Advances in Modeling Language Engineering

Katrin Hölldobler, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann(✉)

Software Engineering, RWTH Aachen University, Aachen, Germany
http://www.se-rwth.de

Abstract. The increasing complexity of modern systems development demands for specific modeling languages capturing the various aspects to be tackled. However, engineering of comfortable modeling languages as well as their tooling is a challenging endeavor. Far too often, new languages are built from scratch. We shed light into the advances of modeling language engineering that facilitates reuse, modularity, compositionality and derivation of new languages based on language components. We discuss ways to design, combine, and derive modeling languages in all their relevant aspects. For each of these activities, we illustrate their application for the model-driven development of a data exploration tool. The tool itself uses a set of meta-information, namely the structural model to derive all necessary software components that help to gather, store, visualize and navigate the data.

The limits of my language
mean the limits of my world

– Ludwig Wittgenstein

1 Motivation

The use of models to understand and shape the world is a very foundational technique that has already been used in ancient Greece and Egypt. Scientists model to understand the world and engineers model to design (parts of) the world. Although modeling has been employed for ages in virtually all disciplines it is fairly new that the form of models is made explicit in so-called modeling languages. Computer science has invented this approach to provide formality and a precise understanding of what is a well-formed model to the communication between humans and machines.

Programming languages in general, SQL [7], XML [2], and the Unified Modeling Language (UML) [22,42,43] in particular have been created to enable highly precise communication. Despite these efforts, it is clear that researchers and practitioners of many domains are dissatisfied by solving domain-specific problems with general purpose languages or unified languages that try to cover everything. The general aspiration of such languages create a conceptual gap between

© Springer International Publishing AG 2017
Y. Ouhammou et al. (Eds.): MEDI 2017, LNCS 10563, pp. 3–17, 2017.
DOI: 10.1007/978-3-319-66854-3_1

the problem domains and the solution domains that raises unintended complexities [18]. As a result, Domain-Specific Languages (DSLs) and Domain-Specific Modeling Languages (DSMLs) [48] were created to match domain specific needs. Due to the ongoing digitization of virtually every domain in our life, work, and society, the need for more specific languages raises. It is apparent, that we need to be able to accommodate new and changing domains with appropriate domain-specific languages – ideally on-the-fly. This raises three questions:

1. How to design new DSLs that fit specific purposes?
2. How to engineer a DSL from predefined components?
3. How to derive DSLs from other DSLs?

In this paper, we give an overview of the current state of the art on the design of DSLs, discuss the mechanisms enabling their composition, and describe how to derive new DSLs from predefined ones, such that we prevent restarting design of the language from scratch each time, but instead successfully engineer language from reusable components. These mechanisms to derive and compose languages are the core of what we today calls *software language engineering* (SLE) [32]: the discipline of engineering software languages, which are not only applied to computer science, but to any form of domain that deals with data, their representation in form of data structures, smart systems that need control, as well as with smart services that assist us in our daily life.

The rest of this paper is organized as follows: First, Sect. 2 presents current language definition techniques and sketches language creation by example. Afterwards, Sect. 3 introduces language composition techniques and illustrates their application, before Sect. 4 highlights language derivation techniques. Section 5 presents the case study of modeling a data explorer application leveraging software language engineering techniques. Ultimately, Sect. 6 concludes this paper.

2 Language Engineering

Model-driven engineering [18] lifts abstract models to primary development artifacts to facilitate software analysis, communication, documentation, and transformation. Automated analysis and transformation of models require that these adhere to contracts that analyses and transformations can rely upon (and be developed against). Such automation is feasible, where models conform to modeling languages. For many popular modeling languages, such as UML [22], AADL [16], or Matlab/Simulink [1], research and industry have produced useful analyses and transformations. These rely on making the constituents and concerns of languages machine processable. To this effect, the discipline of SLE investigates disciplined and systematic approaches to the design, implementation, testing, deployment, use, evolution, and recovery of modeling languages.

Similar to research in natural languages, SLE commonly defines languages as the set of sentences they can produce [3]. Operationalizing languages, however, requires more precise characterizations. To this effect, languages usually

are defined in terms of their syntax (possible sentences) and semantics (meaning) [26], which can be concretized to requiring a concrete syntax (words), an abstract syntax (structure), static semantics (well-formedness), and dynamic semantics (behavior) for language definition [3]. The technical realizations of modeling languages often follow the latter distinction. As "software languages are software too" [15], their technical realizations are as diverse as other representatives of other software categories. This complicates comprehensibility, maintenance, evolution, testing, deployment, and reuse.

