A Compositional Framework for Systematic Modeling Language Reuse

Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University, Aachen, Germany
www.se-rwth.de

ABSTRACT

Many engineering domains started using generic modeling languages, such as SysML, to describe or prescribe the systems under development. This raises a gap between the generic modeling languages and the domains of experience of the engineers using these. Engineering truly domain-specific languages (DSLs) for experts of these domains still is too challenging for their wide-spread adoption. One major obstacle, the inability to reuse multi-dimensional (encapsulating constituents of syntax and semantics) language components in a black-box fashion, prevents the effective engineering of novel DSLs. To facilitate engineering DSLs, we devised a concept of 3D components for textual, external, and translational DSLs that relies on systematic reuse through systematic closed and open variability in which DSL syntaxes can be embedded, well-formedness rules joined, and code generators integrated in a black-box fashion. We present this concept, a method for its systematic application, an integrated collection of modeling languages supporting systematic language reuse, and an extensible framework that leverages these languages to derive novel DSLs from language product lines. These can greatly mitigate many of the challenges in DSL reuse and, hence, can advance the engineering of truly domain-specific modeling languages.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

DSL, Modeling Language, Reuse, Variability

ACM Reference Format:
Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, Andreas Wortmann. 2020. A Compositional Framework for Systematic Modeling Language Reuse. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3365438.3410934

1 INTRODUCTION

Our society thrives on Cyber-Physical System (CPS) that enable communication, education, healthcare, mobility, and more. These systems are engineered in collaboration with experts from multiple domains, such as mechanical engineering, electrical engineering, material sciences, jurisprudence, software engineering, and systems engineering. To cope with the complexity of engineering these systems, domain experts have begun to leverage the benefits of modeling languages [49] to, among others, describe product geometries [17, 37] physical properties [19, 30], or the integration of contributions from different domains [2, 40].

The efficient use of models by domain experts demands well-defined, Domain-Specific Languages (DSLs) that support automated analysis and synthesis of conforming models. Engineering DSLs is a complex endeavor that demands understanding the domain of interest, creating implementations capturing the DSL's syntax and semantics, integrating these properly, and providing tools supporting to their use. Due to these challenges, experts often have to use overly generic modeling languages, such as UML [25] or SysML [22], instead of DSLs precisely tailored to the concepts and notations of their respective domains. This hinders domain experts in employing these languages efficiently. Reusing components to engineer DSLs more efficiently can lead to more precise and specific languages that can foster the adoption of modeling techniques and, ultimately, facilitate engineering complex CPS. The contributions of this paper support the efficient engineering of DSLs through (1) a novel conceptual model of the reuse of 3D DSL components through closed variability of DSL families (product lines) and open customization; (2) a method for its systematic application; (3) a collection of integrated modeling languages to describe DSL families and their constituents; and (4) an extensible framework that supports engineering DSL families as well as deriving DSL components and complete DSLs from these.

The research results presented in this paper extend the findings presented in [6–8] by making extension points explicit on the component level, introducing different kinds of bindings between the DSL families and their components, and providing an integrated feature modeling language that describes how families relate components through features.

In the remainder, Sec. 2 motivates the benefits of systematic language reuse and Sec. 3 presents preliminaries. Sec. 4 introduces our conceptual model and a method for its systematic application. Sec. 5 describes the modeling languages and the framework realizing the conceptual model. Sec. 6 illustrates its application by example. Sec. 7 discusses observations and related research. Sec. 8 concludes.

2 MOTIVATING EXAMPLE

Consider a company engineering different kinds of CPS featuring state-based behavior, such as robotics systems and appliances for smart buildings. Instead of using the same generic modeling language for all three departments, engineers in each department should be enabled to use a DSL closely related to their domain of

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410934

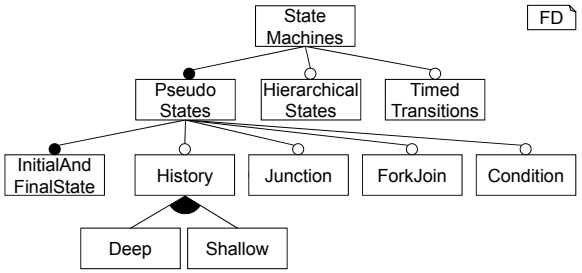

Figure 1: Feature model of an FSM Language Product Line (LPL) (adapted from [38])

expertise. A team of language engineers, therefore, decides to engineer a family of Finite-State Machines (FSMs) DSLs (*cf.* Figure 1). State machines are commonly used behavior descriptions in a multitude of application domains such as robotics [5, 9], aerospace software [21], web applications [23] or game development [35]. Thus, different variants for FSM notations have been brought forth.

For instance, the company has engineered a DSL for describing state-based behavior of robot arms with limited computational power. For this application, performance is crucial and managing state histories and concurrency as induced by join and fork nodes is not desired. However, the behavior of the robot arm should feature time-based triggers on transitions to ease its programming. In another department of the company, a variant of the FSM DSL is used to describe the behavior of web-based smart home appliances. For these, a deep state history can improve the user experience by continuing an interrupted procedure in the state it was interrupted. Furthermore, the language engineers decide that for web applications, junctions in FSMs should simplify user response handling.

Through domain analysis, the engineers of the FSM DSL product line consider these and a set of similarly fashioned applications and decide to create the feature model depicted in Figure 1. Each FSM DSL contains states and transitions that are not explicated in the feature model, as they are contained in every variant of the language. Further, each DSL must contain initial and final pseudo states. These are explicated in the feature model, as the language engineers plan to evolve the feature model in the near future by providing an alternative textual notation for initial and final pseudo states. Moreover, each DSL variant has the option of including deep or shallow history pseudo states, junction pseudo states, fork and join pseudo states, and condition pseudo states. Optionally, an FSM DSL may support modeling hierarchical states that themselves contain states and transitions. Timed transitions, also optional features, enable users to model a passage of time as a transition trigger.

The team developing the above DSL family has four requirements for the reuse of DSL components:

- **R1 Black-box reuse**: To foster DSL reuse across time and involved developers, it must be possible to reuse DSL parts in a black-box fashion without needing to become an expert in their internal implementation details.
- **R2 Structured reuse**: To support reusing DSL parts for building similar DSLs by domain experts without language engineering expertise, it should be possible to arrange the relevant DSL components in the DSL family definition according to their options for composition.
- **R3 Push-button reuse**: To enable domain experts to derive suitable languages with minimal effort, the composition of selected DSL components based on their arrangement in the DSL family should be automated.
- **R4 Open reuse:** To enable extending DSL components with capabilities unforeseen at time of their arrangement, it must be possible to customize these systematically through open variability [13].

