Towards the Black-Box Aggregation of Language Components

Jérôme Pfeiffer and Andreas Wortmann
Institute for Control Engineering of Machine Tools and Manufacturing Units
University of Stuttgart
Stuttgart, Germany
www.isw.uni-stuttgart.de

***Abstract*—Engineering software languages demands considering many different aspects, including syntax, semantics, tooling, variability, usability, and many more. Engineering efficiency depends largely on our capability of reusing language parts in other contexts. Currently, most language reuse focuses on few isolated aspects. We have devised the component-based SCOLAR framework for the novel integrated reuse of concrete syntax, abstract syntax, well-formedness rules, and code generators. SCOLAR supports black-box embedding of language parts, yielding composed languages, and, consequently, models (e.g., expressions embedded into Statecharts). This is insufficient when aiming to integrate languages such that their models should be loosely coupled only. We, therefore, present an extension of SCOLAR towards the black-box integration of language components through language aggregation, yielding a loose coupling between their models (e.g., Statecharts referring to class diagram types). To this end, we analyzed the requirements for language aggregation and devised compact extensions to SCOLAR to also support the latter. This enables a novel integration of language components through aggregation, which can facilitate reusing software languages and, hence, advance their application.**

***Index Terms*—Software Language Engineering, Language Components, Language Aggregation**

I. Introduction

Software engineering always has been a quest for increasing abstraction from punch cards, to assembler code, to programming languages, to modeling languages. One main driver for the success of software engineering is our ability to reuse software parts in a black-box fashion. The success of model-driven engineering thrives on the availability of useful modeling languages. Engineering such languages still is challenging and reusing language parts can mitigate this. Yet, language reuse (e.g., through aggregation, embedding, inheritance, merging, ...) generally still is understood on selected combinations of language implementation aspects only.

To address the challenge of reusing holistic, in the sense of comprising concrete syntax, abstract syntax, well-formedness rules, and code generators, languages, we have devised the SCOLAR framework [1]. With SCOLAR, language implementations are encapsulated into components of stable interfaces, making their provided and required extensions explicit, thus supporting their black-box composition to facilitate reusing language parts. SCOLAR arranges Domain-Specific Language (DSL) components in product lines and supports composition in form of language embedding [2], which produces new languages by embedding the components of selected language product line features into another, yielding languages (and models) featuring selected syntax and code generators from the individual components.

This is insufficient when aiming to integrate languages such that their models should be loosely coupled only. For instance, where Statecharts use class diagram types, their models remain isolated aside from resolving names used in them to check the well-formedness of their integration (such as to check whether methods referenced in the Statechart exist in the class diagram). We, therefore, present a small extension of SCOLAR towards the black-box integration of DSL components through language aggregation, yielding a loose coupling between their models.

Therefore, we analyzed the requirements for language aggregation and devised compact extensions to the language interfaces of SCOLAR components as well as to its composition mechanisms. This paper presents these extensions and illustrates their application. The contributions of this paper, thus, are:

1) An analysis of the requirements of language aggregation.
2) A method for the black-box aggregation of DSL components using the SCOLAR framework.

Overall this is an important step towards enabling the integration of language components through aggregation, which can facilitate reusing software languages and, hence, advance their application.

In the remainder, Sec. II introduces preliminiaries, Sec. III highlights requirements for language aggregation, and Sec. IV explains its integration into the SCOLAR framework. Afterward, Sec. V presents its application and benefits by example, before Sec. VI discusses observations, and Sec. VII relates our apporach to related research. Sec. VIII concludes.

II. Background

A. Language Aggregation

Language aggregation composes multiple languages by employing name-based mapping between them [3], [4]. This means, that named abstract syntax elements of languages can refer to each other. In contrast to the other language composition mechanisms, e.g., embedding, or inheritance, the models

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant no. 441207927.

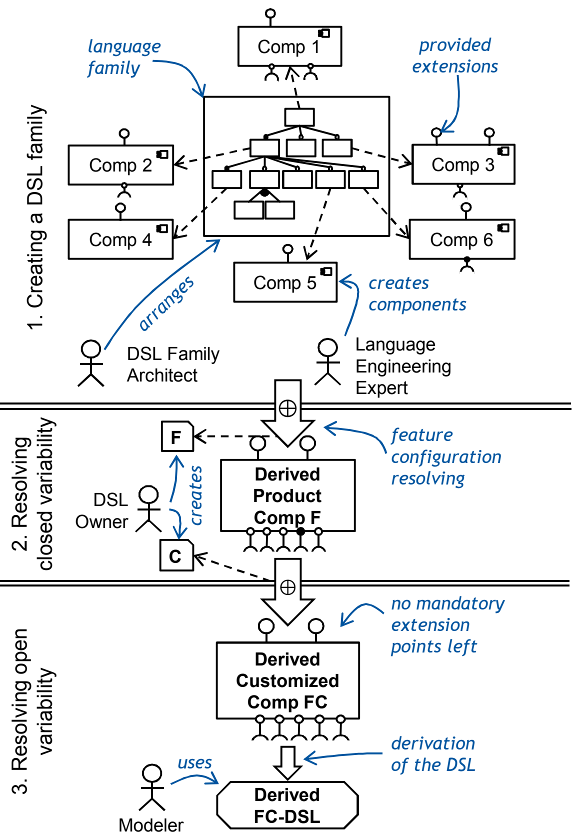


Figure 1: The composition of two DSL components processes all bindings, and updates the interface of the resulting DSL component accordingly. If all mandatory required extensions have been fulfilled, a new DSL can be derived. [1]

remain separated in different artifacts [5], which facilitates loose coupling, and, thus, independent reuse and maintenance. However, models of the composed languages are understood together and contribute to a common modeling goal. Language aggregation is suitable for modeling different aspects of a system where each aspect is described by an individual model, e.g., behavior aspects in statecharts, and structural aspects in class diagrams. With aggregation, a coupling between those models can be achieved, despite, the models can be reused in different combinations and in a modular way.