To shed light onto this diversity, this section presents different mechanisms to define modeling languages and highlights selected language development environments employing these mechanisms. Afterwards, we illustrate development of a language to represent a variant of UML class diagrams that will serve as running example for the subsequent sections.

2.1 Engineering Modeling Languages

Research has produced various means to develop solutions for representing the different concerns of modeling languages. Lately, two different language implementation techniques have been distinguished:

1. Internal modeling languages are realized as fluent APIs [17] in host programming languages whose method names resemble keywords of the language. Omitting syntactic sugar (such as dots and parentheses) as supported by modern programming languages (*cf.* Groovy, Scala) enables to create chains of method calls that resemble models. This method is suitable to language prototyping and yields the benefit of enabling to reuse the host language's tooling (such as parsers, editors, compilers, *etc.*). The expressiveness of the modeling language depends on the host programming language.
2. External modeling languages feature a stand-alone syntax that requires tooling to process its models into machine-processable representations. While this creates additional effort over internal languages, external languages can leverage a greater language definition flexibility. However, language-related tooling must be provided by the language engineer.

The majority of modeling language research focuses on external languages, which yield greater flexibility in language design. Consequently, research has produced more solutions to the definition of external languages, which is why we focus on their realization techniques in the following.

Engineering language syntaxes historically is related to the development and processing of (context-free) grammars [33], which are sets of derivation rules that at least enable describing the languages' abstract syntaxes. Many approaches to grammar-driven language engineering also support specifying a language's concrete syntax in the same grammar as well [21]; hence, enabling efficient language development and maintenance. Metamodels are another popular means to develop the abstract syntax of languages [48]. Here, classes and their relations structure the syntax of a language. While these do not support the integration

of concrete syntax (and, hence, always require providing editors), they enable reifying references between model elements that are name-based in grammars, as first level references.

Concrete syntaxes are either textual [34,49], graphical [10], or projectional [48]. Textual and graphical languages both require parsing, whereas projectional syntaxes (*e.g.,* forms enabling editing the abstract syntax directly [47]) usually are bound to specific editors. In contrast, textual syntaxes enable to reuse established software engineering tooling, such as editors or version control systems.

Whether the well-formedness of models is subject of their syntax or their static semantics is subject to debate. Nonetheless, various techniques have been established to enforce the well-formedness of models with respect to properties that cannot be captured by grammars or metamodels (*e.g.,* preventing to class members of the same name). Popular approaches to well-formedness checking are programming language rules and Object Constraint Language (OCL) [40] constraints. Both require a model's internal representation and raise errors if these are not well-formed according to the individual rule. As OCL is a modeling language itself, this requires interpreting it or translating the constraints to programming language artifacts actually executing the models under development.

Executing models is a popular way to realize their dynamic semantics. This can have the form of interpretation [30] or transformation [6]. With the former, a software (the interpreter) processes the models and executes according to their description. This interpreter can be part of the models or a separate software. Transformations process models and translates these into other formalisms with established semantics, such as a programming language. Model-to-text (M2T) transformations [36] read models of a specific language and translate these to plain text (such as programming language code), whereas model-to-model (M2M) transformations [36] translate models from an input modeling language to an output modeling language. The former lends itself for ad-hoc transformation development using template engines or string concatenation (as the output language is not required), but lacks the structure and verifiability of M2M transformations.

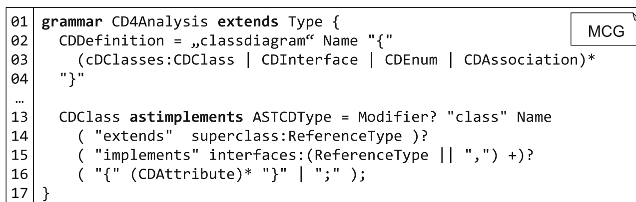
Language workbenches [13] are software development environments supporting language engineering. Based on an, usually fixed, integration of language definition constituents, they facilitate creating languages and corresponding tooling. For instance, GEMOC Studio [5] employs ECore [44] metamodels for abstract syntax, OCL for static semantics, and Kermeta [30] for weaving interpretation capabilities into its languages. Concrete syntax can, *e.g.,* have the form of Xtext [14] grammars or Sirius [46] editors. The meta programming system (MPS) features projectional language engineering on top of a metamodel and combines this with well-formedness checking and execution through M2M transformations. The Neverlang language workbench [45] supports grammar-based language definition and focuses on combining these with language processing tools. It executes models via interpretation.

