting et al. 2017], while for other systems, this is not allowed due to security restrictions. Similarly, some systems support dynamic re-deployment of software components to other systems, while this is not intended for other systems. To reify this properly in the models, the company uses different variants of ADLs — that is, variants of logical and technical viewpoints [Pohl et al. 2012]. These variants have several common language concepts and share large parts of the code generators employed. Without proper language modularization and reuse, these language variants co-exist in the form of cloned-and-owned, monolithic software tools.

In the following, Section 11.2 introduces the MontiCore language workbench, which our solution builds upon. Section 11.3 then introduces our notion of language components, before Section 11.4 explains their composition. Section 11.5 explains how we leverage composable language components to structure language reuse through explicit variability models, which we employed in CrESt to develop variants [Butting et al. 2019] of the MontiArc ADL [Haber et al. 2012] tailored to the use cases of “Autonomous Transport Robots” and “Adaptable and Flexible Factory” (cf. Chapter 1). Section 11.6 concludes this chapter.

11.2 MontiCore

MontiCore [Hölldobler and Rumpe 2017] is a language workbench [Erdweg et al. 2015] that facilitates the engineering of compositional modeling languages. MontiCore languages are based on a context-free grammar (CFG) that defines the (concrete and abstract) syntax of the respective language to which its models must conform. MontiCore uses this CFG to generate a parser that can process models of that language, along with abstract syntax classes that can store the machine-processable representation of the models once they have been parsed.

Abstract syntax tree

After parsing, the models are translated into abstract syntax trees (ASTs) — that is, instances of the abstract syntax classes generated from the grammar. Using MontiCore's extensional function library, these models are checked for well-formedness and other properties, transformed, and ultimately translated into other models, reports, source code, or other target representations. All of these activities rely on MontiCore's modular visitors that process parts of the AST. Visitors

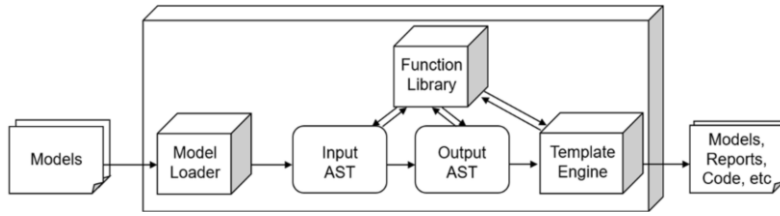


***Fig. 11–2**: The quintessential components of MontiCore's language processing tool chain support model loading, checking, and transformation*

[Gamma et al. 1995] separate operations on object structures from the object structures themselves and thus enable the addition of further operations without requiring modifications to the object structures.

Symbols

To facilitate operation on different nodes of the AST, MontiCore supports the definition of symbols—meaningfully abstracted model parts—based on grammar rules. Symbols are stored in symbol tables and can be resolved within a language as well as by other languages, enabling different forms of language composition.

Using CFGs and symbol tables, MontiCore supports the modular composition of languages through extension, embedding, and aggregation: language extension enables a CFG to extend another CFG, thereby inheriting all productions of the extended CFG. This process produces a new AST that may reuse productions of the extended CFG. This is useful, for example, for extending a base language in different ways with domain-specific extensions that would otherwise

convolute the base. Language embedding is the integration of selected productions of the client CFG into extension points of the host CFG. The resulting AST is the AST of the host CFG with a sub-AST of the client CFG embedded into selected nodes. This supports the creation of (incomplete) languages that provide an overall structure but demand (domain-specific) extension. Language aggregation is the integration of languages through references between their modeling elements. These references are resolved using MontiCore's symbol table framework and do not yield integrated ASTs. Instead, the models of the integrated languages remain separate artifacts. This supports, for example, the separation of different, yet integrated, concerns in models, such as structure and behavior.

For well-formedness checking and code generation, MontiCore provides generic infrastructures that can be customized by adding well-formedness rules (context conditions) and FreeMarker [Forsythe 2013] templates that define the code generation by processing the AST using template control structures and target language text. Consequently, a MontiCore language usually comprises a CFG, context conditions, and FreeMarker templates.

