Software and Systems Modeling
https://doi.org/10.1007/s10270-026-01382-w

REGULAR PAPER

A theory of composable modeling language components

Jérôme Pfeiffer[1] · **Andreas Wortmann**[1]

Received: 20 March 2025 / Revised: 23 February 2026 / Accepted: 2 April 2026
© The Author(s) 2026

Abstract

Modern-day software has become increasingly complex and ubiquitous in many domains. In many cases, the scarcity of software engineering experts means that domain experts must engage in software development tasks. (DSLs) can help bridge this gap by enabling domain experts to express solutions in familiar domain terms. Engineering such (DSLs) is complex. This complexity arises from the variety of artifacts involved—such as grammars, well-formedness rules, and code generators—and from the need to integrate these artifacts across languages. In software language engineering, language workbenches for textual, external modeling languages with translational semantics seem to be a popular, which includes inhabitants such as Xtext, Neverlang, MontiCore, and Spoofax. Yet, reusing existing languages in these environments is often piecemeal and driven by the constraints of specific realization technologies (e.g., Xtend, FreeMarker). This typically requires software language engineers to create DSLs for domain experts, rather than enabling domain experts to reuse and adapt languages themselves. As a result, enabling the systematic reuse of software languages remains a fundamental challenge in software engineering. To reduce the gap between problem domain expertise and software language engineering solution expertise, we have conceived a top-down software language reuse method that enables domain experts to reuse existing languages through formally specified language components that encapsulate the realizations of syntax and semantics of (a fragment of) a language packed ready for systematic reuse. To ease its understanding of our method and its adoption to other technological spaces, we describe our methods and the composition process independent of specific technologies. The presented method of language reuse aims to advance software language engineering for textual, external, translational (DSLs) and may serve as the basis for further investigation of formalizing language reuse.

Keywords DSLs · Language components · Formalization

1 Introduction

Engineering software-intensive systems increasingly involves collaboration with experts from multiple domains [1]. To describe their problems and solutions precisely, domain experts frequently use (DSLs), such as AutomationML [2], SysML [3], SparQL [4], to, e.g., describe domains like production [5], robotics [6], aviation [7], software architectures [8], or systems engineering [9]. The efficient use of these (DSLs) [1] requires a high precision concerning the domain of interest [10]. Engineering precise (DSLs) [11] still is challenging. It requires analyzing the domain, designing syntax and semantics that fit domain needs, integrating the resulting language into existing systems, and providing accessible tooling [12]. This wealth of challenges impedes DSL adoption in practice. A cornerstone of the success of modern software development is the reuse of well-tested, documented, and widely available frameworks and libraries, which improves efficiency and quality. We believe that modularly reusing software languages by composition [13, 14] can bring similar benefits to DSL development, enabling more precise languages with less effort. Therefore, we propose a systematic, top-down, method to reuse languages and language fragments of textual, external (DSLs) with translational semantics.

As part of this method, languages are grouped into language components [15–17] comprising a grammar-based

Communicated by Juan de Lara.

✉ Jérôme Pfeiffer
jerome.pfeiffer@isw.uni-stuttgart.de

Andreas Wortmann
andreas.wortmann@isw.uni-stuttgart.de

[1] Institute for Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Seidenstr. 36, 70174 Stuttgart, Germany

Published online: 24 June 2026

textual syntax, well-formedness rules, and code generators and component interfaces, which make the required extensions and provided extensions as well as their relations explicit [18]. The language components are arranged in language families, which makes their closed variability for planned reuse explicit and accessible to domain experts. By selecting features from the language family, the components and their constituents are composed with relatively little manual effort.

Thus, this paper has the following contributions:

1. A formal definition of the language constituents (1) grammar, (2) well-formedness rules, (3) translational semantics realized by code generators, and (4) their composition using embedding [19], and
2. a formal definition of language components encapsulating the language constituents, their interface, and their composition using embedding,
3. a formal definition of language families that associate features of a feature diagram with language components to guide their composition,
4. algorithms for composing language components and traversing the language family based on a feature configuration.

Overall, on the one hand, this formalization shows a novel method for how to reuse constituents of textual languages in a component-based and systematic fashion, and on the other hand, it provides a definition that is independent of a technological space and therefore can be generalized toward language workbenches that support the concepts defined in this paper. As such, our formal definition can guide researchers and practitioners to apply and implement these concepts. We will exemplify every definition of our approach with two languages: (1) A UML statechart language defining states and transitions between states based on guards; (2) a language defining final states. Each of these languages consists of a grammar defining concrete and abstract syntax, well-formedness rules, and a code generator to realize translational semantics. The goal of our example is to embed the final state language into the statechart language by composing its respective language components. The resulting language then enables modelers to define statecharts with final states.

In the remainder, Sect. 2 introduces the method for systematic component-oriented language reuse. Section 3.1 presents the running example that is leveraged to exemplify the formal definitions that follow. Then Sect. 3 presents the formalization of our method. Section 4 maps our formal approach to the language workbench MontiCore and presents a case study of a language family for OPC UA. Section 5 discusses current limitations of the approach. Section 6 gives an overview of related work in the field. Finally, Sect. 7 concludes.

2 Background

2.1 Software language engineering

Software language engineering (SLE) [20–22] investigates the efficient and reliable engineering, maintenance, deployment, use, and evolution of DSLs to support software engineers and domain experts in efficiently developing software. Although approaches toward a systematic engineering of DSLs exist, most DSLs are still engineered ad hoc, for very specific challenges, and very limited purposes only [23]. Thus, research produced manifold solutions to support the creation of DSLs:

- Abstract syntax can be defined in terms of metamodels [11, 24, 25] or grammars [26–30].
- Concrete syntax can be defined with grammars [26, 29], projections [31], or editors [11].
- Well-formedness rules can be defined using metalanguages [32] or programming languages [33].
- Semantics can be realized through interpreters [11, 34, 35] or model transformations [19, 36, 37]

Furthermore, language workbenches and tools that facilitate the engineering of DSLs have emerged [38]. Popular examples are XText [28], MPS [39], MontiCore [33], Spoofax [32], Neverlang [40], and Melange [41]. Based on the definition artifacts mentioned before, these language workbenches support different notations of DSLs [38], i.e., textual, graphical, or tabular, and different editor functionalities for editing, and syntactic and semantic services. Our definition of DSLs is provided in Sect. 2.4.

2.2 Language components

Language components [17, 18] may provide the three language definition constituents (1) grammar-based syntax, (2) well-formedness rules checking predicates over the abstract syntax tree of the language, and (3) code generators that produce executable code from a well-formed abstract syntax tree (AST) realizing the semantic mapping between the problem and the solution domain. Figure 3 shows an example of a language component for UML Statecharts.

2.2.1 The component interface

In the component interface, we differentiate between required and provided extensions. Provided extensions offer DSL functionality to be reused by other components, whereas required extensions make the missing functionality of a DSL component explicit. In the interface, we differentiate between provided and required extensions.

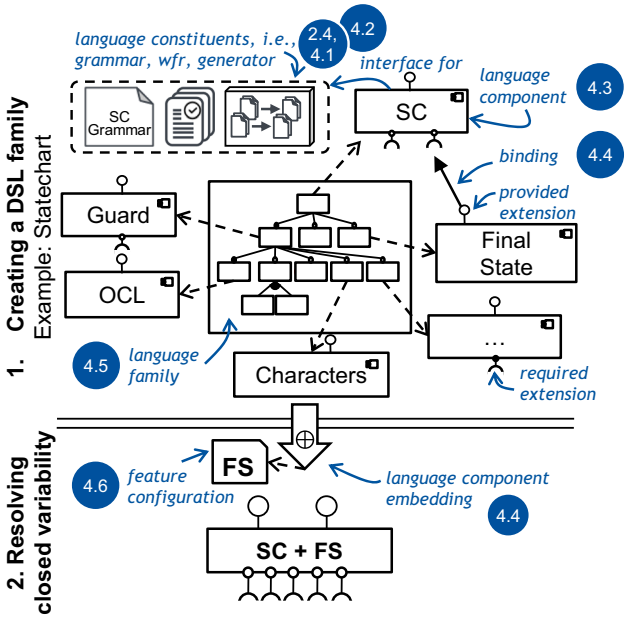

Fig. 1 Systematic language composition process is exemplified with a language family for statecharts. Numbers refer to the subsections that explain the associated concepts

Provided extensions: Provide concepts of a language for reuse.

- Provided grammar extensions reference a production in the grammar that can be reused by other components' grammars. For instance, the production `SC` in the grammar of Fig. 3.
- Provided generator extensions provide a transformation for a specific production of the grammar, with producer and artifact interfaces [19]. The producer interface defines how the generator can be called, and the artifact interface defines how the produced artifacts by the generator can be used. This will be important for the generator composition in the course of this paper. For instance, in Fig. 3, the producer interface is `ISCGen`, and the artifact interface is `IStatechart`.

Required extensions: Make the need for extension by other components provided interface explicit:

- Required extensions for grammars reference productions of a grammar that require extension, e.g., the production `Guard` requires definition.
- Required generator extensions demand an extension for a generator.
- We further differentiate whether they are optional or mandatory. When they are mandatory, they define an underspecification in the underlying language constituents, e.g., one production rule of a grammar does not have a right-hand side, and, thus, no parser can be derived from this grammar. In our approach, to derive a usable language definition, i.e., without underspecification, a component must not have mandatory required extensions.

A special case in the interface are well-formedness rules that are contained in sets that can be provided to other components for reuse, but also be extended by merging sets from other language components into them at the same time.

2.2.2 Composition of DSL components

The composition of two DSL components is the directed application of bindings between these components' interfaces, resulting in a novel language component. For the composition of two language components, the provided extensions of one component are bound to the required extensions of another component. The composition includes two main activities: (1) composing the components' interfaces and (2) composition of the comprised language definition constituents. The composition results in a new language component. Furthermore, whenever a mandatorily required extension is composed, it becomes optional afterward to indicate that it is now possible to derive a complete language definition, i.e., without necessity for extension, from the DSL component.

2.3 Language families

For their systematic reuse, in our approach, language components are arranged in a feature model representing a family of DSLs (cf. Figure 1). In this feature model, each feature either is related to a language component or is an abstract feature [42] for logical grouping. When a feature is associated with a language component, it defines a set of bindings between its language component and the next non-abstract parent feature's language component. This relation guides how the components will be composed when their related features are selected. By selecting features from the family in a feature configuration, the language components are composed according to the defined bindings, and language variants tailored to user requirements can be derived.

2.4 Definition of language constituents

In this paper, we assume that languages are defined through (1) A grammar-based syntax, (2) Well-formedness rules that check predicates over the AST of the language, (3) And generators that produce executable code from a well-formed AST. Since formal definitions of grammars and

well-formedness rules are established in the software engineering literature, we placed them in the background section to have a clear separation between existing work and the contribution of this paper.

Notation 1 *In the following definitions and examples, we denote named elements or sets with the superscript and elements or sets of a specific kind with the subscript.*
Example:

- G^{SC} *is the grammar with the name SC.*
- AST_G *is the AST of a grammar* G.
- *This notation can also be mixed into, e.g.,* $\text{PE}^{\text{SC}}_{gr}$*, which means that this is a set of provided grammar (*gr*) extensions (*PE*) of DSL component SC.*

2.4.1 Grammars

The definition of modeling languages in our concept relies on the grammar definition in the form of a context-free grammar, where possible special properties depend on the language workbench with which they are defined. In this formal view of the concept, we restrict the grammar definition to context-free grammars without any special properties.

Definition 1 (Context-Free Grammar) A *context-free grammar* [43] is defined as the tuple $G = (N, T, R, S)$, where

- N is a finite set of nonterminal symbols,
- T a finite set of terminal symbols,
- $R \subseteq N \times (N \cup T)^*$ a finite set of production rules, i.e., derivation rules, and
- $S \in N$ is the starting nonterminal.

A production rule $(X, \alpha) \in R$ is also denoted as $X \to \alpha$ with $\alpha \in (N \cup T)^*$. If there are multiple production rules for a single nonterminal $X \to \alpha_1$, $X \to \alpha_2$, $X \to \alpha_3$, then these rules might be denoted as a single rule $X \to \alpha_1 \mid \alpha_2 \mid \alpha_3$.