A language engineering approach satisfying the above requirements can help the company to reduce cost and effort for engineering and maintaining modeling languages tailored to each kind of CPS they develop.

3 PRELIMINARIES

Our method for efficient DSL engineering relies on research in Software Language Engineering (SLE) [29, 32, 50] and leverages the MontiCore language workbench as technological space [34] for realization and for the case study.

3.1 Software Language Engineering

A modeling language usually is defined by the set of models it accepts. To make languages machine-processable, language definitions in terms of their constituents have been proposed. These usually require that (1) a language definition comprises a concrete syntax, an abstract syntax, a semantic domain, and a mapping from the abstract syntax to the semantic domain giving meaning [26] to the language's sentences [11]; or that (2) each language definition comprises a concrete syntax, an abstract syntax, static semantics (well-formedness rules), and dynamic semantics (behavior) [12]. The abstract syntax of a modeling language defines the structure of accepted models and is typically defined in terms of grammars [3, 10, 47] or metamodels [16, 41, 43]. The concrete syntax is the representation of models towards the user and can be, *e.g.,* textual, graphical, or mixed. Often, this is defined by the editor used to process models. Well-formedness rules can restrict the abstract syntax further to prevent undesired model properties not expressible through the abstract syntax formalism itself. Interpreters and model transformations can give meaning [26] (and possibly behavior) to models by translating these into other languages.

In the following, we assume language implementations that are

- textual: they feature an integrated definition of concrete and abstract syntax through a grammar;
- external: they are not defined in terms of a host language (in contrast to internal DSLs [15]); and
- translational: they give meaning to models through transformation (in particular through code generation).

3.2 MontiCore

We use the language workbench MontiCore [28] to realize our apporach as proof of concept. MontiCore is a language workbench for the development of textual, external DSLs. The integrated concrete and abstract syntax of a DSL is specified in the form of a context-free grammar. From this, MontiCore generates language

```
01 grammar FSM {                                          MCG
02  StateMachine = "sm" Name "{" (IState | ITrans)* "}";
03  interface ITrans;
04  interface IState = Name ;
05  State implements IState = "state" Name ";" ;
06  Transition implements ITrans = from:Name "->" to:Name ";" ;
07 }
```

concrete syntax only · iteration · interface productions · interface implementation

Figure 2: Example MontiCore grammar of an FSM DSL

tooling including an abstract syntax data structure, a parser that instantiates this data structure, 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 called context conditions 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 [20] template engine.

Each MontiCore grammar begins with the keyword `grammar`, followed by the name of the grammar as depicted by example in Figure 2. The body of a grammar (ll. 2-7) contains grammar productions. By default, the first production is the start production of a grammar. On the left-hand side, each production defines a nonterminal, *e.g.,* `StateMachine` (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. Interface nonterminals can underspecify a right-hand side completely (l. 3) or prescribe abstract syntax elements (l. 4). Other productions can implement interface productions (ll. 5-6). 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 over all nonterminals defined by productions implementing the interface nonterminal.

Moreover, MontiCore supports language inheritance [28], which enables reusing complete grammars by inheriting from them and using all inherited productions in the new grammar. We will leverage this to compose the grammars of DSL components according to their arrangement in the DSL family.

4 A METHOD FOR SYSTEMATIC LANGUAGE ENGINEERING

This section introduces the process of creating families of reusable DSL components and composing these to derive novel DSLs. It further presents a conceptual model describing DSL components, their properties, and their relation to feature models of DSL families. With this in place, it explains the effect of selecting two DSL features as the composition of the two related DSL components.

Our method for systematic language composition relies on encapsulating related language constituents in DSL components, making their provided and required extensions explicit, and composing the language components according to these and guided by a feature model. All these activities are related to roles with specific expertise as illustrated in Figure 3. First, language engineering experts create reusable DSL components for specific purposes, such as the features illustrated in Figure 1. Each of these contains a combination

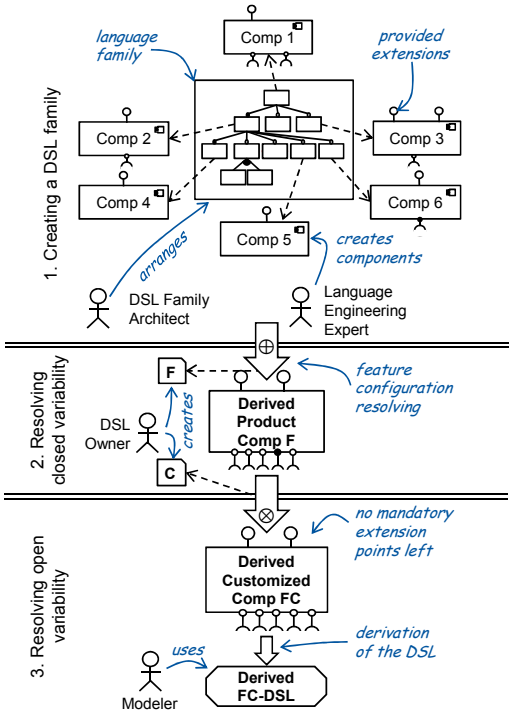

Figure 3: The composition of two DSL components processes all bindings, and updates the interface of the resulting DSL component accordingly. If all mandatory required extensions have been fulfilled, a new DSL can be derived.

of grammars, well-formedness rules, and code generators relating to these grammars. By making their provided extensions (*i.e.,* grammar productions, well-formedness rules, or code generators) and their required extensions (grammar productions or generator extensions) explicit, language family architects can arrange these 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 [42] for logical grouping. By relating features to DSL components and to other features (through their parent-child relation), the language family engineer decides how the components will be composed if their respective features are selected. Once the DSL family is complete, DSL owners, who are experts of the application domains, can derive a suitable DSL by selecting appropriate features from the family. Based on the resulting feature configuration, the selected DSL components are composed and their provided and required extensions are updated accordingly. Through extension points of our framework, the composition of the language constituents (*i.e.,* grammars, well-formedness rules, code generators) is delegated to software modules of the specific technological spaces (such as Neverlang [44], MontiCore [28], or Xtext [18]). The result either is 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 specify additional customization 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 configuring in a

feature model (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 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 supports 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. The modeling languages and the software modules processing these support extension with new language elements and analyses to support extending DSL component definitions and to address challenges of different technological spaces. However, conceptually, our approach assumes the following[1]:

- **A1** Composition leads to *conservative extension* [28], *i.e.,* it is purely additive in terms of language constituents, *i.e.,* composition cannot eliminate grammar productions, well-formedness rules, or generators. Otherwise, composition could eliminate extension points, which yields undesired complexities. Nonetheless, adding new well-formedness rules can restrict the accepted models of the resulting DSL.
- **A2** The grammar language must support identification of extension points. Otherwise, binding extensions to grammars is not possible. This identification, however, can be realized, *e.g.,* through dedicated forms of productions or naming conventions. Hence, many grammar specification formalisms can support this.
- **A3** The well-formedness rules of the technological space must be identifiable and applicable individually. Otherwise, selecting and reusing these rules in different contexts might not be possible. Whether these rules are implemented in OCL [27], a general-purpose programming language [18, 28], or another modeling language [44] then does not matter.
- **A4** The code generators (*producers*) and generated artifacts (*products*) must be defined in a language that supports the notion of object-oriented interfaces and both interfaces (producer and product) must be made explicit by the code generators. Otherwise, the form of adaptation between the generators (producers) or generated artifacts (products) that we propose, will not be possible [6, 8]. This prevents applying our approach to various kinds of target languages and formats (such as CSV, SQL, XML, *etc.*)
- **A5** Each code generator must create a main artifact adhering to the generator's product interface through which that artifact can be invoked during product runtime. If there is no such product, adapting between the required product and the provided product is not possible. While this does not limit the application of our approach technically, enforcing the existence of such a product can make the generated code less efficient. Mitigating this is subject to current research.

[1]The reasoning for the code generator assumptions is discussed in detail in [6, 8].

4.1 A Conceptual Model for Black-Box Language Reuse

Our conceptual model describes the properties of DSL components (**R1**) and DSL families (**R2**) relevant to their systematic reuse. 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 (*cf.* Figure 4).

4.1.1 3D DSL components and interfaces. **DSL components** provide the constituents of a language definition. They are three-dimensional by comprising 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 at least one grammar and can comprise multiple sets of identifiable (**A3**) well-formedness rules, as well as multiple code generator specifications. As both, the well-formedness rules and the generator specifications rely on a grammar for the definition of the abstract syntax data types, it is mandatory for each component. The well-formedness rules are grouped in sets to facilitate their reuse in different contexts. The generator specifications identify a generator as a GPL code class that adheres to an explicit producer interface (**A4**) and creates at least a main GPL artifact that adheres to an explicit product interface (**A5**).

DSL interfaces expose (parts of) these constituents through explicit extensions with cardinalities (optional or mandatory) to the environment (*e.g.,* the language family). For grammars and generators, the interfaces support both, provided and required extensions, whereas for well-formedness rules, only provided extensions can be made explicit. Specifying what is required from a well-formedness rule is subject to ongoing research (*cf.* Sec. 7). For well-formedness rules and code generators, additional parameters can be defined that 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 well-formedness rules extensions offer sets of well-formedness rules for a specific production that can be reused in different contexts. 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 specify missing functionality of a DSL component–*e.g.,* an automaton DSL might need an expression DSL for specifying guards–and can be either optional or mandatory. Required extensions for grammars reference a production of a contained grammar that supports extension. Required generator extensions demand extension for a specific production (such as the guard expressions above), with specific product and producer interfaces as introduced in [6]. Required parameters also are either optional or mandatory and parameterize well-formedness rules or generator specifications, respectively.

Generator specifications describe code generators of components in terms of processed product rules, provided producer and product interfaces, and a set of extension points that follows the

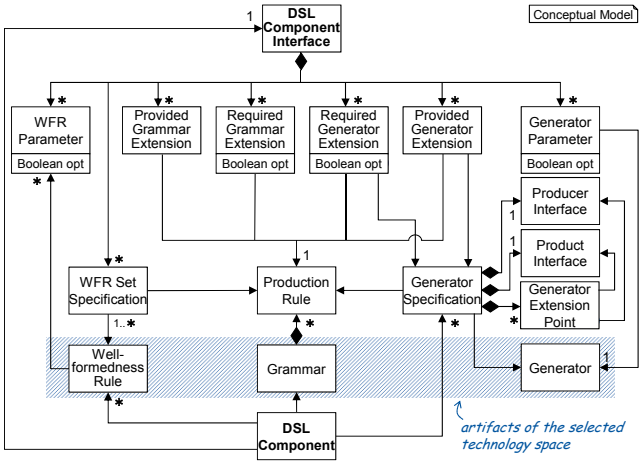

Figure 4: DSL components provide extensions, *i.e.,* parts of language constituents that are exposed by DSL component interfaces. DSL component interfaces also specify required extensions and parameters.

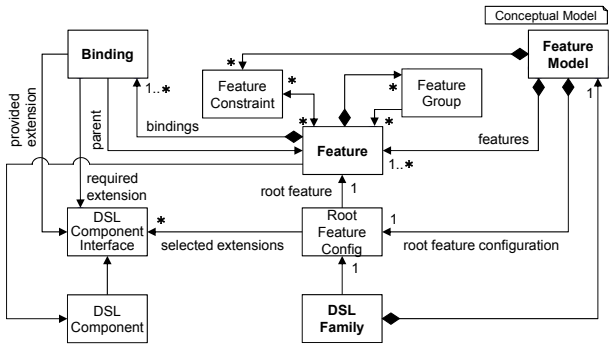

Figure 5: A language family has a feature model with features that reference and bind DSL components.

extension points of the processed grammar. For each required grammar extension, the generator specification provides a generator extension point that describes the required producer interface and a required product interface. The required producer interface prescribes the expected structure of a compatible generator being usable for translating productions embedded into the required grammar extension this extension point relates to. The required product interface prescribes the expected structure of a compatible main artifact produced for the required grammar extension this extension point relates to.

4.1.2 DSL families and bindings. **Language families** [48] describe closed variability through a central feature model [14]. Features relate to the extensions of DSL components and the arrangement of features in this model describes how the components will be composed if their respective features are selected. To this effect, DSL family architects select DSL components for specific purposes and arrange these carefully for DSL owners to use (**R2**).

Figure 5 depicts the conceptual model of DSL families. A family references one or more DSL component(s) and yields a single feature model [1, 4] consisting of features and bindings. A feature references a DSL component of the family that realizes it.

The root (top) feature of the DSL family defines the base DSL component into which the components related to all selected child features are embedded according to the family's feature model. As such, it might yield provided extensions as well, for which the **root feature configuration** can define bindings (*i.e.,* selections) already. This enables using a comprehensive DSL for the root feature while giving the flexibility of reusing only selected parts of it.