The next section illustrates engineering of a textual modeling language for class diagrams (CDs) with the MontiCore language workbench.

2.2 Language Engineering with MontiCore

MontiCore [34] is a language workbench for efficient development of compositional modeling languages. The concrete and abstract syntax of languages are defined as extended context-free grammars and it supports a Java-based well-formedness checking framework as well as model execution through M2M and M2T transformations. From the grammars, MontiCore generates parsers and abstract syntax classes. The parser enables processing textual, conforming models, into instances of the languages' abstract syntax classes. Java context conditions process these instances to check their well-formedness before M2M transformations [27] or template-based code generators [41]. MontiCore supports language inheritance, language embedding, and language aggregation [25] to reuse and combine languages with little effort.

Consider the excerpt of the MontiCore grammar depicted in Fig. 1, which describes the class diagram for analysis (CD4A) modeling language: After the keyword `grammar` and its name, the grammar extends existing types to reuse previously defined grammars (l. 1). Afterwards, a body of productions follows that characterize a variant of class diagrams. Each production is defined by a left-hand-side (*e.g.,* `CDDefinition` in l. 2) and a right-hand-side, which contains terminals (*e.g.,* `"classdiagram"` in l. 2) and non-terminals (*e.g.,* `CDInterface` in l. 3). Different operators (*e.g.,* `*` in l. 3, `?` in l. 13, and `+` in l. 15) define the quantity or presence of a part on the right-hand-side. MontiCore also supports additional grammar constructs to extend the generated AST (*e.g.,* `astimplements` in l. 13).



```
01 grammar CD4Analysis extends Type {                                    MCG
02   CDDefinition = „classdiagram“ Name "{"
03     (cDClasses:CDClass | CDInterface | CDEnum | CDAssociation)*
04   "}"
…
13   CDClass astimplements ASTCDType = Modifier? "class" Name
14     ( "extends"  superclass:ReferenceType )?
15     ( "implements" interfaces:(ReferenceType || ",") +)?
16     ( "{" (CDAttribute)* "}" | ";" );
17 }
```

Fig. 1. An excerpt of a MontiCore grammar for the CD4A language.

With this grammar, the CD4A model shown as an excerpt in Fig. 2 can be created. It describes a simplified banking system consisting of a package declaration (l. 1); an abstract `Account` class to describe different types of accounts (ll. 3–7); an interface to model employees (l. 17) and its implementation (ll. 18–20); and multiple associations (*e.g.,* l. 55). From this grammar, MontiCore produces a parser and an abstract syntax class for each production. The latter captures the production's right hand side by providing members capable of storing its content.

```
01 package banking;                                          CD4A
02 classdiagram BankingSystem {
03   abstract class Account {
04     long number;
05     int balance;
06     int overdraft;
07   }
…
17   interface Employee;
18   class Consultant implements Employee {
19     String name;
20   }
…
55   association [1] Account <-> [[name]] Consultant;
56 }
```

Fig. 2. An example of a CD4A model describing a lightweight banking system.

In addition, an infrastructure to check context conditions, which are predicates defined with respect to the abstract syntax to determine the language's consistency. For example, to restrict the modifiers of classes to abstract only (l. 13 in Fig. 1).

3 Composing Modeling Languages

Model-driven development is successful when initiated bottom-up [50], *i.e.,* developers employ modeling languages considered suitable for their challenges instead of using predefined, monolithic general-purpose modeling techniques. For their efficient development, evolution, validation, and maintenance, such languages should be retained as independent as possible. Ultimately, however, combining such languages mandates their efficient *composition* [3]. Considering, for instance, software of the smart and modular factories imagined with Industry 4.0, these demand integrating business processes, domain models, behavior models and failure models of the automation systems, assembly plan models, manipulator kinematics, *etc.* Integrating these modeling languages into a combined software requires operations for their composition.

