[JR23] N. Jansen, B. Rumpe:
Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages.
In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering,
pp. 163–168, SLE 2023, Association for Computing Machinery, Cascais, Portugal, Oct. 2023.

Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages

Nico Jansen
jansen@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Abstract

In Software Language Engineering, the composition of heterogeneous languages has become an increasingly relevant research area in recent years. Despite considerable advances in different composition techniques, they mainly focus on composing concrete and abstract syntax, while a thorough yet general concept for synchronizing code generators and their produced artifacts is still missing. Current solutions are either highly generic, typically increasing the complexity beyond their actual value, or strictly limited to specific applications. In this paper, we present a concept for lightweight generator composition, using the symbol tables of heterogeneous modeling languages to exchange generator-specific accessor and mutator information. The information is attached to the symbols of model elements via templates allowing code generators to communicate access routines at the code level without a further contract. Providing suitable synchronization techniques for code generation is essential to enable language composition in all aspects.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: Software Language Composition, Code Generation, Generator Composition, CRUD

ACM Reference Format:
Nico Jansen and Bernhard Rumpe. 2023. Seamless Code Generator Synchronization in the Composition of Heterogeneous Modeling Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 6 pages.
https://doi.org/10.1145/3623476.3623530

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623530

1 Introduction

Model-driven engineering (MDE) [26] is a prominent research and application area in which models of various domains, such as automotive [2], robotics [30], and software development [29], represent the central development artifacts [12] in developing modern (often software-intensive) systems. These models conform to modeling languages, prescribing concrete and abstract syntax, additional well-formedness rules, and semantics [8] providing meaning [15]. The discipline of software language engineering (SLE) [19] investigates the efficient design, maintenance, and evolution of such languages (for both modeling and programming).

As software languages evolve and mature [11], and their constant support and development are time-consuming, reuse becomes increasingly critical in SLE [4]. This means not only reusability on a conceptual level but the actual reuse of the implementation, i.e., the language definition and its generated and hand-coded tooling. In this regard, the composition of software languages has been extensively investigated in the last decade [17], establishing libraries of reusable language components [5] and patterns for compositional language design [9].

While composition in a language's front end, i.e., its syntax and tooling, is already pretty sophisticated, composing the back end, usually code generators, is often neglected. Code generation is essential to modeling languages as it translates abstract models into executable program artifacts, thus bridging the gap between the problem and solution domain [22]. Access information must be distributed through the generation process so that generated artifacts can address each other correctly. Figure 1 sketches this challenge and serves as a running example. The left-hand side presents two models of different languages, a class diagram (top) with a class `Person` featuring a `name` and an `age` attribute and an automaton snippet (bottom), granting access once the `age` is at least 18. Thus, the expression at the automaton's transition refers to an attribute definition inside the class diagram. For simplicity reasons, we neglect the type-instance relation at the model level in this example. As the models refer to each other, it is apparent that their generated target artifacts do as well. Therefore, the code generator for the automaton language must adhere to the access information the class diagram generator provides. In scenario (a) (middle), the class is transformed in an intuitive fashion to a Java class

