Modeling Language Variability with Reusable Language Components

Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
<lastname>@se-rwth.de

ABSTRACT

Proliferation of modeling languages has produced a great variety of similar languages whose individual maintenance is challenging and costly. Reusing the syntax and semantics of modeling languages and their heterogeneous constituents, however, is rarely systematic. Current research on modeling language reuse focuses on reusing abstract syntax in form of metamodel parts. Systematic reuse of static and dynamic semantics is yet to be achieved. We present an approach to compose syntax and semantics of independently developed modeling languages through language product lines and derive new stand-alone language products. Using the MontiCore language workbench, we implemented a mechanism to compose language syntaxes and the realization of their semantics in form of template-based code generators according to language product line configurations. Leveraging variability of product lines greatly facilitates reusing modeling language and alleviates their proliferation.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; *Extensible languages*; *Software product lines*;

KEYWORDS

Language Variability, Language Product Lines, Software Language Engineering

ACM Reference Format:
Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *Proceedings of 22nd International Systems and Software Product Line Conference, Gothenburg, Sweden, September 10–14, 2018 (SPLC '18),* 11 pages.
https://doi.org/10.1145/3233027.3233037

1 INTRODUCTION

Modeling to understand and shape the world is an essential human abstraction 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. Whilst humans employed modeling for ages and in virtually all disciplines, it is recent that the form of models is made explicit in modeling languages. Computer science has invented this approach to enable a precise understanding of what is a well-formed model in the communication between humans and machines. The general aspiration of such languages creates a conceptual gap between the problem domains and the solution domains that raises unintended complexities [11]. Consequently, research in industry produced a large body of domain-specific languages (DSLs) [37] to match domain-specific needs. With the ongoing digitization of virtually every domain in our life, work, and society, the need for even more DSLs raises. This proliferation raises three questions:

(1) How to create new DSLs that fit specific purposes?
(2) How to engineer DSLs from predefined components?
(3) How to efficiently derive DSLs from other DSLs?

In this paper, we address the second question through reusable language components arranged as a product line of software languages. From these, product owners can configure language products, *i.e.,* variants of the product line, for specific purposes (*e.g.,* domains, applications) without being forced to understand the intricate details of each participating language. Based on the selected language features, its grammars, well-formedness rules, and code generators are composed automatically, such that the result can be used transparently by the modelers. This extends our previous work on syntactic language reuse [4] with composition of code generators. The contributions of this paper, hence, are:

- A concept for syntactic and semantic modeling language variability based on language product lines over language components.
- An extension to our modeling technique for language components combining grammars and well-formedness rules [4] with code generators. Resulting language components are decoupled from a specific language product line and, hence, can be reused in different contexts as well.
- A composition mechanism for code generators of the participating independent languages.
- A realization of our concept with the MontiCore [15] language workbench [10].

With this extensible language variability mechanism in place, new languages can be configured using existing components more efficiently. Hence, the mechanism reduces the effort in engineering software languages for specific contexts as well as the proliferation of modeling languages.

In the following, Section 2 motivates the benefits of our approach and Section 3 presents preliminaries. Afterwards, Section 4 presents

This research has partly received funding from the German Federal Ministry for Education and Research under grant no. 01IS16043P. The responsibility for the content of this publication is with the authors.

SPLC '18, September 10–14, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-6464-5/18/09…$15.00
https://doi.org/10.1145/3233027.3233037

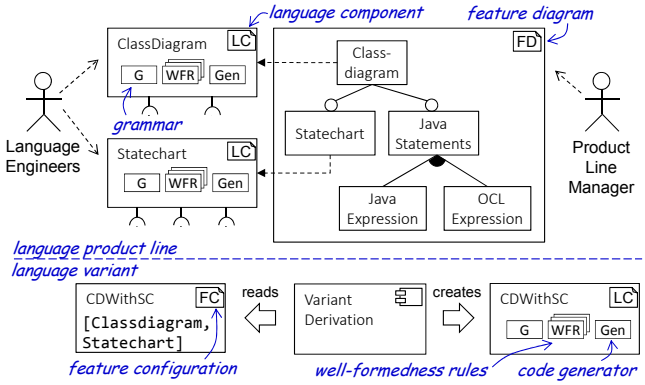

Figure 1: A language product line with a selected variant.

our language variability concept. Section 5 then describes code generators and our mechanism for their integration and Section 6 leverages these to describe language product lines. Section 7 presents an in-depth example, before Section 8 discusses observations and highlights related work. Section 9 concludes.

2 BACKGROUND AND EXAMPLE

Conceptually, there are various techniques to combine two languages, *e.g.,* through merging into embedded models [6] or integration of separate models [15]. Moreover, languages are rarely homogeneous artifacts, but often their definition requires the interaction of multiple meta-languages and programming languages. Popular combinations of these technological spaces, for instance, are ECore [28] metamodels to describe a languages' abstract syntax with OCL [14] for its well-formedness rules and Xtend [2] for code generators. Alternatives are different forms of grammars [30, 39] with GPL well-formedness rules [22] and dedicated languages for model-to-model transformations [19]. There are almost as many technological spaces as there are language workbenches [10] and most support different language composition mechanisms [9].

Consequently, systematic reuse of languages in a meaningful [16] fashion is nearly as complicated as engineering new languages from scratch and capturing the dependencies and relations between the loosely coupled language constituents requires in-depth language engineering expertise. Leveraging dedicated language components that capture these relations and structuring reuse through variability modeling techniques can greatly facilitate language engineering.

Consider developing a language product line for modeling applications, in which the application structure is modeled with class diagrams and there are different options how method bodies are realized with embedded behavior languages. Explicating the variability of such a language product line through feature models over consolidated language components reduces the complexity of identifying and integrating language constituents. Composition of these is performed systematically and does not require an in-depth understanding of the individual language components. An overview of a language product line, the employed language components, and the derived language variant for the example scenario is visualized in Figure 1. The top depicts the language product line comprising a feature model, where each feature references a language component, *i.e.,* constituents of a language definition with respect to a specific language workbench [10]. Each language component contains a grammar (G), well-formedness rules (WFR), and a code generator (Gen), which together realize the syntax and semantics of the language. Moreover, a language component can define named and typed extension points by underspecifying certain parts of its syntax and semantics. Language components are independent of each other and can be developed by different software language engineers individually. The feature model defines combinations of related language components considered valid by a product line manager. Product line managers are language engineers who create language families as feature models that describe possible characteristics of the family's language products. To this end, they collect relevant language components, assign these to features, and define how these realize extension points of language components of their parent features. The latter is realized by mapping an extension point of a parent feature to an extension in the child feature. Further, product line managers ensure that the employed code generators translate to the same or a compatible target language.

