J. Michael, J. Pfeiffer, A. Wortmann (Hrsg.): Modellierung 2022 Satellite Events,
Digital Library, Gesellschaft für Informatik e.V. 32

Embedding Textual Languages in MagicDraw[1]

Florian Drux, Nico Jansen, Bernhard Rumpe, David Schmalzing [2]

Abstract: Stakeholders in model-based systems engineering projects often rely on heterogeneous modeling languages and tools. Efficient and seamless model-based engineering requires analyzing consistency, maintaining tracing information, and propagating changes of models of these languages even in different technological spaces. However, research on software language integration and inter-model consistency often only considers modeling languages and tools within the same technological space. We present a method for language composition across the technological spaces of the graphical modeling framework MagicDraw and the language workbench MontiCore. We realized language integration between these technological spaces by applying concepts of language aggregation to exchange essential model information and performing analysis on this information in an automated toolchain. The presented concepts can guide software language engineers and modeling tool developers on how to combine concepts of language composition to bridge technological spaces.

Keywords: language engineering; language composition; tool integration; model-based engineering

This work was supported by the German Ministry for Education and Research (BMBF) in the SpesML project (https://spesml.github.io/index.html/; grant number 01IS20092B).

1 Introduction

The development of large, complex, model-based systems [Ni15] necessitates the collaboration of experts from different fields, which may use a variety of modeling languages and tools. Managing intermodel consistency and supporting interpreting models together across the boundaries of technological spaces is crucial for seamless model-based engineering but costs a lot of effort and is prone to errors if not done automatically. Software language engineering [Kl08] has given rise to various language composition mechanisms [VV10, HRW18], of which language aggregation enables models of different languages to be interpreted together. However, these mechanisms are mostly constipated for modeling languages originating from the same technological space [GVM12], whereas domain experts may operate in different technological spaces. Furthermore, software languages and workbenches come in different shapes as they may be textual, graphical, or projectional.

Aggregating languages of different shapes and across technological spaces can facilitate automated and seamless model analyzes in model-based systems engineering projects.

[1] This work was supported by the German Ministry for Education and Research (BMBF) in the SpesML project (https://spesml.github.io/index.html/; grant number 01IS20092B).

[2] RWTH Aachen, Lehrstuhl für Software-Engineering, Ahornstraße 55, 52074 Aachen, Germany, {drux/jansen/rumpe/schmalzing}@se-rwth.de

We, therefore, investigate the challenges of integrating textual and graphical languages between different technological spaces at the example of the graphical modeling framework MagicDraw [3] and the language workbench MontiCore [4]. We do this by making essential model information explicit that can then be exchanged between languages. Using the example of embedding textual expressions into graphical state machines, we then show how this model information can be utilized in modeling tools to analyze inter-model conformity automatically. The presented results can guide language engineers on how to combine language embedding with language aggregation to systematically develop a shared understanding of models from different technological spaces.

In the remainder, we illustrate the challenges of language integration across technological spaces in section 2. We then present the background of MontiCore and MagicDraw in section 3 before we then present a systematic concept for language embedding in MagicDraw through essential model information in section 4. Next, we explore language embedding using the example of embedding textual expression languages into graphical state machines in MagicDraw in section 5. Finally, we present related work on language composition and tool integration in section 6 and discuss our solution in section 7.

2 Challenges

Integrating textual with graphical modeling languages across heterogeneous technological spaces poses several challenges. The different language constituents must be embedded appropriately, capable of interacting with each other and provided to the user in a convenient form. With respect to ISO 25010 [IS10], the norm for software product quality, the occurred challenges are related to functional suitability, compatibility, and usability.

Functional suitability refers to providing the required functionalities in the product. For embedding textual languages in graphical MagicDraw diagrams, this involves enabling the modeler to seamlessly define graphical and textual portions in the MagicDraw editor, which are automatically checked for validity. An integrated editing environment is essential to prevent the modeler from switching back and forth between multiple tools, which could disturb the overall modeling effort. Compatibility basically means the ability of a software system to interoperate with others. This is particularly important in our case, as the main task is to connect two different technological spaces. Thus, model elements of both spaces must be able to interact with each other, and well-formedness must be checked on the integrated (i.e., the entire) model. Usability refers to the applicability of a product to the end-user, considering usability, intuitiveness of functionalities, and general user interface. Accordingly, textual languages must be seamlessly integrable into graphical model elements. Furthermore, the modeler requires direct feedback on the well-formedness of the modeled elements and support for the modeling process itself.