11.3 Language Components

Component-based software engineering is a paradigm for increasing software reusability by means of modularization. This paradigm is successfully applied in different domains and well suited for the engineering of embedded systems. The techniques of this paradigm can be applied to software languages as well. As a consequence, all advantages of component-based software engineering, such as increased reusability and better maintainability, can be leveraged to facilitate SLE. Similar to [Clark et al. 2015], we use the term *language component* for modular, composable software language realizations.

Definition 11-3*: Language component*

A language component is a reusable unit encapsulating a potentially incomplete language definition. A language definition comprises the realization of syntax and semantics of a (software) language.

This definition reduces the notion of language components to the constituents of the language infrastructure without being dependent on a specific technological space [Kurtev et al. 2002]. Ultimately, this means that a language component is a set of artifacts that form a

reusable unit. This set includes both handwritten as well as generated artifacts of language-processing tooling. For textual languages, it may include, for example, a grammar as a description of the syntax, the source code realizing well-formedness rules, a generated parser, and a generated AST data structure. In other technological spaces, a language component may contain a metamodel instead of a grammar and parser. Some language workbenches, such as MontiCore, enable language engineers to customize generated artifacts. Such handwritten customizations are part of a language component as well.

Extension points

Ideally, software components are black boxes whose internal workings are not relevant in their environment [McIlroy 1968]. Consequently, language components may also hide implementation details from their environment. To this end, language engineers can plan explicit extension points of a language component for which other language components can provide extensions. The realization of the extension points and extensions depends on the technological space used to realize the language components. In MontiCore, for example, syntax extension points can be realized through underspecification in grammars realized as interfaces or external productions [Hölldobler and Rumpe 2017]. Other language constituents, such as code generators, may yield different mechanisms for extension points and extensions.

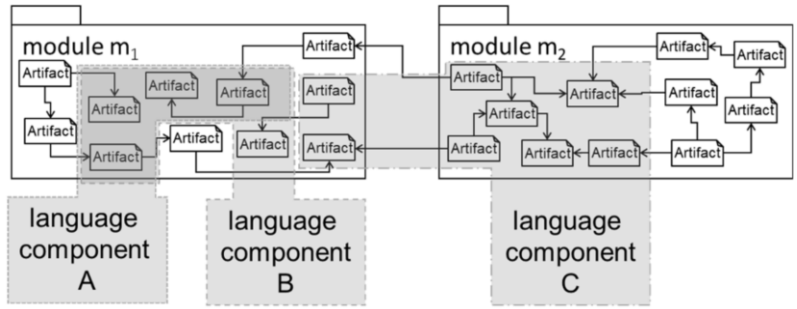


Fig. 11–4: *Artifacts of a language component can be distributed among software modules and some artifacts belong to multiple language components*

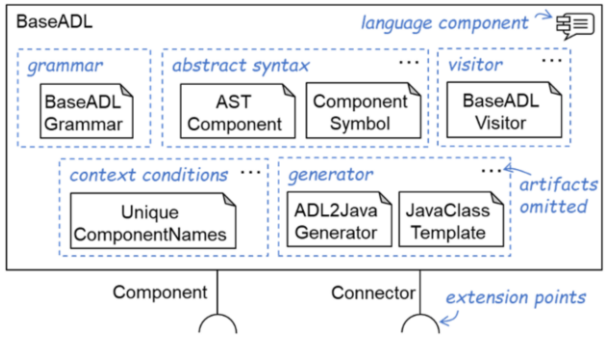
Artifact organization

A language component consists of many interrelated artifacts that may be distributed across different software modules and a single software module may contain artifacts for one or more language components (cf. Figure 11–4). This is due to the fact that the modularization of software into modules is typically driven by build tools (e.g., Maven or Gradle) that intend a different level of granularity.

Furthermore, an artifact may be part of multiple language components.