In this example, every language variant of the product line uses class diagrams to model the application's structure. The behavior of method bodies can be modeled with embedded Statecharts or statements of the Java/P [7] action language. These Java/P statements rely on expressions, which are either realized as Java expressions, OCL expressions, or both. The bottom left part of Figure 1 depicts the feature configuration `CDWithSC` defining a language variant. The feature configuration is selected by a language product owner, who is an expert in the domain that the language variant is to be used in. The variant includes the `Classdiagram` and the `Statechart` feature. Given this feature configuration as input, the variant derivation tool derives a language variant by composing the language components of all selected features, resulting in a new language component (depicted at the bottom right). This language component has a composed grammar, aggregated well-formedness rules, and a composed code generator. Afterwards, a language workbench, given this language component as input, produces tooling capable of processing models conforming to the language variant. This typically includes a parser, abstract syntax data structure, a checking infrastructure for the well-formedness rules, and coordination of code generation. The tailored tooling can be used by modelers to develop models conforming to the language variant.

3 PRELIMINARIES

This section presents the language workbench MontiCore [15, 27] and the concept for composition of independent MontiCore grammars as presented in [4].

3.1 MontiCore Language Workbench

Our concept for language variability is realized with the MontiCore language workbench [15, 27]. MontiCore supports development of modular modeling languages. It comprises a grammar modeling language and a tool chain for the efficient engineering of textual languages and their infrastructure (parsers, analyses, transformations, code generators). MontiCore employs context-free grammars

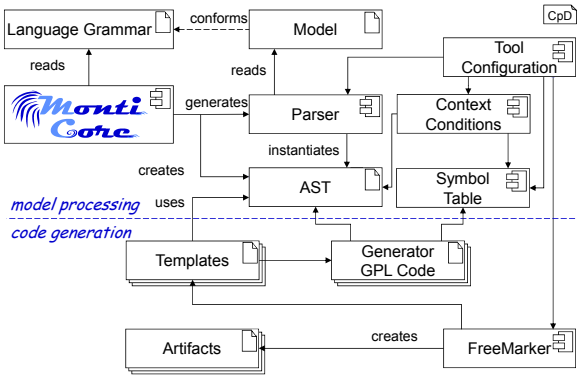

Figure 2: The quintessential components and artifacts of a MontiCore language: From a grammar, MontiCore generates a parser, abstract syntax classes as well as infrastructure for context condition checking and code generation.

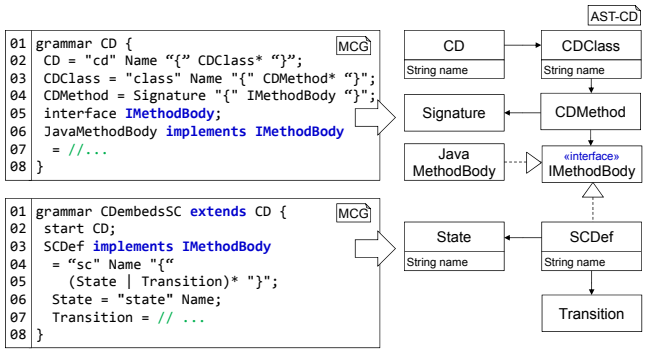

Figure 3: Grammar inheritance in MontiCore.

for integrated definition of abstract and concrete syntax. The grammars describe, which models are principally possible and Java well-formedness rules restrict these. Each grammar contains production rules, may extend other grammars, and yields a dedicated start rule. From the grammars, MontiCore generates model processing infrastructure to parse textual models into abstract syntax trees (ASTs), which store the content of models, such as their elements and their relations. MontiCore supports compositional Java context conditions checking the models' well-formedness that a language-specific, generated visitor applies to the ASTs. Template-based code generators realize the DSMLs' semantics. To this end, MontiCore provides an extensible code generation framework based on the FreeMarker template engine [1]. Figure 2 illustrates the quintessential components and artifacts of MontiCore and their relations.

MontiCore also supports compositional integration of modeling languages through inheritance, embedding, and aggregation [15]. *Inheritance* enables modeling languages to extend and override production rules of their (possibly multiple) parent languages. From inheriting DSMLs, MontiCore produces refined AST classes that inherit from the AST classes of the overridden production rules. MontiCore also features *interface* production rules, which enable underspecification in grammars by prescribing only the required abstract syntax elements of implementations. We leverage this through inheritance to integrate new production rules into these well-defined extension points as depicted in Figure 3.

The grammar `CD` (top left) is an excerpt of a grammar describing textual class diagrams. Each of these class diagram comprises classes that have a name and can contain methods (ll. 2-3). Methods have a signature and a method body, where the latter is realized as an interface production rule that underspecifies a concrete production rule body (l. 5). The interface production rule can be implemented by other production rules, *e.g.*, by the production rule `JavaMethodBody` (ll. 6-7). From this grammar, MontiCore generates six AST classes (depicted top right), out of which `IMethodBody` is an interface implemented by the AST class `JavaMethodBody`. Interface production rules can be used through grammar inheritance. For instance, the grammar `CDembedsSC` (Figure 3, bottom) extends the grammar `CD` (l. 1) and provides further implementation of the interface `IMethodBody`, which in addition to the Java method bodies, features Statecharts as method bodies (ll. 3-5). Accordingly, MontiCore generates new AST classes for newly introduced production rules and reuses all modeling elements of `CD`. The start production of a grammar determines the root of the generated AST and, thereby, also the return type of a parser. MontiCore uses the first production rule of a grammar as start production by default. With the keyword `start` (*cf.* l. 2), MontiCore can be set to use a different production rule than the one of a grammar as start production. In this example, MontiCore is set to use the start production `CD` of the grammar `CD` for the grammar `CDembedsSC`.

3.2 Composing Grammars and Context Conditions

The composition of grammars for achieving language embedding as explained in the last section requires that the embedded grammar has a dependency to the embedding grammar. As one of our concerns is *independent* development of language components, this mechanism is not feasible. Composition of independent grammars relies on syntactic extension points (interface production rules) of a base grammar and a binding indicating, which production rules of the implementing grammar connect to which extension point [4]. To achieve the composition of two grammars, we employ a specific variant of language embedding [15], which combines the syntaxes of the two languages through multiple-inheritance in a generated, third grammar. This new grammar leverages MontiCore's production rule extension and production rule implementation mechanisms to implement extension points of the base grammar with production rules of the embedded grammar. This mechanism especially enables to integrate arbitrary production rules of the embedded grammar into the base grammar and does not require any language developers being aware of this possible interaction. Extending the embedded production rule causes the generated abstract syntax classes of the extending production rules to become subclasses of the classes generated from the extended production rule. Consequently, all model analyses and transformations implemented against the original abstract syntax class can be reused without additional effort.