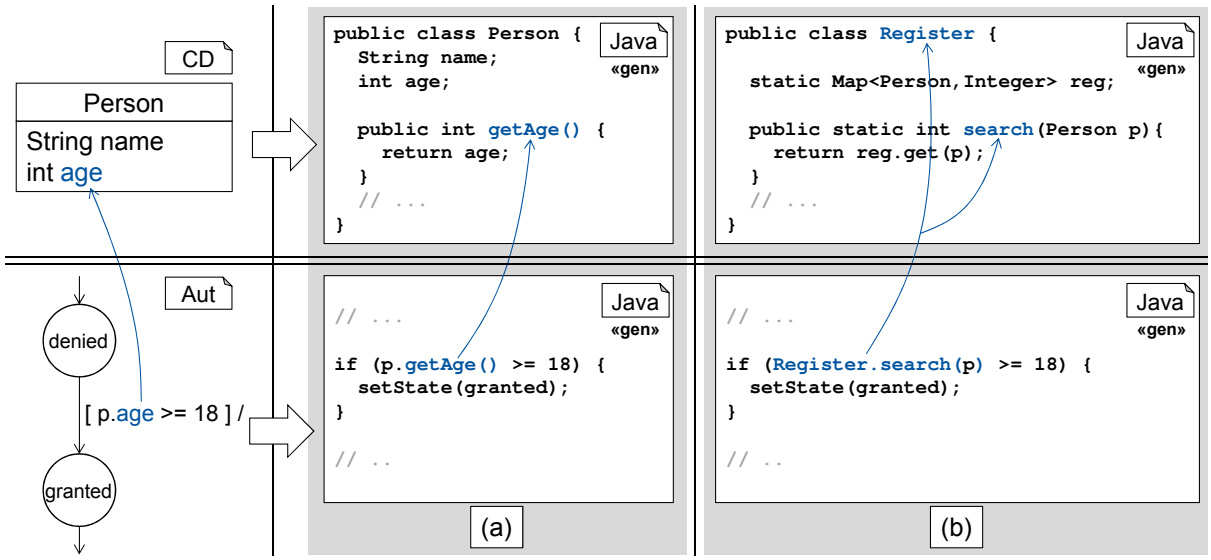

Figure 1. Model excerpts of a class diagram and an automaton (left) with an inter-model cross reference and two alternative generated code snippets (middle and right). In each scenario, these generated artifacts must match the respective access.

with attributes accessible via corresponding getters. Thus, the generated code for the automaton can utilize this getter. However, scenario (b) (right) depicts a rather different translation of the class diagram, resulting in a `Register` with a static access map requiring the corresponding person as key to retrieve the age. Therefore, assuming a provided `getAge` method, as in the first case, is not applicable anymore, the target code of the automaton must adapt. While scenario (b) is a contrived situation highlighting the underlying challenge, it is well within the realm of possibility. In reality, multiple cases of divergent access situations exist, such as employing builders or factories for object instantiation instead of native constructors or translating an automaton concerning different design patterns (e.g., the state pattern) [13].

Currently, there are no well-established solutions for composing generators or their generated target artifacts. This results in a gap when integrating models of distinct modeling languages, as their outputs must be synchronized, demanding additional manual effort. Some approaches exist but are either tied to integrating explicit generators of particular application domains [24], require strict compliance with generation rules, or are overly generic [23], raising the complexity beyond their practical usability. A general, lightweight solution is still missing.

In this paper, we envision a novel approach to synchronizing generated outputs via accessor template-enriched symbol tables. This approach harnesses the capabilities of already established composition techniques for languages' front ends and extends these to synchronize their generators as well. Our proposed solution is based on the symbol management infrastructure of a language enabling inter-model cross-referencing. Augmenting symbol tables with target access templates enables the adaptive generation of accessor code. An application programming interface (API) for synchronizing generated artifacts should be as lightweight as possible and ideally require down to no additional knowledge of another generator's intricacies. Therefore, we propose CRUD-like accessors for this API, as these operations are commonly known, language agnostic, and the notation is easily understandable. Our work focuses on template-based code generators that produce artifacts of a common yet arbitrary object-oriented, general-purpose programming language. Furthermore, we concentrate on harmonizing the target code for models of aggregated languages, as this composition technique preserves models as separated artifacts and, thus, only establishes a loose coupling [6]. Our main contributions are:

- An approach for enriching the symbol table with accessor templates enabling target code synchronization
- A conceptual API based on CRUD-like operations as a lightweight generator synchronization contract

The remainder of the paper is structured as follows: section 2 discusses the current state of the art comprising related approaches and preliminary work. section 3 presents our concept of integrating templates in symbol tables to synchronize generated artifacts. Finally, section 4 discusses our solution, states open challenges, and section 5 concludes.

2 State of the Art and Related Work

While language composition, in general, is a broad field of research in many language workbenches [10], there are currently only a few approaches to the integration of their generators. These attempts are often either tailored solutions

for specific applications, too shallow to deliver a general concept, or too generic to be effortlessly applied in practice.

Most preliminary work in this field is often domain- or even application-specific. For example, there are approaches in robotics [24, 25] coupling generators specifically for component and connector systems via a separate generator configuration and an orchestrator. Other investigations follow a round-trip engineering approach defining framework-specific modeling languages [1]. However, these approaches are tightly coupled to the underlying application frameworks and do not aim for generalizable generator composition.

An interesting approach is based on reverse engineering existing target artifacts to extract corresponding accessor information [3, 14]. While this attempt can generally enable to generate syntactically well-formed and consistent code, it lacks the link to the original model elements. Therefore, it does not solve the issue of deriving target accessor information based on inter-model relationships.