Bindings relate features to DSL components. Our concept supports three kinds of bindings, matching the different kinds of required and provided extensions (grammar, well-formedness rule, generator). Bindings are defined within features, *i.e.,* each feature describes how (a subset of) the provided extensions of its related DSL component will be bound to (a subset of) the required extensions of the DSL component related to its parent feature. To this end, each non-abstract feature must define at least one such binding and can define as many bindings as there are provided extensions in its DSL component. The different kinds of bindings are:

Grammar bindings map a provided grammar extension of the embedded component (*e.g.,* of a child feature) to a required grammar extension of the embedding component (*e.g.,* of a parent feature). The effect of such a binding is that everything producible from the provided grammar extension will become producible from the required grammar extension as well. For instance, when embedding the provided grammar extension for arithmetic expressions into a required grammar extension for Boolean expressions, arithmetic expressions become an alternative to the former. This can be realized through production inheritance [28] or adding an alternative supporting the provided productions to the requiring productions [5, 44]. This composition is supported by adhering to (**A1**) and (**A2**).

Generator bindings map a provided generator extension of the embedded component to the required generator extension of the embedding component. Such a binding entails that the provided generator will be used whenever the required generator is called. For instance, embedding a generator for translating arithmetic expressions to Java into another generator requiring that translation entails, per construction detailed in [8], that the embedding generator will call the embedded generator via an adapter between the required producer for arithmetic expressions and the provided producer interface of the generator for arithmetic expressions. This composition is enabled by (**A1**), (**A3**), and (**A4**).

Well-formedness rule embeddings join a well-formedness rule set of the embedded component into a well-formedness rule set of the embedding component. The result is a novel component with the same number of well-formedness rule sets than before, but more well-formedness rules in its sets. This enables refining DSLs by adding additional rules to its provided well-formedness rules.

Well-formedness rule addition adds a complete set of well-formedness rules of the embedded component en-bloc to the embedding component. Through this, a novel set of well-formedness rules becomes present in the resulting component. Both forms of well-formedness rule set composition rely on (**A1**) and (**A5**).

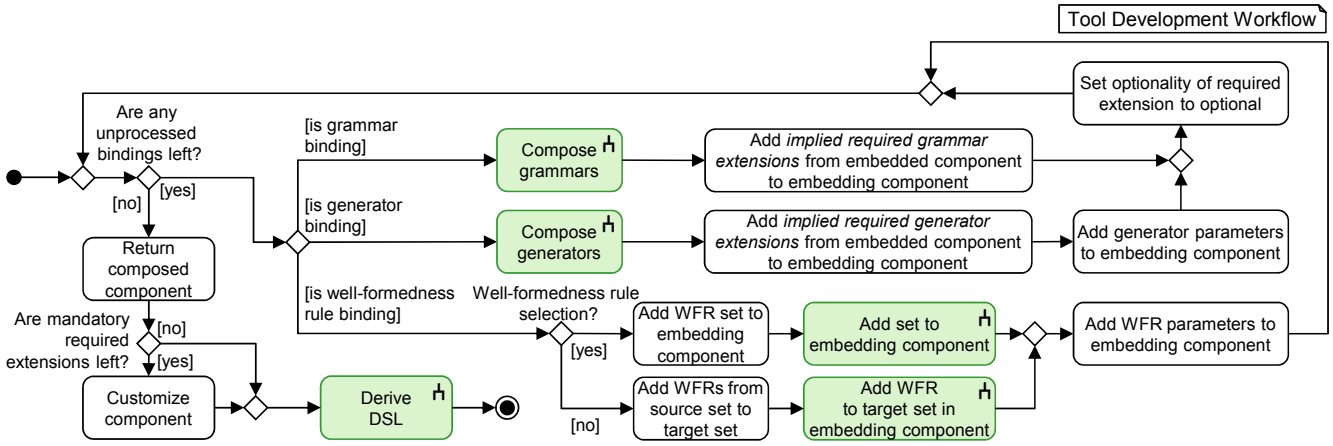

Figure 6: DSL components are composed according to the bindings defined between DSL family features.

4.2 Composing DSL Components

The composition of two DSL components is the directed application of bindings between these components. It produces a novel component resulting from adding selected provided extensions of the embedded component into the respective required extensions of the embedding component. This comprises two main activities: (1) Composition of the components' interfaces; and (2) Composition of the comprised language definition constituents (grammars, well-formedness rules, code generators);

Our method of reusing DSLs and DSL parts is independent of the actual composition of language constituents in the different technological cases as long as these adhere to (**A1**)-(**A5**). Consequently, the method and its realization anticipate extension with software modules specific to the technological space of choice that take care of the technical composition (*cf.* Sec. 5.4).

The process of composing two DSL components along their interface is illustrated in Figure 6: As long as there are unprocessed bindings, these and the related artifacts are passed to the technology space-specific composition components (used by green activities with fork icon) to perform the composition of DSL constituents. Afterwards, the required extensions of the language interface of the embedding component are updated accordingly by (a) setting fulfilled extensions to be optional and (b) adding implied required extensions of the embedded component. Provided extensions of the embedded components are not added to the interface of the embedding component as this would add options for reuse unintended by the DSL family. For well-formedness rules, either a set of the embedded component was meant to be reused en-bloc, in which case the complete set is added to the interfaces of the embedded component, or individual rules shall be reused. In this case, these are added to the set of well-formedness rules of the embedding component as indicated by the respective bindings.

Where an embedded component yields parameters, these are added to the interfaces of the embedding component. If all required extensions are fulfilled, the resulting DSL component can be translated into a new DSL automatically. Otherwise, it needs subsequent customization.

For a feature configuration relative to a language family, the feature tree of the language family is traversed bottom-up. If a feature is selected in the feature configuration, all associated bindings of this particular feature are applied, and the components are composed pairwise. The application of the bindings is similar to the composition process stated in the former part of this section. The traversing of the feature tree ends with the root feature configuration, if present. When applying the root feature configuration, all provided extensions and well-formedness rule sets not stated in the root configuration are removed. If the component has no mandatory required extension or component parameter, a usable DSL can be derived from it automatically (**R3**). Otherwise, customizing the component (**R4**) is necessary to obtain a usable DSL. To this effect, the bindings between the embedded and the customized component are applied and the components are composed as if they were related to a parent feature and its child.

5 MODELING LANGUAGES AND FRAMEWORK

Based on the example of the FSM family of Figure 1, this section presents the modeling languages for DSL components and families.

5.1 DSL Components

The DSL component language reifies our conceptual model of DSL components and interfaces (**R1**) in form of a MontiCore modeling language. Following the conceptual model, each DSL component references exactly one grammar, zero to many generator contexts (describing producers and products), as well as various provided and required extensions and well-formedness rule sets.

Figure 7 illustrates this by example of the `TransitionSystem` DSL component It references a grammar via its fully qualified name `mc.FSM` (l. 2) and specifies the generator `FSMG` with context `FSMGenerators` (l. 3, Figure 8). The generator context is a class diagram describing the generator and its interfaces.