Embedding, for instance, Statecharts into class diagrams as depicted in Figure 4 requires that the `CD` grammar provides an interface

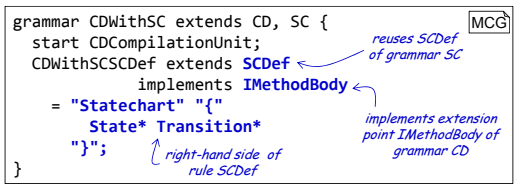

Figure 4: The composition of two independent grammars.

production rule as extension point (here `IMethodBody`) and a binding from a Statechart production rule (here `SCDef`) to the interface production rule extension point. In the example of `CDWithSC`, the binding realizes the integration between method bodies and the definition of Statecharts. This composition enables using the syntax of Statecharts in method bodies of class diagrams to describe their behavior. The grammar generated from this composition extends both the base grammar `CD` and the embedded grammar `SC`. The grammar `CD` is as depicted in Figure 3 and the grammar `SC` is similar to the grammar `CDembedsSC` depicted in the same figure. In contrast to `CDembedsSC`, `SC` does not require to extend the `CD` grammar as it is independent of this. Further, it has no reference to the start production of `CD` and `SCDef` does not implement the interface `IMethodBody`. By generating a new, composed grammar extending both individual grammars, all production rules from these become available in the new grammar. Our grammar composition does not prohibit dependencies between the base grammar and embedded grammar per se [4]. However, embedding production rules of a grammar into a base grammar on which the embedded grammar depends may break the composition, which MontiCore detects.

The integration of well-formedness rules (context conditions) is less complicated. These are realized as Java classes implementing an interface specific to the AST class of the production rule they operate on. As the synthesized production rules extend from production rules of the embedded grammar, the context conditions generally are applicable to these as well. Our integration hence collects the context conditions from both language components and registers these to a generated visitor that applies these accordingly. Through integration of handcrafted code, MontiCore also supports adding inter-language context conditions specific to the integration of both languages that cannot be defined for one language alone [13].

4 A CONCEPT FOR CODE GENERATOR VARIABILITY

Creating a language product line begins with language engineers developing the modeling languages that the language product line combines. We assume the languages are defined in terms of grammars, well-formedness rules, and code generators. Consequently, development begins with the grammars, which prescribe the languages *AS* and *CS*, as well as its possible extension points using, *e.g.,* underspecification, in the grammar description mechanism [4]. Afterwards, the language engineers create well-formedness rules to enable rejection of models not considered well-formed. This usually is necessary as the meta-languages used with common grammar-based language engineering methods lack sufficient mechanisms to describe well-formedness without additional rules. We assume

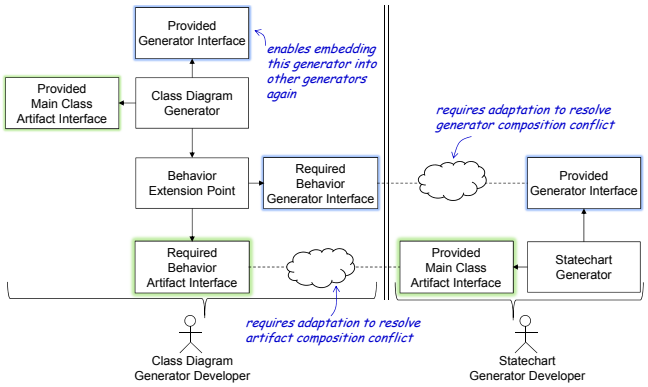

Figure 5: Conceptual representation of composable generators with required and provided interfaces on the example of class diagrams that embed Statecharts.

that the dynamic semantics of a language are realized through code generators that translate models into executable GPL artifacts. Variability of language components consisting of grammar-based languages using Java well-formedness-rules is presented in [4] and briefly recapitulated in Section 3.2.

To extend this notion to code generators, we include a reference to a code generator class into language component models. Based on selected features, we compose the related generators by *embedding* these into another. However, if the language components' generators were not developed for usage in a language product line, the product line manager's feature model raises two kinds of conflicts between features and their immediate parent features:

- *Generator composition conflict:* The code generator of a feature implements an interface not expected for embedding with the generator of the parent feature.
- *Artifact composition conflict:* The code generator of a feature produces artifacts of a type the embedding generator of the parent feature is unaware of.

The language variability infrastructure presented in this paper synthesizes adapters together with a lookup and instantiation framework for both kinds of conflicts based on the feature model and their language components' generator properties. However, the specific adaptation generally is inaccessible to automation as the interfaces on both levels are completely unknown at design time and, hence, may differ significantly. Consequently, the infrastructure generates a framework expecting handcrafted implementation of generator adaptation and artifact adaptation by the product line manager. Adding these two implementations per pair of composed generators enables automated execution of code generation and ensures structural compatibility of generated code by construction.

To resolve the generator composition conflict, generators for grammars with extension points, *i.e.,* underspecified elements, must support similar extension points for responsible code generators. For instance, a code generator for a class diagram grammar supporting embedding various behavior languages for method body implementations must yield an *extension point for generators* responsible for translating behavior models. As the class diagram

generator generally is unaware of any concrete behavior generator it prescribes a code *generator interface* for compatible behavior generators. One possible realization of behavior models could be Statecharts. Statechart generators, however, are generally unaware of the generator interfaces prescribed by the class diagram generator. To compose both generators nonetheless, we also require that each code generator implements a dedicated *code generator interface* itself. Making both, the required and the provided interfaces, explicit enables *automated construction of code generator adapters*. As the specific adaptation cannot be derived, the implementations of the generated adapters must be handcrafted and extend the generated abstract adapter base classes. Our framework then uses these for actual adaptation. Through this, *e.g.,* the class diagram generator can delegate generation of embedded Statechart models to the Statechart generator. This includes to translate available information into parameters that the Statechart generator requires.

This, however, does not resolve the artifact composition conflict, *i.e.,* that the jointly generated code is structurally compatible. The target language of code generation is typically a general purpose language, *e.g.,* an object-oriented programming language. These generally are not expressive enough to define contracts on arbitrary statements, expressions, or blocks. Therefore, we must rely on *structural compatibility between classes* through contracts, *e.g.,* through abstract base classes, implementation of interfaces, *etc.* Enabling class-wise compatibility between code produced by different generators requires that

- each embedded generator produces a *dedicated main class* (which may interact with other classes produced by the same generator);
- embedding generators specify the contract required by possible implementations (*e.g.,* behavior implementations); and
- embedded generators specify the contract provided by the generated main class as its *artifact interface*.