The left-derivation relation $\Rightarrow$ is defined as

$$\begin{aligned}\alpha \Rightarrow \beta \text{ iff. } &\alpha = wX\delta \in (N \cup T)^*,\\ &\beta = wa_1 \dots a_k\delta \in (N \cup T)^*,\\ &w \in T^*, \delta \in (N \cup T)^*, \text{ and } X \to a_1 \dots a_k.\end{aligned}$$

We write $\alpha \Rightarrow^* \gamma$ for $\alpha \in (N \cup T)^*$ and $\gamma \in T^*$, if there is a sequence of derivations

$$X \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_k \Rightarrow \gamma \text{ for } \alpha_i \in (N \cup T)^*.$$

The set of models recognized by the context-free grammar G is defined as

$$L(G) := \{w \in T^* \mid S \Rightarrow^* w\}.$$

To have the capability of using names as identifiers in grammar, we assume that the Name nonterminal and the terminal symbols for the letters of the alphabet are available in all grammars. Name can derive instances of the regular expression $[a - zA - Z]+$, corresponding to elements of $\{a, \dots, z, A, \dots, Z\}^+$. This enables shorter grammar definitions in the examples without a repeating definition of the production and its referenced elements. We use quotation marks in the definition of production rules to mark terminal symbols, e.g., *"state"*.

Example 1 The context-free grammar $G^{\text{SC}} = (N^{\text{SC}}, T^{\text{SC}}, R^{\text{SC}}, \text{SC})$ defines statecharts, where $N^{\text{SC}} = \{\text{SC}, \text{Elem}, \text{State}, \text{Trans}, \texttt{Guard}, \text{Name}\}$, $T^{\text{SC}} = \{\textit{"state"}, \textit{";"}, \textit{"}\to\textit{"}\}$, and R^{SC} is defined by the following productions:

SC $\to$ Elem | Elem *";"* Elem

Elem $\to$ State | Trans

State $\to$ *"state"* Name

Trans $\to$ Name `Guard` *"*$\to$*"* Name

Statecharts consist of states and transitions. In this example the guard element of the transitions is missing a production rule and is thus underspecified.

We can derive the model *state A* using the following left-derivation sequence:

SC $\Rightarrow$ Elem $\Rightarrow$ State $\Rightarrow$ *state* Name $\Rightarrow$ *state A*

It follows that `state A` $\in L(G^{\text{SC}})$. Since the `Guard` nonterminal has no production rule, it may not be used in the derivation of a model in the language of G^{SC}. Therefore, to allow transitions in models of the grammar, a production rule for `Guard` is required.

To correctly compose language components, which we define later, we need to be able to determine reachable nonterminals from a certain nonterminal. We formalize this notion with the following definition.

Definition 2 ((Directly) Reachable Nonterminals) The set of *directly reachable nonterminals* for a grammar $G = (N, T, R, S)$ from a nonterminal $x \in N$ is defined as

$$D_G(x) = \bigcup_{x \to \gamma \in R} \left\{ N_1, \dots, N_k \;\middle|\; \begin{array}{l} \gamma = w_0 N_1 w_1 \dots N_k w_k, \\ \forall i \in [0, k] : \; w_i \in T^*, \\ \forall j \in [1, k] : N_j \in N \end{array} \right\}.$$

The set $D^*_G(x) \subseteq N$ is the reflexive transitive closure of reachable nonterminals for nonterminal $x \in N$ in the grammar G and is the smallest set, such that:

If $x' \in D_G(x)$, then $x' \in D_G^*(x)$.
If $x' \in D_G^*(x)$ and $y \in D_G^*(x')$, then $y \in D_G^*(x)$.

In the example grammar G^{SC}, the `State` nonterminal is reachable from `SC`, but not from `Trans`. We have $D_G(\text{State}) = \text{Name}$, $D_G(\text{SC}) = \{\text{Elem}\}$ and $D_G^*(\text{SC}) = N_{\text{SC}}$.

2.4.2 Abstract syntax tree

Well-formedness rules (WFRs) and generators rely on the structure used for the derivation of the models, that is, the AST. The AST captures the left-derivation process of the models of the grammar.

To define the AST, we first introduce the notion of labeled trees that are leveraged to define partial abstract syntax trees (PASTs). The latter is also relevant to the definition of WFRs and generators that may only refer to parts of the complete AST. We then extend PASTs to the definition of the AST.

Definition 3 (Labeled Tree) A *labeled tree* [44] is a tuple (M, C, r, l, L) where M is a finite set of nodes in the tree, $C \subseteq M \times M$ is the directed child relation, $l : M \to L$ the labeling function and $r \in M$ the root node of the AST. The set L defines the labels of the tree.

The children relation has to be cycle-free, and for all non-root nodes $m \in M \backslash \{r\}$ there has to be a single other node $m' \in M, m' \neq m$ such that $(m', m) \in C$. This means that each node, except the root node, has to have a parent node in the tree, which ensures that it is only possible to define one tree at a time. Furthermore, if $(m, m') \in C$, then there is no m'' such that $(m'', m') \in C$. This condition guarantees that the defined structure is indeed a tree, where each node has at most one parent node, and not a directed, acyclic graph.

Example 2 An example for a labeled tree is given as follows:

$$\text{LT}_1 = (\{0, 1, 2, 3\}, \{(0, 1), (0, 2), (2, 3)\}, 0, \{l(0) = A, l(1) = B, l(2) = C, l(3) = A\}).$$

We say that node `1`, labeled with `B`, is a child of node `0`, which is labeled with `A`. We display the labeling function as the second element in the tuple. In some cases where the name of the nodes is irrelevant, we only show the labels of the nodes in the graphical tree representation.

We can compare labeled trees by identifying whether a labeled tree is a subtree of another labeled tree.

Definition 4 (Subtree of a Labeled Tree) A labeled tree (M, C, r, l) is the *subtree* of another labeled tree (M', C', r', l'), if $M \subseteq M', C \subseteq C'$, and $l = l'|_M$. The map $l'|_M : M' \to L$ is the restriction of l' to the nodes in M. This means that $l'|_M(m) := l'(m)$ for all $m \in M$. We denote the subtree relation as $(M, C, r, l) \sqsubseteq (M', C', r', l')$.

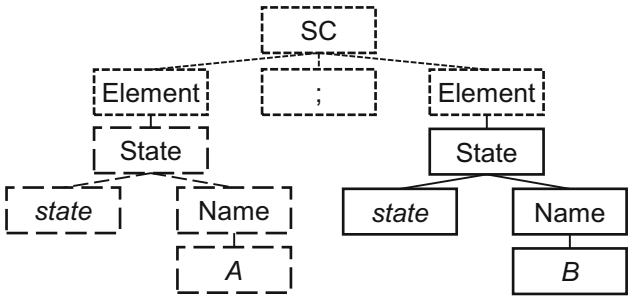

Fig. 2 Abstract syntax tree AST_{Ex} of the model *state A; state B* $\in L(G^{\text{SC}})$. The subtree highlighted with long dashes shows a partial abstract syntax tree, as well as the subtree indicated by short dashes

Example 3 The labeled tree $\text{LT}_2 = (\{2, 3\}, \{(2, 3)\}, 2, \{l(2) = C, l(3) = A\})$ is a subtree of the labeled tree that is given in Example 2. In other words, $\text{LT}_2 \sqsubseteq \text{LT}_1$.

An AST is a labeled tree with certain properties. We will first define the notion of a partial AST, which we then extend with additional properties to obtain the definition for an AST. The definition of partial ASTs is necessary for the definition of WFRs and generators in what follows (2).

Definition 5 (Partial Abstract Syntax Tree) A *partial abstract syntax tree* (PAST) for a given grammar $G = (N, T, R, S)$ is a labeled tree $A = (M, C, r, l)$ with $l : M \to N \cup T$, with properties:

1. If there is a node $x \in M$ with $l(x) = K$ and the children of x, are $y_1, \ldots, y_n \in M$ with $l(y_i) = K_i \in N \cup T$, then there has to be a corresponding production rule $K \to K_1 \ldots K_n$.
2. For each parent node $x \in M$, i.e., $\{y \mid (x, y) \in C\} \neq \emptyset$ with $l(x) = K$, it holds that $K \in N$.

We define the *set of all PASTs of grammar* G as PAST_G and the *set of all PASTs of grammar* G *with root label* $A \in N$ as $\text{PAST}_G(A)$.

An *abstract syntax tree (AST)* for a model $m \in L(G)$ with $G = (N, T, R, S)$ is a PAST_G with the following additional properties:

3. For all leaf nodes $x \in M$, i.e., $\{y \mid (x, y) \in C\} = \emptyset$ with $l(x) = K$, it holds that $K \in T$.
4. The label of the root node r is the starting nonterminal, i.e., $l(r) = S$.

An AST is complete/non-partial if and only if there is a derivation of a model in the language equal to the ordered leafs of the tree. This AST enables to reconstruct the derivation. We define the set of all ASTs of grammar G as AST_G.

Remark 1 It holds that $\text{AST}_G \subseteq \text{PAST}_\text{G}(S)$.

Example 4 Figure 2 shows an example AST, called AST_Ex, for the model `state A; state B` $\in L(G^{\text{SC}})$ in the example grammar G^{SC}. The example hides the names of the nodes and only shows the labels since the names of the nodes are irrelevant in the notation and only required for the formal definition. The subtree indicated with long dashes highlights a partial abstract syntax tree, which derives the first state of the statechart. This subtree fulfills the property (3) of the AST definition, but it does not start at the starting nonterminal and therefore, does not have property (4). The subtree indicated with short dashes is also a PAST. It has property (4) in addition to the PAST properties. However, not all leaves of this PAST are labeled with terminal symbols. Therefore, property (3) is not given and it is therefore no AST.

2.4.3 Well-formedness rules

Well-formedness rules (WFRs) constrain the models of a modeling language. We define them formally as WFRs that are applied to partial ASTs at all nodes.

Definition 6 (Well-Formedness Rules) Given a grammar $G = (N, T, R, S)$, a WFR w is defined as

$$w : \text{PAST}_\text{G} \rightarrow \mathbb{B},$$

where $\mathbb{B}$ denotes the set of boolean values.

This means, that a WFR takes a partial AST as input and decides, whether the partial model is well-formed.

Definition 7 (Set of all WFRs*for a grammar)* We define the *set of all WFRs* for a given context-free grammar G as

$$\mathbb{B}^{\text{PAST}_\text{G}}$$

Definition 8 (Application of a Well-Formedness Rule) For the *application of a WFR wfr* to the instantiated AST m of a model with $m \in M \backslash \{r\}$ and r being the root node, we define the function *apply* as follows:

$$\begin{aligned}
&apply : \mathbb{B}^{\text{PAST}_\text{G}} \times \text{AST}_G \rightarrow \mathbb{B} \\
&apply(wfr, m) := \\
&\begin{cases}
\text{false}, & \text{if there is a } x \in \text{PAST}_\text{G} \text{ s.t.} \\
& x \sqsubseteq m \wedge \neg\exists x' \in \text{PAST}_\text{G} : (x \sqsubseteq x' \sqsubseteq m \wedge r^x = r^{x'}) \\
& \wedge wfr(x) = \text{false} \\
\text{true}, & \text{otherwise}
\end{cases}
\end{aligned}$$

In other words, the definition of the application function provides each rule with a maximal PAST of the specific grammar. That is, the complete subtree of the input AST starts at a certain node. If the rule is not fulfilled by one of the subtrees, then the complete AST is not well formed either. This behavior mimics a traversal of the AST [33], which applies the WFRs at each level, granting the rules access to all child nodes at each application.

Example 5 We define an example WFR for the example statechart grammar G^{SC}. The rule checks whether the number of states in the statechart exceeds two.
The rule $min2States : \text{PAST}_{G^{\text{SC}}} \times \mathcal{P}(I) \rightarrow \mathbb{B}$ is defined as

$$\begin{aligned}
&min2States(M, C, r, l) = \\
&\begin{cases}
\text{false}, & \text{if } l(r) := \text{SC} \wedge \neg\exists x_1, \ldots, x_j \in M : \\
& \forall 1 \leq i < j \leq 2 : (x_i \neq x_j \wedge l(x_i) = l(x_j) \\
& = \text{State}) \\
\text{true}, & \text{otherwise}
\end{cases}
\end{aligned}$$

The application $apply(min2States, \text{AST}_\text{Ex})$ of the rule *min2States* on the example AST_Ex from Figure 2 returns true. This is the case, as there is only one maximal PAST of AST_Ex that starts at `SC`, which is AST_Ex itself, and there are two nodes with label `State`.