A few framework solutions aim for the complete integration of languages and thus also incorporate their generators. CompoSE [20] provides for the integration of languages and generates glue code for the individual language components' target artifacts. Similarly, the SCOLAR framework also provides the ability to compose language components in the large [7, 23]. However, these works mainly concentrate on language embedding, i.e., a stronger coupling of the models. While such integrated framework solutions still address the problem of generator composition, they generally have the disadvantage that languages must be integrated into their respective ecosystem. Additionally, the defined communication interfaces are usually very generic and, therefore, uncomfortable to employ for arbitrary modeling languages.

Similarly, the Genesys project [18] provides for generator development conforming to predefined frameworks. It supports services for communication accessor information. However, the results are bound to the jABC ecosystem [27], and a more seamless composition of generators is considered future work.

Finally, a few lightweight approaches exist that provide for an exchange of information via the symbol tables of the models [21, 22]. The advantage of these attempts is that mainly already existing composition techniques are employed without creating over-complicated new infrastructures. So far, however, these approaches have only been weakly studied for predefined modeling and programming language combinations. A general solution that seamlessly composes generated artifacts of heterogeneous modeling languages still needs to be established.

3 Distributing Access Information Exchange via Symbol Tables

Our approach builds upon the concepts of [22], who propose enriching the symbol table with accessor and mutator code snippets of the model elements' corresponding target artifacts. While the basic idea is promising, their proposal only considers a fixed source (i.e., modeling) and target (i.e., general-purpose programming) language. Thus, the described symbol table extension and mapping are bound to these technological spaces. While further adaptions are possible, they require one mapping for each language combination, ultimately convoluting the symbol table infrastructure when incorporating more and more languages. Thus, our approach envisions a more generalized extension of the symbol table, which is as language-agnostic as possible and allows for arbitrary accessor and mutator mappings concerning different symbol kinds.

3.1 An Extended Symbol Table Infrastructure for Managing Target Access

For efficient cross-referencing, modern language workbenches support the concept of symbol tables, either directly (such as MontiCore [16]) or implicitly (such as MPS [28]). In a symbol table, symbols of language-specific kinds are ordered hierarchically inside scopes managing their visibility and accessibility. This principle is used, for instance, in language aggregation to compose models of different artifacts by resolving their respective inter-model references.

Utilizing the cross-referencing of symbols, our approach extends the symbol table infrastructure by enriching it with further generator information. Therefore, we augment the symbols further with access information templates of the target code. Figure 2 depicts our concept of the extended symbol table infrastructure. Similar to existing approaches, we foresee the extension of symbols with a `GeneratorInfo`, defining the overall API of the generator synchronization mechanism. As we strive for a general solution not bound to a definitive technological space, the signature is specified in a language-agnostic way.

In a first attempt, we propose CRUD operations for a generalized API to `create`, `read`, `update`, and `delete` constructs in an object-oriented sense. Thus, the `GeneratorInfo` attached to each symbol offers the corresponding methods independent of its kind. For proper access derivation, each operation expects a respective context in which the access occurs. For instance, in our running example, the context is the variable `p` of type `Person`, which is used quite differently in both scenarios. Next, as updating an object constitutes mutating access, it requires an additional value parameter to write, i.e., the new value to update with. This value, of course, is only provided at the model level. However, it is crucial to consider it here, as the generator needs to insert it in the mutator template. Finally, we conceive a collection of additional optional parameters that can be used to parametrize the access further. For instance, when updating only a single value inside a list, these parameters can provide the respective position.

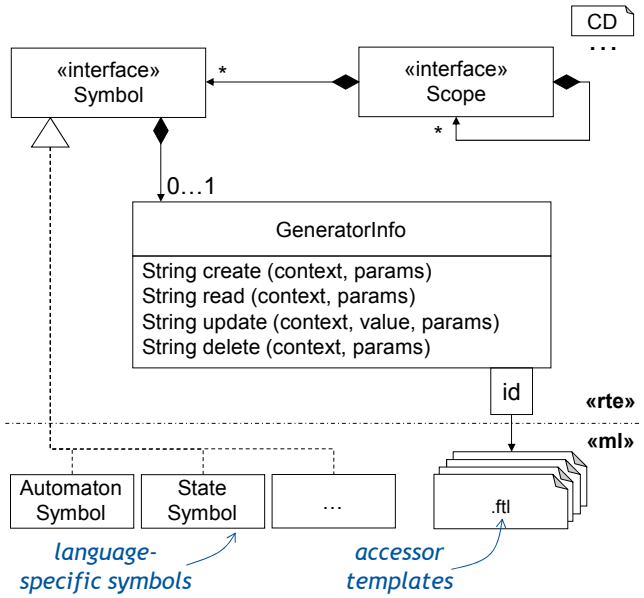

Figure 2. Extended symbol table infrastructure for storing generator-specific CRUD accessor templates for each symbol.