Software engineering itself is another prime example of a domain leveraging language composition to facilitate development, evolution, and maintenance. To this effect, research and industry have produced languages for 1. modeling structure and behavior of the software under development, such as UML [22]; 2. describing database interaction, such as SQL [7] or HQL [29]; 3. describing software build processes, such as Maven's Project Object Models [37]; 4. describing configuration of product lines [4], such as feature diagrams [6]; 5. describing model changes in a structured fashion, such as delta modeling languages [24] 6. extending models with additional, external information (tagging languages [20]); 7. coordinating the use of different modeling languages, such as the BCOol language [35]; 8. transforming of models of other languages, such as ATL [31] or the FreeMarker [39] template language; and 9. describing the syntax and semantics of modeling languages, such as ECore [44], Kermeta [30], or MontiCore [34].

Consequently, structured reuse of language parts is crucial to enable efficient SLE. And while research on language integration has produced reuse concepts and related these to language definition concerns [3], the diversity of language realization techniques has spawned very different reuse mechanisms [11]. Generally, we distinguish *language integration*, which produces a new language, from existing languages, from *language coordination*, in which the sentences of two languages (*i.e.,* their models) are related to enable achieving a common goal.

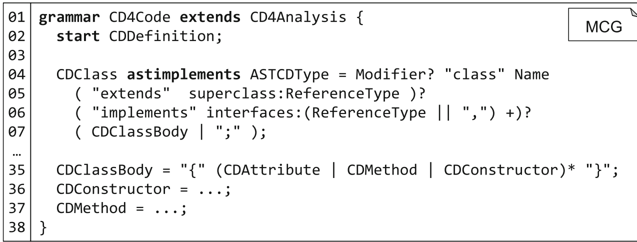
For integrating languages, concepts such as merging of metamodels [9], inheriting and embedding of grammars [25], and importing of metamodel and grammar elements [12] have been conceived. These mechanisms enable a white-box integration to extend and refine existing abstract syntaxes to domain requirements, but rarely consider including integration of other language concerns. For instance, efficient creation of models of language produced through of merging, inheritance, or importing of parts of other languages requires creating or extending proper editors. Even when editors for the base languages exist, this requires handcrafting editing capabilities for the extensions. The same challenges arise for reusing language semantics. As these usually are realized through interpretation or transformation, the corresponding tools of extended languages must be extended also. Yet, there are only few approaches that support compositional semantics realizations, such as code generator composition mechanisms [41].

Coordination of modeling languages is less invasive but mandates means to reason over models of coordinated languages – either for their joint analysis or their joint execution. The former, for instance requires checking the validity of feature models or model transformations with respect to the referenced models. To prevent tying the referencing languages to abstract syntax internals of the referenced languages, abstraction mechanisms, such as the symbol tables of MontiCore [25] have been developed. Joint execution of model of different languages requires exposing and combining their execution mechanisms. Where languages originate from the same language workbench, this integration has been addressed (*e.g.,* by exposing the executable interfaces of model elements [8]). Truly heterogeneous, generalizable coordination has yet to be achieved.

In the next section, we sketch how applying language integration mechanisms to the CD language enables preparing it for code generation.

3.1 Extending and Refining a MontiCore Language

To enable describing software-related properties of domain models more precisely, we extend the CD4A language with additional language constructs such as constructors, methods, and visibility. To reuse the CD4A language, we refine it by removing modeling of method bodies from the modeling language. For the former, we employ MontiCore's language inheritance, for the latter, we introduce new well-formedness rules. An excerpt of the newly created CD4Code language is shown in Fig. 3. It extends the CD4A language (l. 1) and reuses the start production (l. 2). In addition, the `CDClass` production is extended with a `CDClassBody` production, which adds methods and constructors (l. 35).



```
01 grammar CD4Code extends CD4Analysis {
02   start CDDefinition;
03
04   CDClass astimplements ASTCDType = Modifier? "class" Name
05     ( "extends"  superclass:ReferenceType )?
06     ( "implements" interfaces:(ReferenceType || ",") +)?
07     ( CDClassBody | ";" );
…
35   CDClassBody = "{" (CDAttribute | CDMethod | CDConstructor)* "}";
36   CDConstructor = ...;
37   CDMethod = ...;
38 }
```


Fig. 3. An excerpt of the CD4Code extension of CD4A.

4 Deriving Languages

Software engineering leverages modeling languages to mechanize working with models of other languages, such as transformation languages [28,31], delta modeling languages [24], or tagging languages [20]. Such languages have in common that these are either overly generic or are specifically tied to a *host language* (*i.e.*, the languages whose models are transformed or tagged). The former requires developers to learn completely new languages that are independent of a (possibly well-known) host language, while the latter raises the challenge of engineering and maintaining specific languages as well as their specific tooling (editors, analyses, transformations), which is hardly viable.