3 Formalizing systematic component-oriented language reuse

This section formally defines the different parts of our concept for systematic component-oriented language reuse as depicted in Fig. 1. For a better understanding of the concepts, besides their formal definition, we leverage a running example based on the statechart language. For this, we first extend the definition of language constituents that are relevant to our definition of language components with generators realizing translational semantics. To make these language constituents reusable, we define their reuse by composition through embedding. The definition of language components then enables their reuse via a defined interface, where we

show the algorithm used to compose language components via bindings between interfaces of language components. To systematically compose language components, we define language families. Finally, the configuration is defined, and the traversal algorithm used when applying such configurations is presented.

3.1 Running example

We introduce a running example that is used to exemplify the formalized concepts. Figure 3 shows two language components `SC` and `FS`. The component `SC` has two provided extensions, where the grammar provided extension refers to a production `SC` in the grammar of the component, and the provided generator extension with the name `sc2J` refers to the provided generator interface `ISCGen` and the provided artifact interface `IStatechart`. Both interfaces provide a signature (see Definition 10. The generator also provides two required interfaces `StateJGen` and `StateArt` that are exposed via the required generator extension `s2J`. The component `FS` comprises two provided extensions `fs` and `fs2J`. These provided extensions are bound to the required extensions of the component `SC` via the feature declaration in the language family. When the features `SCBase` and `Final State` are selected, these two components are composed according to these bindings (see Algorithm 2. A detailed version of the component `SC` in the technological space of MontiCore is provided in Fig. 5. Details about the realization of the component `FS` and also of the generator interfaces can be investigated in the accompanying repository of this paper .[1]

3.2 Extended definition of language constituents: generators

We define a generator as a transformation from partial ASTs to a set of target models in a target grammar J. Implementations of generator interfaces for the partial AST of one nonterminal of the grammar expose their interface via signatures for their producer interface, i.e., the interface to call when using the generator, and the artifact interface, i.e., the interface that the generated artifacts must conform to [19]. For the embedding of new generators, these new generators need to adapt to these interfaces to be compatible and be registered as the embedding generator.

Definition 9 (Transformation) Given a grammar $G = (N, T, R, S)$, a *transformation* starting at the AST of the nonterminal $x \in N$ to a set of models M in a target language L(J) (see Definition 1) is defined as a function

$$t : \mathrm{PAST_G}(x) \rightarrow M \subseteq L(J).$$

Definition 10 (Signature) The set of signatures is defined as $Sig = Name \times PAR$, where $Name$ is the universe of names, and $PAR = Name \times UTYPE$ where $UTYPE$, as defined in UML [45], is the universe of type names with a carrier set CAR of values from universe $UVAL$ associated with each type: $CAR : Type \rightarrow \mathcal{P}(UVAL)$. Signatures that realize a transformation t are defined as Sig_t.

Example 6 Referring to our running example, an example of a signature can be given based on the generator interface class `ISCGen` that defines its signature as:

$$sig^{ISCGen} = \{("transSC", \{AST\,SC\ ast\})\}$$

Definition 11 (Generator interface) The set of generator interfaces GI over a grammar $G = (N, T, R, S)$ is defined as:

$$\mathrm{GI} = \bigcup_{x \in N} (Sig_t \times Sig \times \mathrm{PAST_G}(x)),$$

where we identify $p \in Sig_t$ as the generator's *producer interface* and $a \in Sig$, as its *artifact interface*.

Definition 12 (Generator) A *generator* over a grammar G = (N, T, R, S) is based on our previous work [19] and is defined as Gen = (PGI, RGI, GR, AR) with

- $PGI \subseteq$ GI is set of provided generator interfaces
- RGI $= \mathcal{P}$(GI) the set of required generator interfaces.
- GR $= \mathcal{P}$(GenReg) is the set of all registered adapted generators for a ProdI $\in$ RGI with GenReg = ProdI $\times$ Adapt where Adapt $\subseteq$ AD (see Definition 13)
- AR $= \mathcal{P}$(ArtReg) is the set of all registered adapted artifact interfaces for a ArtI $\in$ RGI with ArtReg = ArtI $\times$ Adapt where Adapt $\subseteq$ AD (see Definition 13)

Definition 13 (Generator adapter) For embedding generators into another generator, we need to adapt the provided generator interface, i.e., its producer and artifact interface, of one generator to a required generator interface of another generator. Therefore, we define a generator adapter as a function $adapt$ that is an element of a set of functions AD where

$$\begin{aligned} AD = \{adapt_{ij} \mid adapt_{ij} : \mathrm{ProdI}_i \rightarrow \mathrm{ProdI}_j \text{ where} \\ \mathrm{ProdI}_i \in \mathrm{pgi}_i \text{ of a generator } \mathrm{Gen}_i \\ \text{and } \mathrm{ProdI}_j \in \mathrm{rgi}_j \text{ of a generator } \mathrm{Gen}_j\} \end{aligned}$$

The artifact interface adaption works identically.

[1] https://doi.org/10.5281/zenodo.19553862

Example 7 The generator Gen^{SC} can be defined as:

$\text{Gen}^{SC} = (\text{PGI}^{SC}, \text{RGI}^{SC}, \text{GR}^{SC}, \text{AR}^{SC})$,

with

- $\text{PGI}^{SC} = \{\text{pgi}^{SC2J}\}$ that is the set of provided generator interfaces,
- $\text{pgi}^{SC2J} = \{(\text{ProdI}^{ISCGen}, \text{ArtI}^{IStatechart}, \text{AST}^{SC})\}$ that is the provided generator interface translating the nonterminal `SC` to Java and referencing the producer and artifact interface and the AST class of the nonterminal `SC`,
- $\text{ProdI}^{ISCGen} = (\text{"}transSC\text{"}, (\text{AST}^{SC})) \in \text{pgi}^{SC2J}$ that is the signature of the producer interface with the name of the function, the parameters and the transformation it provides,
- $\text{ArtI}^{IStatechart} = (\text{"}exec\text{"}, \{\}) \in \text{pgi}^{SC2J}$ that is the artifact interface,
- $\text{RGI}^{SC} = \{\text{rgi}^{G2J}, \text{rgi}^{S2J}\}$ that is the set required extensions for guards and states,
- $\text{rgi}^{G2J} = (\text{ProdI}^{Guard2JGen}, \text{ArtI}^{IGuardArt}, \text{AST}^{Guard})$ that is stating the producer and artifact interface and the AST class of the nonterminal `Guard` that they can process,
- $\text{ProdI}^{Guard2JGen} = (\text{"}translate\text{"}, \text{AST}^{Guard}) \in \text{rgi}^{G2J}$
- $\text{ArtI}^{IGuardArt} = (\text{"}check\text{"}, \{\}) \in \text{rgi}^{G2J}$,
- $\text{rgi}^{S2J} = (\text{ProdI}^{IStateJGen}, \text{ArtI}^{IStateArt}, \text{AST}^{State})$ that is stating the producer and artifact interface and the AST class of the nonterminal `State` that they can process,
- $\text{ProdI}^{IStateJGen} = (\text{"}generate\text{"}, \text{AST}^{State}) \in \text{rgi}^{S2J}$ that state the method and the parameters of the producer interface `StateJGen`
- $\text{ArtI}^{IStateArt} = (\text{"}step\text{"}, \{\}) \in \text{rgi}^{S2J}$ that state the method and the parameters of the artifact interface `StateArt`,
- $\text{GR}^{SC} = \emptyset$, $\text{AR}^{SC} = \emptyset$ are the set of registered adapters for the generator required extensions. Since no generator has been embedded they are initially empty.

3.3 Composing language constituents through embedding

Language constituents can be reused via composition. In this paper, we focus on the language composition operator embedding [46]. The operator is directed and takes one language definition that is embedded into an embedding language definition, together with two productions. Models that conform to the new language can then use rules of the embedded grammar production in place of the selected embedding production in the same model. For instance, one could embed statecharts into method bodies of class diagram methods to describe their behavior.

In the following, we present the parameters and effects of applying this composition operator to the language constituents grammar and generators. We do not include well-formedness rules because they are defined based on a PAST. The PAST, when embedded into the PAST of another language, remains unchanged such that the well-formedness rules remain compatible, and, thus, has no implications on the application of WFRs.

3.3.1 Grammars

We define the composition of grammars by embedding as the addition of a new alternative in the production rule of a nonterminal. We give a formal definition of this embedding procedure using the notion of the slice of a grammar at a certain nonterminal.

Definition 14 (Slice of a Context-Free Grammar) We define the slice of a context-free grammar $G = (N, T, R, S)$ at a nonterminal $x \in N$ as

$$
\begin{aligned}
&sl(G, x) = (N', T', R', x), \text{ where}\\
&\quad N' = D^*_G(x),\\
&\quad T' = \bigcup_{x \to \gamma \in R} \left\{ t_1, \ldots, t_k \;\middle|\; \begin{array}{l} \gamma = t_0 N_1 t_1 \ldots N_k t_k,\\ \forall i \in [0,k] : t_i \in T^*,\\ \forall j \in [1,k] : N_j \in N \end{array} \right\}\\
&\quad R' = R \setminus \{y \to \gamma \in R \mid y \in N \setminus N'\}
\end{aligned}
$$

This means that a slice of a grammar is a part of the source grammar, which only contains the rules of the nonterminal at which the grammar was sliced, as well as all transitively reachable nonterminals and their production rules. We translate the concept of grammar embedding to the formalization using the slice of grammars as follows:

Definition 15 (Grammar Embedding) Given the grammars $G^1 = (N^1, T^1, R^1, S^1)$ and $G^2 = (N^2, T^2, R^2, S^2)$, we define the embedding of $G^1 = (N^1, T^1, R^1, S^1)$ of nonterminal $x \in N_1$ into $G^2 = (N^2, T^2, R^2, S^2)$ at nonterminal $y \in N^2$ as

$$
\begin{aligned}
G^1 \rhd_{x \to y} G^2 := (&N^2 \cup N^3, T^2 \cup T^3, R^2 \cup R^3 \cup \{y \to x\}, S^2),\\
&\text{where } (N^3, T^3, R^3, x) = sl(G^1, x).
\end{aligned}
$$

This means we only add the reachable part of grammar G^1 starting at $x \in N^1$ to G^2. We also add a new connecting transition to the resulting grammar while retaining the old starting nonterminal of G^2.

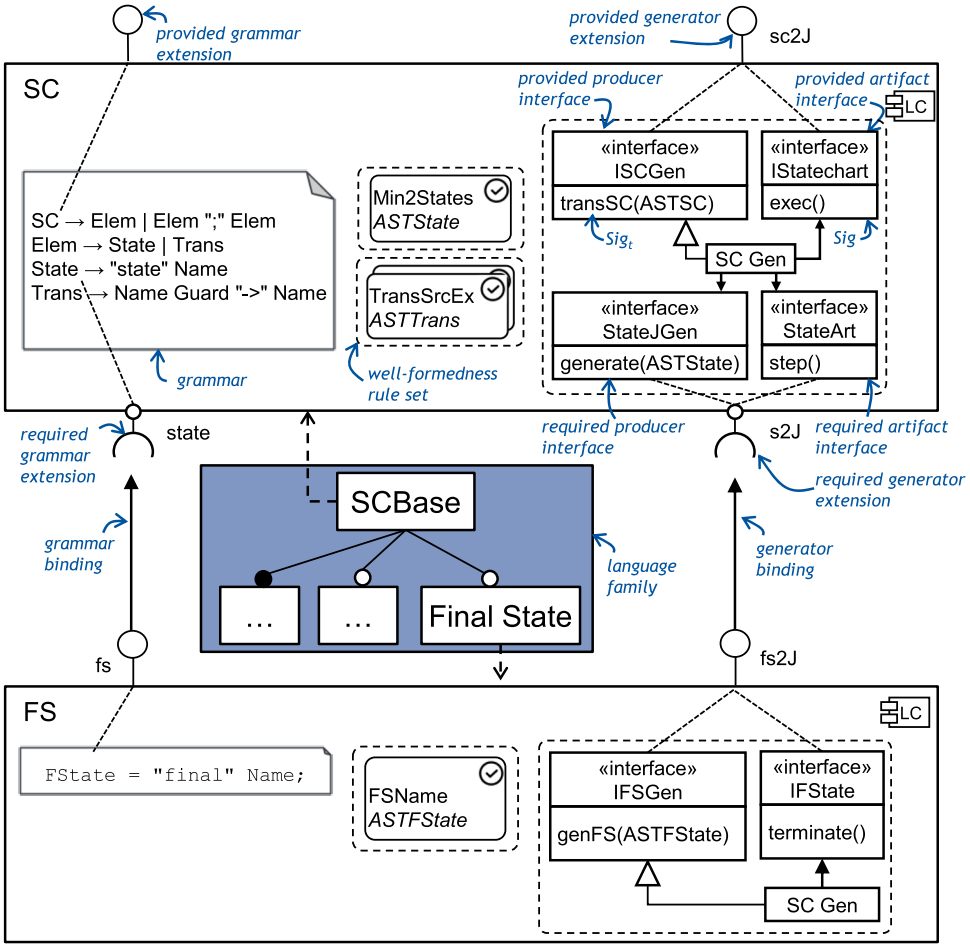

Fig. 3 Running example for the formalization of the method for component-oriented language reuse. We reuse two language components SC and FS that are composed via the selection of the associated features in the language family