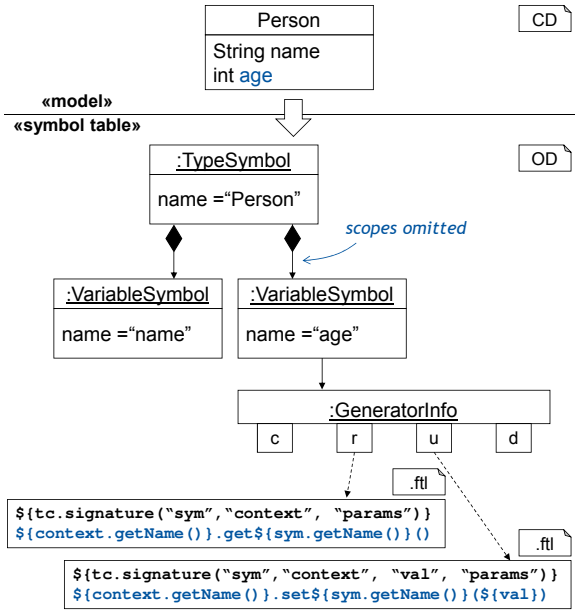

Figure 3. Object diagram representation of the symbol table for a corresponding class diagram model with attached read and update templates for accessing the generated artifacts.

The proposed symbol table extension with generator information based on CRUD operations is independent of a particular modeling language. However, this only constitutes a general API via which generators can communicate the access data. The methods delegate to corresponding accessor templates which are, in contrast, attached to the symbols (resp. to their generator information) on the language level. Thus, each symbol kind comes with a particular set of templates carrying the accessor information for the respective generator's target code. In fact, it is even possible to provide different symbols of the same kind with distinct templates based on their use.

The extended symbol table infrastructure enables modular generators to look up the respective accessor code adaptively. Figure 3 shows a simplified view of a symbol table instance, providing accessor templates for the variable `age` of our running example. That is, the generator information of the variable symbol carries templates for `read` and `update` operations adhering to scenario (a). In this example, `create` and `delete` commands are not allocated. While the approach convinces the solution for arbitrary template engines, the presented templates are written in FreeMarker syntax (.ftl) for demonstration purposes.

Considering the template for the `read` operation, the first line defines the signature of the template, containing the variable symbol itself, the context, and optional parameters. The second line describes the derivation of the respective accessor (here, a getter method) by first referencing the corresponding context, followed by the static part `.get` and the dynamic name of the symbol itself, concluded by `()`. Please note that the presented template is a slight simplification as for a proper getter signature, the first letter of the attribute's name is capitalized, which is not reflected here. During generation, this template gets evaluated by the corresponding engine providing the required accessor code.

3.2 A Light-Weight Infrastructure for Composing Template-based Generators

Employing the extended symbol table infrastructure, we can further design compositional generator tools. For this purpose, we propose a generator architecture encapsulating language-specific printers that use the cross-language API of the generator information now delivered with the particular symbols. Figure 4 gives an overview of this architecture. Generally, such a tool comprises the standard components, such as a parser transforming the input models into an AST and a template engine for code generation. Additionally, we attach a composition printer for dealing with the incorporated symbol tables. Whenever an accessor code is required, this printer is incorporated, resolving the respective symbol and retrieving the corresponding accessor template.

It is important to note that such a printer only needs to know the constructs of its own language (usually but not limited to expressions) and not those of the referenced symbols. For instance, a printer for the automaton generator (cf. Figure 1) would know how to translate expressions, such as `x >= y`, in general, but fetches the concrete target code accessors from the templates of the loaded symbol table. This mechanism enables the seamless composition of target artifacts while simultaneously preserving the loose coupling of language aggregation.