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

This leads to the following overall challenges:

C1 Textual model elements must be editable and storable in MagicDraw's graphical editor.

C2 Textual model parts must be checkable for correctness with respect to the context of the graphical model components.

C3 Graphical model parts must be referencable via textual qualifiers.

C4 Textual model parts and corresponding functionalities must be embedded seamlessly into MagicDraw.

C5 The user requires direct feedback on modeled elements and modeling support.

Since this paper focuses on embedding textual languages into MagicDraw, functional suitability (C1) and compatibility (C2, C3) are mandatory goals. While usability (C4, C5) often cannot be determined objectively and requires user studies, we can at least qualitatively evaluate whether these challenges have been addressed appropriately.

3 Preliminaries

MontiCore [HKR21] is a workbench and a framework for engineering modular textual modeling languages from context-free grammars (CFGs), which define a language's concrete and abstract syntax. From a CFG, MontiCore generates the basic infrastructure for the engineering of modeling languages, which includes a parser, a symbol table, model consistency and model transformation infrastructure, and a common workflow. Language engineers can implement language specifics by filling provided extension points and extending the generated infrastructure. The common workflow of modeling languages engineered with MontiCore starts with parsing a textual model to construct its abstract syntax tree, deriving a model's symbol table, performing handcrafted context-condition checks, and serializing essential model information in exchangeable artifacts. To facilitate modular development, MontiCore provides syntax-focused mechanisms for composing language constituents. A provided library of literals, expressions, and types further facilitates language composition [Bu20].

MagicDraw is a graphical collaborative and extendable modeling tool that implements various standardized languages, including the UML, SysML, and BPMN. In MagicDraw, profiling enables language engineers to adapt predefined language profiles and create custom and specialized languages on top of the provided profiles. MagicDraw furthermore enables customization of modeling language-specific tools by supporting the development of custom plugins. MagicDraw provides an open development API whose code can be reused, extended, and modified, to facilitate profiling and plugin development.

4 Embedding Textual Languages

Besides the graphical modeling capabilities, MagicDraw offers a set of predefined textual languages. These include scripting languages such as Groovy or JavaScript and a specific OCL dialect. Additionally, MagicDraw supports structured expressions, which are assembled in a tree-like fashion via the editor and are encoded in XML. Nevertheless, the possibilities for textual modeling are severely limited. The existing textual languages have only rudimentary access to the elements of the containment tree in MagicDraw, and thus elements can often only be referenced very unintuitively via detours. Otherwise, there is only the possibility of inserting plain text. However, the resulting string is neither checked in itself nor concerning the context of other model elements. An integrated language requires a seamless connection between graphical and textual modeling. Furthermore, MagicDraw does not provide any direct hook points for mounting additional textual languages besides the existing ones.

Therefore, to embed an external, textual language into an otherwise graphical DSL in MagicDraw, it must be integrated in such a way that it: (1) accepts only the allowed set of syntactic sentences, and (2) these are validated for well-formedness concerning their context. To meet the first requirement, we have to integrate a parser for the textual language into MagicDraw. The basic idea is to use MagicDraw's plain text panels to enable textual modeling. For this purpose, we can either reuse existing plain text fields or create new fields for the MagicDraw language under development by creating them in the corresponding metamodel. In both cases, we have to bind a parser to the plain text field for validation. This parser can be provided externally by using appropriate technology to create textual domain-specific languages, such as MontiCore [HKR21]. Here we provide a context-free grammar that defines the concrete and abstract syntax of the textual language and automatically generates a parser for its models. MagicDraw allows adding customized validation rules for model elements. These rules are checked at modeling time and return an error message to the modeler in case of a violation. We exploit this extension point by specifying a validation rule for the designated plain text field for embedding the textual language. The input string is forwarded to the parser, which converts it to an AST, or in case of erroneous input, returns an error message, which in turn is displayed to the modeler.