We assume for the embedding that either $G^1 = G^2$, i.e., the embedding grammar is the same as the embedded grammar, or it holds that $N^1 \cap N^2 = \emptyset$. This restriction will guarantee that a sequence of embedding different grammars with common nonterminals at different target nonterminals do not interfere with one another. For example, let G^1 be defined by the production rule $S \rightarrow_{R^1} A\ B$, G^2 by $X \rightarrow_{R^2} x$, and G^3 by $X \rightarrow_{R^3} y$. Then, the result of the embedding $G^2 \rhd_{X \rightarrow A} G^1 = G^4$ is defined by the production rules on the top and the result of $G^5 = G^3 \rhd_{X \rightarrow B} G^4$ on the bottom:

$$S \rightarrow_{R^4} A\ B \qquad A \rightarrow_{R^4} X \qquad X \rightarrow_{R^4} x$$
$$S \rightarrow_{R^5} A\ B \qquad A \rightarrow_{R^5} X \qquad B \rightarrow_{R^5} X$$
$$X \rightarrow_{R^5} x \mid y$$

While the embedding produces a valid context-free grammar, it does not produce the intended semantics. After embedding G^3, one can derive the terminal symbol y from the nonterminal A. This contradicts the intended derivations, where the embedding of G^2 in G^1 should only allow the derivation of x from A. The same occurs for the nonterminal B of G^5, which can derive x, even though it should only be able to derive y.

Therefore, we implicitly assume that if the input grammar of the grammar embedding operator is not the same, then the set of nonterminals of the grammars is disjoint. To mitigate this problem and the according restrictions, we also implicitly assume the renaming of the nonterminals of embedded grammar, which already occur in the embedding grammar. This renaming also applies to any WFRs and generator functions defined on the embedded grammar, which use these nonterminals. With the renaming, we obtain the intended results from the compositions. Embedding G^3 into G^4, we first rename the nonterminal X of G^3 to X'. Then, we get the following productions defining the composed grammar G^5:

$$S \rightarrow_{R^5} A\ B \qquad A \rightarrow_{R^5} X \qquad B \rightarrow_{R^5} X'$$
$$X \rightarrow_{R^5} x \qquad X' \rightarrow_{R^5} y$$

Now it is not possible to derive the terminal symbol y from nonterminal A, or x from B.

Example 8 We define the grammar $G^{\text{FS}} = (N^{\text{FS}}, T^{\text{FS}}, R^{\text{FS}}, \text{FState})$, where $N^{\text{FS}} = \{\text{FState}\}$, $T^{\text{FS}} = \{\textit{"final state"}\}$ and $R^{\text{FS}} = \{\text{FState} \rightarrow \textit{"final state"}\ \text{Name}\}$.

The result of embedding G^{FS} into G^{SC} where FState is added to the State rule is given as follows:

$$G^{\text{FS}} \rhd_{\text{FState}\rightarrow\text{State}} G^{\text{SC}}$$
$$= (N^{\text{SC}} \cup \{\text{FState}\}, T^{\text{SC}} \cup \{\textit{"final state"}\}, R^{\text{SC}}$$
$$\cup R_{\text{FS}} \cup \{\text{State} \rightarrow \text{FState}\}, \text{SC}) = G^{\text{SC}'}$$

3.3.2 Generators

We define the composition of two generators as a directed embedding of one generator's pgi into another generator's rgi.

Definition 16 (Generator embedding) For two generators

$$Gen_1 = (\text{PGI}_1, \text{RGI}_1, \text{GR}_1, \text{AR}_1), \text{ and}$$
$$Gen_2 = (\text{PGI}_2, \text{RGI}_2, \text{GR}_2, \text{AR}_2)$$

we define the embedding of the generator Gen_2 into Gen_1, noted as $Gen_2 \rhd_{\text{pgi}_1\rightarrow gep_2} Gen_1$, as follows

$$emb(\text{Gen}_1, \text{Gen}_2, \text{pgi}_1, gep_2)$$
$$= \{\text{PGI}_1, \text{RGI}_1 \cup \text{RGI}_2, \text{GR}_1 \cup adapt_{\text{pgi}_{1.\text{ProdI}}\rightarrow gep_{2.\text{ProdI}}}$$
$$\cup\, \text{GR}_2, \text{AR}_1 \cup adapt_{\text{pgi}_{1.\text{ArtI}}\rightarrow gep_{2.\text{ArtI}}} \cup \text{AR}_2\}$$

This means that the provided generator interfaces of the embedded generator are maintained, and the required generator interfaces of the embedded generator are added to the set of required generator interfaces of the embedding generator to retain further extensibility. For the embedding generator to be able to call the embedded generator, the generator and artifact interface adapters between the composed generators need to be registered and added together with the existing adapters.

Example 9 To illustrate the embedding of generators, we define a generator Gen^{FS} for the grammar G^{FS} (see Example 8). We embed this generator into the generator Gen^{SC} (see Example 7) to enable the generation of code for the concept of final states from the generator Gen^{SC}. We define the generator Gen^{FS} with:

$$\text{Gen}^{FS} = (\text{PGI}^{FS}, \emptyset, \emptyset, \emptyset),$$

with

- $\text{PGI}^{FS} = \{\text{pgi}^{FS2J}\}$,
- $\text{pgi}^{FS2J} = \{\text{ProdI}^{IFSGen}, \text{ArtI}^{IFState}, \text{AST}^{FState}\} \in \text{PGI}^{FS}$
- $\text{ProdI}^{IFSGen} = (\text{"}genFS\text{"}, (\text{AST}^{FState}))$,
- $\text{ArtI}^{IFState} = (\text{"}terminate\text{"}, \{\})$

We perform the embedding $\text{Gen}^{FS} \rhd_{\text{pgi}^{FS2J}\rightarrow\text{rgi}^{S2J}} \text{Gen}^{SC}$ resulting in the GR^{SC} and AR^{SC} as follows:

$$\text{GR}^{SC} = \{\text{ProdI}^{IStateJGen}, (\text{ProdI}^{IFSGen} \rightarrow \text{ProdI}^{IStateJGen})\}$$
$$\text{AR}^{SC} = \{\text{ArtI}^{IStateArt}, (\text{ProdI}^{IFState} \rightarrow \text{ArtI}^{IStateArt})\}$$

3.4 Language components

We present a formal representation of language components using the previously introduced constituents of modeling languages. As in the concept, there is a division between the artifact and component interface levels.

Definition 17 (Language Component Interface) The interface of a language component is defined as

$$\text{CI} = (\text{PE}_{\text{gr}}, \text{RE}_{\text{gr}}, \text{WFR}_{\text{sets}}, \text{PE}_{\text{gen}}, \text{RE}_{\text{gen}}, impl_{gr}, impl_{gen}, o_{gr}, o_{gen}),$$

where

- $\text{PE}_{\text{gr}} \subseteq N$ is the set of grammar provided extensions,
- $\text{RE}_{\text{gr}} \subseteq N$ is the set of grammar required extensions,
- $\text{WFR}_{\text{sets}} \subseteq Name \times \mathcal{P}(Name \times \mathbb{B}^{\text{PAST}_G})$ is the set of all WFR sets (see Definition 6 in the language component's interface. Each WFR set has a name and references multiple WFRs by name.
- $\text{PE}_{\text{gen}} \subseteq (Name \times \text{PGI})$ is the set of generator provided extensions (see Definition 11 in the language component's interface referenced by their name,
- $\text{RE}_{\text{gen}} \subseteq (Name \times \text{RGI})$ the set of generator required extensions (see Definition 11 in the language component's interface referenced by their name,
- $impl_{gr} : \text{PE}_{\text{gr}} \to \mathcal{P}(\text{RE}_{\text{gr}})$ with $impl_{gr}(x) := \{y \in \text{RE}_{\text{gr}} \mid y \in D^*_G(x)\}$ is the implication relationship (see Definition 19) between the grammar's provided extensions and the grammar's required extensions, which will become relevant when composing language components,
- $impl_{gen} : \text{PE}_{\text{gen}} \to \mathcal{P}(\text{RE}_{\text{gen}})$ with $impl_{gen}(x) := \{(y, t) \in \text{RE}_{\text{gen}} \mid y \in D^*_G(x)\}$ is the implication relationship between the generator's provided extensions and their implied generator required extensions,
- $o_{gr} : \text{RE}_{\text{gr}} \to \{opt, mand\}$ indicates the optionality of grammar required extensions, and
- $o_{gen} : \text{RE}_{\text{gen}} \to \{opt, mand\}$ indicates the optionality of generator required extensions.

The optionality of required extensions depends on whether the referenced nonterminal has a production associated with it or not, and, hence, can be set to optional after it has received rules after execution of the embedding operator (see Algorithm 1).

Definition 18 (Language Component) A language component is defined as

$$C = (G, \text{WFR}, \text{Gen}, \text{CI}),$$

where

- $G = (N, T, R, S)$ is the context-free grammar,
- $\text{WFR} \subseteq \mathbb{B}^{\text{PAST}_G}$ is a set of WFRs,
- Gen is a generator for well-formed models of $L(G)$, and
- CI is the language component's interface.

Example 10 The *Statechart* language component is defined as

$$\text{SC} = (G^{\text{SC}}, \text{WFR}^{\text{SC}}, \text{Gen}^{\text{SC}}, \text{CI}^{\text{SC}}),$$

where G^{SC} is the grammar defined in Example 1. The component contains the WFR $min2States_n$, introduced in Example 5. Furthermore, there are two rules which ensure that the source and target states of transitions exist. These are *transSrcEx* and *transTgtEx*, respectively. Thus, the set of well-formedness rules is defined as $\text{WFR}^{\text{SC}} = \{min2States, transSrcEx, transTgtEx\}$. The component defines the generator Gen^{SC} (see Example 7). The component interface for `SC` is defined as

$$\text{CI}^{\text{SC}} = (\text{PE}^{\text{SC}}_{\text{gr}}, \text{RE}^{\text{SC}}_{\text{gr}}, \text{WFR}^{\text{SC}}_{\text{sets}}, \text{PE}^{\text{SC}}_{\text{gen}}, \text{RE}^{\text{SC}}_{\text{gen}}, impl^{\text{SC}}_{gr}, impl^{\text{SC}}_{gen}, o^{\text{SC}}_{gr}, o^{\text{SC}}_{gen}),$$

where

- $\text{PE}^{\text{SC}}_{\text{gr}} = \{\text{SC}\}$, $\text{RE}^{\text{SC}}_{\text{gr}} = \{\texttt{Guard}, \text{State}\}$,
- $\text{WFR}^{\text{SC}}_{\text{sets}} = \{W^{\text{SC}}_{\text{state}}, W^{\text{SC}}_{\text{trans}}\}$,
- $\text{PE}^{\text{SC}}_{\text{gen}} = \{\text{SC2J}\}$, $\text{RE}^{\text{SC}}_{\text{gen}} = \{\text{G2J}, \text{S2J}\}$,
- $o^{\text{SC}}_{gr} = \{(\text{State}, opt), (\texttt{Guard}, mand)\}$, and
- $o^{\text{SC}}_{gen} = \{(\text{S2J}, opt), (\text{G2J}, mand)\}$,
- $W^{\text{SC}}_{\text{state}} = \{min2States_n\}$ and
- $W^{\text{SC}}_{\text{trans}} = \{transSrcEx, transTgtEx\}$.