Example 11-5: BaseADL language component in MontiCore

The BaseADL language component contains a context-free grammar to describe the concrete and abstract syntax of a basic architecture description language (ADL). From this grammar, MontiCore generates a set of AST and symbol table classes that represent the abstract syntax data structure, a parser, a visitor infrastructure, and an infrastructure for realizing and checking context conditions. The handwritten context conditions, code generator classes, and templates are part of the language component as well.



In this example, the language engineers have planned two extension points for the BaseADL language component. One extension point can be extended to introduce a new notation for components and another one to introduce a new kind of connector. The extension point for components, for example, can be extended to add dynamic components that contain a mode automaton (cf. Example 11-1).

To identify, analyze, compose, and distribute language components, the large number of source code artifacts that realize the language component have to be extracted from the software modules. The constituents of a language component can be described and typed through a suitable *artifact model* [Butting et al. 2018b]. This produces the opportunity to identify the constituents of a language component by means of an artifact data extractor in a semi-automated process. This process collects potential artifacts of a language component, starting with a central artifact such as a grammar or a metamodel. With an underlying artifact model, an artifact data extractor can extract all associations from this artifact to other artifacts. For instance, in the technological space of MontiCore, this automated

extraction handles the identification of all Java classes that realize context conditions that can be checked against abstract syntax classes generated from a grammar.

However, the result of this automatic extraction (1) can produce artifacts that are not intended to be part of a language component or (2) can lack artifacts intended to be part of the language component. Therefore, handwritten adjustments of this result must be considered. In other technological spaces, these data extractors must be provided accordingly.

11.4 Language Component Composition

Forms of language composition

In general, the engineering of language components as described in Section 11.3 is the basis for building languages by composing language components. There are various forms of language

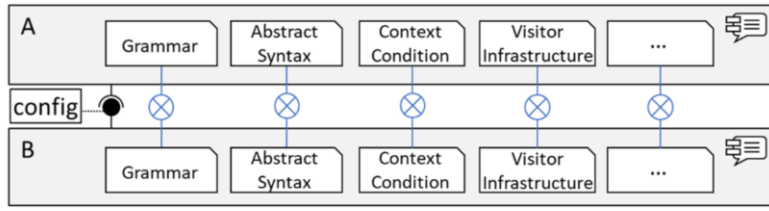


Fig. 11–6: *Composing two language components A and B requires composition of their constituents*

composition [Erdweg et al. 2012] that are supported by different language workbenches [Méndez-Acuña et al. 2016]. Some forms of language composition produce composed languages that can process integrated model artifacts, while other forms—such as language aggregation—integrate languages whose models remain in individual artifacts. Certain kinds of language composition—for example, language extension and language inheritance—require that one language depends on another language. These forms are not suitable for independent engineering of the participating languages and, when applied to language components, may introduce dependencies to the language component context. Some forms of language composition also require configuration with integration "glue," such as adapters between two kinds of symbols [Nazari 2017]. Therefore, care must be taken to select a suitable form of language composition.

Language component composition operators

For the composition of language components, we generalize the concrete form of language composition and denote that each composition of two language components is specified through a

configuration, as depicted in Figure 11–6. The configuration connects an extension point of a language component with an extension of another language component and states which form of composition has to be applied. Depending on the form of composition, the composition may also have to be configured with glue code. The actual composition of two language components is realized through the composition of their constituents. To this end, composition operators must be defined for each kind of constituent individually.

For example, MontiCore enables the composition of language components through embedding. The actual embedding has to be performed for handwritten constituents—such as grammars, context conditions, and generators—but also for generated constituents such as the AST data structures, the symbol table, and the visitor infrastructure. Thus, for all these constituents, an individual composition operator that realizes the embedding must be defined.

Composing grammars

MontiCore enables grammars to inherit from one or more other grammars. If a grammar inherits from another (super-)grammar, it can reuse and, optionally, extend or override the productions of the super-grammar. This influences the syntax through the generated parser and the integrated AST infrastructure, but also affects many other parts of the language-processing infrastructure generated from a grammar. Multi-inheritance in grammars can be used to compose two independently developed grammars and through this, realize language embedding. Therefore, the composition operator for embedding a MontiCore grammar into another MontiCore grammar produces a new grammar that inherits from both source grammars [Butting et al. 2019]. Furthermore, a grammar production integrating extension point and extension are generated, depending on the kind of syntax extension point (e.g., an interface production) and the kind of extension (e.g., a parser production).