B. A Method for Systematic Language Engineering

Our method for systematic language engineering encapsulates language constituents in DSL components. Components make their provided and required extensions explicit via an interface and can be composed according to these guided by a feature model. All activities are associated with roles with specific expertise as shown in Figure 1. First, language engineering experts create reusable DSL components for specific purposes. Each component contains a combination of grammars, well-formedness rules, and code generators relating to these grammars. Language family architects arrange DSL components into a feature model representing a family of DSLs. In this feature model, each feature either is related to a language component or is an abstract feature [6] for logical grouping. Through this relation, the DSL family architect decides how the components will be composed when their related features are selected. Once the DSL family architect completes the DSL family, DSL owners, who are experts of the application domains derive a suitable DSL for their application domain, by selecting appropriate features from the family in a feature configuration. Based on this, the related DSL components are composed and their provided and required extensions are updated accordingly. To compose the language constituents (*i.e.,* grammars, well-formedness rules, code generators) of the specific technological spaces (such as Neverlang [7], MontiCore [8], or Xtext [9]), our framework provides extension points to delegate to modules specific to the technological space used. Composition results either in a new DSL component, if mandatory extensions were not provided through the family, or a new DSL otherwise. In the former case, the DSL owner can further customize the component with information that was either not available during family creation (*e.g.,* the action language needed for automata transitions for a specific domain) or not suitable for the language family (such as numerical parameters). If the DSL family was well-defined, *i.e.,* options for all required extensions of its components were provided, the DSL owner does not need to have any expertise in Software Language Engineering (SLE) but can derive the most suitable DSL variant on a push-button basis.

To foster DSL reuse, we have conceived and integrated modeling languages for describing DSL components and DSL families. They are tailored to language engineering experts and support making provided and required DSL component extensions explicit. Their models form the basis of component composition. The latter language is an extension of features models that support describing DSL families and the binding of features to extension points of DSL components. A customization language supports implementing required extensions of DSL components not provided by their language family.

1) A Metamodel for Black-Box Language Reuse: Our metamodel describes the properties of DSL components and DSL families relevant to their systematic reuse (see Figure 2). For this purpose, the DSL components do not provide closed variability themselves, but support customization through their required extensions. DSL families comprise feature models to describe closed variability of potential DSLs by arranging DSL components.

DSL components provide the constituents of a language definition. They comprise elements of each of the three essential language definition constituents: (1) syntax, (2) well-formedness rules, and (3) semantics-based code generators. To this end, each DSL component comprises one grammar and can comprise multiple sets of identifiable well-formedness rules, as well as multiple code generator specifications. Components expose these constituents through explicit extensions with cardinalities (optional or mandatory) in their **DSL com-**

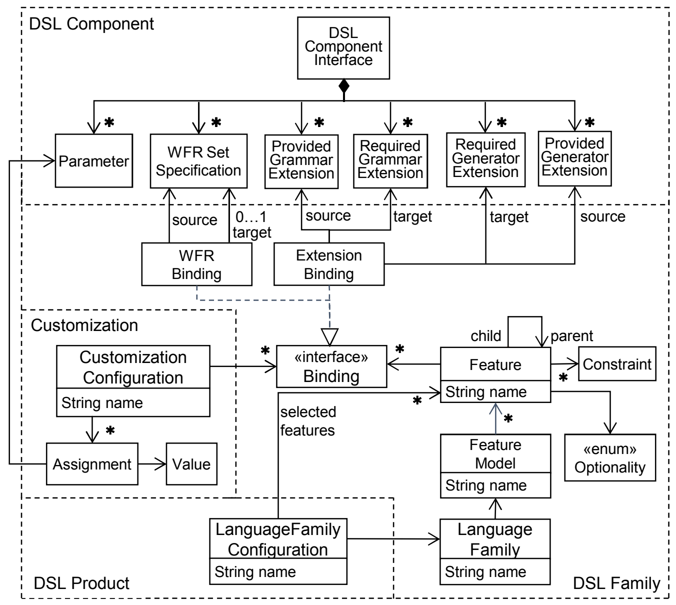


Figure 2: Overview of the metamodel of our approach, consisting of DSL component, Customization, and Language Family.

ponent interfaces. For grammars and generators, the interfaces support both, provided and required extensions, whereas well-formedness rules are contained in sets that can act as both provided and required extensions at the same time. Parameters can be either for well-formedness rules or code generators and enable more fine-grained customization (such as numerical constraints, paths, *etc.*). **Provided extensions** offer DSL functionality to be reused by other components. Provided grammar extensions reference a production in the grammar that can be reused by other components' grammars. Provided generator extensions reference a production for which they provide a transformation, a reference to a GPL class, and the interfaces of producer and product. **Required extensions** make missing functionality of a DSL component explicit and can be either optional or mandatory. Required extensions for grammars reference a production of a grammar that requires extension. Required generator extensions demand extension for a specific production, with specific product and producer interfaces [10]. **Parameters** can also be either optional or mandatory and parameterize generators or well-formedness rules. **DSL families** [11] describe closed variability via a central feature model [12], [13]. Features refer to DSL components and the arrangement of features define how the related components will be composed if the respective features are selected.

Customization configuration describes open variability by defining multiple bindings between multiple DSL components to the DSL component that needs customization. Additionally, the configuration can assign values to parameters.

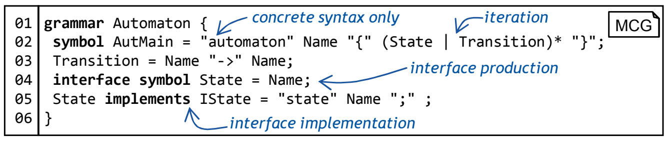
Bindings relate to the connection between DSL components. The current approach supports two kinds of bindings:

- **Extension bindings** map provided grammar or generator extensions of one DSL component to required grammar or generator extensions of another component.
- **Well-formedness rule binding** can be either a rule embedding by joining a well-formedness rule set of one component to the set of another component or be an addition where a complete well-formedness rule set is added en-bloc to the composed component.

2) Composing DSL Components: The composition of two DSL components is the directed application of bindings between these components. After the application of a binding, the cardinality of required extensions becomes optional. Currently, SCOLAR supports embedding of extensions of other DSL components by binding their provided extensions to required extensions of the embedding component. This produces a novel component resulting from adding selected provided extensions of the embedded component into the respective required extensions of the embedding component. This consists of two main activities: (1) Composing the components' interfaces; and (2) Composition of the comprised language definition constituents (grammars, well-formedness rules, code generators); When composing the components' interface, we differentiate between the kinds of bindings. In the case of extension binding, the required extensions of the embedded component are added to the embedding component together with associated generator parameters (only when the extension binding is between generator extensions). In the case of well-formedness rule set composition, either the set is joint with a set of the embedding component or is added next to existing sets. The composition of the comprised language definition constituents is specific to the technological space used. Thus, our toolchain [1] provides extension points for software modules realizing the technological space-specific composition of the associated artifacts.