Explicating these provided and required contracts enables generating abstract *artifact adapters* between, *e.g.,* the interface expected for behavior implementations by the class diagram generator and the interface provided by Statechart behavior implementations by the Statechart generator. Similar to the generator interfaces, the actual adaptation requires specific handcrafted implementations to extend the generated adapter base classes, which then are incorporated by the framework automatically. Figure 5 illustrates the generators and interface related to embedding Statecharts into class diagrams. Our method to black-box composition of code generators therefore raises the following requirements:

RQ1 Each participating code generator implements a dedicated *provided generator interface* that describes its usage.

RQ2 Code generators for grammars with extension points support registration of other code generators responsible for translating implementations of these extension points. For each extension point, they describe the *required generator interface* and the *required artifact interface* related to the generated code.

RQ3 Each code generator produces a *single main class* and specifies its *provided artifact interface*.

With this in place, the interfaces of *generator adapters* and of *artifact adapters* (one of each per generator pair to be composed) can

```
01 language ClassDiagram {                    LC
02   grammar CD;                  selected grammar
03   root CDCompilationUnit;      start rule of reused language part
04   cocos {
05     CDDiagramNameUpperCase,    selected context conditions
06     CDClassNameUnique
07   }
08   generator CD2JavaGen;        code generator for root syntax element
09 }
```

```
01 language Statechart {                       LC
02   grammar SC;
03   root SCDef;
04   cocos {
05     UniqueStateNames,
06     // ...
07   }
08   generator SCJavaGen;
09 }
```

Figure 6: Two language component aggregating class diagram and Statechart language constituents, respectively.

be derived automatically. Provided handcrafted implementations are integrated into the composed generators automatically. We lift this composition to language product lines by applying it to all embedding (*i.e.,* in a parent feature) and embedded (from its child feature) code generators automatically. Hence, the product line manager must implement twice the number of participating code generators as adapters only.

The next section presents the realization of this variability concept that addresses both generator composition conflicts. The subsequent section explains how the product line manager leverages this to create language product lines.

5 COMPOSING INDEPENDENT CODE GENERATORS

The quintessential building blocks for language integration are language components. These aggregate the syntax and the realization of the semantics of a MontiCore language in terms of a grammar (concrete and abstract syntax), Java context conditions (static semantics), and a code generator (realizing dynamic semantics). For instance, Figure 6 illustrates a language component for class diagrams, which references its grammar (l. 2), its well-formedness rules (ll. 4-7), and its generator (l. 8). Optionally, a root production (l. 3) of the grammar can be selected. This enables to reuse only a subtree of the related grammar's abstract syntax (*e.g.,* to reuse only Java expressions of a grammar describing the syntax of Java classes). If no root production is specified, the default start production of the grammar is used. Each interface rule of the grammar becomes an extension point of the language component, where the name of the extension point is the name of the respective interface rule. Interfaces in the grammars therefore are used for typing extension points. Deriving a variant of the language product line entails composing the language components of all selected features. The different language's constituents require different composition mechanisms. The composition mechanisms for grammars and context conditions have been explained in Section 3, this section presents the realization of the black-box composition mechanism for code generators.

Composing code generators is an ongoing challenge that raises the questions of syntactic and semantic conformance. Our approach to code generator composition enables syntactic integration of code generators and generated code, which, by construction, ensures that generators and generated code interact syntactically. Whether behavior of generators or generated code generally is meaningful [16] is as complex as the halting problem [29] and not part of our composition.

Our general approach is to employ adaptation between the explicit interfaces of code generators to enable their interaction at generation time. Through the language components, it is clear which generators must be combined and the grammars of their language components prescribe the generators' extension points (*e.g.,* the generator for `CD` must be able to invoke generators responsible for translating `IMethodBody` instances. Through architectural constraints (such as implementing a single execution interface per generator), abstract adapter classes for the composition of two language components' generators can be generated. Using the generation gap pattern [33], the developer composing two language components with their code generators must provide a proper implementation of the interface imposed by the generated adapter. By construction, the generator of the base language then can call the generators of the embedded languages, pass model parts to these and invoke code generation. Leveraging the assumption that each generator produces at least a main GPL artifact (*e.g.,* a Java class) responsible for interacting with the generated code, the base language generator can produce code instantiating this and using it as intended.

For the integration of generated code artifacts, the embedding generator prescribes for each extension point what it expects from the code produced by generators registered for this extension point within a required artifact interface. For instance, the generator translating `CD` instances could prescribe that code produced for realizations of `IMethodBody` must implement a specific GPL interface. As language components can be developed independent of each other, this expectation, however, rarely is fulfilled. Hence, we impose that all generators also explicate the interface of the main GPL artifact they generated. Generators are mapped transitively to language component extension points for which the generator of the embedding language component prescribes a specific interface. The code produced by the embedded generator provides a mapping between expected GPL artifact interface and GPL artifact interface. Thus, we also can leverage adaptation between these interfaces and generate abstract adapters accordingly. For these, usually one per pair of generators, also a proper implementation must be provided. This also is integrated through the generation gap pattern [33].

For adaptations to work, all generators must be implemented in the same GPL and produce code of the same GPL. The GPLs for code generator implementation and for generated code may differ. Future work on cross-GPL code generator invocation and platform-independent artifact interfaces can mitigate this. Consequently, we make the following assumptions for a realizing the generator composition:

(1) All code generators are implemented in the same GPL and all artifacts are implemented in the same GPL.
(2) Each code generator produces at least one main GPL artifact for which it explicates its GPL interface. The generator ensures that all generated main artifacts comply to this interface.
(3) All extension points of generators have to be explicated at design time of the generator. Generator extension points are typed with the main abstract syntax element they translate (*e.g.,* `IMethodBody`) and the interface expected from a generator registered to translate instances of this element.
(4) Operation of code generators may not rely on assumptions regarding code generated by other generators that are not made explicit through their interfaces.

We assume that each code generator constitutes a generator class (*e.g.,* `CD2JavaGen` and `SC2JavaGen` in Figure 7) realizing the actual code generation. This class performs the code generation, *e.g.,* by invoking a template engine. Each generator class implements an interface describing types of the input and output of the generator (*e.g.,* `ICDGenerator` and `ISCGen` in Figure 7) as presented in [1]. Further, each generator references the type of an interface *typing* the main artifact of the generated code (*e.g.,* `IJavaClassArtifact` and `IStatechart` in Figure 7). For each extension point that is foreseen in the generated code of a generator, an additional class (*cf.* `ExtensionPointInfo`) describes the type of the main artifact in the code that is generated by a generator (*e.g.,* `IBehaviorArtifact`). Further, the additional class describes the interface of the generator producing the main artifact of the generator implementing the extension point (*e.g.,* `IBehaviorJavaGen`). Additionally, an extension point info has a reference to the abstract syntax type (*e.g.,* `IMethodBody`) that is being translated. To this effect, extending an extension point is realized by adapting an extension point interface of the generator defining the extension point to an interface of a generator realizing the extension.