Composing context conditions

In the context of language composition, we distinguish between *intra-language* and *inter-language* context conditions. Intra-language context conditions check the well-formedness of the syntax of a single language component, while inter-language context conditions affect syntax elements of more than one language component. Intra-language context conditions are part of a language component, whereas we regard inter-language context conditions as part of the configuration of the composition. Context conditions in MontiCore are evaluated against the abstract syntax by means of a visitor. To this end, composing context conditions of different language components requires the composition of the underlying visitor infrastructures. This is realized via inheritance and delegator visitors [Heim et al.

2016]. Once the visitors are integrated, the context conditions can be checked against the integrated structure.

Composing code generators

Code generators are commonly used for translating models into implementations that can be executed on embedded systems. However, few techniques for the composition of code generators exist, and these rarely enable composition of independent code generators. Code generator composition is challenging, as the result of the composition should produce correct code. While this is generally impossible, we can support language engineers in developing code generators that produce code that is structurally compatible with code generated by other code generators [Butting et al. 2018a]. This is realized by requiring each generator to indicate an *artifact interface* to which the generated code conforms. An adapter resolves potential conflicts between the artifact interfaces of two different code generators.

A further challenge in code generator composition is the coordination of the code generator execution. For some forms of composition, such as language embedding, code generators have to exchange information and thus comply with each other in a similar way to the generated code. To this end, generators provide *generator interfaces* to which the code generators conform. Again, potential conflicts between two code generators that are to be composed are resolved via adapters.

11.5 Language Product Lines

Reuse of languages or language parts is not only beneficial for language engineers due to the decreasing development cost and the increase in the language tooling quality, but also for language users, as the accidental complexity [Brooks 1987] posed by the effort of learning the syntax of new languages is reduced. In the context of engineering CESs and CSGs, language product lines are very applicable. Despite the variety in fields of application for which CESs and CSGs are employed, their model-driven engineering often relies on the same general-purpose modeling languages (e.g., UML) to describe aspects such as the geometry of physical entities of CESs, their system functions, collaboration functions, their communication paradigms, architectures, goals, capabilities, and much more.

This raises a gap between the problems in the application domain and the ability to express these in the modeling languages in a compact and understandable way. Enriching general purpose

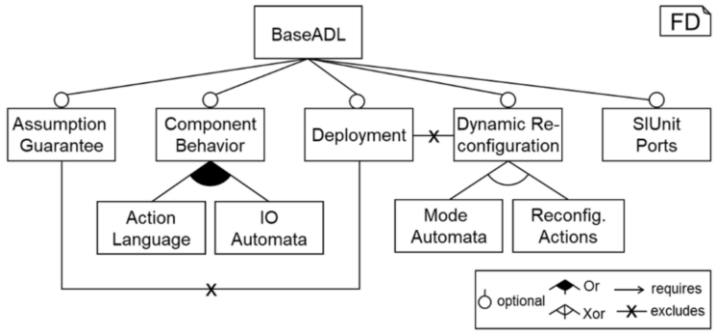
modeling language with application domain-specific language concepts helps to bridge this gap. Modular language engineering in terms of developing language components as presented in Section 11.3 and composing these as presented in Section 11.4 can be used to realize product lines of languages [Butting et al. 2019]. Such *language product lines* enable systematic reuse of language components for a family of similar languages and, therefore, enable individual tailoring of the modeling languages to the application fields of CESs and CSGs.

Modeling language product lines

The variability of the language product line in terms of language features is modeled as a *feature diagram*, where language features are realized as language components. Therefore, a *binding* of the product line connects features with the language components that realize them. Furthermore, the binding configures the pairwise language component compositions that occur in all products of the language product line.