Since `State` and `Guard` are reachable from the production rule `SC` (cf. Example 1), and `SC` together with the generator provided extension `SC2J` both reference `SC`, we get the following implications in the interface:

$$impl^{\text{SC}}_{gr} = \{(\text{SC}, \{\text{State}, \texttt{Guard}\})\} \text{ and}$$
$$impl^{\text{SC}}_{gen} = \{(\text{SC2J}, \{\text{S2J}, \text{G2J}\})\}.$$

Example 11 The *Final State* component is defined as

$\text{FS} = (G^{\text{FS}}, \text{WFR}^{\text{FS}}, \text{Gen}^{\text{FS}}, \text{CI}^{\text{FS}})$, where

- G^{FS} defined in Example 8,
- $\text{WFR}^{\text{FS}} = \{fsName\}$ is the set of WFRs,
- $fsName : \text{PAST}^{G^{\text{FS}}} \rightarrow \mathbb{B}$ checks that the names of the final states are defined uppercase, and
- Gen^{FS} as defined in Example 9.

The language component interface CI^{FS} has no required extensions is defined as

$$\text{CI}^{\text{FS}} = ((\text{PE}_{\text{gr}}^{FS}, \text{RE}_{\text{gr}}^{FS}, \text{WFR}_{\text{sets}}^{FS}, \text{PE}_{\text{gen}}^{FS}, \text{RE}_{\text{gen}}^{FS}, impl_{gr}^{FS}, impl_{gen}^{FS}, o_{gr}^{FS}, o_{gen}^{FS})),$$

where

- $\text{PE}_{\text{gr}}^{\text{FS}} = \{\text{FState}\}$,
- $\text{RE}_{\text{gr}}^{FS} = \emptyset$,
- $\text{WFR}_{\text{sets}}^{\text{FS}} = \{W^{\text{FS}}\}$ with $W^{\text{FS}} = \{fsName\}$,
- $\text{PE}_{\text{gen}}^{\text{FS}} = \{\text{FS2J}\}$ with $\text{FS2J} = \text{pgi}^{\text{FS2J}} \in \text{PGI}^{FS}$
- $\text{RE}_{\text{gen}}^{FS} = \emptyset$,
- $impl_{gr}^{FS} = \emptyset$,
- $impl_{gen}^{FS} = \emptyset$,
- $o_{gr}^{FS} = \emptyset$, and
- $o_{gen}^{FS} = \emptyset$

To select which extensions are relevant for the composed component that is the result of the language component composition, we have to determine the provided extensions that imply a certain required extension.

Definition 19 (Implying Provided Extension) For a language component $C = (G, \text{WFR}, \text{Gen}, \text{CI})$ with the $\text{CI} = (\text{PE}_{\text{gr}}, \text{RE}_{\text{gr}}, W, \text{PE}_{\text{gen}}, \text{RE}_{\text{gen}}, \ldots)$, the set of implying provided extensions is defined as:

$$\begin{aligned} implying_{gr}(e_x) &:= \{p \in \text{PE}_{\text{gr}} \mid e \in impl_{gr}(p)\}, \\ implying_{gen}(e_y) &:= \{p \in \text{PE}_{\text{gen}} \mid e \in impl_{gen}(p)\}, \\ &\text{for } e_x \in \text{RE}_{\text{gr}} \text{ and } e_y \in \text{RE}_{\text{gen}} \end{aligned}$$

This means that for the required extension of a certain type, the $implying_{gr}$ function determines the set of provided extensions of the same type that have the referenced nonterminals in its set of reachable nonterminals. In other words, $implying_{gr}$ is the inverse image of $impl_{gr}$ and reflects a dependency between the provided and required extensions of one component.

Example 12 Let $impl_{gr} = \{(X, Y), (Z, \{Y, A\})\}$.
Then, $implying_{gr}(Y) = \{X, Z\}$ and $implying_{gr}(A) = \{Z\}$.

Definition 20 For a set of language components CMP, we define

$$\begin{aligned} \text{PE}_{\text{y}}^{CMP} &:= \bigcup_{x \in CMP} \text{PE}_{\text{y}}^{x}, \\ \text{RE}_{\text{y}}^{CMP} &:= \bigcup_{x \in CMP} \text{RE}_{\text{y}}^{x} \text{ for } y \in \{gr, gen\}, \text{ and} \\ \text{WFR}_{\text{sets}}^{CMP} &:= \bigcup_{x \in CMP} \text{WFR}_{\text{sets}}^{x} \end{aligned}$$

3.5 Language component composition

In this section, we introduce an algorithm for the composition of language components for a given set of bindings. This algorithm is based on our previous work for systematic language reuse [17, 18] that showcase this algorithm with practical examples and give a walk through in the technological space of MontiCore.

Definition 21 (Adjustment) We define for a function $f : A \rightarrow B$ with $(a, b) \in f$, adjustment as

$$f[x \mapsto z] := (f \backslash \{(x, y)\}) \cup \{(x, z)\}$$

This is used, e.g., When composing language components where the optionality of mandatory required extensions is adjusted to optional in their component interface.

Definition 22 (Bindings) To compose two language components, we define bindings $\text{B} = (B_{gr}, B_{gen}, B_{wfr}^{mer}, B_{wfr}^{sel})$ between the provided extensions of an embedded language component C^1 and required extensions of an embedding language component C^2 as follows:

- $B_{gr} \subseteq \text{PE}_{\text{gr}}^{1} \times \text{RE}_{\text{gr}}^{2}$ that defines the binding from one grammar provided extension to one grammar required extension,
- $B_{gen} \subseteq \text{PE}_{\text{gen}}^{1} \times \text{RE}_{\text{gen}}^{2}$ that defines the binding from one generator provided extension to one generator provided extension,
- $B_{wfr}^{mer} \subseteq \text{WFR}_{\text{sets}}^{1} \times \text{WFR}_{\text{sets}}^{2}$ that defines that one $\text{WFR}_{\text{sets}}^{1}$ is merged into another set $\text{WFR}_{\text{sets}}^{2}$ in the composed component (similar to a union of two sets), and
- $B_{wfr}^{sel} \subseteq \text{WFR}_{\text{sets}}^{1}$ that defines that one set of a language component is selected to be included in the composed component.

Algorithm 1: `compose` Function – Language Component Composition

Input:
Embedded language component C^1 and embedding language component C^2, where $C^i = (G^i, \text{WFR}^i, \text{Gen}^i, \text{CI}^i)$ and $\text{CI}^i = (\text{PE}^i_{\text{gr}}, \text{RE}^i_{\text{gr}}, \text{WFR}^i_{\text{sets}}, \text{PE}^i_{\text{gen}}, \text{RE}^i_{\text{gen}}, \text{Params}^i, impl^i_{gr}, impl^i_{gen}, o^i_{gr}, o^i_{gen})$.
Bindings $B = (B_{gr}, B_{gen}, B^{mer}_{wfr}, B^{sel}_{wfr})$
Result: The composed language component

```
// Initialize resulting component with C^2
1  C^res ← C^2        // Includes G^res ← G^2, CI^res ← CI^2, ...

// Handle grammar bindings
2  foreach grammar binding (X, Y) ∈ B_gr do
       // Compose the grammars (cf. Definition 15)
3      G^res ← G^1 ▷_{X→Y} G^res
       // Add the implied (cf. Def. 19) grammar required extensions of X to C^res
4      RE^res_gr ← RE^res_gr ∪ impl^1_gr(X)
       // Determines the provided extensions of C^2, which imply Y and store them in I
5      I ← implying^2_gr(Y)
       // Add the implications from I and X to the result
6      impl^res_gr ← impl^res_gr[I' ↦ impl^res_gr(I') ∪ impl^1_gr(X) | I' ∈ I]
       // Adjust optionality of bound required extension Y (cf. Definition 21)
7      o^res_gr ← o^res_gr[Y ↦ opt]

// Handle WFR set bindings
8  foreach well-formedness rule set binding (X, Y) ∈ B^mer_wfr do
9      WFR^res_sets ← (WFR^res_sets \{Y}) ∪ {Y ∪ X}
10     WFR^res ← WFR^res ∪X
11 foreach well-formedness rule set selection X ∈ B^sel_wfr do
12     WFR^res_sets ← WFR^res_sets ∪{X}
13     WFR^res ← WFR^res ∪X

// Handle generator bindings
14 foreach generator binding (X, Y) ∈ B_gen do
       // Compose the generators (cf. Def. 16)
15     Gen_res ← Gen_2 ▷_{X→Y} Gen_1
       // Add the new EPs
16     RE^res_gen ← RE^res_gen ∪ impl^1_gen(X)
       // Add the new implications (cf. Def. 19)
17     I ← implying^2_gen(Y)
18     impl^res_gen ← impl^res_gen[I' ↦ impl^res_gen(I') ∪ impl^1_gen(X) | I' ∈ I]
       // Adjust optionality of target EP (cf. Definition 21)
19     o^res_gen ← o^res_gen[Y ↦ opt]

20 return C^res
```

The composition algorithm is defined in Algorithm 1. In the following algorithms, $X \leftarrow Y$ denotes the assignment of Y to the variable X.

It first initializes the variables for the resulting language component C^{res} with the embedding component C^2, creating a working copy of all elements (l. 1).

Then, it iterates over all grammar bindings in B_{gr} (ll. 2–7). For each binding, the algorithm uses the grammar embedding (see Definition 15), composing the grammar of the source component with the current result grammar along the rules in the current binding (l. 3). It then adds the grammar required extensions $impl^1_{gr}(X)$ of the embedded component C^1, implied by the bound provided extension X, to the resulting set $\text{RE}^{\text{res}}_{\text{gr}}$ of the composed component (l. 4). Next, it determines the provided extensions of C^2, which imply the required extension Y referenced in the binding, and saves them as I (l. 5). It adds new implications from all provided extensions in I to the added extensions $impl^1_{gr}(X)$ (l. 6). As the last step for each binding, the optionality $o^{\text{res}}_{gr}(Y)$ of the bound required extension Y changes to optional (l. 7).

The handling of the WFR bindings for merging (ll. 8–10) and selection (ll. 11–13) of rule sets follows the same structure, iterating over the given bindings. For merging, the set Y is first removed from the resulting component and afterward the merged set of both X of the embedded component and Y is added to the composed component again (l. 9). In the case of selection, the set is simply added as a new set to $\text{WFR}^{\text{res}}_{\text{sets}}$ in the interface of the resulting component C^{res} (l. 12). Then, the WFRs are added to the WFRs of C^{res} (ll. 10 and 13).

The algorithm iterates over the generator bindings in B_{gen} and adapts the resulting component accordingly. (ll. 14–19).

For each generator binding, it composes the generators using the generator operator defined in Definition 16 (l. 15).

The remaining steps are analogous to the handling of the grammar bindings.

Finally, the algorithm returns the modified language component C^{res} (l. 20).

Example 13 In this example we are instantiating the algorithm from Algorithm 1. In our example, the component FS (see Example 11) is considered to be C^1 and SC (see Example 10) to be C^2.