Afterwards, `TransitionSystem` defines a provided and two required grammar extensions of different optionalities (ll. 5-7) using productions from the `mc.FSM` grammar. This defines that the

```
01 dsl component TransitionSystem {                                  LC
02   grammar mc.FSM;          <- grammar reference
03   gen FSMG context fsm._gen.FSMGenerators;   <- generator context
04
05   provides production StateMachine;
06   requires optional production IState;        <- grammar extensions
07   requires mandatory production ITrans;
08
09   provides gen FSMMainGen for StateMachine with FSMG;
10   requires optional gen StateGen for IState with FSMG;
11   requires optional gen TransGen for ITrans with FSMG;
12                                                <- generator extensions
13   wfrs TransitionsCorrect {
14     fsm._cocos.TransitionSourceStateExists;
15     fsm._cocos.TransitionTargetStateExists;
16   }                                            <- provided sets of well-formedness rules
17   wfrs TSCorrect {
18     fsm._cocos.AllStatesReachable;
19     fsm._cocos.NamesAreUpperCase;
20   }
21 }
```

Figure 7: A component representing a transition system DSL. It provides and requires extensions for the language's grammar, generator, and well-formedness rules.

component enables extension for the productions `IState` and `ITrans`. For code generation, the component defines a provided and two required generator extensions (ll. 9-11) relative to the generator context represented in Figure 8. The provided generator extension `FSMMainGen` enables reusing the component's generator. The required generator extensions `StateGen` and `TransGen` enable to extend code generation of this component for states and transitions accordingly. Ultimately, `TransitionSystem` also defines two provided sets of well-formedness rules of two rules each (ll. 13-20). The FSM grammar itself is illustrated in Figure 2 and comprises four productions: it defines two interfaces that can act as grammar extension points (ll. 3-4) and defines a transition system as a named collection of instances of these interfaces (l. 2). For states and transitions, it provides a default implementation (ll. 5-6).

Figure 8 depicts the generator context for the transition system component. The top three classes, `IFSMProducer`, `IFSMProduct`, and `IFSMSystemGenerator` define how this the `FSMGenerator` can be embedded into other components, *i.e.,* that it can act as an `IFSMProducer` and that its generated artifacts will adhere to the `IFSMProduct` interface. Moreover, `FSMGenerator` yields registration methods corresponding to its two extension points. For extension of states, *e.g.,* the `FSMGenerator` expects a producer of type `IStateProducer` and that this producer generates a main artifact of type `IStateProduct`, hence the corresponding code interfacing with implementations of this interface can be generated.

When bindings between two DSL components specify embedding `FSMGenerator` as `IFSMProducer` into a generator expecting another particular producer interface (as defined in the embedding component's generator context), an adapter between the expected producer interface and `IFSMProducer` and a factory for its injection is generated. When the adaptation is non-trivial, this generated adapter needs to be extended with handcrafted adaptation functionality using the generation gap pattern [24]. For the product interfaces, the same mechanism is applied. The classes of the generator context and signatures of the registration methods follow framework-wide naming conventions [6, 8].

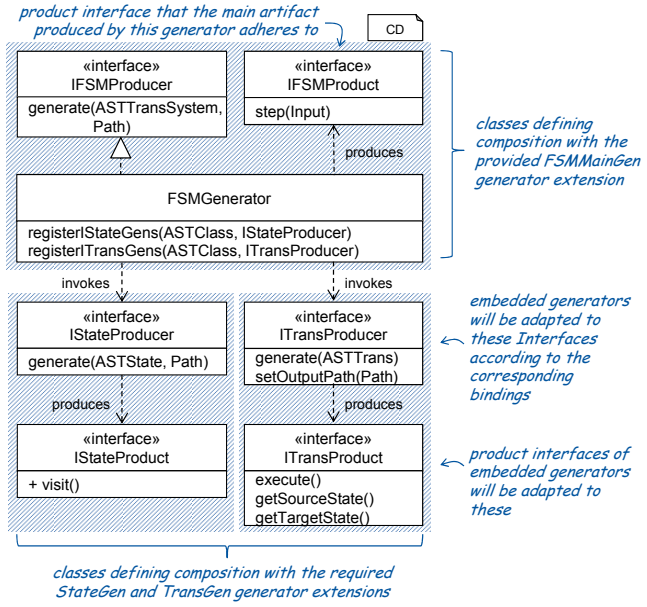

Figure 8: The generator context for the transition system DSL component. It contains the generator interfaces and classes for the extensions of the component.

5.2 DSL Families

DSL families consist of a feature model, components that realize features of the model, and bindings between these components.

Figure 9 depicts an excerpt of the `StateMachineFamily` that describes a family of FSM languages. The family contains a textual feature model to arrange the components within the family (**R2**). It is an excerpt of the one presented in Figure 1 (ll. 3-10). After the feature model, the family defines features in terms of names, realizing component, and bindings (ll. 12-31) such that each feature of the feature model is realized through a DSL component. For instance, in the `StateMachineFamily` the feature `StateMachines` (ll. 12ff) is realized through the DSL component `TransitionSystem` (l. 13). A detailed insight into this component is given in Figure 7.

All non-root features yield bindings that connect the provided extensions of their DSL components with the required bindings of their parent feature's component–or its ancestor, if the parent feature is abstract. For instance, the `InitialAndFinalState` feature is realized through the component `InFinState` (ll. 17ff), which is illustrated in Figure 12. Afterwards (ll. 19-23), bindings for productions, generators and well-formedness rules are defined between the feature-realizing component `InFinState` and the grandparent feature's component `TransitionSystem` (*cf.* Figure 7). Through these bindings, *e.g.,* the provided grammar extension `InitialState` of `InFinState` is bound to the required grammar extension `IState` of `TransitionSystem`.

5.3 DSL Component Customization

The DSL component customization realizes open variability (**R4**). In contrast to the DSL family configuration, it enables the DSL owner to customize a DSL component by binding its required extensions to other DSL components that might not have been part of the

```
01 family StateMachineFamily {                                          LF
02
03   feature diagram StateMachines {        <- root feature
04     mandatory PseudoStates {
05       mandatory InitialAndFinalState;
06       abstract History {…} or Junction or //...;
07     }
08     optional HierarchicalStates;
09     optional TimedTransitions;
10   }
11                                  <- feature definition
12   feature StateMachines {
13     component ts.comp.TransitionSystem;
14     // Bindings of the StateMachines feature
15   }
16
17   feature InitialAndFinalState {
18     component ps.comp.InFinState;
19     bind production InitialState  -> IState;
20     bind production FinalState    -> IState;
21     bind generator  InitStateGen  -> StateGen;
22     bind generator  FinalStateGen -> StateGen;
23     bind wfrs       CheckStateCardinality;
24   }
25
26   feature TimedTransitions {
27     component tt.comp.TransitionsWithTiming;
28     bind production TimedTrans -> ITrans;
29     bind generator  TTGen      -> TransGen;
30     bind wfrs       TimingCorrectness;
31   }
32   // Definitions of further features
33 }
```

Figure 9: Excerpt of a textual model of the `StateMachineFamily` DSL family (*cf.* Figure 1).