To address the latter, methods to develop new languages by deriving their syntaxes from related host languages have been developed. These methods rely on processing the host languages' (abstract) syntaxes and creating new (abstract) syntaxes from these. Where the host languages are defined through grammars, such derivation can produce derived concrete syntaxes. For metamodel-based language definition, this would require deriving editor (parts) instead. Automating creating well-formedness rules and behavior implementations of derived languages is more challenging as both may differ from the host languages completely. Where, for example, Statecharts describe state-based behavior, a transformation language derived from Statecharts describes how to translate Statechart models into something else. The behaviors of both languages are unrelated. The same holds for their well-formedness rules.

The next section applies language derivation to the CD4A language to create a domain-specific transformation language from it.

4.1 Deriving a Domain-Specific Transformation Language

In [28] derivation rules to derive a domain-specific transformation language (DSTL) form a given modeling language were presented. A DSTL is composed of a common base grammar that provides modeling language independent parts of the DSTL as well as a derived grammar for the modeling language dependent parts. The derived grammar is created according to the derivation rules presented. The derivation rules create the non-terminals for the different operators

of the DSTL and the start symbol. The start symbol combines the non-terminals provided by the base and the derived grammar to form a transformation rule. This derivation process was applied to create the DSTL CDTrans that is suitable to describe transformations for class diagrams modeled using the modeling languages described in Sect. 2.2. Figure 4 demonstrates the derivation rules for the non-terminal `Attribute` of the CD4A grammar. The non-terminal of CD4A is depicted at the top, the derived non-terminals at the bottom.

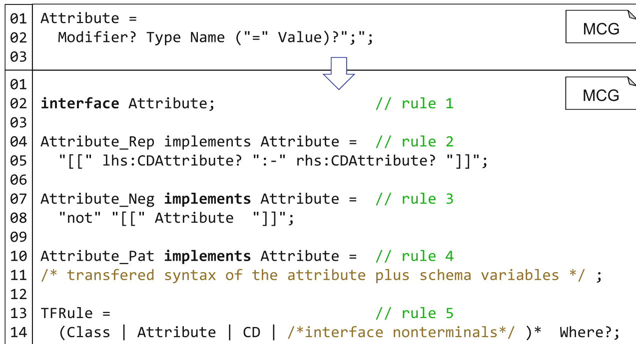


Fig. 4. Application of the derivation rules described in [28].

In [28] there are basically five derivation rules described. The first of which derives interface non-terminals for the non-terminals and keywords of the modeling language (`Attribute`, l. 2). The second rule derives non-terminals for the replacement of each model element (cf. `Attribute_Rep`, l. 4), while the third rule derives non-terminals to forbid model elements[1] (`Attribute_Neg`, l. 7). The forth rule derives the non-terminals to transfer the concrete syntax of the model elements to the DSTL (`Attribute_Pat`, l. 10) and allows to use schema variables (consisting of a name that starts with a $-sign), *e.g.,* for names of modeling elements such as the attribute name. Finally, the start symbol that combines the interface non-terminals to an alternative and adds the option to specify an application constraint is created in the fifth derivation rule (`TFRule`, l. 13). For further explanation please refer to [28].

A transformation rule modeled using CDTrans is shown in Fig. 5. This transformation matches an arbitrary class (indicated by the schema variable `$_`) that has a public attribute (l. 2). The public visibility of the attribute is changed to private (l. 2) and public access methods are added (ll. 4–5). Please not that transformation rules modeled via CDTrans use an integrated notation of the left-hand side (LHS) and right-hand side (RHS) of a transformation rule. Thus, modification within the pattern are expressed directly at the pattern element affected by the modification (cf. Fig. 5, ll. 2, 4–5). The left part of the replacement operator (`[[ :- ]]`) (*i.e.,* left of the `:-` is part of the pattern), while the

[1] This corresponds to negative application conditions [23].

```
01 class $_ {
02   [[public  :- private]] $type $attrname;
03
04   [[ :-  public $type $get(); ]]
05   [[ :-  public void $type $attrname); ]]
06 }
07
08 where {
09   $get = "is"  +  capitalize($attrname);
10   $set = "set" +  capitalize($attrname);
11 }
```
MTR

Fig. 5. A model transformation rule to encapsulates attributes by changing its visibility to private and adding public access methods.

part right of it replaces the left part or is added if the left part is left blank. Finally the where clause is used to calculate the values of the variables $`get` and $`set` used for the names of the added access methods. `capitalize(...)` is a built in function to capitalize a string value, *e.g.,* names.