C. MontiCore

To realize our approach, we use the language workbench MontiCore [8] as proof of concept. MontiCore is a language workbench for the development of textual, external DSLs. Context-free grammars comprise the definition of the integrated concrete and abstract syntax of a DSL. From this, MontiCore generates language tooling including an abstract syntax data structure, a parser that instantiates this data structure, a symbol management infrastructure, a visitor infrastructure for traversing the abstract syntax, and infrastructures for defining and checking well-formedness rules as well as for generating code from models conforming to the grammar. Well-formedness rules in MontiCore are realized as Java classes and are checked against the abstract syntax leveraging the generated visitor infrastructure. Code generation is realized through template-based code generators based on the FreeMarker [14] template engine. Figure 3 shows a MontiCore grammar by example. Each MontiCore grammar starts with the with keyword `grammar`, followed by the name of the grammar (see l. 1). The body of a grammar (ll. 2-6) contains grammar productions. By default, the first production is the start production of a grammar. On the left-hand side, each



```
01 grammar Automaton {
02  symbol AutMain = "automaton" Name "{" (State | Transition)* "}";
03  Transition = Name "->" Name;
04  interface symbol State = Name;
05  State implements IState = "state" Name ";" ;
06 }
```

Figure 3: Example MontiCore grammar of an automaton DSL

production defines a nonterminal, *e.g.,* `AutMain` (l. 2). On the right-hand side, a production can contain terminals (in double quotes) and nonterminals (starting with upper case letter) as well as iterations ('`*`', '`+`' ,'`?`'), alternatives ('`|`'), and concatenations (' ') thereof. Nonterminals starting with the keyword `symbol` define symbols (l. 2). In MontiCore, Symbols are named nonterminals of the language that enable resolving names within and across models. Interface nonterminals can underspecify a right-hand side or prescribe abstract syntax elements (l. 4). Other productions can implement interface productions (l. 5). If the right-hand side prescribes abstract syntax elements, implementing nonterminals must provide these. The generated parser treats the usage of an interface nonterminal equal to an alternative overall nonterminals defined by productions implementing the interface nonterminal.

III. Aggregating DSL Components

Performing language aggregation between DSL components is an extension to our existing SCOLAR framework. Thus, to make language aggregation fit into SCOLAR, requirements derived from 1) language aggregation and, 2) from our existing approach exist. The composition workflow for DSL components has to fulfill all of these requirements. Therefore, the following presents requirements to language aggregation in our approach, and introduces extensions to the composition workflow to enable both, language embedding and, now, also language aggregation.

A. Requirements Derived from Language Aggregation

Extending the SCOLAR framework with aggregation entails several requirements that are presented in the following. Some are derived from the characteristics of language aggregation, and others are general concepts of our framework that have to hold for language aggregation too.

R1: Keep grammars separated. When aggregating two languages, their models remain separated afterward. However, they contribute to one modeling goal. To facilitate extending and reusing both aggregated languages, but make their interconnection explicit, their grammars should be kept separated in the composed DSL component.

R2: Aggregation is uni-directional. In aggregation, the direction matters. For instance, it makes a difference whether automatons aggregate class diagrams or class diagrams aggregate automatons. In the former case, in automaton types, defined in class diagrams are referenced per name. In the latter, class diagrams reference automatons by their name to for instance, define a method's behavior with an external automaton model, which happens to have the same name as the method. Hence, aggregation is a uni-directional binding from a provided extension of one DSL component to a required aggregation extension of another DSL component.

R3: Required extensions make the purpose of aggregation explicit. Because aggregation is unidirectional and because the Language Engineering Expert knows where aggregation could be possible, she should decide on the language elements that require aggregation and be able to make them explicit. With this, the DSL Family Architect can use DSL components as black-boxes and can identify via their interface which required extensions should be bound for aggregation.

R4: Provided extensions are not exclusive for aggregation or embedding. Since aggregation is uni-directional, it is not necessary for the provided extensions to know whether it is embedded or aggregated to other DSL component's required extensions. For instance, when aggregating class diagrams to automatons, the automaton language requires aggregation for, e.g., the definition of variable types and uses classes contained in a class diagram for this. Thus, the automaton language knows the need for aggregation and also knows class diagrams that define variable types. The class diagrams however do not know anything about how they are used. They just provide the named language elements that can be either used to be embedded or aggregated.

R5: Once bound, required aggregation extensions are optional. When a DSL component contains mandatory extensions, its language misses an implementation for this extension. Thus, it is not possible to derive the language's implementation without this required extension being bound. To indicate that a required aggregation extension has been bound in the composition step, the extension's optionality should change to optional afterward.

R6: Provided extensions of aggregated language stay available. Since both languages stay separated within a DSL component, they can be refined further independently. Thus, other languages should be able to reuse provided extensions as well as parameters.

R7: Required extensions of aggregated language stay available. Furthermore, to keep DSL components extensible and enable evolution beyond one composition step, all required extensions of both involved components should stay available after the composition.

R8: Derive language infrastructure automatically. If the DSL component resulting from the aggregation does not have any remaining mandatory required extensions, our approach can automatically derive a DSL from the component. For this, it is required that aggregation on the level of the technological space is possible and that the implementation of the mapping infrastructure is derivable.

R9: Referenced productions must have a named right-hand side. Whenever two language elements are aggregated, they are mapped via their name. This requires both productions, the production of the required extension and the one of the provided extension to have a name on their right-hand side.