These adapters are necessary to enable decoupled development of the involved code generators. While the interface of the adapter can be generated, it is impossible to automatically generate the implementation of the adapter as it requires in-depth understanding of the behavior and meaning of the generated code. The same mechanism is applied at artifact interface level, where the expected artifact interface of an extension point is adapted to the provided artifact interface of the embedded generator. We combine the classical adapter pattern [12] with a mechanism to integrate a handwritten implementation of the concrete adapter (the TOP mechanism [27]). Each generated adapter is an abstract class that has a *target* (or adaptee) that it implements, which is the (artifact or generator) interface required by the extension point. Further, each generated adapter yields an attribute *delegate* of the (artifact or generator) interface provided by the embedded generator. The class realizing the adapter has to be handcrafted and extends the generated abstract adapter class. The name of the generated adapter, therefore, is fix and can be derived automatically by the embedding generator to interact with (a) the registered, embedded generators, and (b) artifacts produced by these generators. The names of handcrafted adapters are also fix, as these have to be identical to the generated adapter names without the suffix `TOP`. This property simplifies instantiation of the correct adapters, which is explained in Section 7. The above mechanism is applied to all combinations of extension point and extension that are possible within the generators of a language product line to relieve product owners from being language engineers. The developer combining two language components, hence, is responsible for registering the generator of the embedded language component with the generator of the embedding language component. Also, the developer of the embedding generator must be aware of the general existence of adapters.

Considering the example of embedding `SC` into `CD`, a class diagram describes the architecture of an application and Statecharts

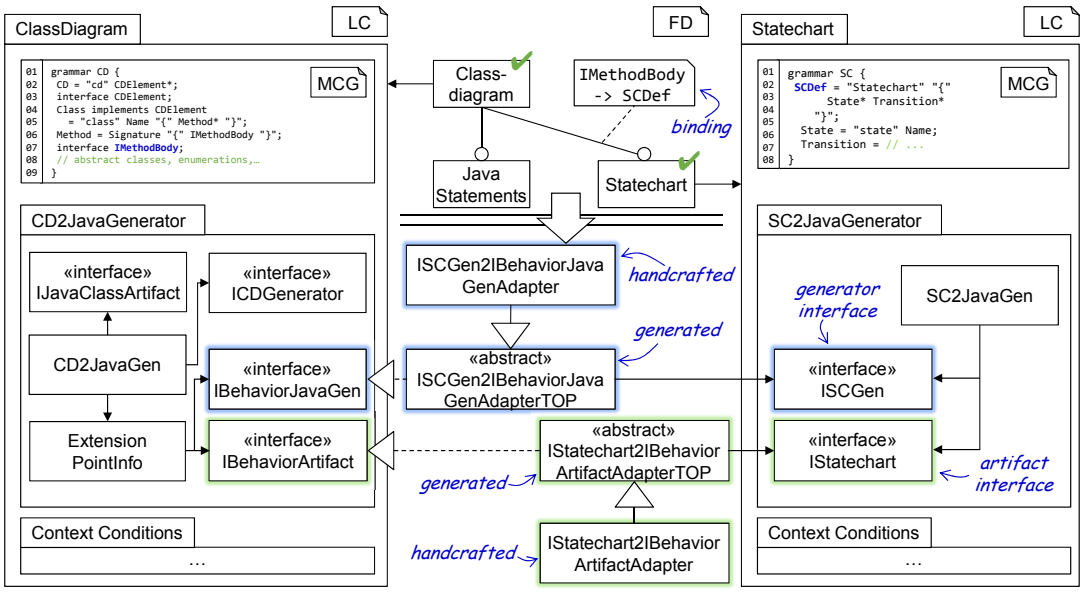

Figure 7: Constituents for a composition of two language components `ClassDiagram` and `Statechart`: The left depicts grammar, context conditions, and a code generator for class diagrams and the right respective constituents for Statecharts. The middle depicts the connection between these independent language components: The top visualizes the feature model and the binding between extension point and extension. The bottom presents the generated adapters and their handcrafted implementations.

can be employed to describe the behavior of methods modeled within the class diagram's classes. Figure 7 depicts the constituents of the individual generators of the class diagram language component and the Statechart language component whose functionality has been explained above. The generator for class diagrams has to be aware of the existence of adapters adapting any (generator or artifact) interface to the required (generator or artifact) interface of each extension point. From the mapping within the language product line, the code generator receives a map from the type of each extension point (*e.g.,* `IMethodBody`) to the concrete type of the extension (*e.g.,* `SCDef`). This information is used to invoke the correct generator. For instance, parsing a class diagram model containing a method body with a statechart results in an abstract syntax tree containing an object of type `CDWithSCSCDef` (*cf.* Figure 4). This type is a subtype of `SCDef` and therefore, the class diagram code generator invokes the generator for Statecharts via the registered generator adapter.

6 FEATURE-ORIENTED LANGUAGE ENGINEERING

This section explains how the variability and composition of language components as explained in Section 5 (the solution space [5]) integrates with the variability model and the derivation of variants in the problem space. With the feature model at hand, all possible compositions of language components are described at product line level. The composition mechanism and the definition of language components are loosely coupled to an employed feature modeling tool, which is therefore easily exchangeable. The set of bindings for a concrete feature model is realized as a model conforming to a dedicated small-scale DSL. The feature model restricts the cardinality of the bindings between several extensions and the extension point. To this effect, it also realizes the differentiation between optional and mandatory extension points. Although in general, dependencies between different language components may lead to unpredicted issues, such dependencies sometimes can be very useful. Hence, the feature model may indicate that a certain feature *requires* another feature, which allows the respective language component to have (*e.g.,* grammar inheritance) dependencies to other language components. Further, the composition of two specific language components can be forbidden using *excludes* in the feature model.

The language product line manager assigns language components to features and aligns the feature model meaningfully. After completing the product line, she generates adapters for all generators and artifacts of language components, which are directly related and implements the adaptations accordingly using the generation gap pattern.

Afterwards, a product owner can configure a language product by selecting desired features. After validating the feature configuration against the feature model, the language components and their constituents are resolved. Based on these, the new grammar that extends from the grammars of all related features is synthesized and for each binding, a new production rule is created that realizes this binding through grammar rule extension and implementation. Based on this grammar, MontiCore generates a parser, AST classes, context condition interfaces, and visitors. Further, all context conditions are collected and a wrapper for the generated context condition checking visitor is generated that parametrizes the latter with the collected context conditions. Thus, parsing and checking models of the language product already is possible.