5 Engineering a Data Explorer

To demonstrate the applicability of the presented concepts and methods, we present a use case for model-driven development of data-centric applications from structural models, *i.e.,* CD4A models (cf. Sect. 2.2). This demonstrates (a) the use of CD4A for generating executable data-centric applications, and (b) the use of domain-specific transformations for code generation. In general, a data-centric applications manages structured and consistent information by providing SCRUD (search, create, read, update, and delete) functionality [38] through a graphical user interface. The strength of data-centric applications is that the generated source code is aware of the managed data. For example, from the CD4A model in Fig. 2, the data-centric application shown in Fig. 6 is generated.

As only one kind of input models is used as input, adaptation and customization concerns are addressed by the code generator and in the generated code. An overview of different adaptation approaches for generated code is given in [19]. Where adapting the generated code is not feasible, code generator customization can be achieved by integrating transformation- and template-based code generation using the CD4Code language (cf. Sect. 3.1) as an intermediate representation of the object-oriented structure of the target code.

An overview of the code generation approach is shown in Fig. 7. After parsing the CD4A model, the resulting AST is transformed into a CD4Code AST, which is gradually transformed (cf. Sect. 4.1) until the CD4Code AST describes the object-oriented structure of a data-centric application. Since CD4Code does not contain target language specific source code, templates are attached to CD4A method and constructors to realize their bodies. In addition, default templates are added to describe the mapping of CD4Code language concepts to Java source code. Finally, the transformed CD4Code AST and the templates are passed to a template engine to generated Java source code.

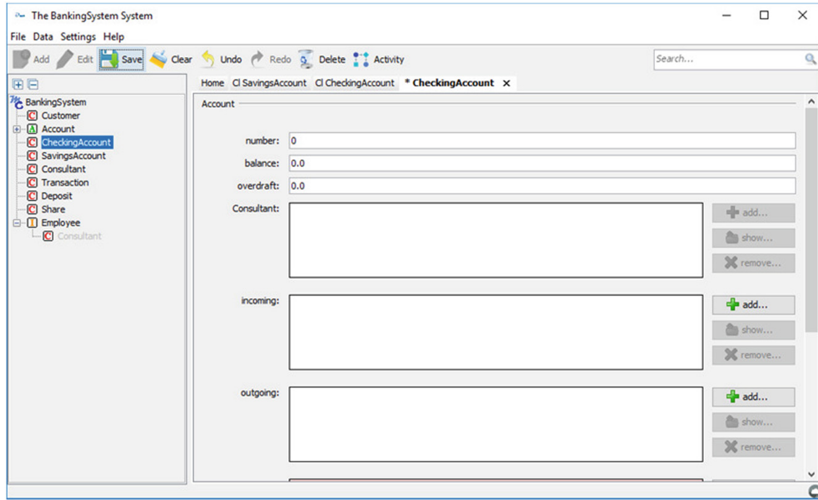


Fig. 6. Part of the data explorer generated from the CD4A model in Fig. 2.

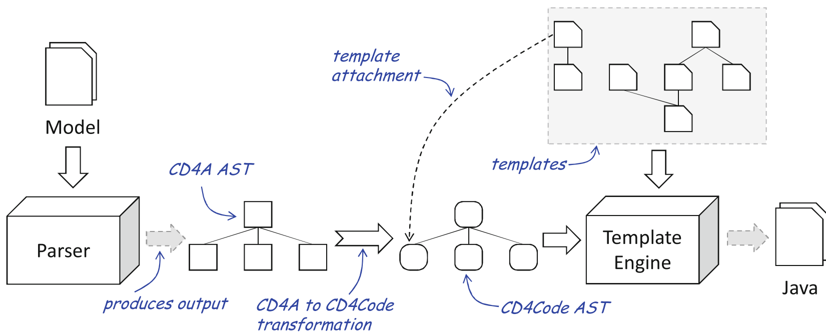


Fig. 7. An overview of the code generation activities that uses domain-specific transformations on the CD4Code AST and additional templates.