The second requirement must be validated on the created AST (thus, after parsing the textual input), checking if the syntactically correct model is also well-formed. Therefore, the nodes of the textual model's AST must be checked against the other elements, both textual and graphical. The particular characteristic of the well-formedness rules is highly dependent on the language. Thus, in the following, we discuss the different types of validation rules, as well as a general approach for addressing these. Generally, these rules arise from known context conditions for modeling languages [HKR21] but pose a special challenge in the case of language embedding and aggregation across different technological spaces. First, basic direct checks exist on an AST node, such as whether the name of an element begins with an uppercase letter. Furthermore, it can be checked whether certain elements can be referenced in a textual model, such as variables or method calls. This also includes checking whether these elements are accessible and, e.g., in the case of methods, the signature of

the definition fits the signature applied in the usage. Finally, the types of the referenced elements must be compatible with each other.

While basic validation rules on a few AST nodes of the textual model can usually be implemented easily, references between model elements pose a major challenge, especially for references between the different technological spaces. To resolve these references, programming and textual modeling languages introduce the concept of the symbol table [HMSNR15, Ha15]. A symbol table is an infrastructure that stores uniquely named model elements (so-called symbols) and enriches them with additional information. It elevates the tree structure of the AST to a graph structure allowing for quick navigation and cross-referencing. To achieve referencing of graphical model elements in embedded textual language models, we extend this symbol table concept to the entire containment tree in MagicDraw. We traverse the entire containment tree and create a symbol for each model element uniquely identifiable by its qualified name. Here, we opt for the application of the Visitor Pattern, which is supported by in MagicDraw. For each relevant element, a corresponding symbol is created.

Depending on the element's type, we instantiate a symbol of a respective kind. Thus, the symbol for, e.g., an attribute is distinct from a method's as they need to provide different information and should be referenceable separately. The precise assignment of symbols is language-specific, although there are common candidates, such as (object-oriented) type definitions, fields, functions, or methods. After creating the symbols, they are bound to their model elements (e.g., via a map). Sophisticated language workbenches such as MontiCore follow a more integrated approach between AST nodes and symbols. In the case of MagicDraw, we have opted for a more straightforward binding mechanism for simplicity. Ultimately, the referenced symbols can be resolved, thus accessing the heterogeneous model elements throughout the entire containment tree. Well-formedness rules for referencing other model elements can be added via a validation rule in MagicDraw, similar to the parser.

Finally, the implemented validation rules can be packed into a MagicDraw plugin together with the customized language and the embedded parser. Since MagicDraw is based on the Java programming language, it is useful to rely on compatible technologies for parsing and symbol table creation.

5 Case Study

To evaluate our concept for tool interoperability, we extend MagicDraw's graphical state machines with textual expressions. Our goal is to use state machines to model the behavior of distributed systems. To this end, we want to interpret state machines together with internal block diagrams and block definitions. Unfortunately, while MagicDraw offers behavior descriptions, these don't fulfill the specific semantics we require. Using strongly typed expressions consisting of boolean operations, variable assignments, and method calls, we

want to enable modelers to define transition triggers and effects that can reason over port and variable values.

MagicDraw's API enables us to embed expressions into transition guards and effects on a syntactical level. Using an expression parser provided by MontiCore's library of literals, types, and expressions and attaching it to a plain text panel in MagicDraw's state machines, we can now parse plain text input and derive their abstract syntax tree. While this provides us with at least the most basic checks for ensuring correct concrete syntax, we can not yet check the well-formedness of expressions with respect to the availability of referenced symbols and type conformity; that is, we can not yet check inter-model well-formedness.

To enable well-formedness checks between models, we need to make the available system elements, ports, variables, and types known to the type system. However, here two problems arise. First, MagicDraw does not provide a straightforward way to export such information; secondly, and more importantly, type access differs significantly for graphical and textual languages. The graphical framework MagicDraw stores elements in a containment tree and enables access to elements through direct object inks, whereas textual languages manage access to symbols either through namespaces or scopes, with MontiCore supporting the latter. That is, symbols are available in certain scopes, and access to elements is provided through their fully qualified name. Not all symbols in scope are available from the outside, though, as scopes may only export certain symbol kinds. Furthermore, a symbol may shadow another symbol of the same name in another scope.