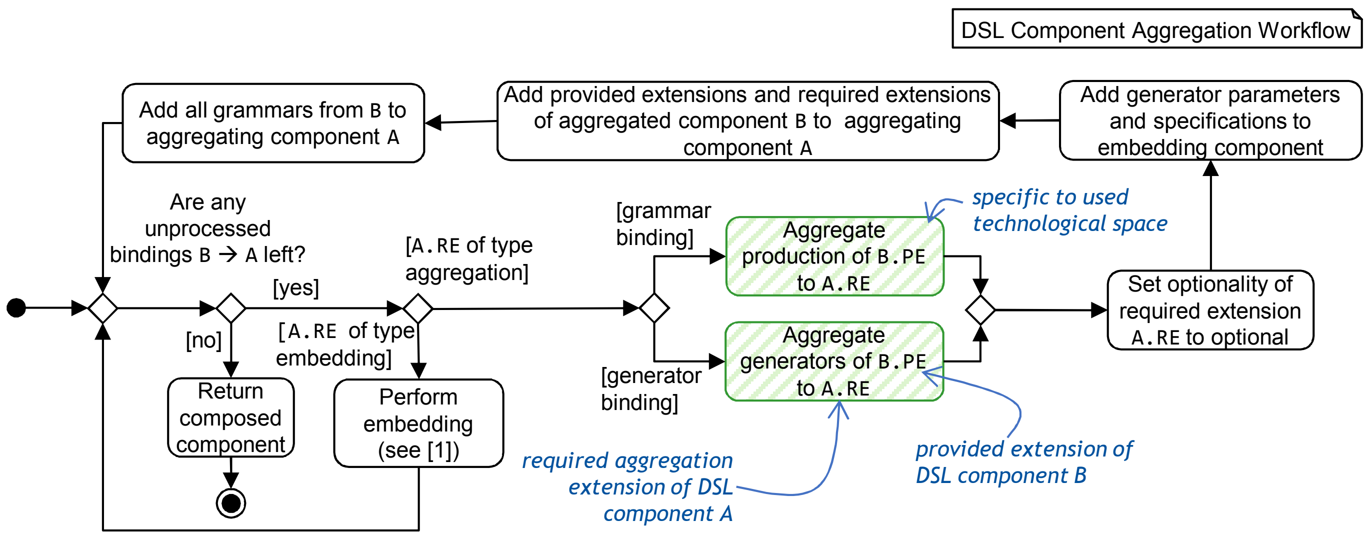

DSL Component Aggregation Workflow
Add all grammars from B to aggregating component A
Add provided extensions and required extensions of aggregated component B to aggregating component A
Add generator parameters and specifications to embedding component
Are any unprocessed bindings B → A left?
[yes]
[no]
[A.RE of type aggregation]
[A.RE of type embedding]
[grammar binding]
[generator binding]
Aggregate production of B.PE to A.RE
Aggregate generators of B.PE to A.RE
specific to used technological space
Set optionality of required extension A.RE to optional
Return composed component
Perform embedding (see [1])
required aggregation extension of DSL component A
provided extension of DSL component B

Figure 4: DSL components are composed according to the bindings defined in the language family. The step `Perform Embedding` is described in detail in [1].

B. Integrating Aggregation into the DSL Component Composition Workflow

The extension of our approach with aggregation leads to a new DSL component composition workflow depicted in Figure 4. It describes the composition of two DSL components `A` and `B`. As long as there are unprocessed bindings (**R2**) between both components, e.g., originating from unprocessed features of the DSL family, the composition takes place. If the required extension of `A` is of type embedding, the provided extension of `B` is embedded into `A`. Binding well-formedness rules is supported out-of-the-box in our approach. When aggregating, well-formedness rule sets can be either added to the composed component separately or added into an existing well-formedness rule set to contribute to the common modeling goal of the aggregated languages. Both ways are already possible in the existing composition workflow (cf. [1]). If the binding binds a provided extension of component B (**R4**) to a required aggregation extension of component A (**R3**), the following step is executed in the particular technological space that is used (indicated by green box and fork icon) (**R8**). In the case of a grammar extension binding, the production of the provided extension of component `B` is aggregated to the production of the required aggregation extension of `A`. In the latter case, the generator of the provided extension of `B` is aggregated to `A`. In both cases, the optionality of the bound required extension of component `A` are set to optional (**R5**). Afterwards, all generator parameters and specifications, all provided and required extensions (**R6**, **R7**), and all grammars (**R1**) of the aggregated component `B` are added to the aggregating component `A`. If there are no unprocessed bindings left, the process returns the composed component and terminates.

Figure 5 visualizes the aggregation between two DSL components, `CD` and `Aut`. They each contain a grammar realizing their abstract and concrete syntax. The component `CD` provides a grammar extension `ClassDef` for the definition of classes in class diagrams, and requires a extension `IMethod` for the implementation of method bodies in classes of class diagrams. The component `Aut` provides an extension for the definition of automatons, requires a mandatory aggregation `VarType` for defining types of variables, and requires an optional extension `IGuard` for the realization of guards on transitions. When aggregating `ClassDef` ⊗ `VarType`, a novel DSL component `CD` ⊗ `Aut` is created. This component comprises all provided and required extensions of the aggregated components. Furthermore, it contains both grammars of the aggregated extension separately. To realize the aggregation from `ClassDef` to `VarType`, an adapter is generated in the respective technological space.

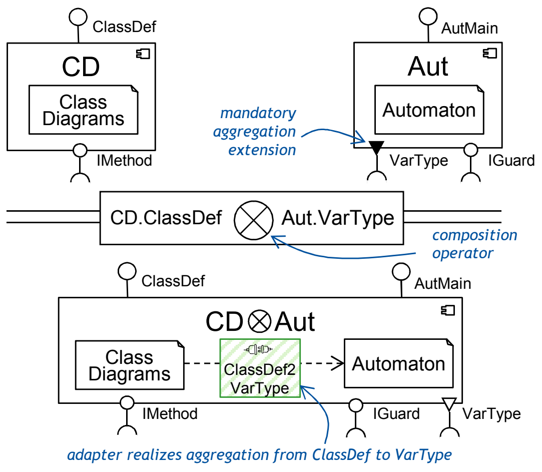


Figure 5: Aggregation of two DSL components. The automaton language aggregates class definitions from the class diagram language to use them for specifying variable types.