Adaptability and customizability of the code generation approach is achieved by employing transformations on the CD4Code AST and attaching templates to individual CD4Code AST nodes in the intermediate representation. This code generation approach shows the effective use of transformations to reduce complexity of template-based code generation by outsourcing computations on the AST to pattern matching, which is used in transformation-based approaches. It, furthermore, shows that transformations in code generation may enable reuse if the same intermediate representation is used.

6 Conclusion

Ludwig Wittgenstein once said that the limits of his language are the limits of his world. While programming languages are pretty expressive in describing structure and operations and data, and general-purpose modeling languages like the UML are good in specifying structure, architecture, behavior of software systems, these languages suffer from not being very domain oriented.

Today many domains are being digitalized and a lot more non-software people have to deal with encoding their information, knowledge, methods and procedures. Thus good languages for domain people to describe their information are needed. This includes models of various unforeseen forms and thus needs a strong field of language engineering.

Language engineering includes a systematic way of development of language components, integrating and composing them into larger languages, modifying and extending language components as desired, to easily accommodate the evolution of digitalized domains.

In this paper, we have discussed these techniques on three levels: a data exploration tool for a concrete data structure (level 1) is generated using a data exploration generator (level 2), which in turn is developed using a typical Language Workbench, called MontiCore (level 3). Only on the level of language workbenches, language engineering techniques become feasible.

Even though the principles are to some extent understood, it still takes some time to make industrial capital out of these techniques.

References

1. Abell, J.: MATLAB and SIMULINK. Modeling Dynamic Systems. CreateSpace Independent Publishing Platform, Seattle (2016)
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML). World Wide Web J. **2**(4), 27–66 (1997)
3. Clark, T., den Brand, M., Combemale, B., Rumpe, B.: Conceptual model of the globalization for domain-specific languages. In: Cheng, B.H.C., Combemale, B., France, R.B., Jézéquel, J.-M., Rumpe, B. (eds.) Globalizing Domain-Specific Languages. LNCS, vol. 9400, pp. 7–20. Springer, Cham (2015). doi:10.1007/978-3-319-26172-0_2
4. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley, Boston (2002)
5. Combemale, B., Deantoni, J., Barais, O., Blouin, A., Bousse, E., Brun, C., Degueule, T., Vojtisek, D.: A solution to the TTC'15 model execution case using the GEMOC studio. In: 8th Transformation Tool Contest. CEUR (2015)
6. Czarnecki, K.: Generative programming-principles and techniques of software engineering based on automated configuration and fragment-based component models. Ph.D. thesis, Technical University of Ilmenau (1998)
7. Date, C.J., Darwen, H.: A Guide to the SQL Standard, vol. 3. Addison-Wesley, New York (1987)
8. Deantoni, J.: Modeling the behavioral semantics of heterogeneous languages and their coordination. In: Architecture-Centric Virtual Integration (ACVI), pp. 12–18. IEEE (2016)

9. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: a meta-language for modular and reusable development of DSLs. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 25–36. ACM (2015)
10. Ellner, S., Taha, W.: The semantics of graphical languages. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2007, pp. 122–133. ACM, New York (2007)
11. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA 2012. ACM, New York (2012)
12. Erdweg, S., Kats, L.C.L., Rendel, T., Kästner, C., Ostermann, K., Visser, E.: Library-based model-driven software development with SugarJ. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, pp. 17–18. ACM (2011)
13. Erdweg, S., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Wyk, E. (eds.) SLE 2013. LNCS, vol. 8225, pp. 197–217. Springer, Cham (2013). doi:10.1007/978-3-319-02654-1_11
14. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH 2010, pp. 307–309. ACM, New York (2010)
15. Favre, J.-M., Gasevic, D., Lämmel, R., Pek, E.: Empirical language analysis in software linguistics. In: Malloy, B., Staab, S., Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 316–326. Springer, Heidelberg (2010). doi:10.1007/978-3-642-19440-5_21
16. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley, Boston (2012)
17. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Boston (2010)
18. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering (FOSE 2007), no. 2, pp. 37–54 (2007)
19. Greifenberg, T., et al.: Integration of handwritten and generated object-oriented code. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) MODELSWARD 2015. CCIS, vol. 580, pp. 112–132. Springer, Cham (2015). doi:10.1007/978-3-319-27869-8_7
20. Greifenberg, T., Look, M., Roidl, S., Rumpe, B.: Engineering tagging languages for DSLs. In: Conference on Model Driven Engineering Languages and Systems (MODELS 2015), pp. 34–43. ACM/IEEE (2015)
21. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore: a framework for the development of textual domain specific languages. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, 10–18 May 2008, Companion Volume, pp. 925–926 (2008)
22. Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3, 03 May 2010
23. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3), 287–313 (1996)
24. Haber, A., Hölldobler, K., Kolassa, C., Look, M., Müller, K., Rumpe, B., Schaefer, I., Schulze, C.: Systematic synthesis of delta modeling languages. J. Softw. Tools Technol. Transf. (STTT) **17**(5), 601–626 (2015)