Using the above-presented concept for language aggregation, we make symbol information explicit to type check on expressions and thus bridge the gap between the technological spaces of MagicDraw and MontiCore. We do this by traversing the containment tree using a visitor that collects element information along the way. We then synthesize, aggregate, and serialize the symbol table of the stored model elements from this information. While the serialized symbol table functions as a pivot model between the two technological spaces, it is pruned down to the essential model information for conformity checks.

To integrate well-formedness and type checks provided by MontiCore, we rely on the MagicDraw's validation suite, which provides validation rules which can be applied to model elements to check conformity. Filtering model elements for textual elements of state machines, we can use these to check the conformity of the embedded expressions. First, the embedded parser receives the abstract syntax tree. Then, the type check is applied to the abstract syntax tree, consuming aggregated symbol information to check well-formedness. Usages of names must be available in the symbol table as names of ports and fields. Their symbols provide information about their type, visibility, and access rights. Symbol information is made visible to the scope of an expression through import statements, with the containment tree hierarchy of elements defining their fully qualified name. The type check then consumes this information to check the conformity of the left-hand and right-hand sides of expressions, or that expressions in transition guards evaluate to `Boolean`.

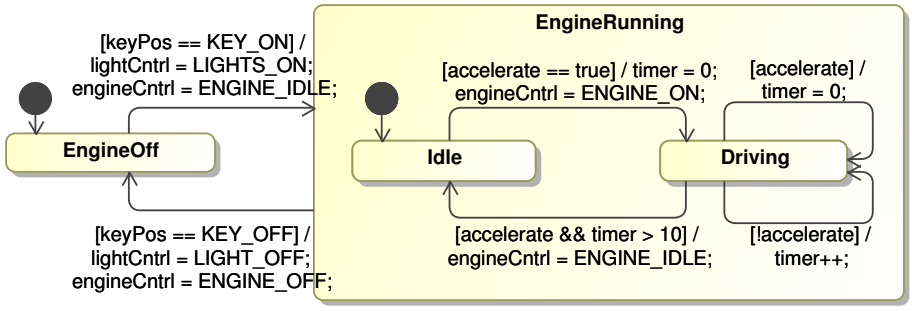

Fig. 1: A visual MagicDraw model of a car integrating textual model elements of our language both inside the rectangular brackets and after the slashes; The car may be turned on or off. When accelerating with the engine in its idle state the engine is turned on. After not accelerating for a while, the engine returns to its idle state.

Example: In Fig. 1 in the top left the guard `keyPos == KEY_ON` can be seen. When validating the model, the provided text is parsed to an AST with an equality expression containing two names. Afterward, referenced names are resolved in the symbol table. As both symbols have the same type, they are comparable, and thus the type of the expression is of type `Boolean`.

To report on ill-formed elements, we use the annotations-Application Programming Interface (API) provided by MagicDraw. Error messages are annotations on elements which are then displayed in the Graphical User Interface (GUI). By adding the annotations-API, MagicDraw will highlight annotated elements as erroneous. In addition, the errors are also listed in an additional window of GUI, which results from fetching the error output of the embedded expression language.

6 Related Work

Most related work on language composition only considers homogeneous technological spaces, e. g., languages developed in a common language workbench. MontiCore supports systematic, syntax-oriented reuse of individually developed language components through language aggregation, language embedding, and language inheritance [KRV10, Lo13, Ha15]. These mechanisms mostly enable language integration as defined in [EGR12]. Furthermore, language aggregation enables to combine independent modeling languages such that models of varying aspects defined in different artifacts can be interpreted together. Serialization and deserialization of model essentials provides a basic foundation for integration of languages of heterogeneous technological spaces. However, MontiCore only focuses on syntactic language composition and does not provide support for composing semantics or development environments.

The language workbench MPS [Vo13] supports language composition of concrete syntax with mechanisms similar to MontiCore, though they are called differently. In addition to syntax-oriented reuse of languages, MPS also addresses modularity of type systems, generators, and development environments. However, mechanisms that enable composition of modeling languages of heterogeneous technological spaces, such as language aggregation, are missing in MPS.