Both components are composed according to the following bindings:

$B_{gr} = \{(\text{FState}, \text{State})\}$, $B_{gen} = \{(\text{FS2J}, \text{S2J})\}$,
$B^{mer}_{wfr} = \{(W^{\text{FS}}, W^{\text{SC}}_{\text{state}})\}$, $B^{sel}_{wfr} = \emptyset$

- **Line 1:** The component C^{res} is initialized with SC.
- **Line 2:** The algorithm iterates over the list ob grammar bindings B_{gr} and performs the following step for the the binding (FState, State)
  - **Line 3:** We obtain the resulting grammar $G^{\text{res}} = G^{\text{SC}'}$, as seen in Example 8 which results in the addition of a new production rule for the State nonterminal which delegates to the nonterminal FState.
  - **Line 4:** Since the grammar provided extension FState does not imply any grammar required extensions, i.e., $impl(\text{FState}) = \emptyset$, the composition adds no new grammar required extensions and implications.
  - **Line 5:** Since no grammar required extensions are implied by `FState`, the set of existing implications in SC can remain unchanged.
  - **Line 6:** The set of implications in C^{res} remains unchanged from its initialization based on SC.
  - **Line 7:** As the required grammar extension State is already optional, the optionality of the required extension does not change.
- **Line 8:** for the B^{mer}_{wfr} $(W^{\text{FS}}, W^{\text{SC}}_{\text{state}})$ do the following:
  - **Line 9:** removes the target set $W^{\text{SC}}_{\text{state}}$ and then adds the merged set to $\text{WFR}^{\text{res}}_{\text{sets}}$, resulting in $\text{WFR}^{\text{res}}_{\text{sets}} = \{W^{\text{SC}}_{\text{state}}, W^{\text{SC}}_{\text{trans}}\}$, where $W^{\text{SC}}_{\text{state}} = \{min2States_n, fsName\}$
  - **Line 10:** The rules of the sets are added to the constituent set of the resulting component:

$$\begin{aligned}\text{WFR}^{\text{res}} &= \{min2States_n, transSrcEx, transTgtEx\} \cup W^{\text{FS}} \\ &= \{min2States_n, transSrcEx, transTgtEx, fsName\}.\end{aligned}$$

- **Line 11-13:** Since $B^{sel}_{wfr} = \emptyset$ this part of the algorithm is not executed.
- **Line 14:** The algorithm iterates over the B_{gen} with the first and only binding (FS2J, S2J) and performs the following steps:
  - **Line 15:** Use the generator composition operator (cf. Definition 16) to compose the generator provided extension FS2J into the generator extension of the SC component specified by S2J. This results in the generator Gen^{res}.
  - **Line 16-18:** Similar to the case of the grammar bindings, there are no generator required extensions in the FS component. Therefore, it adds no new required extensions and implications to the resulting language component.
  - **Line 19:** As the required generator extension S2J is already optional, the optionality of the required extension does not change
- **Line 20:** The algorithm creates the resulting language component C^{res} and returns it.

3.6 Language families

Our conceptual model of a language family comprises a feature diagram, the mapping to the realizing language components, the root configuration, and the bindings between the realizing components of features. Consequently, we first define feature diagrams.

Definition 23 (Feature Diagram) A feature diagram [47] is defined as

$$\text{FD} = (\text{F}, A, r, \lambda, DE, CE),$$

where

- F is the set of features,
- $A \subseteq F$ is the set of abstract features,
- $r \in F$ the distinguished root feature,
- $\lambda : F \rightarrow FT$ labels each feature with an operator from FT = {and, xor, or, opt, mand},
- $DE \subseteq F \times F$ is the set of decomposition edges; $(f, f') \in DE$ will rather be noted $f \rightarrow f'$,
- $CE \subseteq F \times CT \times F$ is the set of constraint edges with CT = {requires, excludes},

and the following well-formedness conditions hold:

- Only r has no parent: $\forall f \in F(\nexists f' \rightarrow f) \iff f = r$.
- DE is acyclic: $\nexists f_1, ..., f_k \in F, f_1 \rightarrow ... \rightarrow f_k \rightarrow f_1$.
- DE is a tree: $\nexists f_1, f_2, f_3 \in F, f_1 \rightarrow f_2 \wedge f_3 \rightarrow f_2 \wedge f_1 \neq f_3$.

When applying a language family configuration, we traverse the feature tree and, thus, need to determine the children of a feature. Therefore, we introduce the children mapping, which determines the children of a feature, independent of their relationship.

Definition 24 Given a feature diagram FD as defined in Definition 23, we define the children mapping as follows:

$$\text{children}(f) := \{c \in \text{F} | (f, c) \in DE\}$$

With the definition of feature diagrams, we can now define the formal representation of language families.

Definition 25 (Language Family) Is defined as

$$\text{LF} = (\text{FD}, \text{CP}, \text{real}, \text{RC}, (B_{gr}, B_{gen}, B^{mer}_{wfr}, B^{sel}_{wfr})), \text{where}$$

- $\text{FD} = (\text{F}, A, r, \lambda, DE, CE)$ is the feature diagram of the language family.
- CP is a non-empty set of realizing language components (cf. Definition 18) for the features,
- $\text{real} : \text{F} \rightarrow \text{CP}$ defines the realizing language components of the features in the feature diagram,
- RC is the root configuration,
- $B_{gr} \subseteq \text{F} \times \text{PE}^{\text{CP}}_{\text{gr}} \times \text{F} \times \text{RE}^{\text{CP}}_{\text{gr}}$ are the grammar bindings (cf. Definition 22),
- $B_{gen} \subseteq \text{F} \times \text{PE}^{\text{CP}}_{\text{gen}} \times \text{F} \times \text{RE}^{\text{CP}}_{\text{gen}}$ are the generator bindings (cf. Definition 22),
- $B^{mer}_{wfr} \subseteq \text{F} \times \text{WFR}^{\text{CP}}_{\text{sets}} \times \text{F} \times \text{WFR}^{\text{CP}}_{\text{sets}}$ are the WFR bindings (cf. Definition 22), and
- $B^{sel}_{wfr} \subseteq \text{F} \times \text{WFR}^{\text{CP}}_{\text{sets}}$ are the selected WFR sets of the components.

As abstract features only perform a grouping function, we do not consider them in this section, and thus, the real mapping is not a partial function.

The root configuration, i.e., the selection of WFR sets and provided extensions of the root component used to restrict the provided functionality of the starting component, is defined as the triple

$$\text{RC} = (\text{RC}_{gr}, \text{RC}_{gen}, \text{RC}_{wfr}),$$

where $\text{RC}_{gr} \subseteq \text{PE}_{\text{gr}}$ is the selection of grammar provided extensions, $\text{RC}_{gen} \subseteq \text{PE}_{\text{gen}}$ the selection of generator provided extensions, and $\text{RC}_{wfr} \subseteq \text{WFR}_{\text{sets}}$ the selection of WFR sets. PE_{gr}, PE_{gen}, and WFR_{sets} are the set of provided extensions and WFR sets of the interface of the root component $\text{real}(r)$.

The four sets defining grammar, generator, and WFR set bindings, as well as the WFR set selections have certain restrictions. For all grammar bindings $(s, p, t, e) \in B_{gr}$, it has to hold that $p \in \text{PE}^{C}_{\text{gr}}$, where $C = \text{real}(s)$, and $e \in \text{RE}^{C'}_{\text{gr}}$, where $C' = \text{real}(t)$. Furthermore, we require $s \in \text{children}(t)$ for all $(s, p, t, e) \in B_{gr}$. For all generator bindings $(s, p, t, e) \in B_{gen}$, it has to hold that $p \in \text{PE}^{C}_{\text{gen}}$ where $C = \text{real}(s)$ and $e \in \text{RE}^{C'}_{\text{gen}}$ where $C' = \text{real}(t)$. Once again, we require that $s \in \text{children}(t)$ for all $(s, p, t, e) \in B_{gen}$. For all WFR bindings $(s, w, t, w') \in B^{mer}_{wfr}$, it has to hold that $w \in \text{WFR}^{C}_{\text{sets}}$ where $C = \text{real}(s)$ and $w' \in \text{WFR}^{C'}_{\text{sets}}$ where $C' = \text{real}(t)$. As seen above, for all $(s, w, t, w') \in B^{mer}_{wfr}$ the source feature s has to be a child of the target feature t, i.e., $s \in \text{children}(t)$.

The conditions for the bindings ensure that they may only be defined between provided extensions of the realizing component of a child feature to the required extensions of the realizing component of its parent feature (4).

Example 14 We give the formal representation of the reduced language family from the running example with features Statecharts, FinalStates, and the optional Characters feature. Figure 4 shows the feature diagram and the bindings. The `FinalStates` and `Characters` features are both optional subfeatures of the root feature `Statecharts`.

Therefore, the formal definition of the feature diagram in the language family is given as

$$\begin{aligned}\mathrm{FD}^{\mathrm{Ex}} = (&\{\text{Statecharts, FinalStates, Characters}\}, \emptyset, \{\text{Statecharts}\}\\ &\{\text{FinalStates} \rightarrow \mathit{opt}, \text{Characters} \rightarrow \mathit{opt}),\\ &\{\text{Statecharts} \rightarrow \text{FinalStates, Statecharts} \rightarrow \text{Characters}\}, \emptyset).\end{aligned}$$

For the example feature model, the children function returns the following values:

$$\begin{aligned}&\text{children(Statecharts)} = \{\text{FinalStates, Characters}\} \text{ and}\\ &\text{children(FinalStates)} = \text{children(Characters)} = \emptyset.\end{aligned}$$

The set of language components in the language family is defined as $\mathrm{CP}^{\mathrm{Ex}} = \{\mathrm{SC}, \mathrm{FS}, \mathrm{CG}\}$, with $\mathrm{real}^{\mathrm{Ex}}(\text{Statecharts}) = \mathrm{SC}$, $\mathrm{real}^{\mathrm{Ex}}(\text{FinalStates}) = \mathrm{FS}$, and $\mathrm{real}^{\mathrm{Ex}}(\text{Characters}) = \mathrm{CG}$. The language components for Statecharts and FinalStates are defined in Example 10 and Example 11, respectively.

The component realizing the Characters feature is defined as

$$\mathrm{CG} = (G^{\mathrm{CG}}, \mathrm{WFR}^{\mathrm{CG}}, \mathrm{Gen}^{\mathrm{CG}}, \mathrm{CI}^{\mathrm{CG}}),$$

where $G^{\mathrm{CG}} = (\{\text{Char}\}, \{a, \ldots, z\}, R^{\mathrm{CG}}, \text{Char})$ and $R^{\mathrm{CG}} =$ $\{\text{Char} \rightarrow a \mid b \mid \ldots \mid z\}$. The component contains no WFRs, i.e., $\mathrm{WFR}^{\mathrm{CG}} = \emptyset$. Furthermore, the set of generators is defined as $\mathrm{Gen}^{\mathrm{CG}} = \{\mathrm{PGI}^{CG}, \emptyset, \emptyset, \emptyset\}$ with $\mathrm{PGI}^{CG} = \{\mathrm{pgi}^{CG2J}\}$

The language component interface of CG is defined as

$$\mathrm{CG} = (\mathrm{PE}^{\mathrm{CG}}_{\mathrm{gr}}, \emptyset, \emptyset, \mathrm{PE}^{\mathrm{CG}}_{\mathrm{gen}}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset),$$

where $\mathrm{PE}^{\mathrm{CG}}_{\mathrm{gr}} = \{\text{Char}\}$, $\mathrm{PE}^{\mathrm{CG}}_{\mathrm{gen}} = \{\text{CG2J}\}$.

We define the bindings of the language family as

$$\begin{aligned}B^{\mathrm{Ex}} = (&\{(\text{FinalStates, FState, Statecharts, State}),\\ &\quad(\text{Characters, Char, Statecharts, } \texttt{Guard})\},\\ &\{(\text{FinalStates, FS2J, Statecharts, S2J}),\\ &\quad(\text{Characters, CG2J, Statecharts, G2J})\},\\ &\{(\text{FinalStates}, W^{\mathrm{FS}}, \text{Statecharts}, W^{\mathrm{SC}}_{\mathrm{state}})\}, \emptyset).\end{aligned}$$

For the root configuration, we want to provide all elements of the SC component. Thus, the root configuration has exactly the interface elements of SC:

$$\begin{aligned}\mathrm{RC}^{\mathrm{Ex}} &= (\{\mathrm{SC}\}, \{\mathrm{SC2J}\}, \{W^{\mathrm{SC}}_{\mathrm{state}}, W^{\mathrm{SC}}_{\mathrm{trans}}\})\\ &= (\mathrm{PE}^{\mathrm{SC}}_{\mathrm{gr}}, \mathrm{PE}^{\mathrm{SC}}_{\mathrm{gen}}, \mathrm{WFR}^{\mathrm{SC}}_{\mathrm{sets}})\end{aligned}$$

The complete example language family is defined as

$$\mathrm{LF}^{\mathrm{Ex}} = (\mathrm{FD}^{\mathrm{Ex}}, \mathrm{CP}^{\mathrm{Ex}}, \mathrm{real}^{\mathrm{Ex}}, \mathrm{RC}^{\mathrm{Ex}}, B^{\mathrm{Ex}}).$$