25. Haber, A., Look, M., Mir Seyed Nazari, P., Navarro Perez, A., Rumpe, B., Völkel, S., Wortmann, A.: Composition of heterogeneous modeling languages. In: Desfray, P., Filipe, J., Hammoudi, S., Pires, L.F. (eds.) MODELSWARD 2015. CCIS, vol. 580, pp. 45–66. Springer, Cham (2015). doi:10.1007/978-3-319-27869-8_3
26. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? IEEE Comput. **37**(10), 64–72 (2004)
27. Hermerschmidt, L., Hölldobler, K., Rumpe, B., Wortmann, A.: Generating domain-specific transformation languages for component & connector architecture descriptions. In: Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp 2015). CEUR Workshop Proceedings, vol. 1463 (2015)
28. Hölldobler, K., Rumpe, B., Weisemöller, I.: Systematically deriving domain-specific transformation languages. In: Conference on Model Driven Engineering Languages and Systems (MODELS 2015), pp. 136–145. ACM/IEEE (2015)
29. Iverson, W.: Hibernate: A J2EE (TM) Developer's Guide. Addison-Wesley Professional, Boston (2004)
30. Jézéquel, J.-M., Barais, O., Fleurey, F.: Model driven language engineering with kermeta. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 201–221. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18023-1_5
31. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like transformation language. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (2006)
32. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley, Boston (2008)
33. Knuth, D.E.: Semantics of context-free languages. Theory Comput. Syst. **2**(2), 127–145 (1968)
34. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(5), 353–372 (2010)
35. Larsen, M.E.V., Deantoni, J., Combemale, B., Mallet, F.: A behavioral coordination operator language (BCOoL). In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2015)
36. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
37. Miller, F.P., Vandome, A.F., McBrewster, J.: Apache Maven (2010)
38. Mir Seyed Nazari, P., Roth, A., Rumpe, B.: Mixed generative and handcoded development of adaptable data-centric business applications. In: Domain-Specific Modeling Workshop (DSM 2015), pp. 43–44. ACM (2015)
39. Radjenovic, J., Milosavljevic, B., Surla, D.: Modelling and implementation of catalogue cards using freemarker. Program **43**(1), 62–76 (2009)
40. Richters, M., Gogolla, M.: On formalizing the UML object constraint language OCL. In: Ling, T.-W., Ram, S., Lee, M. (eds.) ER 1998. LNCS, vol. 1507, pp. 449–464. Springer, Heidelberg (1998). doi:10.1007/978-3-540-49524-6_35
41. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Language and code generator composition for model-driven engineering of robotics component & connector systems. J. Softw. Eng. Robot. (JOSER) **6**(1), 33–57 (2015)
42. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer, Cham (2016). doi:10.1007/978-3-319-33933-7
43. Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer, Cham (2017). doi:10.1007/978-3-319-58862-9

44. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education, London (2008)
45. Vacchi, E., Cazzola, W.: Neverlang: a framework for feature-oriented language development. Comput. Lang. Syst. Struct. **43**, 1–40 (2015)
46. Viyović, V., Maksimović, M., Perisić, B.: Sirius: a rapid development of DSM graphical editor. In: 2014 18th International Conference on Intelligent Engineering Systems (INES), pp. 233–238. IEEE (2014)
47. Voelter, M., Solomatov, K.: Language modularization and composition with projectional language workbenches illustrated with MPS. In: Software Language Engineering, SLE, vol. 16, p. 3 (2010)
48. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages (2013). dslbook.org
49. Wachsmuth, G.H., Konat, G.D.P., Visser, E.: Language design with the spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)
50. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014)