DSL family. Furthermore, the customization can assign values to parameters of the identified DSL component. Figure 10 illustrates a customization by example.

The customization `RobotArmWithClock` customizes the component `RobotArmLang` (see Figure 14). Bindings in the customization have the same syntax as in the feature definition of the language family and are applied in the same fashion. The left side of the binding is the source, *i.e.,* a provided extension or well-formedness rule set, and the right side of the binding is the target, *i.e.,* a required extension or well-formedness rule set. In the customization the source of the binding is the fully qualified name of the language component and the name of the respective provided extension or well-formedness rule set. The required extension or well-formedness set which is the target of the binding, always originates from the customized component. The customization `RobotArmWithClock` binds a grammar production and a generator for a clock expression (ll. 3ff) of a DSL component `ce.comp.Clock` to the customized component `RobotArmLangComp`. Furthermore, it limits the number of initial states to one by setting the corresponding parameter (l. 6). Customization produces a new composed component that contains the bound extensions and no longer contains the set parameter.

5.4 Language Engineering Framework

For demonstration of the feasibility of our approach, we have implemented the framework for deriving languages from the family (**R3**) in four different modules that each relate to different activities and their modeling languages (*cf.* Figure 11).

The `DSL Component Processor` (top left) is responsible for parsing, processing, and validating DSL components (*cf.* Figure 4) and used by the language engineering expert. Therefore, the

```
01 import ce.comp.Clock;                                                CC
02
03 customization RobotArmWithClock for RobotArmLangComp {
04   bind production Clock.ClockExpr        -> ITimedExpr;
05   bind generator  Clock.ClockGen         -> TimerGen;
06   assign          numberOfInitialStates  =  1;
07 }
```

Figure 10: The textual model of a customization for the DSL component `RobotArmWithClock` (*cf.* Figure 14). It contains two bindings and a parameter assignment.

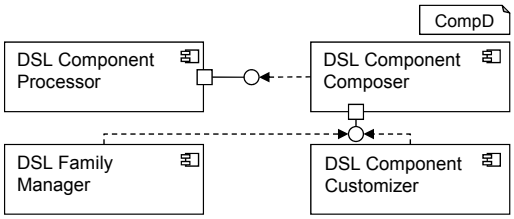

Figure 11: Our framework comprises modules for processing, composing, customizing DSL components, and managing DSL families.

module holds the corresponding language processing tools (parser, lexer, well-formedness checker) as well as an interface for calculating implications. A MontiCore-specific implication calculator implements the calculator interface to resolve and validate implications. After processing a DSL component model, it produces a DSL component that can be reused in DSL families and other contexts.

The `DSL Component Composer` (top right) takes two DSL components with a set of bindings and composes these as presented in Sec. 4.2. It relies on the DSL Component Processor to parse components, before it composes their interfaces and artifacts. For the latter, it provides an interface to integrate modules specific to the technological space operated within. The `DSL Family Manager` (bottom left) evaluates and resolves the DSL families and related feature configurations. To this end, it comprises three modules that process feature models, feature configurations, and resolve the latter. It also interacts with the DSL Component Composer for composing DSL components in the process of applying the DSL family configuration and deriving a new DSL (of component).. The `DSL Component Customizer` (bottom right) reads and applies the customization configuration. It parses and validates customization configurations and applies these.

6 APPLICATION EXAMPLE

This section provides an insight into applying our concepts on the example of deriving an FSM DSL used for describing state-based behavior of a robot arm (*cf.* Sec. 2) [39]. This includes selecting required features from the DSL family, showing the feature-realizing DSL components, their composition, and the artifact composition exemplified via the composition of grammar productions.

Consider language engineering experts that design DSLs for different concerns of Figure 1 and create DSL components accordingly. For instance, the DSL component `InFinState` (see Figure 12) contains a grammar and a generator (ll. 2-3) to provide productions

```
01 dsl component InFinState {                                   LC
02   grammar mc.InitialAndFinalState;        <- contained grammar
03   gen IFG context infinstate._gen.InFinGenerators;  <- generator
04
05   provides production InitialState;
06   provides production FinalState;          <- provided grammar extensions
07
08   provides gen InGen  for InitialState with IFG;
09   provides gen FinGen for FinalState   with IFG;   <- provided generator extensions
10
11   wfrs CheckStateCardinality {
12     infinstate._cocos.InitStatesCardinality;
13   }
14                                            <- well-formedness rule parameter
15   wfr parameter Integer numberOfInitialStates for
16                 infinstate._cocos.InitStatesCardinality;
17 }
```

```
01 grammar InitialAndFinalState {                                MCG
02   InitialState = "initial" "state" Name;
03   FinalState   = "final"   "state" Name;
04 }
```

Figure 12: The DSL component `InFinState` that provides grammar productions and generator for the language elements initial and final state (top) and its grammar (bottom).

```
01 dsl component TransitionsWithTiming {                        LC
02   grammar mc.TimedTransition;
03   gen TTG context time._gen.TTGenerators;
04
05   provides production TimedTrans;
06   requires optional production ITimedExpr;
07
08   provides gen InGen    for TimedTrans with TTG;
09   requires gen TimerGen for ITimedExpr with TTG;
10
11   wfrs TimingCorrectness {
12     time._cocos.IsTimingPositive;
13   }
14 }
```

```
01 grammar TimedTransition {                                     MCG
02   interface ITimedExpr;
03   TimedTrans = Name "-" timer:ITimedExpr ">" Name;
04   IntegerTimer implements ITimedExpr = IntLiteral "sec";
05 }
```

Figure 13: The DSL component `TransitionsWithTiming` (top) and its grammar (bottom).

and generators for initial and final state definitions (ll. 5-9). In addition, it contains a well-formedness rule that limits the number of initial states. The number itself is configurable via the parameter `numberOfInitialStates` (ll. 15ff). Figure 12 depicts the referenced grammar with the productions provided by the DSL component. The `TransitionsWithTiming` (see Figure 13) provides a production and a generator for timed transitions with a counter that decreases over time and is extensible with additional timing expressions. A DSL family architect then models a family for FSMs. Therefore, she arranges several DSL components resulting in the DSL family `StateMachineFamily` (*cf.* Figure 9).