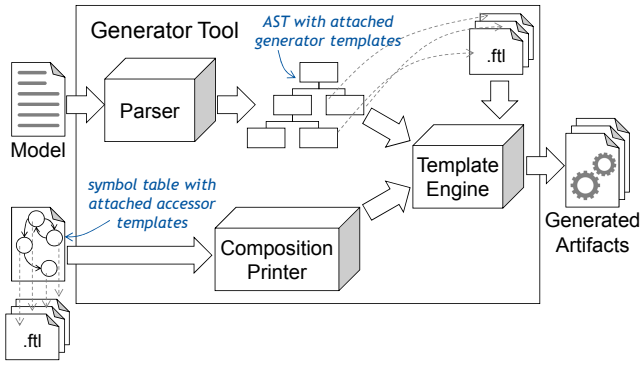

Figure 4. Conceptual generator tool with a built-in printer utilizing symbol-attached templates to supply the generator engine with accessor information on external artifacts.

4 Discussion and Open Challenges

The presented approach to seamlessly compose generated artifacts of heterogeneous modeling languages is based on template-enriched symbol tables and a lightweight interface using CRUD-like operations. Considering previous work, we envision extending existing concepts for accessor provisioning via symbols and establishing a more general solution for object-oriented target languages.

Previous approaches of related attempts require knowledge of the access signatures for each symbol type [22]. This hampers the loose coupling of some composition techniques, such as language aggregation, since generally, we cannot assume global knowledge of all loaded symbol kinds. Our proposal delegates access to CRUD-like operations, thus relaxing the tight coupling of generators. Moreover, previous approaches were limited to a fixed relation of the modeling and target programming languages, while our proposal aims for more generality.

Different approaches to composing generators or their generated artifacts on a more general level rely on highly generic, often bulky composition interfaces or require their explicit definition [7, 20]. Our technique avoids these drawbacks since we mainly build on existing composition mechanisms and rely only on a simple, standardized API. To provide for composability, a generator developer simply creates access templates and attaches these to symbols according to predefined CRUD operations. No further knowledge about employing generators is required. In turn, the developer of a generator, which requires access information, does not need any knowledge of the providing language despite its anyways provided symbols. A modeler using the languages does not need any knowledge of the generation process, the communication mechanism, or the attached templates.

However, our proposal for lightweight generator synchronization is a mere starting point with open challenges that need further investigation. For instance, the mentioned `context` in a generator template (cf. Figure 3) is often more complicated than in the highlighted example. In general, all symbols traversed during resolution must be considered, resulting in a collection of contextual symbols of which some might be relevant while others are omitted. While this is a task for the engineer providing the access templates (i.e., no issue that hampers the composability), it could turn out to be challenging to create the required templates for complicated situations in the first place. Thus, future investigations are required to evaluate whether the composition mechanism is as lightweight and applicable as envisioned.

Moreover, while our approach is generalized, it fits particularly well on infrastructures that provide an explicit symbol management system accessible to the language (or generator) developer. For frameworks that do not allow for customization or augmentation of symbols, concrete implementation, while generally possible, may turn out more difficult.

Furthermore, our approach does not yet consider that a model element can be mapped to several conflicting target elements. For instance, we can translate a single class of a class diagram into multiple classes on the code level. This makes the target access ambiguous. Generally, generators can always be modularized to minimize this issue. Other attempts propose an identifier, making the mapping unique [22]. However, both approaches have the disadvantage that knowledge about the generation process of the source generator is required.

Finally, the proposed API needs further investigation. While we envision a lightweight and straightforward generator synchronization interface, CRUD-like operations might not be sufficient. The requirements on the API strongly depend on which information of the target artifacts is relevant for synchronization. However, as a state is per se no type on the model level, this information does not directly emerge from the CRUD operations but requires additional consideration. Thus, while the CRUD-like API already covers a large set of necessary access patterns, it cannot be considered final. We need a detailed analysis of common symbol kinds and their potential target code information to extend further and refine the access interface.

5 Conclusion

As language engineering becomes increasingly sophisticated with composition techniques at concrete and abstract syntax level, so must it become for integrating generated artifacts. For this purpose, we presented an approach that envisions augmenting the symbol table, used in language aggregation, with additional templates to exchange access information of the target artifacts. We highlighted the need for a lightweight interface between generators and discussed a simple API based on CRUD-like operations. Furthermore, we have depicted challenges for further investigation regarding the portion of the information that must be exchanged. Seamlessly synchronizing generators is a crucial step for completely integrating languages.