Example 11-7: MyADL language product line

The company developing CESs described in Example 11-1 can employ a language product line for their ADLs to eliminate clones of redundant language parts and the resulting effort in maintaining and evolving these individually. All ADL variants have a common base language, and different combinations of extensions to this base language are considered in the product line. The optional behavior of software components can be modeled via input-output automata, an action language, or both. Some application scenarios benefit from using SI units as data types for messages sent via ports.



A product of the product line is specified via a *feature configuration*. The language components of all selected features are composed in pairs, as specified via the binding. The result of composition is a language component. Derivation of languages from

the product line is automated, but the resulting language component can be customized manually (optional). Engineering reusable language components and using these within language product lines fosters separation of concerns among different roles, as depicted in Figure 11–8.

❑ *Language engineers* develop language components and their extension points independently of one another. The artifacts of a language component are identified and collected via an artifact data extractor.

❑ A *product line manager* selects suitable language components for a field of application scenarios, arranges these in the form of a feature model, and configures the composition of the language components in a binding.

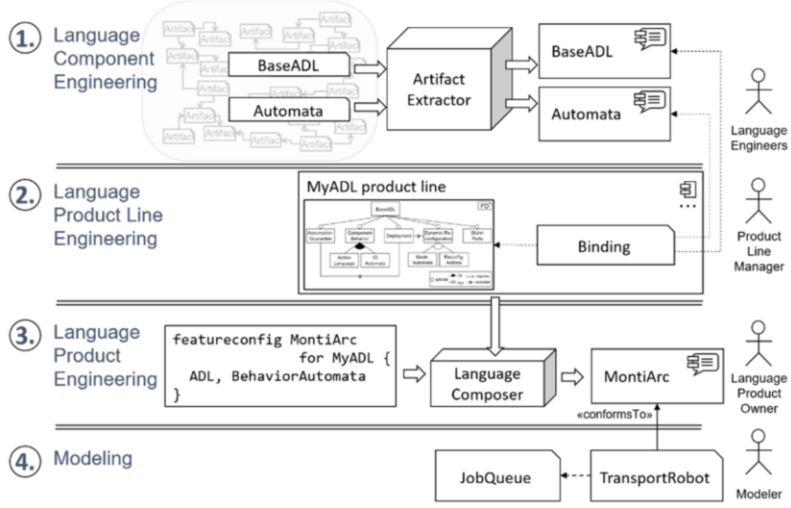


Fig. 11–8: *Processes and stakeholders involved in engineering language product lines*

❑ A *language product owner* selects features of a language product line that are useful for a concrete application and, on a pushbutton basis, can use generated language-processing tools for this language. The generated tooling can be customized (optional). In Figure 11–8, the language product is an ADL with the name "MontiArc."

❑ A *modeler* uses a language product through the generated language-processing tools without being aware of the language product line — for instance, to model specific system functions or collaboration functions of collaborative transport robot systems.

11.6 Conclusion

We have presented concepts for composing modeling languages from tried-and-tested language components. Leveraging these concepts facilitates engineering of the most suitable domain-specific languages for the different stakeholders involved in systems engineering. This mitigates an important barrier in the model-driven development of CESs and CSGs. Future research should encompass generalization of language composition beyond technical spaces and support for language evolution.

11.7 Literature

[Brooks 1987] F. P. Brooks, Jr.: No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer (20:4), 1987, pp 10-19.

[Butting et al. 2017] A. Butting, R. Heim, O. Kautz, J. O. Ringert, B. Rumpe, A. Wortmann: A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In: Proceedings of MODELS 2017. Workshop ModComp, CEUR 2019, 2017.

[Butting et al. 2018a] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Modeling Language Variability with Reusable Language Components, In: International Conference on Systems and Software Product Line (SPLC'18), 2018, ACM.

[Butting et al. 2018b] A. Butting, T. Greifenberg, B. Rumpe, A. Wortmann: On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In: Software Technologies: Applications and Foundations, Springer, 2018, pp. 146-153.