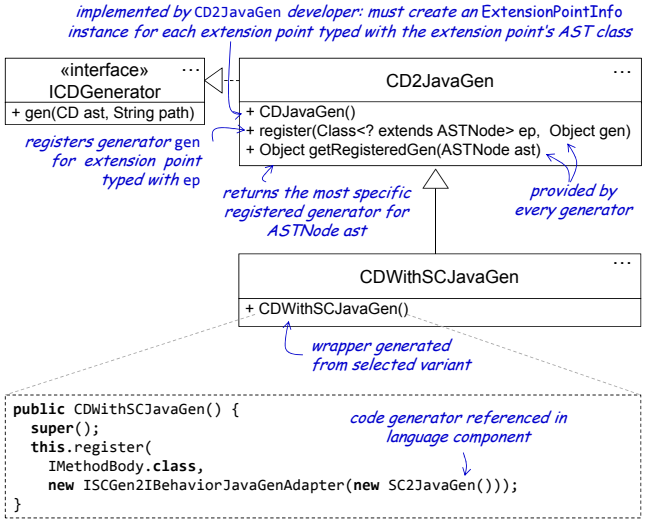

Figure 8: The generator `CDWithSCJavaGen` is generated from the selected variant.

Generators are composed by generating a wrapper for the generator of the feature model's root feature that parametrizes that generator with the other selected generators. This is realized by instantiating the respective (handcrafted) generator adapter class. Where multiple levels of generators are selected, the parametrization is nested accordingly. This wrapper then is registered with the new language model processing framework generated by MontiCore. On artifact level, adapters typically are instantiated within the templates of the embedding generator. As the name of artifact adapters is fixed (*cf.* Section 5) and derived from provided and required artifact interfaces, the name can be calculated within templates. Finally, a new language component is generated that uses the synthesized grammar, the context condition checker wrapper, and the synthesized code generator to process and transform models. This composed language component can be reused, *e.g.,* in a new language product line.

7 EXAMPLE REVISITED

The previous sections explained composition mechanisms for the grammars, context conditions, and generators of language components. A more detailed explanation of the composition of grammars is given in [4]. This section provides an insight into the composition of code generators alongside of the example with the composition of the Statechart language component and class diagrams (Figure 6).

Figure 7 depicts the involved classes of the `CD2JavaGen`, the `SC2JavaGen`, and the adaptation, where the abstract adapters are generated at product line level. The `CD2JavaGen` initializes its extension points in its constructor and yields a method to obtain a registered generator for an extension point based on a given node of the AST as depicted in Figure 8. For a single extension point there might be multiple registered generators, but for a concrete type of ASTNode (*e.g.,* `SCDef`) there must be only a single generator translating it. To use a generator with other generators embedded via generator adapters, a wrapper is generated (*cf.* Section 6). This

```
01 <#-- requires: ASTClass ast, String package, CD2JavaGen cdGen -->          FTL
02 class ${ast.getName()} {
03 <#list ast.getMethods() as met>                    find most
04 <#assign name = met.getMethodHead().getName()>     specific      generate
05 <#assign fqName = package + name>                  generator     artifact
06                                                        ↓            ↓
07 <#assign g = cdGen.getRegisteredGen(met.getMethodBody())>
08 <#assign genClassName = g.generate(met.getMethodBody(), fqName)
09                         .getGenClassName()>
10 IBehaviorArtifact ${name}Body =
11   new ${g.getTargetInterface()
12        .getSimpleName()}2IBehaviorArtifactAdapter(new ${genClassName}());
13                                                                  ↑
14 public void ${name}(Map<Object,Object> _input) {                 instantiate
15   ${name}Body.compute(_input);                                   generated artifact
16 }                                  method of                    through adapter
17 </#list> <#-- ... -->              IBehaviorArtifact
18 }                                  interface
```

Figure 9: Excerpt of a template called by `CD2JavaGen`.

wrapper extends the host generator and in its constructor, registers the employed handwritten extensions of the generator adapters. Each generator adapter has to be parametrized with the concrete instance of its adaptee. In the example, the employed generator adapter adapts the `SC2JavaGen` as realization of an `ISCGen` to an `IBehaviorJavaGen` and registers the adapter at the extension point `IMethodBody`.

If invoked to translate a concrete model, the `SC2JavaGen` executes its main template as depicted in Figure 9. The template produces a new Java class (l. 2) for each class of the class diagram. For each method of the class to translate, it generates an artifact adapter attribute (ll. 10-12) and a Java method (ll. 14-16). The correct artifact adapter is determined according to the embedded generator that translates the specific type of method body (*i.e.,* in the example only `SCDef`) and the fix naming scheme for adapters. The generated method delegates the execution of the method of the class diagram method to the respective artifact adapter. For generating an artifact for an embedded `IMethodBody`, the template retrieves the method-specific generator (l. 7) and executes it to synthesize the artifact (ll. 8-9). Afterwards, the template instantiates the artifact adapter and parametrizes it with a new instance of the generated artifact it delegates to (ll. 11-12).

Figure 10 overviews the artifact adapter infrastructure. The artifact interfaces of the code generators, in the example `IBehaviorArtifact` of the `CD2JavaGen` and `IStatechart` of the `SC2JavaGen`, are handwritten. The notion behind the `IBehaviorArtifact`, for instance, is that the `CD2JavaGen` expects all generators corresponding to productions bound to the `IMethodBody` extension point to produce classes that implement the `IBehaviorArtifact` interface. To this effect, it provides an implementation of the method `compute()` that has the given parameters and return type. `IStatechart2IBehaviorArtifactAdapterTOP` is a generated, abstract adapter class. It implements the interface it adapts to and holds an attribute for the adaptee. When initialized, each adapter has to specify the concrete instance of the adaptee. Concrete adapter implementations have to be handcrafted (cf. Section 5). For instance, the class `IStatechart2IBehaviorArtifactAdapter` is a concrete adapter implementation that extends the abstract adapter and follows the naming scheme. In this concrete adapter, all methods that are required by the interface that the adapter adapts to, have to be implemented while delegating to the methods provided by the adaptee

```java
public interface IBehaviorArtifact {
  public void compute(Map<Object,Object> _input);
}
```
Java

```java
public abstract class IStatechart2IBehaviorArtifactAdapterTOP
                                    implements IBehaviorArtifact {
  private IStatechart adaptee;
  public IStatechart2IBehaviorArtifactAdapterTOP(IStatechart adaptee) {
    this.adaptee = adaptee;
  }
}
```
generated

```java
public class IStatechart2IBehaviorArtifactAdapter extends
                IStatechart2IBehaviorArtifactAdapterTOP {
  public IStatechart2IBehaviorArtifactAdapter(IStatechart adaptee) {
    super(adaptee);
  }

  @Override
  public void compute(Map<Object,Object> _input) {
    String scInput = (String)_input.get("trigger");
    boolean wasFinal = false; int i = 0;
    while(i<scInput.length() || !isFinal)
      isFinal = getAdaptee().updateCurrentState(entry.charAt(i));
    }
  }
}
```
handcrafted

```java
public interface IStatechart {
  public boolean updateCurrentState(char trigger);
}
```