Based on this family, a DSL owner then can derive an FSM DSL for describing the state-based behavior of robot arms. For this, she selects the features `StateMachines`, the abstract feature `PseudoStates`, `InitialAndFinalState`, and `TimedTransitions` in a feature configuration `RobotArmLang`. The feature `StateMachines` is realized by the `TransitionSystem` component, the feature `InitialAndFinalState` is realized by the component `InFinState` (see Figure 12), and the feature `TimedTransitions` is realized by the `TransitionsWithTiming` component. Based on the feature selection, the DSL components `InFinState` and `TransitionsWithTiming` are composed with the DSL component `TransitionSystem`. This results in a composed DSL component `RobotArmLangComp` (see Figure 14). Since bound provided extensions are embedded into the component that is the target of the binding, the provided extensions of the embedded components are no longer available in the composed component. However, to preserve further extension, the required extensions of the embedded components remain available in the composed component. Thus, the composed component adopts the provided extensions of the component `TransitionSystem` (ll. 7, 12). Furthermore, the required extensions of the embedded component `TransitionsWithTiming` are available (ll. 10, 15). Through binding the well-formedness rule set in the feature `TimedTransitions`, the set `TimingCorrectness` is present in the composed component (ll. 25ff) as well as the well-formedness rule sets of the component `TransitionSystem` (ll. 17-20 and ll. 21-24). As bindings of the feature `InitialAndFinalState` include the well-formedness rule set `CheckStateCardinality` (ll. 28ff) of component `InFinState`, the parameter `numberOfInitialStates` is available in the composed component (ll. 31ff).

The composition of the language artifacts according to the bindings defined by the selected features results in a composed grammar `RobotArmLangGrammar` (l. 2) named after the DSL family configuration and a new generator context `RobotArmLangGenerators` (l. 5). Our approach implements the composition of grammars, generators, and well-formedness rule sets as presented in [6, 7]. Hence, the generator context contains the abstract adapter classes between the producer and product interfaces of the extended and embedded generators that are necessary to compose the implementations of the bound generators.

Figure 15 depicts the composed grammar `RobotArmLangGrammar`. The grammar results from applying the bindings defined in the selected features of the DSL family (see Figure 9). The feature `InitialAndFinalState` defines grammar bindings that bind the provided extensions `InitialState` and `FinalState` of the component `InFinState` to the required extension `IState` of component `TransitionSystem` (ll. 20ff). Identifying the grammar of the provided production requires insights into the DSL component. The component model of `InFinState` (*cf.* Figure 12) references the grammar `InitialAndFinalState` depicted in Figure 12. It contains the two productions referenced by the provided extensions of the DSL component. The required extension of the component `TransitionSystem` references the interface production `IState` of the grammar `TS` (*cf.* Figure 7). From this, the tooling generates the composed grammar depicted in Figure 15. For the grammar bindings of the feature `InitialAndFinalState`, the composed grammar extends the grammars `TS` and `InitialAndFinalState` referenced by the bound components (l. 2). Also, the grammar defines two productions `InitialState2IState` and `FinalState2IState` adapting the bound productions to another. Processing the grammar bindings of the feature `TimedTransitions` is similar to the ones of the feature `InitialAndFinalState`. Here, the composed grammar introduces a new

LC
```
dsl component RobotArmLangComp {
  grammar mc.RobotArmLangGrammar;
  gen FSMG context fsm._gen.TSGenerators;
  gen TTG context time._gen.TTGenerators;
  gen RAG context ra._.RobotArmLangGenerators;

  provides          production TransSystem;
  requires optional production IState;
  requires optional production ITrans;
  requires optional production ITimedExpr;

  provides          gen TSMainGen for TransSystem with FSMG;
  requires optional gen StateGen  for IState      with FSMG;
  requires optional gen TransGen  for ITrans      with FSMG;
  requires optional gen TimerGen  for ITimedExpr  with TTG;

  wfrs TransitionsCorrect {
    fsm._cocos.TransitionSourceStateExists;
    fsm._cocos.TransitionTargetStateExists;
  }
  wfrs TSCorrect {
    fsm._cocos.AllStatesReachable;
    fsm._cocos.NamesAreUpperCase;
  }
  wfrs TimingCorrectness {
    time._cocos.IsTimingPositive;
  }
  wfrs CheckStateCardinality {
    infinstate._cocos.InitStatesCardinality;
  }
  wfr parameter Integer numberOfInitialStates for
                infinstate._cocos.InitStatesCardinality;
}
```

Figure 14: The component resulting from the feature configuration contains the DSL components added by the bindings defined in the selected features.

production implementing the interface of the required extension and extends the production of the provided extension.

The DSL owner needs a timed expression to define a specific trigger time. As this is not available in the DSL family, she customizes the derived DSL component (see Figure 14). With the customization depicted in Figure 10, she binds a production and generator realizing the expression that enables her to define a condition based on a certain time. Here, she limits the number of initial states to one state by setting the parameter `numberOfInitialStates`. The composition produces a new DSL component containing the language features added via the customization. A modeler then can use an FSM DSL tailored specifically to her needs.

MCG
```
grammar RobotArmLangGrammar
        extends FSM, TimedTransition, InitialAndFinalState {
  start StateMachine;

  InitialState2IState extends InitialState implements IState
                     = "initial" "state" Name;
  FinalState2IState extends FinalState implements IState
                     = "final" "state" Name;
 TimedTrans2ITrans extends TimedTrans implements ITrans
                     = Name "-" timer:ITimedExpr ">" Name;
}
```

Figure 15: The composed grammar after applying the feature configuration. It adapts the provided grammar productions to the productions of the required extensions.

7 DISCUSSION AND RELATED WORK

Encapsulating constituents relating to a language concern in an explicit DSL component eases their reuse as it mitigates the challenge of identifying how the usually only loosely coupled language constituents can be reused without becoming an expert in their implementation (**R1**). Through arranging DSL components in families, their systematic reuse can be guided, which eases composing these components accordingly (**R2**). This separation of concerns along the different roles also can liberate domain experts enacting as DSL owners from needing in-depth language engineering expertise (**R3**). Customization enables open variability of DSL components with capabilities not foreseen in the DSL family (**R4**). However, our approach entails additional efforts in defining language components and empirically measuring their impact demands further research.