Another approach [Se17] describes integrating the textual modeling language Alf [Se14] into MagicDraw UML diagrams. As an action language, a modeler can use Alf code snippets to specify behavior. For instance, the operation body of a method could be written in Alf, or smaller snippets could be used in conjunction with graphical UML behavior diagrams (such as activity diagrams or state machines). In general, this concept is similar to our work. A textual language is also embedded in MagicDraw, which is evaluated context-sensitively with respect to the graphical model elements. The main difference lies in the actual application, as the described Alf plugin adds additional modeling capability and does not, as in our case, formalize existing model elements syntactically through a language. Furthermore, the article describes the features and benefits of the Alf plugin rather than providing a general concept for embedding textual languages in MagicDraw.

Generally related are various works in the area of tool integration. Highly heterogeneous models are linked together, and information is exchanged between them. For example, the FTG+PM framework [Mu12] defines model transformations as graph structures to automatically pass information between different representations. Furthermore, there are attempts to represent information of different models via knowledge base systems [Fe15], via which rules for the consistency of information can then be derived and inferred. Besides many other works describing the linking of information across heterogeneous models and tool landscapes [Br10, Ch15, ZLVH18, DDFV09, SWA10, Mi09, CMM09, Zi07, Ba10], these approaches are usually only loosely related to our solution. While generally, an aggregation of models and their information takes place, our approach describes a concrete embedding in the MagicDraw modeling platform. Consequently, elements do not have to be exchanged between different tools but are directly implemented together in an integrated environment.

7 Discussion

Our presented approach of embedding textual languages into graphical languages is based on the modeling framework MagicDraw. For designing and integrating the textual language, we rely on the MontiCore language workbench because it is based on the same technological space (i.e., Java) and highly fosters the use of a symbol management infrastructure, which is heavily used in our solution. The particular challenge in our work was not only to make a textual language accessible within a predominantly graphical modeling environment but also to integrate it seamlessly into the graphical model. This means that the model must be not only syntactically correct in itself but also well-formed under the context of the graphical

model elements. Table 1 presents the evaluation results of our approach concerning the initially identified challenges (cf. section 2).

Tab. 1: Fulfillment of embedding challenges for textual languages in MagicDraw (• = fulfilled, ⊙ = partially fulfilled, ∘ = not fulfilled)

Challenge	Description	Fulfilled
C1	Textual model elements must be editable and storable in MagicDraw's graphical editor.	•
C2	Textual model parts must be checkable for correctness with respect to the context of the graphical model components.	•
C3	Graphical model parts must be referencable via textual qualifiers.	⊙
C4	Textual model parts and corresponding functionalities must be embedded seamlessly into MagicDraw.	•
C5	The user requires direct feedback on modeled elements and modeling support.	⊙

Generally, our approach extends language composition techniques known from textual languages and applies these to the graphical modeling space of MagicDraw (**C1**). Using the symbol table to resolve elements by their qualified name enables the modeler to cross-reference elements over the boundaries of the textual model. By integrating the resolve mechanism into a custom validation suite, we can check the well-formedness of the textual elements automatically (**C2**). However, the overall referencing could be seen as incomplete from a graphical modeling point of view, as not all parts are uniquely accessible by their names (**C3**). In MagicDraw, all model elements are theoretically accessible independent of their name. Referencing is accomplished via internal identifiers enabling arbitrary cross-referencing. Therefore, both ways of referencing are not fully compatible, resulting in potential conflicts and ambiguities. However, since internal identifiers are unintuitive in textual modeling and we can usually assume that relevant model elements have meaningful (and uniquely qualified) names in graphical modeling as well, we decided to follow this approach. A potential extension could combine these concepts by presenting the textual representation to the user while simultaneously managing MagicDraw's identifiers in the symbol table.