References

[1] Michał Antkiewicz and Krzysztof Czarnecki. 2006. Framework-Specific Modeling Languages with Round-Trip Engineering. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 692–706. https://doi.org/10.1007/11880240_48

[2] Hans Blom, Henrik Lönn, Frank Hagl, Yiannis Papadopoulos, Mark-Oliver Reiser, Carl-Johan Sjöstedt, De-Jiu Chen, Fulvio Tagliabo, Sandra Torchiaro, Sara Tucci, et al. 2013. EAST-ADL: An architecture description language for Automotive Software-Intensive Systems. *Embedded Computing Systems: Applications, Optimization, and Advanced Design: Applications, Optimization, and Advanced Design* (2013), 456. https://doi.org/10.4018/978-1-4666-3922-5.ch023

[3] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. MoDisco: a Model Driven Reverse Engineering Framework. *Information and Software Technology* 56, 8 (2014), 1012–1032. https://doi.org/10.1016/j.infsof.2014.04.007

[4] Barrett Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. 2015. *Globalized Domain Specific Language Engineering*. Springer, 43–69. https://doi.org/10.1007/978-3-319-26172-0_4

[5] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology* 19, 3 (October 2020), 3:1–16. https://doi.org/10.5381/jot.2020.19.3.a4

[6] Arvid Butting, Judith Michael, and Bernhard Rumpe. 2022. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology* 21 (October 2022), 4:1–13.

[7] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 35–46. https://doi.org/10.1145/3365438.3410934

[8] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09) (LNCS 5795)*. Springer, 670–684. https://doi.org/10.1007/978-3-642-04425-0_54

[9] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology* 21, 4 (October 2022), 4:1–13. https://doi.org/10.5381/jot.2022.21.4.a4

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217.

[11] J-M Favre. 2005. Languages evolve too! Changing the Software Time Scale. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE, 33–42. https://doi.org/10.1109/IWPSE.2005.22

[12] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)* (May 2007), 37–54.

[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. 1995. Elements of Reusable Object-Oriented Software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company* (1995).

[14] I Garcıa, M Polo, and M Piattini. 2004. Metamodels and architecture of an automatic code generator. In *2nd Nordic Workshop on the Unified Modeling Language, NWUML*.

[15] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (October 2004), 64–72. https://doi.org/10.1109/MC.2004.172

[16] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag, Aachen.

[17] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures* 54 (2018), 386–405. https://doi.org/10.1016/j.cl.2018.08.002

[18] Sven Jörges, Tiziana Margaria, and Bernhard Steffen. 2008. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering* 4, 4 (2008), 361–384. https://doi.org/10.1007/s11334-008-0071-2

[19] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Pearson Education.

[20] Thomas Kuhn, Soeren Kemmann, Mario Trapp, and Christian Schäfer. 2009. Multi-Language Development of Embedded Systems. In *9th OOPSLA DSM Workshop, Orlando, USA*.

[21] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. 2015. Management of Guided and Unguided Code Generator Customizations by Using a Symbol Table. In *Domain-Specific Modeling Workshop (DSM'15)*. ACM, 37–42. https://doi.org/10.1145/2846696.2846702

[22] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. 2016. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference (LNI, Vol. 254)*. Bonner Köllen Verlag, 133–140.

[23] Jérôme Pfeiffer and Andreas Wortmann. 2021. Towards the Black-Box Aggregation of Language Components. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 576–585. https://doi.org/10.1109/MODELS-C53483.2021.00088

[24] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2014. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *Model-Driven Robot Software Engineering Workshop (MORSE'14) (CEUR Workshop Proceedings, Vol. 1319)*. 66 – 77.

[25] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2015. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)* 6, 1 (2015), 33–57.

[26] Bran Selic. 2003. The Pragmatics of Model-Driven Development. *IEEE software* 20, 5 (2003), 19–25. https://doi.org/10.1109/MS.2003.1231146

[27] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. 2007. Model-Driven Development with the jABC. In *Hardware and Software, Verification and Testing: Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers 2*. Springer, 92–108.

[28] Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1449–1450.

[29] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

[30] Dennis Leroy Wigand, Arne Nordmann, Niels Dehio, Michael Mistry, and Sebastian Wrede. 2017. Domain-Specific Language Modularization Scheme Applied to a Multi-Arm Robotics Use-Case. *Journal of Software Engineering for Robotics* (2017).

Received 2023-07-07; accepted 2023-09-01