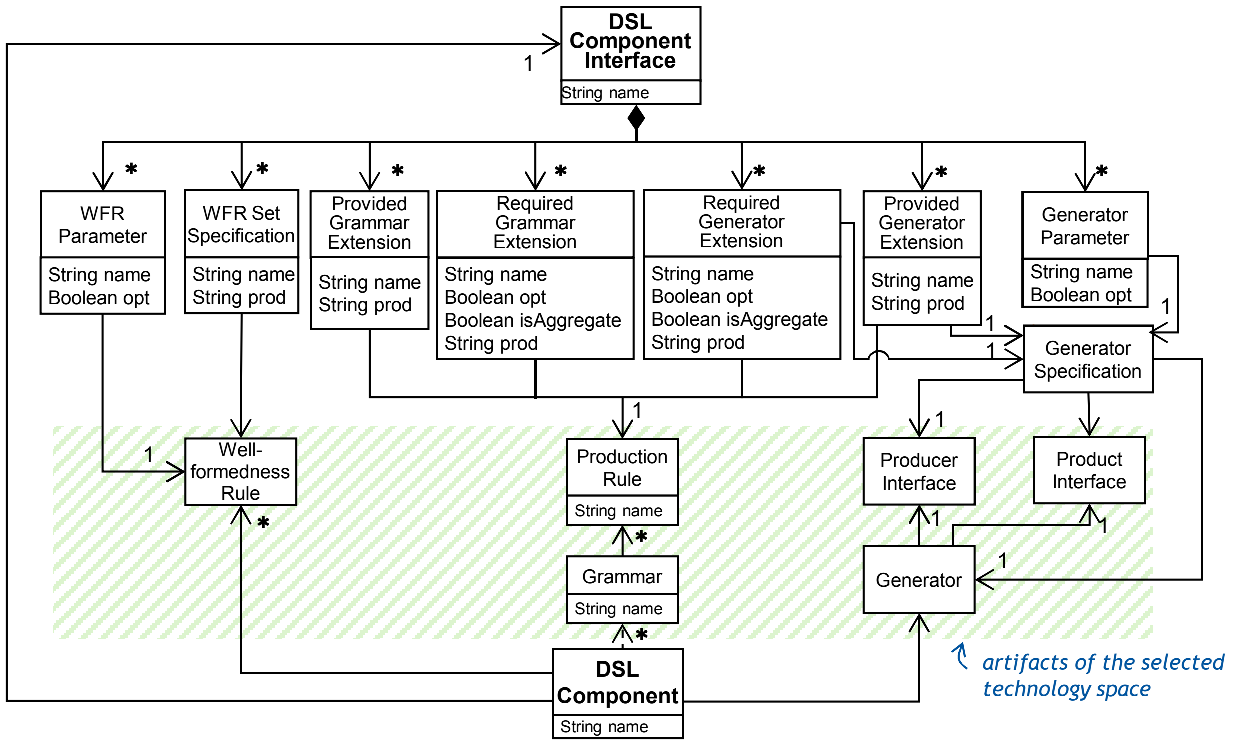


Figure 6: DSL components provide extensions, *i.e.,* parts of language constituents that are exposed by DSL component interfaces. DSL component interfaces also specify required extensions and parameters.

IV. Implementing Language Aggregation

This section describes how language aggregation is integrated into the existing approach. For this, the metamodel and the toolchain are adjusted to realize the aggregation of DSL components and to fulfill the presented requirements.

A. Extensions of the Metamodel

The requirements derived from language aggregation demand for an extension of the metamodel of our approach. These changes on one hand affect the DSL component interface and on the other hand the artifacts of the selected technology space. The extended metamodel of DSL components is depicted in Figure 6. In the artifacts of the technological space, the cardinality of referenced grammars are no longer limited to only one referenced grammar. Because, in contrast to embedding, the grammars remain separated and exist independent from each other after the aggregation took place, it is now possible to reference 1 to many grammars to fulfill **R1**.

Requirement **R2** is already fulfilled by our existing metamodel through the extension binding (see Figure 2), that is between a source, provided extension, and a target, required extension. So it is directed from provided to required extension per definition. To keep a black-box view on DSL components, required grammar and generator extensions are extended with one boolean attribute `isAggregate`. This indicates whether a required extension is expected to be extended by aggregation (true) or embedding (false) (**R3**). The provided extensions stay

```
LC
1  dsl component Automaton {
2    grammar AutomatonBase;
3    provides production AutomatonMain;
4    requires mandatory aggregate production VariableType;
5    requires optional embedding production Guard;
6    requires optional embedding production State;
7  }
```

Figure 7: A DSL component `Automaton` with an aggregate production `VariableType`.

the same as before because as stated in **R4** it is not necessary for them to be constrained to embedding or aggregation. With this, language engineering experts can make the purpose of required extensions explicit. Hence, the required composition operation is directly visible for the DSL family architect when designing a language family and defining bindings.

B. Extension of the Toolchain

Besides the changes of the metamodel, also the toolchain (see Figure 8) has to be extended to fulfill the stated requirements. For realizing the changes to the metamodel, the `DSL Component Processor` is extended. It now can process DSL component models with multiple grammars and with explicit aggregation and embedding required extensions (see Figure 7, ll. 4-6). Thus, fulfills **R1** - **R4**. To fulfill **R5** - **R7**, the `DSL Component Composer` is extended to realize the aggregation composition workflow presented Sec. III-B.

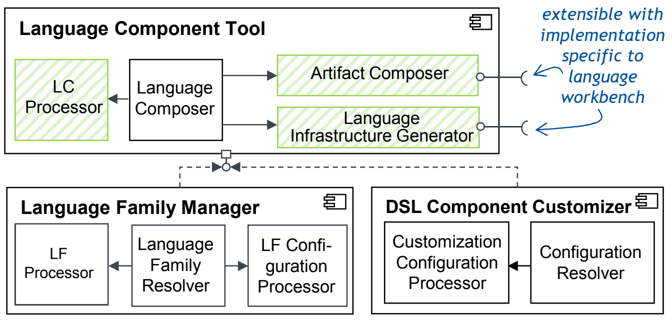


Figure 8: Our framework consists of three main modules for composing and processing DSL components, managing language families, and customizing DSL components.