Finally, we primarily concentrated on the general feasibility and interoperability of the languages' constituents in an integrated fashion (**C4**). Accordingly, there is still a considerable lack of editor functionalities for the embedded textual language, such as syntax highlighting or autocompletion. However, by integrating validation rules, we can provide instant feedback, which satisfies at least the most rudimentary requirements of this challenge (**C5**). The integration of these functionalities, for example, via the language server protocol [BK19], still needs to be explored. In this context, extending the integration regarding refactorings of the graphical elements (such as renaming or deletion) and automatically tracing these

changes into the textual model parts is important. Currently, the integrated validation rules would directly detect the results of such modifications but without automated adaptations.

Our solution demonstrates the integration of textual MontiCore languages into the predominantly graphical modeling environment of MagicDraw. While the approach is specific to both technological spaces, it also provides a foundation for generally integrating textual languages into graphical tools. First, although strongly established by MontiCore, the general concept for building a symbol table, which maps individual model elements to uniquely resolvable symbols, can be applied commonly. Moreover, the realization approach can also be considered generalizable as it is based on the prevalent visitor pattern. Here, we assume that most modeling tools manage the model internally in a tree-like construct, which immensely benefits the use of this pattern. Otherwise, the underlying technology stack and the supported extensibility of the tools used are crucial for the integrability of different languages. Our approach benefits heavily from the fact that MagicDraw supports Java extensions, and MontiCore can easily fill this hook point. However, for too diverse or hardly extensible tools, realizing such integration can be difficult.

8 Conclusion

We presented a concept for embedding textual modeling languages in graphical languages of MagicDraw, discussing different challenges for usability and interoperability. At its core, the required infrastructure to resolve textual references on graphical model elements and vice versa is elaborated. Additionally, we presented a case study, realizing our concept based on context-free grammars written in MontiCore. We integrated a textual language in consideration of mutual access to the different technological spaces by leveraging integrated symbol tables, thus bridging the gap not only between heterogeneous languages but also tools. Ultimately, our approach provides a contribution to fostering a seamless transition between graphical and textual modeling.

Bibliography

[Ba10] Baumgart, Andreas: A Common Meta-Model for the Interoperation of Tools With Heterogeneous Data Models. Citeseer, 2010.

[BK19] Bünder, Hendrik; Kuchen, Herbert: Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol. In: International Conference on Model-Driven Engineering and Software Development. Springer, pp. 225–245, 2019.

[Br10] Brecher, C; Özdemir, D; Feng, J; Herfs, W; Fayzullin, K; Hamadou, M; Müller, AW: Integration of Software Tools with Heterogeneous Data Structures in Production Plant Lifecycles. IFAC Proceedings Volumes, 43(4):48–53, 2010.

[Bu20] Butting, Arvid; Eikermann, Robert; Hölldobler, Katrin; Jansen, Nico; Rumpe, Bernhard; Wortmann, Andreas: A Library of Literals, Expressions, Types, and Statements for

Compositional Language Design. Special Issue dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology, 19(3):3:1–16, October 2020. Special Issue dedicated to Martin Gogolla on his 65th Birthday.

[Ch15] Chen, DeJiu; Maffei, Antonio; Ferreirar, João; Akillioglu, Hakan; Khabazzi, Mahmood R; Zhang, Xinhai: A Virtual Environment for the Management and Development of Cyber-Physical Manufacturing Systems. IFAC-PapersOnLine, 48(7):29–36, 2015.

[CMM09] Crnković, Ivica; Malavolta, Ivano; Muccini, Henry: A Model-Driven Engineering Framework for Component Models Interoperability. In: International Symposium on Component-Based Software Engineering. Springer, pp. 36–53, 2009.

[DDFV09] Didonet Del Fabro, Marcos; Valduriez, Patrick: Towards the Efficient Development of Model Transformations Using Model Weaving and Matching Transformations. Software & Systems Modeling, 8(3):305–324, 2009.

[EGR12] Erdweg, Sebastian; Giarrusso, Paolo G.; Rendel, Tillmann: Language Composition Untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. LDTA '12, Association for Computing Machinery, New York, NY, USA, 2012.

[Fe15] Feldmann, Stefan; Herzig, Sebastian JI; Kernschmidt, Konstantin; Wolfenstetter, Thomas; Kammerl, Daniel; Qamar, Ahsan; Lindemann, Udo; Krcmar, Helmut; Paredis, Christiaan JJ; Vogel-Heuser, Birgit: Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems. IFAC-PapersOnLine, 48(3):916–923, 2015.