[Butting et al. 2019] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann: Systematic Composition of Independent Language Features. In: Journal of Systems and Software, 152, 2019, pp. 50-69.

[Cabot and Gogolla 2012] J. Cabot, M. Gogolla: Object Constraint Language (OCL): A Definitive Guide. In: International School on Formal Methods for the Design of Computer, Communication and Software Systems, Springer, Berlin, Heidelberg, 2012, pp. 58-90.

[Clark et al. 2015] T. Clark, M. v. d. Brand, B. Combemale, B. Rumpe: Conceptual Model of the Globalization for Domain-Specific Languages. In: Globalizing Domain-Specific Languages (Dagstuhl Seminar), LNCS 9400, Springer, 2015, pp. 7-20.

[Dubinsky et al. 2013] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, K. Czarnecki: An Exploratory Study of Cloning in Industrial Software Product Lines. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, CSMR '13, Washington, DC, USA, 2013, pp. 25–34.

[Erdweg et al. 2012] S. Erdweg, P. G. Giarrusso, T. Rendel: Language Composition Untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, 2012, pp. 1-8.

[Erdweg et al. 2015] S. Erdweg et al.: Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. In: Computer Languages, Systems & Structures 44, 2015, pp. 24-47.

[Eysholdt et al. 2009] M. Eysholdt, S. Frey, W. Hasselbring: EMF Ecore based meta model evolution and model co-evolution. In: Softwaretechnik-Trends 29.2, 2009, pp. 20-21.

[Favre 2005] J. M. Favre: Languages Evolve Too! Changing the Software Time Scale. In: Eighth International Workshop on Principles of Software Evolution (IWPSE'05) IEEE, 2005, pp. 33-42.

[Forsythe 2013] C. Forsythe: Instant FreeMarker Starter. Packt Publishing Ltd, 2013.

[France and Rumpe 2007] R. France, B. Rumpe: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering 2007 at ICSE. Minneapolis, IEEE, 2007, pp. 37-54.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Haber et al. 2012] A. Haber, J. O. Ringert, B. Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, 2012.

[Harel and Rumpe 2004] D. Harel, B. Rumpe: Meaningful Modeling: What's the Semantics of "Semantics"?. In: IEEE Computer, Volume 37, No. 10, 2004, pp 64-72.

[Heim et al. 2016] R. Heim, P. Mir Seyed Nazari, B. Rumpe, A. Wortmann: Compositional Language Engineering using Generated, Extensible, Static Type-Safe Visitors. In: Conference on Modelling Foundations and Applications (ECMFA'16), LNCS 9764. Springer, July 2016, pp. 67–82.

[Hölldobler and Rumpe 2017] K. Hölldobler, B. Rumpe: MontiCore 5 Language Workbench Edition 2017. In: Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, 2017.

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

[Kelly and Tolvanen 2008] S. Kelly, J. P. Tolvanen: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, 2008.

[Kurtev et al. 2002] I. Kurtev, J. Bézivin, M. Aksit: Technological Spaces: An Initial Appraisal. In: 4th International Symposium on Distributed Objects and Applications (DOA), 2002.

[McIlroy 1968] M. D. McIlroy: Mass-Produced Software Components, Software Engineering Concepts and Techniques. NATO Conference on Software Engineering, Van Nostrand Reinhold, 1976, pp. 88-98.

[Medvidovic and Taylor 2000] N. Medvidovic, R. N. Taylor: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26.1, 2000, pp. 70-93.

[Méndez-Acuña et al. 2016] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, B. Baudry: Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. Computer Languages, Systems & Structures, 46, 2016, pp. 206-235.

[Mens and van Gorp 2006] T. Mens, P. Van Gorp: A Taxonomy of Model Transformation. Electronic notes in theoretical computer science 152, 2006, pp. 125-142.

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

[Pohl et al. 2012] K Pohl, H. Hönninger, R. Achatz, M. Broy (Eds.): Model-Based Engineering of Embedded Systems, Springer-Verlag, Berlin Heidelberg, 2012.