Figure 10: Parts of the artifact adaption infrastructure.

interface. In this example, the compute method has to be implemented to adapt the execution of a method of the class diagram to the execution of a Statechart. The implementation retrieves a parameter `trigger`, which it assumes is present and of type `String`. Then it iterates over the concrete String and updates the state of the Statechart by invoking the method `updateCurrentState()` with the next character of the String, until either the Statechart is in a final state or the input word is processed completely.

The presented approach relieves product owners from being software language engineers completely. A product owner should know the language concepts required for a certain product, but does not require to implement adapters or any form of "glue code". Further, all composition mechanisms applied during derivation of a variant are completely automated. Product line engineers have to be software language engineers as these have to connect extension points to extensions and realize, *e.g.,* the adapters for code generators. Nonetheless, product line managers do not have to be aware of intricate details of the implementations within individual language components, which greatly facilitates reusability of these.

8 DISCUSSION AND RELATED WORK

Our work is a first approach to realize parts of the VCU (variability, customization, use) model of reuse [21] for software languages as sketched in [18]. To this end, we support aggregating language concerns – including syntax and semantic realizations – through features. We therefore currently investigate whether partial configuration of language products is suitable.

Our approach to compose code generators is limited to composing independent generators. Adding, *e.g.,* aspect-like functionality through a single feature is not supported and we are currently investigating this extension. Moreover, our approach limits the variability of language syntax, well-formedness, and dynamic semantics to a single dimension. While this reduces the effort of modeling a language product line, it may require to produce multiple language components that rely on common constituents. For example, if a language can be translated either to Java or C using two different code generators, our approach relies on two different language components. These have references to the same grammar and well-formedness rules, but each employs a different code generator. Our approach also leverages language embedding as the composition mechanism of choice. With this, a loose coupling of languages, in which their abstract syntaxes are not composed – such as language aggregation [15] – is not supported, but subject to ongoing research. Another open challenge is to make language components an active unit of systematic reuse, for instance through inheritance of language components. Some approaches consider code generation as the last step in a pipeline of tools that process a model. In this representation, the language's semantics is realized via applying several model-to-model transformations in a certain order and then translating the transformed model into text by employing a code generator. Currently, our approach does not explicate model-to-model transformations within language components. Considering model-to-model transformations as first phase of executing a code generator, however, is possible.

Our generator variability mechanism ensures compatibility between composed code generators, but cannot guarantee correctness of the generated code. However, the syntactical correctness (*e.g.,* avoiding that two generators generate a file with the same names) can be checked. Also, the restriction to code generators producing standalone artifacts enables a better investigation of their compatibility, but limits their modularity to be coarser grained. While many use cases of code generator composition can be realized following this premise, we are aware of its limitations. For instance, code generators producing, a return statement of method bodies only, are too fine-grained for our composition approach. Future work will investigate how the composition of generators and artifacts can be realized with finer grained modularity of generators.

For composition of generators and generated artifacts, the product line engineer has to provide two adapters per combination of code generators on product line level. If, however, we assume that the product manager is also a software language engineer, the adaptation could be performed later – while deriving the variant. This reduces the number of adapters to be created to twice the number of selected generators. Also, the product owner could perform variant-specific integration of handcrafted code to further customize the other constituents of the language components (such as integration of novel inter-language well-formedness rules). Ultimately, our composition mechanism also relies on the constituents of all language components to be implemented in the same technological space. This prevents, *e.g.,* defining the language components of a single product line in different language workbenches. Such inter-space language definitions also subject to ongoing research.

Future work on the benefits of code generator composition through feature must also investigate efficiency and usability considerations. Our approach requires only little overhead (a language component model per language and the interfaces per code generator) and leverages this to yield black-box composition of reusable code generators. These interfaces are very compact and the language composition merely aggregates existing artifacts. Code generator composition without explicit interfaces for generators and

artifacts requires sophisticated and costly white-box generator investigation. To uncover the benefits of generator variability, future work could investigate and apply metrics regarding size and complexity of artifacts as well as empirical metrics regarding usability.

Research and practice have produced a number of language workbenches, *i.e.,* software tools that support developing and (re)using modeling languages [10]. These language workbenches employ different language definition paradigms, *e.g.,* to (1) define concrete syntax and abstract syntax of languages (usually grammars [2, 15, 30, 32], metamodels [6, 28], or projectional editing [38]); (2) develop the well-formedness rules applied to the abstract syntax (typically OCL [17] or GPL rules [15]); and (3) describe the behavior of models (interpretation [3] or code generation [5]). Due to this wealth of technological spaces and fragmentation in different solution techniques for language development, support for reusing syntactic and semantic language components is rare [31]. Consequently, language reuse is an ongoing research challenge and different approaches [25] exist to address this challenge. Some approaches employ plain negative variability to derive variants of a 150% metamodel [40], which limits their extensibility. There are, however, few approaches addressing both syntax and semantics of modeling languages. The revisitor approach [24] is one of these. It uses a new pattern to enable independent extensions of executable DSMLs covering both metamodel and the realization of the semantics. It supports to extend a language without foreseeing explicit extension points at its design time and reusing language components without recompilation. To the best of our knowledge, it does not support to develop extending language and extended language independent of another.

Neverlang [30] is a language development framework that enables to develop compositional languages components comprising a grammar-based syntax definition and several evaluation phases realizing that, among other things, include type checking and code generation. Extension points in these grammars are placeholders, which are unused nonterminal names. To the best of our knowledge, there is no dedicated typing system for placeholders. AiDE [23], built on top of Neverlang, guides language developers in composition of the language components by extracting dependencies between language components. From these, AiDE synthesizes a feature model for a language product line fully automated. There is an extension to Neverlang using the common variability language for organizing the variability across language components [31]. However, both extensions to Neverlang require dependencies between the language components. Compared to our approach, this limits their reuse in different contexts as language components cannot be developed independently.