[GVM12] Giachetti, Giovanni; Valverde, Francisco; Marín, Beatriz: , Interoperability for Model-Driven Development: Current State and Future Challenges, 2012.

[Ha15] Haber, Arne; Look, Markus; Mir Seyed Nazari, Pedram; Navarro Perez, Antonio; Rumpe, Bernhard; Völkel, Steven; Wortmann, Andreas: Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In: Model-Driven Engineering and Software Development Conference (MODELSWARD'15). SciTePress, pp. 19–31, 2015.

[HKR21] Hölldobler, Katrin; Kautz, Oliver; Rumpe, Bernhard: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HMSNR15] Hölldobler, Katrin; Mir Seyed Nazari, Pedram; Rumpe, Bernhard: Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In: Domain-Specific Modeling Workshop (DSM'15). ACM, pp. 23–30, 2015.

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

[IS10] ISO/IEC: ISO/IEC 25010 System and Software Quality Models. Technical report, International Standardization Organization (ISO), Madrid, Spain, 2010.

[Kl08] Kleppe, Anneke: The Field of Software Language Engineering. In: International Conference on Software Language Engineering. Springer, pp. 1–7, 2008.

[KRV10] Krahn, Holger; Rumpe, Bernhard; Völkel, Stefen: 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.

[Lo13] Look, Markus; Navarro Pérez, Antonio; Ringert, Jan Oliver; Rumpe, Bernhard; Wortmann, Andreas: Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In: Globalization of Modeling Languages Workshop (GEMOC'13). volume 1102 of CEUR Workshop Proceedings, 2013.

[Mi09] Milanovic, Nikola; Cartsburg, Mario; Kutsche, Ralf; Widiker, Jürgen; Kschonsak, Frank: Model-Based Interoperability of Heterogeneous Information Systems: An Industrial Case Study. In: European Conference on Model Driven Architecture-Foundations and Applications. Springer, pp. 325–336, 2009.

[Mu12] Mustafiz, Sadaf; Denil, Joachim; Lúcio, Levi; Vangheluwe, Hans: The FTG+PM Framework for Multi-Paradigm Modelling: An Automotive Case Study. In: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling. pp. 13–18, 2012.

[Ni15] Nielsen, Claus Ballegaard; Larsen, Peter Gorm; Fitzgerald, John; Woodcock, Jim; Peleska, Jan: Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. ACM Computing Surveys, 48(2), September 2015.

[Se14] Seidewitz, Ed: UML With Meaning: Executable Modeling in Foundational UML and the Alf Action Language. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. pp. 61–68, 2014.

[Se17] Seidewitz, Ed: A Development Environment for the Alf Language within the MagicDraw UML tool (Tool Demo). In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. pp. 217–220, 2017.

[SWA10] Seifert, Mirko; Wende, Christian; Aßmann, Uwe: Anticipating Unanticipated Tool Interoperability using Role Models. In: Proceedings of the First International Workshop on Model-Driven Interoperability. pp. 52–60, 2010.

[Vo13] Voelter, Markus: Language and IDE Modularization and Composition with MPS. In (Lämmel, Ralfand Saraiva, João; Visser, Joost, eds): Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 383–430, 2013.

[VV10] Völter, Markus; Visser, Eelco: Language Extension and Composition with Language Workbenches. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. OOPSLA '10, Association for Computing Machinery, New York, NY, USA, p. 301–304, 2010.

[Zi07] Ziemann, Jörg; Ohren, Oddrun; Jäkel, Frank-Walter; Kahl, Timo; Knothe, Thomas: Achieving Enterprise Model Interoperability Applying a Common Enterprise Metamodel. In: Enterprise Interoperability, pp. 199–208. Springer, 2007.

[ZLVH18] Zou, Minjie; Lu, Boyang; Vogel-Heuser, Birgit: Resolving Inconsistencies Optimally in the Model-Based Development of Production Systems. In: 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE). IEEE, pp. 1064–1070, 2018.