Algorithm 2: Language Family Configuration Application

Input:

$$\text{Fam} = (\text{FD}, \text{CP}, \text{real}, (\text{RC}_{gr}, \text{RC}_{gen}, \text{RC}_{wfr}), (B_{gr}, B_{gen}, B^{mer}_{wfr}, B^{sel}_{wfr}))$$

and the language family configuration $X \subseteq \text{F}$ for Fam

Result: The resulting language component that is the product of the application process

1 **Function** *traverse*:
  Input: Language family Fam, current feature f, selected feature set X
  Result: Language component for the selection of the partial feature tree starting at f
2 cp ← real(f)
3 **foreach** *child* $c \in$ children(f) *with* $c \in X$ **do**
4 $C^{\text{res}} \leftarrow$ `traverse`(Fam, c, X)
  `// Compose components (Algorithm 1)`
5 cp ← `compose`(C^{res}, *cp*, (B_{gr}, B_{gen}, B^{mer}_{wfr}, B^{sel}_{wfr}))
6 **if** $f \neq r$ **then**
  `// Propagate WFR set to root`
7 $B^{sel}_{wfr} \leftarrow B^{sel}_{wfr} \cup \{(f, w) \mid (c, w) \in B^{sel}_{wfr}\}$
8 **return** *cp*
9 rc ← real(r)
  `// Handle root configuration`
10 $\text{PE}^{\text{res}}_{\text{gr}} \leftarrow \text{RC}_{gr}$
11 $\text{PE}^{\text{res}}_{\text{gen}} \leftarrow \text{RC}_{gen}$
12 $\text{WFR}^{\text{res}}_{\text{sets}} \leftarrow \text{RC}_{wfr}$
  `// Remove rules from unselected sets`
13 $\text{WFR}^{\text{res}} \leftarrow (\text{WFR}^{\text{res}} \setminus \text{WFR}^{\text{rc}}) \cup (\bigcup_{X \in \text{RC}_{wfr}} X)$
14 **return** C^{res}

3.7 Language family configuration application

A configuration of a language family is a valid configuration of the encoded feature diagram. Thus, a language family configuration for a given language family LF with feature set F is a collection of features LFC $\subseteq$ F, which should fulfill the constraints of the tree relations (see [48] for details).

The formal specification of the algorithm is given in Algorithm 2. The application of a configuration to a language family is a bottom-up composition process applying the defined bindings between the associated language components. The `Traverse` function (ll. 1–8) recursively traverses the tree in a depth-first manner. For each feature, the function iterates over all selected child features (l. 3). It embeds the resulting component C^{res} of the subtree into the realizing component `cp` of the current feature using the `compose` function (see Algorithm 1). It saves the resulting, composed component C^{res} and continues with the next child feature. Once the traversal is complete, the algorithm applies the root configuration (ll. 10–13), where it removes the unselected provided extensions and WFR sets from the resulting language component's interface and constituent sets. After the composition, it modifies the WFR set selection bindings, if necessary. This adds the WFR set selections from the embedded component to the bindings of the current, embedding component's feature (l. 6), propagating the selected set to the realizing component of the root feature. Finally, it returns the C^{res} component with the applied root configuration as the result of the configuration application.

Example 15 We show the application of the language family configuration selecting all features, i.e.,

$$\text{LFC}^{\text{Ex}} = \{\text{Statecharts, FinalStates, Characters}\},$$

of the LF^{Ex} language family from Example 14.

The application of the configuration starts with the depth-first traversal of the feature tree using the `Traverse` function. It is called recursively for each selected child feature of the currently visited feature. The root feature Statecharts has two child features FinalStates and Characters, which are both selected.

The `Traverse` function call thus first calls itself for the FinalStates feature. It loads the realizing language component FS using the real mapping. The FinalStates has no child features.

Therefore, the unmodified component FS is returned to the `Traverse` function call for the Statecharts feature.

It then composes the FS component, which the `Travers` calls to FinalStates produced, with the current component, SC. For this, it uses the bindings defined between the features and the realizing components as input for the `compose` function.

The result of the composition is the language component SC′. It is temporarily saved as C^{res}.

The WFR set selection bindings do not have to be adapted to accommodate WFR selections since the current feature is already the root feature.

The next child feature of Statecharts is Characters, for which the `Traverse` call once again only returns the CG language component since the Characters has no child features. Then, the algorithm composes the CG component with the currently saved component SC′, according to these bindings. Resulting in the language component

$$\text{SC}'' = (G^{\text{SC}''}, \text{WFR}^{\text{SC}''}, \text{Gen}^{\text{SC}''}, \text{CI}^{\text{SC}''}), \text{ where}$$

$$\text{CI}_{\text{SC}''} = \begin{pmatrix} \text{PE}_{\text{gr}}^{\text{SC}''}, \text{RE}_{\text{gr}}^{\text{SC}''}, \text{WFR}_{\text{sets}}^{\text{SC}''}, \text{PE}_{\text{gen}}^{\text{SC}''}, \text{RE}_{\text{gen}}^{\text{SC}''}, \\ \text{Params}^{\text{SC}''}, \mathit{impl}_{gr}^{\text{SC}''}, \mathit{impl}_{gen}^{\text{SC}''}, o_{gr}^{\text{SC}''}, o_{gen}^{\text{SC}''} \end{pmatrix}.$$

Overall, the language component interface $\text{CI}_{\text{SC}''}$ is similar to the interface of the SC′ LC. They differ in the aspect, that the required extensions for the grammar and generator of guards are now optional instead of mandatory.

The traversal ends with the second composition and the algorithm saves the resulting component SC″. Then, it applies the root configuration, which removes all unselected provided extensions and WFR sets. The root configuration in the example selects all provided extensions and WFR sets. Therefore, the component remains unchanged and the algorithm returns the SC″ language component.

4 Evaluation

The evaluation of this approach is twofold. First, we evaluate how well the method can be adapted to the language workbench MontiCore and explain how the language component and language constituent composition are applied in this language workbench. In the second part of our evaluation, we present a language family for the language OPC UA to evaluate how well our method scales for larger language families and to demonstrate the effectiveness of the method compared to an iterative whitebox approach. The implementation of the framework together with its adaption to the language workbenches MontiCore and Xtext, and one tutorial is available in our replication package on GitHub[1].

4.1 Adaptability of the method

We adapted the presented method to the language workbenches MontiCore. In this section, we first demonstrate its adaptation to MontiCore and afterward present a summarizing table that presents the LoC required for the adaptation effort for MontiCore.

4.1.1 DSL components in MontiCore

Figure 5 shows how our formal approach can be mapped to the language workbench MontiCore. A DSL component interface for the language defines the provided and required extensions for the syntax (ll. 3–4), the generator (ll. 6–15), and the well-formedness rules (ll. 17–20). The component interface definition enables compatibility with language workbenches that support defining these constituents, e.g., Xtext, Neverlang, and MontiCore. To what extent our formalism applies to other possible language constituents like metamodels or languages with executable semantics is up to future research. Furthermore, we assume that for opportunistic reuse, code generators must be built as proposed in [19] and defined in Sect. 3.2. Consequently, generators need a defined provided and required interface of which each comprise a producer interface and artifact interface.

4.1.2 Grammar embedding

MontiCore grammars support the composition operator language embedding. For this, the grammar that contains the required extensions should define these productions with the keyword `interface` (cf. Fig. 6, grammar `GSC`, ll. 3). In this example, we define a binding from the provided grammar extension `FinalState` to the required grammar extension `State`. Applying these bindings results in a composed grammar that embeds the grammars `FinalState`, i.e., their bound productions in the grammar `GSC`. The composed grammar `GSCWithFinalStates` extends the

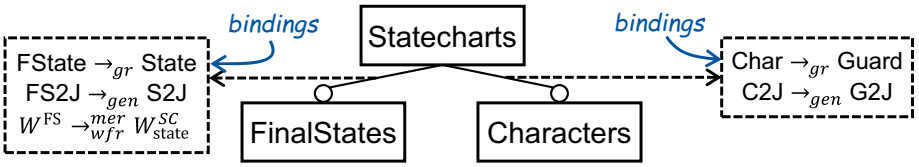

Fig. 4 Graphical representation of the example language family LF^{Ex} with feature diagram FD^{Ex} and the bindings B^{Ex}, defined in Example 14. The realizing components are not pictured

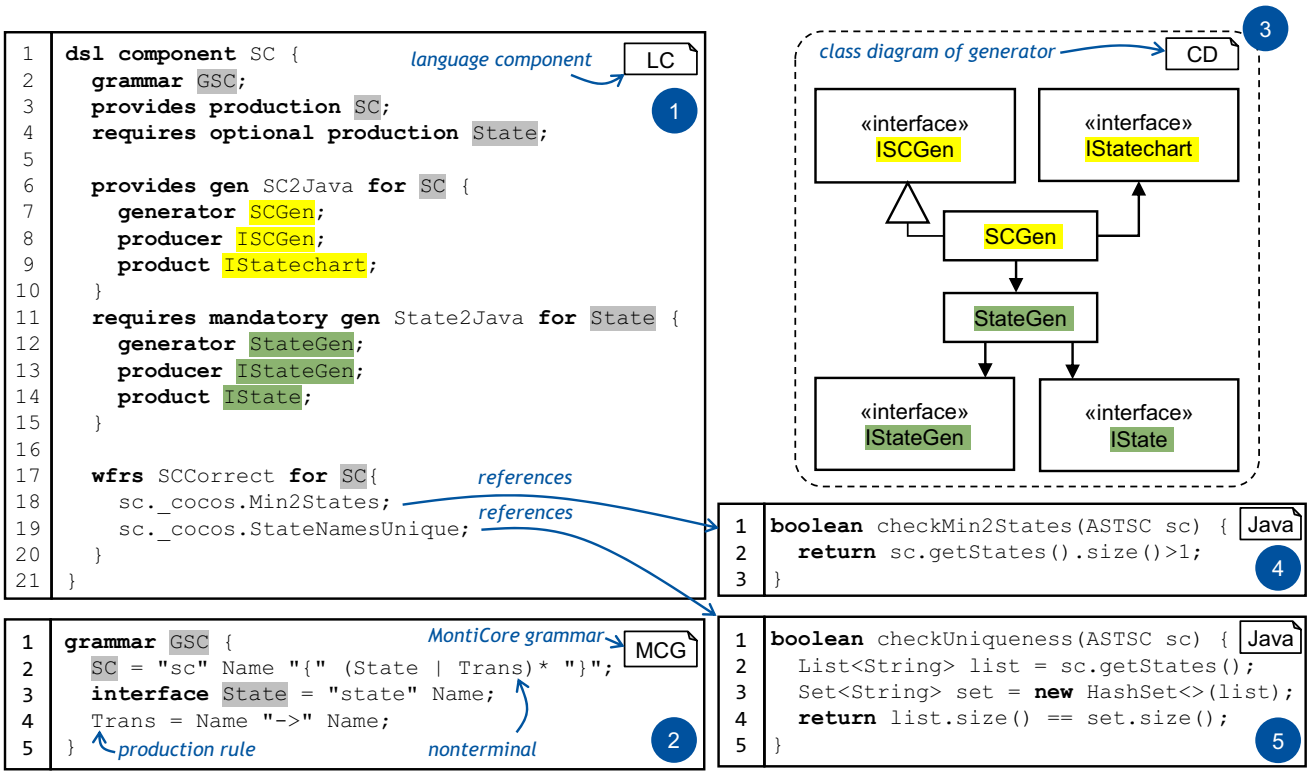

Fig. 5 Component SC as implemented in the technological space of MontiCore. The DSL component interface (1) references the language constituents grammar (2), generator classes including product and producer interfaces (3), and two well-formedness rules to check that at least two states are defined (4), and that the names of states are unique (5)

grammar FinalState (cf. GSCWithFinalState, l. 1 in Fig. 6). In MontiCore, this has the effect that all productions of the extended grammars are available. With this, the interfaces of the extended grammars can be implemented, and the productions can be extended. In the composed grammar, a new production is created for each binding. These productions each extend the embedded productions and implement the interfaces of the required extension (cf. GSCWithFinalState, l. 2 in Fig. 6).