The approach presented in [8] enables developing programming languages gradually from independent language modules containing context-free grammars as their syntaxes and action semantics. Action semantics differ from the denotational semantics realized with code generators as presented in our approach. Action semantics modules are built from action notation symbols and as such limited to the expressiveness of the underlying symbols. This yields the advantage that composing semantics modules is more controllable compared to our approach. Further, such symbols are interpretable from different target GPL interpreters. To the best of our knowledge, the approach lacks an explicit variability model to build up product lines of languages.

mbeddr [35, 36] is an extensible set of language modules that builds upon C and the language workbench MPS [34]. It is, therefore, limited to use C as common base language and, to the best of our knowledge, lacks a variability model to properly manage available language modules and their conceptual interrelations. However, it has great usability in terms of editors through MPS. The MPS code generator is extensible by resolving all generator rules and mapping configurations of the individual languages and then building a generation plan. The order of executing the single code generators is specified via priorities. Similarly, ableC [20] is an extensible language framework that builds upon C. It uses Silver and Copper as underlying technologies for the definition of attribute grammars to describe the syntax, and provides several mechanisms to realize composition of these.

In [26], generator composition is realized similar to our approach. Here, each code generator implements one out of three predefined interfaces and provides basic information required by the main generator to call other generators. To this end, each generator yields a model describing its interface-related properties. This approach enables to compose generators, but the composed generators have to implement the predefined interfaces and possible extensions are restricted to existing generator interfaces. The runtime dimension is not covered as all possibly existing generator types are known a priori and, hence, generated source code matches by construction. However, this approach is limited to embedding behavior languages into architecture description languages.

9 CONCLUSION

We have introduced a method to reuse modeling language (parts) through syntactic and semantic embedding of language components. Based on this, modeling language product lines foster systematic reuse of languages and related tooling. Our method relies on abstract syntax descriptions that support underspecification and code generators that, yielding explicit interfaces, produce target GPL artifacts whose contracts (*e.g.,* interfaces), they make explicit. With this, language product line engineers can arrange features representing language components such that various domain-specific language variants can be derived easily. All language engineering efforts (such as integration of inter-language well-formedness rules or code generator adapters) are with the language product line manager. Hence, all composition is transparent to the product line users and modelers. Ultimately, this can facilitate reducing the proliferation of (domain-specific) modeling languages.

REFERENCES

[1] Kai Adam, Arvid Butting, Oliver Kautz, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2018. Retrofitting Type-safe Interfaces into Template-based Code Generators. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18)*. SciTePress, 179 – 190.

[2] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd.

[3] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the gemoc studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 84–89.

[4] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Controlled and Extensible Variability of Concrete and Abstract Syntax with Independent Language Features. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'18)*. ACM, 75–82.

[5] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[6] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States.

[7] Thomas Degueule, Tanja Mayerhofer, and Andreas Wortmann. 2017. Engineering a ROVER Language in GEMOC STUDIO & MONTICORE: A Comparison of Language Reuse Support. In *Proceedings of MODELS 2017. Workshop EXE (CEUR 2019)*.

[8] Kyung-Goo Doh and Peter D Mosses. 2003. Composing programming languages by combining action-semantics modules. *Science of Computer Programming* 47, 1 (2003), 3–36.

[9] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. 2012. Language Composition Untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. ACM, New York, NY, USA.

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D.P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*. Springer International Publishing.

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

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[13] Timo Greifenberg, Katrin Hoelldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Mueller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. 2015. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[14] Object Management Group. 2010. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01). (2010).

[15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Voelkel, and Andreas Wortmann. 2015. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Scitepress, Angers, France.

[16] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer* 37, 10 (2004), 64–72.

[17] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. 2010. Integrating OCL and textual modelling languages. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 349–363.

[18] Jean-Marc Jézéquel, Manuel Leduc, Olivier Barais, Tanja Mayerhofer, Erwan Bousse, Walter Cazzola, Philippe Collet, Sébastien Mosser, Benoit Combemale, Thomas Degueule, Robert Heinrich, Misha Strittmatter, Jörg Kienzle, Gunter Mussbacher, Matthias Schöttle, and Andreas Wortmann. 2018. Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering. *Computer Languages, Systems & Structures* 54 (2018), 139 – 155.

[19] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. 2006. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 719–720.

[20] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 98 (Oct. 2017), 29 pages. https://doi.org/10.1145/3138224

[21] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. 2016. VCU: The Three Dimensions of Reuse. In *International Conference on Software Reuse*. Springer, 122–137.

[22] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: a Framework for Compositional Development of Domain Specific Languages. In *International Journal on Software Tools for Technology Transfer (STTT)*.

[23] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. 2015. Choosy and picky: configuration of language product lines. In *Proceedings of the 19th International Conference on Software Product Line*. ACM, 71–80.

[24] Manuel Leduc, Thomas Degueule, Benoît Combemale, Tijs Van Der Storm, and Olivier Barais. 2017. Revisiting Visitors for Modular Extension of Executable DSMLs. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*. Austin, United States. https://hal.inria.fr/hal-01568169

[25] David Méndez-Acuña, José A Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures* 46 (2016), 206–235.

[26] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2014. Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. In *1st International Workshop on Model-Driven Robot Software Engineering (MORSE 2014) (CEUR Workshop Proceedings)*, Vol. 1319. York, Great Britain, 66 – 77.

[27] Bernhard Rumpe and Katrin Hölldobler. 2017. *MontiCore 5 Language Workbench. Edition 2017*. Shaker Verlag.

[28] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework* (2. ed.). Addison-Wesley, Boston, MA.

[29] Alan Mathison Turing. 1937. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society* 2, 1 (1937), 230–265.

[30] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

[31] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. 2013. Variability Support in Domain-Specific Language Development. In *Software Language Engineering*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer International Publishing, Cham, 76–95.

[32] Tijs van der Storm. 2011. *The Rascal Language Workbench*. CWI. Software Engineering [SEN].

[33] John Vlissides. 1998. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. online at http://www.research.ibm.com/designpatterns/pubs/gg.html.

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

[35] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, New York, NY, USA, 121–140. https://doi.org/10.1145/2384716.2384767

[36] Markus Voelter, Jos Warmer, and Bernd Kolb. 2015. Projecting a Modular Future. *IEEE Software* 32, 5 (2015), 46–52.

[37] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C L Kats, Eelco Visser, and Guido Wachsmuth. 2013. *{DSL} Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[38] Markus Völter and Eelco Visser. 2010. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 301–304.

[39] Guido H Wachsmuth, Gabriel DP Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *Software, IEEE* 31, 5 (2014), 35–43.

[40] Jules White, James H Hill, Jeff Gray, Sumant Tambe, Aniruddha S Gokhale, and Douglas C Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE software* 26, 4 (2009).