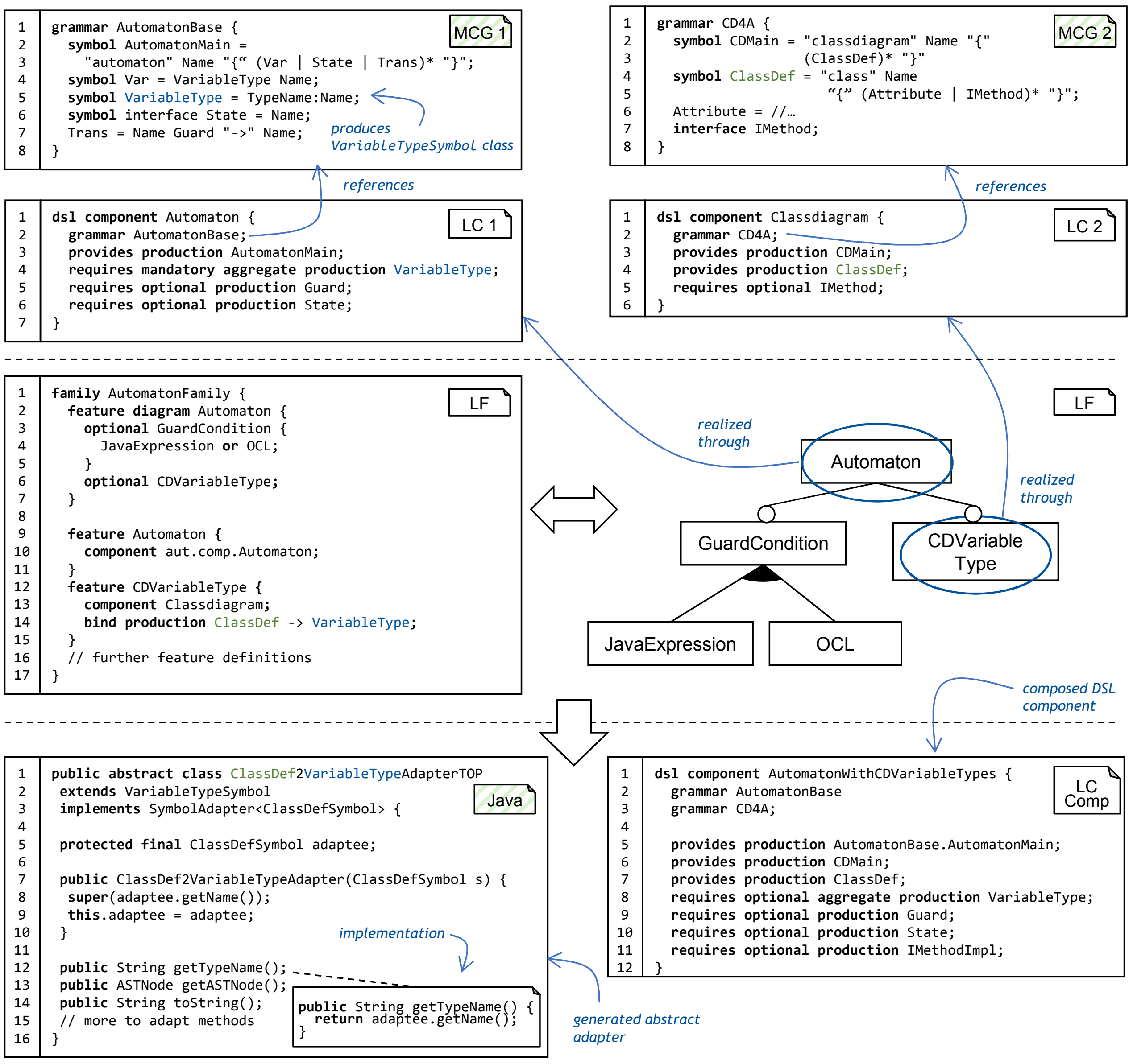
Additionally, the `DSL Component Composer` is extended to realize the aggregation composition workflow of DSL components including changing the optionality of required extensions, and adding all constituents of the aggregated DSL component to the composed component. To perform the aggregation of the language artifacts of the technological space used (**R8**), the interface of the `Artifact Composer` is extended with methods to generate the aggregation of grammar production and generators. Since only the metamodel of DSL components changed, there is no need to change anything in the `Language Family Manager` or the `DSL Component Customizer`. Furthermore, to fulfill **R9** the `DSL Component Processor` comprises a well-formedness rule interface, that checks whether both, the aggregated grammar rule and the rule that requires aggregation, provide a named element on their right-hand side. This well-formedness rule is specific to the used technological space. The well-formedness rule is applied at design time, after the DSL family architect connected features with realizing DSL components and defined the bindings between them.

V. Case Study

This section presents an insight into applying our concept on the example of deriving a DSL for describing automatons with the specialty that variable types are defined in external class diagram models. This includes showing the language family and selecting the required features from it, disclose the involved DSL components and the referenced grammars and syntax rules, their composition, and the artifact composition, exemplified by the aggregation of grammar productions. The artifacts shown in this example correspond to the technical space of MontiCore. The artifacts specific to the technological space are marked green and the fork icon. Consider a language family `AutomatonFamily` (see Figure 9). The family enables to enhance automatons with guard conditions formulated with Java or OCL expressions and the feature `CDVariableType` enables using types of external class diagrams referenced by name when defining automaton variables. The DSL owner chooses the root feature `Automaton` and `CDVariableType` in her language family configuration. For the feature `Automaton`, the language family references a DSL component `aut.comp.Automaton` (l. 10, LF). The DSL component defines a mandatory aggregate production `VariableType` (l. 4, LC 1). This production is contained in the grammar `AutomatonBase` referenced by the DSL component (l. 5, MCG 1). The feature `CDVariableType` is realized through a component `Classdiagram` (l. 13, LF). This component references a grammar `CD4A` (l. 2, LC 2) that comprises the syntax for specifying class diagrams, including a rule for class definitions (ll. 4-5, MCG 2), that is referenced in the DSL component as a provided extension (l. 4). The binding defined in the feature definition of feature `CDVariableType` connects the provided extension `ClassDef` of DSL component `ClassDiagram` to the required aggregation extension `VariableType` of DSL component `Automaton` (l. 14, LF). Based on the feature selection the composition of the DSL components `Automaton` and `Classdiagram` is performed, resulting in a novel DSL component `AutomatonWithCDVariableTypes`. Since the binding included an aggregation, the composed component contains both of the grammars referenced by the bound extensions (ll. 2-3). Furthermore, it contains all provided and required extensions of both participating DSL components (ll. 5-11). A generated adapter abstract `ClassDef2VariableTypeAdapterTOP` realizes the aggregation on grammar level in the technological space by providing abstract methods. Since the adaptation is not trivial, these abstract methods have to be implemented by hand. MontiCore employs named symbols to resolve names within and across models. The grammar production `VariableType` is represented by a symbol `VariableTypeSymbol` and the production `ClassDef` is represented by a symbol `ClassDefSymbol`. Thus, for `ClassDefSymbol` to be recognizable under the name of `VariableTypeSymbol`, the adapter pretends to be a `VariableTypeSymbol` and is findable when resolving under this kind of symbol, but internally the adapter refers to a `ClassDefSymbol` as adaptee. Technology space-specific well-formedness rules ensure, that the grammar productions referenced by the required aggregation extension and the provided extension have a name on their right-hand side. Figure 10 shows an example of possible models conform to the DSL resulting from the aggregation. The automaton specifies the behavior of timed light control that automatically turns off the light after 10 minutes. The timer is introduced as a variable of the automaton (l. 4). The type of the timer is defined as a class in an external class diagram `LightDataTypes`. There, the class defines one attribute and two methods to start the timer and to stop the timer. Through aggregation, classes, their attributes, and their methods specified in class diagrams are now referenceable by name in automatons.

VI. Discussion

Extending our method for systematic language engineering with the capability to integrate DSL components via aggregation supports aggregation of new language features without



```
grammar AutomatonBase {
  symbol AutomatonMain =
    "automaton" Name "{“ (Var | State | Trans)* "}";
  symbol Var = VariableType Name;
  symbol VariableType = TypeName:Name;
  symbol interface State = Name;
  Trans = Name Guard "->" Name;
}
```

```
grammar CD4A {
  symbol CDMain = "classdiagram" Name "{"
                  (ClassDef)* "}"
  symbol ClassDef = "class" Name
                  “{” (Attribute | IMethod)* "}";
  Attribute = //…
  interface IMethod;
}
```

```
dsl component Automaton {
  grammar AutomatonBase;
  provides production AutomatonMain;
  requires mandatory aggregate production VariableType;
  requires optional production Guard;
  requires optional production State;
}
```

```
dsl component Classdiagram {
  grammar CD4A;
  provides production CDMain;
  provides production ClassDef;
  requires optional IMethod;
}
```

```
family AutomatonFamily {
  feature diagram Automaton {
    optional GuardCondition {
      JavaExpression or OCL;
    }
    optional CDVariableType;
  }

  feature Automaton {
    component aut.comp.Automaton;
  }
  feature CDVariableType {
    component Classdiagram;
    bind production ClassDef -> VariableType;
  }
  // further feature definitions
}
```

```
public abstract class ClassDef2VariableTypeAdapterTOP
 extends VariableTypeSymbol
 implements SymbolAdapter<ClassDefSymbol> {

 protected final ClassDefSymbol adaptee;

 public ClassDef2VariableTypeAdapter(ClassDefSymbol s) {
  super(adaptee.getName());
  this.adaptee = adaptee;
 }

 public String getTypeName();
 public ASTNode getASTNode();
 public String toString();
 // more to adapt methods
}
```

```
public String getTypeName() {
  return adaptee.getName();
}
```