Our approach to DSL engineering is limited to textual, external, and translations DSL and has comprehensive requirements for compatible technological spaces. Based on these assumptions, it uses specific composition operations, namely embedding (grammars), merging (context conditions), and adapted embedding (code generators). Removing parts of a language by selecting features, thus, is not possible. While the set of valid models can be restricted through adding new features (that contain suitable context conditions), the non-terminals, context conditions, and code generators selected by other features remain part of the language (family). Moreover, we currently use the technological space of MontiCore for realizing our concepts as well as for engineering language families. This might introduce biases towards MontiCore in our concepts, we are currently experimenting with the language workbenches Neverlang [44] and Xtext [18].

Several language engineering tools such as MPS [46], Spoofax [47], and Melange [16] 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 [3, 33, 44, 45], or do not support all three component dimensions [16, 36].

Overall, our approach builds upon ideas formulated as concern-oriented language development [13, 31], 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.

8 CONCLUSION

We have presented concepts for reusing 3D DSL components through closed variability of DSL families (product lines) and open customization. These concepts are intended to be used in a systematic fashion by different stakeholders involved in language engineering, who are supported by a collection of integrated modeling languages to model DSL families and their constituents. While our concepts and their application method are currently limited to textual, external, and translational DSLs, they greatly facilitate DSL reuse and, hence, foster the adoption of modeling languages by domain experts. In the future, we plan to relax our method's assumptions (**A1**-**A5**) and integrate further language definition dimensions.

REFERENCES

[1] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Conference on Software Product Lines*. Springer, 7–20.

[2] Olaf Berndt, Uwe Freiherr von Lukas, and Arjan Kuijper. 2015. Functional Modelling And Simulation Of Overall System Ship-Virtual Methods For Engineering And Commissioning In Shipbuilding.. In *ECMS*. 347–353.

[3] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[4] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. 2004. Variability management with feature models. *Science of Computer Programming* 53, 3 (2004), 333–352.

[5] Jonathan Bohren and Steve Cousins. 2010. The SMACH High-Level Executive. *IEEE Robotics & Automation Magazine* 17, 4 (2010), 18–20.

[6] 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.

[7] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM.

[8] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2019. Systematic Composition of Independent Language Features. *Journal of Systems and Software* 152 (2019), 50–69.

[9] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. 2015. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*.

[10] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2–Componentised Language Development for the JVM. In *International Conference on Software Composition*. Springer, 17–32.

[11] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2009. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09) (LNCS 5795)*. Springer, 670–684.

[12] Tony Clark, Mark G. J. van den Brand, Benoit Combemale, and Bernhard Rumpe. 2015. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*. Springer, 7–20.

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

[14] Krzysztof Czarnecki and Ulrich W Eisenecker. 2000. Generative Programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen* (2000), 15.

[15] Manuela Dalibor, Nico Jansen, Johannes Kästle, Bernhard Rumpe, David Schmalzing, Louis Wachtmeister, and Andreas Wortmann. 2019. Mind the Gap: Lessons Learned from Translating Grammars Between MontiCore and Xtext. In *International Workshop on Domain-Specific Modeling (DSM'19)*, Jeff Gray, Matti Rossi, Jonathan Sprinkle, and Juha-Pekka Tolvanen (Eds.). ACM, 40–49.

[16] 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, 25–36.

[17] Jiwang Du, Qichang He, and Xiumin Fan. 2013. Automating generation of the assembly line models in aircraft manufacturing simulation. In *2013 IEEE International Symposium on Assembly and Manufacturing (ISAM)*. IEEE, 155–159.

[18] Moritz Eysholdt and Heiko Behrens. 2010. 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, 307–309.

[19] Stefan Feldmann, Sebastian JI Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan JJ Paredis, and Birgit Vogel-Heuser. 2015. Towards effective management of inconsistencies in model-based engineering of automated production systems. *IFAC-PapersOnLine* 48, 3 (2015), 916–923.

[20] Charles Forsythe. 2013. *Instant FreeMarker Starter*. Packt Publishing.

[21] Ricardo Bedin Franca, Jean-Paul Bodeveix, Mamoun Filali, Jean-Francois Rolland, David Chemouil, and Dave Thomas. 2007. The AADL behaviour annex–experiments and roadmap. In *null*. IEEE, 377–382.

[22] Sanford Friedenthal, Alan Moore, and Rick Steiner. 2014. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann.

[23] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. 2010. Using state machines for a model driven development of web service-based sensor network applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. 2–7.

[24] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, 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 *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*. SciTePress, 74–85.

[25] Object Management Group. 2010. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03).

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

[27] 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.

[28] Katrin Hölldobler and Bernhard Rumpe. 2017. *MontiCore 5 Language Workbench Edition 2017*. Shaker Verlag. http://www.se-rwth.de/phdtheses/MontiCore-5-Language-Workbench-Edition-2017.pdf

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

[30] Botond Kádár, Walter Terkaj, and Marco Sacco. 2013. Semantic Virtual Factory supporting interoperable modelling and evaluation of production systems. *CIRP Annals* 62, 1 (2013), 443–446.

[31] 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.

[32] Anneke Kleppe. 2008. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley.

[33] 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.

[34] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. 2002. Technological Spaces: an Initial Appraisal. *CoopIS, DOA* 2002 (2002).

[35] Aung Sithu Kyaw. 2013. *Unity 4. x Game AI Programming*. Packt Publishing Ltd.

[36] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 11.

[37] Michael Lütjen and Daniel Rippel. 2015. GRAMOSA framework for graphical modelling and simulation-based analysis of complex production processes. *The International Journal of Advanced Manufacturing Technology* 81, 1-4 (2015), 171–181.

[38] David F. Méndez Acuña. 2016. *Leveraging Software Product Lines Engineering in the Construction of Domain Specific Languages*. Ph.D. Dissertation. INRIA Rennes.

[39] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2014. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag.

[40] Chantal Steimer, Jan Fischer, and Jan C Aurich. 2017. Model-based design process for the early phases of manufacturing system planning using SysML. *Procedia CIRP* 60 (2017), 163–168.

[41] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional.

[42] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract features in feature modeling. In *2011 15th International Software Product Line Conference*. IEEE, 191–200.

[43] Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 819–820.

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

[45] Markus Völter and Konstantin Solomatov. 2010. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE* 16 (2010), 3.

[46] 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.

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

[48] 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), 47–53.

[49] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. 2019. Modeling Languages in Industry 4.0: an Extended Systematic

Mapping Study. *Software and Systems Modeling* 19, 1 (January 2019), 67–94. http://www.se-rwth.de/publications/Modeling-languages-in-Industry-4-0-an-extended-systematic-mapping-study.pdf

[50] Steffen Zschaler, Dimitrios S Kolovos, Nikolaos Drivalos, Richard F Paige, and Awais Rashid. 2009. Domain-specific metamodelling languages for software language engineering. In *International Conference on Software Language Engineering*. Springer, 334–353.