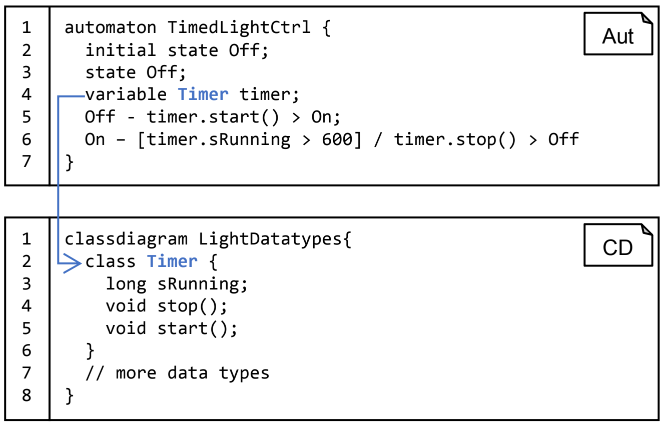
```
dsl component AutomatonWithCDVariableTypes {
  grammar AutomatonBase
  grammar CD4A;

  provides production AutomatonBase.AutomatonMain;
  provides production CDMain;
  provides production ClassDef;
  requires optional aggregate production VariableType;
  requires optional production Guard;
  requires optional production State;
  requires optional production IMethodImpl;
}
```

Figure 9: A language family for automaton language variants (middle). The top shows the DSL component realizing the features `Automaton` and `CDVariableType`. The bottom section shows the resulting abstract adapter for the aggregated grammar production, and the resulting DSL component after the composition took place.

requiring deep expertise in their implementation. To this end, DSL components make their provided and required extensions explicit and distinguish required extensions between embedding and aggregation per keyword. This supports language engineering experts, who know the details of the DSL implementation, in defining the components' stable interfaces while foreseeing planned reuse through required extensions for embedding or aggregation. DSL family architects then can construct language families based on black-box DSL components and their interfaces without knowing specific details about their implementation or the technological space used. In our approach, we opted for a single feature model to represent both dimensions of reuse (embedding, aggregation) instead of using different feature models for each dimension. While we believe that working with a single feature tree instead of a feature forest eases the understanding of the complete language family and facilitates its evolution, future experiments need to validate this. Usability also might benefit from bundled extensions and extension points that expose syntax, generator parts, and well-formedness rules that should be reused together only. This also needs to be investigated.

Currently, our approach is limited to textual, external, translational DSLs and makes some strong assumptions [1], [10], [15] about the supported composition mechanisms as well as how grammars and how code generators are specified. For instance, our approach requires that the extension points can be



```
automaton TimedLightCtrl {
  initial state Off;
  state Off;
  variable Timer timer;
  Off - timer.start() > On;
  On - [timer.sRunning > 600] / timer.stop() > Off
}
```

```
classdiagram LightDatatypes{
  class Timer {
    long sRunning;
    void stop();
    void start();
  }
  // more data types
}
```

Figure 10: The models that can be processed by the DSL resulting from the composed DSL component from Figure 9.

uniquely identified, that composition operates on the level of abstract syntax types (*e.g.,* if a part of a language is embedded into an extension point of another language, this embedding holds for all instances of that extension point), and that required generator extensions expose the required interfaces of both the generator to be embedded as well as of the artifact produced by the embedded generator such that it can be called from the product of the embedding generator (*cf.* [1]). Thus, the black-box aggregation of code generators currently requires that both products of the generators of aggregated language parts adhere to interfaces defined at generator design time. This is a very strong restriction that we plan to alleviate in the future, *e.g.,* by enabling both aggregated generators to exchange information about the shape of the produced artifacts dynamically.

Moreover, our approach requires composition to be additive only (*i.e.,* it cannot remove abstract syntax elements). While being a restriction, it (a) ensures that all language products remain compatible w.r.t. the abstract syntax to the DSL components that they have been composed of, *e.g.,* model checkers, analyses, *etc.* can be reused with the resulting language product as all types of the abstract syntax are still available and (b) adding additional well-formedness rules can be used to restrict languages by prohibiting the occurrence of instances of particular abstract syntax times nonetheless.

Moreover, even though the optionality of required extensions changes after being bound once, it is not possible to fully close and remove required extensions from DSL components due to composition being additive only to rule out future extensions at this point. Whether such behavior is desirable needs to be investigated as well.

Further future research will address the inclusion of Language Server Protocol[1] clients to support the composition of editors, leveraging model-to-model transformation for DSL components, and the application of SCOLAR to technological spaces [16] of graphical or projectional software languages.

[1]https://langserver.org

VII. Related Work

Various approaches apply software product line techniques [17] to SLE [11]. However, only a few support (closed) variability and (open) customization across the constituents concrete and abstract syntax, well-formedness rules, and code generators of DSLs. There are two major approaches for representing language variability in feature models [18]:

(1) Features represent language modules comprising syntax and semantics. This limits the potential to describe, *e.g.,* presentational variability [19] and semantic variability [20].
(2) Abstract features contain selected dimensions of a language (module), such as realizations of abstract syntax or semantics. Subfeatures complete these by providing different realizations for missing dimensions, such as *e.g.,* a textual a graphical concrete syntax.

Our approach enables DSL components to optionally contain multiple realizations per constituent kind and, thus, supports both kinds. Also, our approach uses a loose coupling between the feature model and DSL components and thus supports developing DSL families both top-down and bottom-up [21]. Several language engineering tools such as MPS [22], Spoofax [23], and Melange [24] provide means for language composition and customization, but do not provide methods for systematic reuse through DSL families.

Other approaches for systematically reusing language parts do not make their interfaces explicit, which hampers reusing these modules [7], [25]–[27], or do not support all three component constituents [24], [28].

Overall, our approach builds upon ideas formulated as concern-oriented language development [29], [30], which proposes to engineer languages based on components (called "concerns") with three kinds of interfaces representing their variability, customization, and use. In this vision, concerns comprise artifacts linked with each other that conform to meta-languages which are typed by "perspectives" contained in libraries. With respect to this vision, our approach addresses the componentization of languages and their systematic reuse only. However, we are unaware of any other similar comprehensive realizations of this part of the vision.

Another approach [31] introduces variability for graphical modeling languages by analyzing the metamodel of the language and automatically deriving the allowed model variability. In contrast, our approach enables variability across textual languages and their constituents by composing them via aggregation or embedding.

A different line of research of the authors presents a similar approach for building Language Product Lines (LPLs) [15], [32]. However, our approach is based upon the novel, integrated modeling languages for DSL components and DSL families. In models of these languages, provided and required extension points of DSL components are explicated to enhance black-box reusability. Furthermore, the approach introduces parameters for well-formedness rules and explicit open variability through DSL component customization.

VIII. Conclusion

Efficient software engineering thrives on reuse. For software languages, efficient reuse is a crucial. Many approaches towards reusing language parts focus on a subset of concrete syntax, abstract syntax, well-formedness, or semantics realizations. SCOLAR combines these but supports language embedding only. We have presented a small extension of the SCOLAR framework to support language aggregation as well. This extension rests on the same assumptions than SCOLAR and requires only a single new modeling element. This enables DSLs to be aggregated through their DSL component's interface without knowing specific DSL implementation details in the respective technological space used. Thus, language engineering experts can decide which language parts are required for language embedding or aggregation and make these explicit in the component's interface. This greatly facilitates the reuse of DSLs and makes language development more accessible. Yet, this extension limited to specific forms of code generator aggregation, *i.e.,* those where the interfaces of generated artifacts can be defined at code generator design-time. We are working on alleviating this constraint.

References

[1] A. Butting, J. Pfeiffer, B. Rumpe, and A. Wortmann, "A Compositional Framework for Systematic Modeling Language Reuse," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, October 2020, p. 35–46.

[2] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[3] A. Horst and B. Rumpe, "Towards compositional domain specific languages." in *MPM@ MoDELS*, 2013, pp. 1–5.

[4] B. Rumpe, "Towards model and language composition," in *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, 2013, pp. 4–7.

[5] A. Haber, M. Look, P. Mir Seyed Nazari, A. Navarro Perez, B. Rumpe, S. Völkel, and A. Wortmann, "Composition of Heterogeneous Modeling Languages," in *Model-Driven Engineering and Software Development*, ser. Communications in Computer and Information Science, vol. 580. Springer, 2015, pp. 45–66.

[6] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund, "Abstract features in feature modeling," in *2011 15th International Software Product Line Conference*. IEEE, 2011, pp. 191–200.

[7] E. Vacchi and W. Cazzola, "Neverlang: A framework for feature-oriented language development," *Computer Languages, Systems & Structures*, vol. 43, pp. 1–40, 2015.

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

[9] M. Eysholdt and H. Behrens, "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*. ACM, 2010, pp. 307–309.

[10] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "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, January 2018, pp. 75–82.

[11] J. White, J. H. Hill, J. Gray, S. Tambe, A. S. Gokhale, and D. C. Schmidt, "Improving Domain-Specific Language Reuse with Software Product Line Techniques," *IEEE software*, vol. 26, no. 4, pp. 47–53, 2009.

[12] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *International Conference on Software Product Lines*. Springer, 2005, pp. 7–20.

[13] D. Beuche, H. Papajewski, and W. Schröder-Preikschat, "Variability management with feature models," *Science of Computer Programming*, vol. 53, no. 3, pp. 333–352, 2004.

[14] C. Forsythe, *Instant FreeMarker Starter*. Packt Publishing, 2013.

[15] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Modeling Language Variability with Reusable Language Components," in *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[16] I. Kurtev, J. Bézivin, and M. Aksit, "Technological spaces: An initial appraisal," *CoopIS, DOA*, vol. 2002, 2002.

[17] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer Science & Business Media, 2005.

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

[19] M. V. Cengarle, H. Grönniger, and B. Rumpe, "Variability within Modeling Language Definitions," in *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, ser. LNCS 5795. Springer, 2009, pp. 670–684.

[20] S. Maoz, J. O. Ringert, and B. Rumpe, "Semantically Configurable Consistency Analysis for Class and Object Diagrams," in *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, ser. LNCS 6981. Springer, 2011, pp. 153–167.

[21] T. Kühn and W. Cazzola, "Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines," in *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 2016, pp. 50–59.

[22] M. Völter and E. Visser, "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, 2010, pp. 301–304.

[23] G. H. Wachsmuth, G. D. Konat, and E. Visser, "Language Design with the Spoofax Language Workbench," *IEEE software*, vol. 31, no. 5, pp. 35–43, 2014.

[24] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, "Melange: A Meta-language for Modular and Reusable Development of DSLs," in *8th International Conference on Software Language Engineering (SLE)*, Pittsburgh, United States, 2015, pp. 25–36.

[25] T. Kühn, W. Cazzola, and D. M. Olivares, "Choosy and Picky: Configuration of Language Product Lines," in *Proceedings of the 19th International Conference on Software Product Line*. ACM, 2015, pp. 71–80.

[26] M. Völter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with MPS," *Software Language Engineering, SLE*, vol. 16, p. 3, 2010.

[27] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[28] J. Liebig, R. Daniel, and S. Apel, "Feature-Oriented Language Families: A Case Study," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2013, p. 11.

[29] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. Deantoni, J. Klein, and B. Rumpe, "Vcu: the three dimensions of reuse," in *International Conference on Software Reuse*. Springer, 2016, pp. 122–137.

[30] B. Combemale, J. Kienzle, G. Mussbacher, O. Barais, E. Bousse, W. Cazzola, P. Collet, T. Degueule, R. Heinrich, J.-M. Jézéquel, M. Leduc, T. Mayerhofer, S. Mosser, M. Schöttle, M. Strittmatter, and A. Wortmann, "Concern-Oriented Language Development (COLD): Fostering Reuse in Language Engineering," *Computer Languages, Systems & Structures*, vol. 54, pp. 139 – 155, 2018.

[31] A. Garmendia, M. Wimmer, E. Guerra, E. Gómez-Martínez, and J. de Lara, "Automated variability injection for graphical modelling languages," in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2020, pp. 15–21.

[32] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic Composition of Independent Language Features," *Journal of Systems and Software*, vol. 152, pp. 50–69, June 2019.