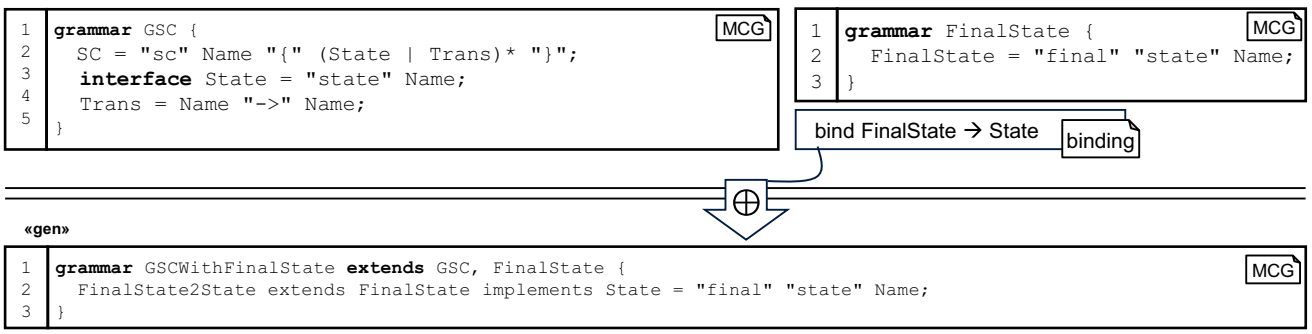

Fig. 6 Example of grammar embedding realized in MontiCore

4.1.3 Well-formedness rule composition

WFRs are called context conditions in MontiCore. In MontiCore, the context conditions of a language are created in a class called after the language to which they refer, followed by the suffix `CoCos`. Figure 7 shows the context conditions that are contained in the DSL components for the statechart and final state languages. The class `GSCCoCos` has a method to create a context condition checker for statechart models. Each checker method represents a WFR in the DSL component interface. In addition, the class `FinalStateCoCos` provides a checker for the final state language. In this example, a binding defines that the WFR set `CheckStateCardinality` should be merged into the set `SCCorrect`. Based on this binding, a novel context conditions class is generated with the name of the composed language `GSCWithFinalStatesCoCos`. Based on the selected binding, i.e., merging or adding, the WFR composition adds methods to the novel context condition class. In the case of merging, the name of the method that is the target of the binding remains in the composed context condition class. In this example, the method `createSCCorrectChecker()` is created in the composed class and calls the two methods of `GSCCoCos` and `FinalStateCoCos` that realize the WFR sets that are merged. In the second case, the method representing the added WFR set is simply added and calls the method of origin.

4.1.4 Code generator composition

The composition of two generators entails the adaptation between their generator and artifact interfaces. In Fig. 8, the composition of the generators `StateGen` and `FSGen` in MontiCore is depicted. The `StateGen` has two interfaces as required generator extensions. The artifact interface `IState` and the generator interface `IStateGen`. In addition, it has a method to register generators that comply with the interface and to return all registered generators. The generator `FSGen` uses the generator interface `IFSGen` and the artifact interface `IFState`. Based on the binding between both generators, a composed generator `StateWIthFSGenerator` and two adapter classes for artifact and generator interface, respectively, are generated. The composed generator calls the constructor of the extended generator `StateGen` and internally registers the adapted generator `IStateGen2IFSGen`. Thus, all registered adapted generators can be called at the runtime of the generator.

4.1.5 Adaptation effort in MontiCore vs. Xtext

Table 1 shows the result for the LoC required for the implementation of the different tasks that we presented before, i.e., grammar composition, WFR composition, and code generator composition. For grammar composition, the effort is similar. The WFR composition requires considerably more effort in MontiCore. This difference in WFR composition (381 #LoC in MontiCore vs. 53 #LoC in Xtext) is due to the inherent complexity of the respective WFR infrastructures. MontiCore's required effort reflects the complexity of instrumenting and composing WFRs, which are implemented as Java-based context conditions and integrate tightly with its generated visitor infrastructure. In contrast, Xtext's underlying framework simplifies the instrumentation of SCOLAR's general WFR concepts, e.g., rule sets, and their embedding by offering a more abstract checking mechanism, reducing the boilerplate required for the adaptation to our method. The LoC required for the generator composition is similar between both language workbenches. This highlights the stability of this abstraction, since the generator assumptions, i.e., explicit producer and artifact interfaces, enforce a highly structured approach that is relatively independent of the underlying template engine, i.e., FreeMarker for MontiCore or Xtend for Xtext. In conclusion, the effort for adapting the method remains manageable, with even the most complex integration requiring fewer than 1000 lines of glue code for each language workbench. This variation in lines of code mainly reflects the differences in API design and tooling maturity across the underlying technology spaces and does not indicate any weakness in the SCOLaR method's conceptual model or its composition workflow.

4.2 Case study: OPC UA language family

This section presents the in-depth case study of our approach with the DSL OPC UA (Open Platform Communications Unified Architecture). OPC UA is a cross-platform, open-source standard for industrial automation that enables secure and reliable data exchange between diverse devices and systems. It goes beyond simple data transport by using a sophisticated information model to provide rich semantic context, allowing machines to "understand" each other regardless of the manufacturer or operating system. Similar to UML class diagrams, the OPC UA information modeling language provides means to model different data types, variables, links between data types (references), and methods. Figure 9 depicts the OPC UA language family of this case study. For the modularization of the monolithic OPC UA language, we follow the following process:

1. We determined the fundamental OPC UA functionality that forms the core language for defining a namespace,

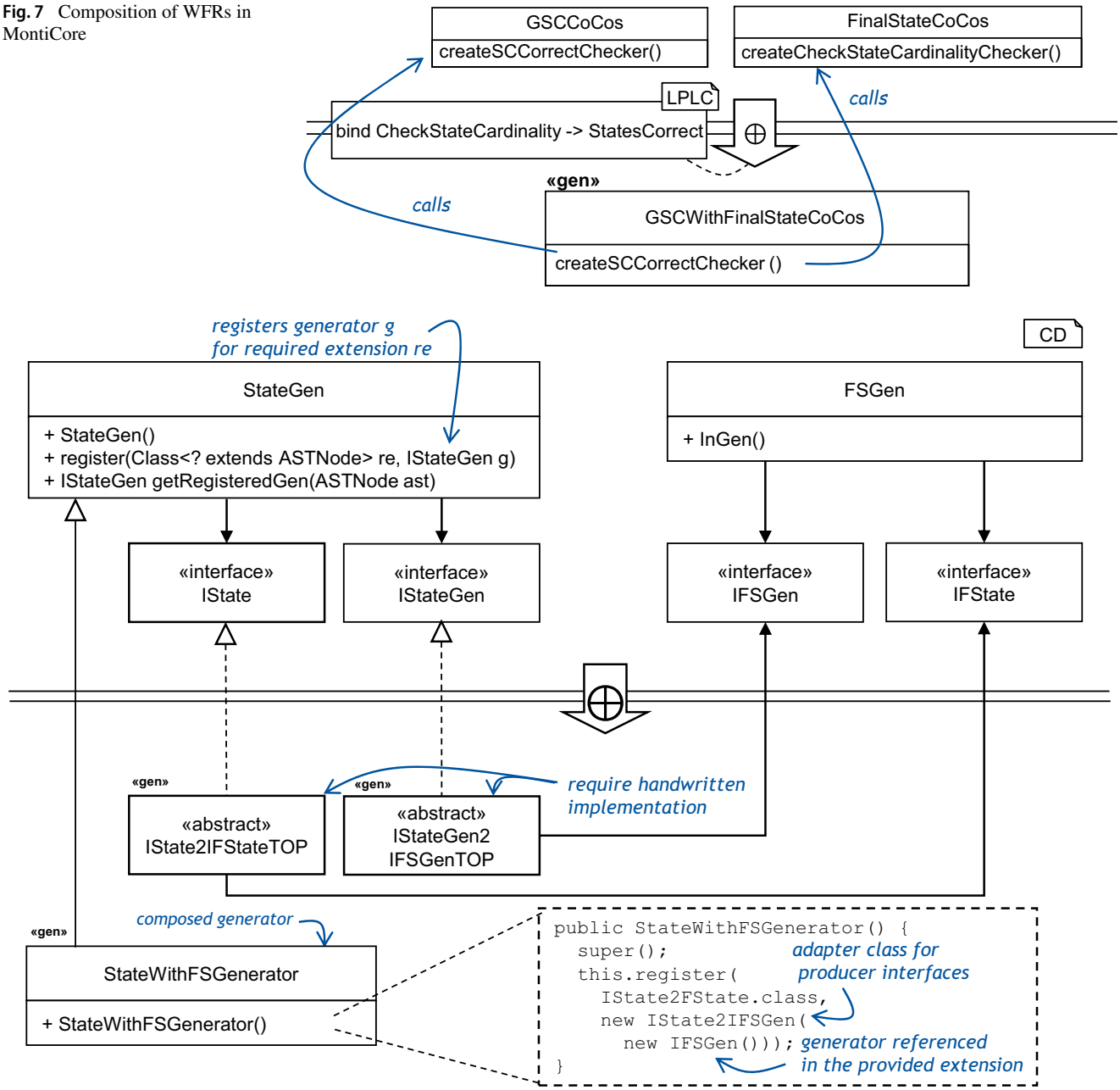

Fig. 7 Composition of WFRs in MontiCore

Fig. 8 Generator composition in MontiCore. The composed generator registers the adapted generator to be able to call it at execution time

Table 1 Adaptation effort measured in lines of code (#LoC)

Task	MontiCore #LoC	Xtext #LoC
Grammar composition	208	199
Well-formedness rule composition	381	53
Generator composition	408	373
Total	997	625

object types, and their associated properties to construct simple data structures.

2. We identified extension elements that can be employed to enhance the language in line with the OPC UA stan-

dard. These elements include, for instance, methods and variable type definitions.
3. We specified language component interfaces both for the basic OPC UA functionality and for the extension elements, so that they can be reused within a language family.

In the following, we first introduce two OPC UA language components, and then present the language family and demonstrate the language component composition by configuring this family. Finally, we compare the size of the different components of the language family to demonstrate the effectiveness of our approach. Therefore, we compare the LoC that need to be comprehended when using our approach vs. the iterative approach, where each language needs to be comprehended for its reuse.

4.2.1 OPC UA base language component

Figure 10 shows the language component interface for the base OPC UA functionality. It references the MontiCore grammar `OPCUA` (cf. l. 1). Moreover, it specifies three for grammar productions: `CustomDataTypeDef`, `ObjectTypeDefElements`, and `OPCUAElement` (cf. ll. 4–6). Of these, the last two are optional, while the first is not, as the grammar itself does not include an implementation for this production. In addition, the language component interface defines two WFR sets. The first WFR set (cf. ll. 8–10) verifies that the metadata is present in the model and that the endpoint specified in the model is actually reachable. The second WFR set (cf. ll. 12–14) validates the model's referential integrity by checking whether the employed data types exist in OPC UA, ensuring that all names are written in uppercase, and confirming that the default values of properties are compatible with their respective data types. Regarding generator extensions, the component defines one required extension for each mandatory production (cf. ll. 16–32).

4.2.2 OPC UA language family

The OPC UA language family describes the variability of the OPC UA language specification by combining a feature model with modular, composable language components. The feature model represents the variability of the OPC UA information modeling standard, e.g., the option to use methods or enums only when they are explicitly needed in the department's use cases. The language components provide interfaces that enable future extensions of the language family if required.

A textual model of the OPC UA language family is shown in Fig. 11. The feature diagram (cf. ll. 5–9) captures the essential structure and variability of OPC UA:

- `OPCUA` is the root feature and includes the minimal set of concepts needed to represent an OPC UA information model.
- `DataTypes` and `ReferenceTypes` are mandatory abstract features intended solely for logical organization.
- `VariableTypeDefinition` is an optional feature, since it introduces additional complexity on top of the core OPC UA information modeling language.
- The `Datatype` subfeatures explicitly encode the variability between `Enum` (optional) and `Struct` (mandatory), mirroring OPC UA's differentiation between simple and complex custom datatypes.
- `ReferenceTypes` are further refined into the most commonly used reference types, namely `HasProperty` and `HasComponent`.
- `Method` is modeled as an optional feature, in line with the specification, because methods belong to object types but are not required in every model.

Each non-abstract feature is implemented by a language component. The base component that implements the feature `OPCUA` supplies the fundamental language constructs and necessary extensions that are needed to support the incremental augmentation of the language with further features (cf. ll. 11–15).

The remaining features in the language family connect their offered extensions to the required extensions of the `OPCUA` feature's component. Nonetheless, if any component within the feature hierarchy declares a required extension, it can likewise serve as an extensibility point in the language family. Thus, the language family stays extensible even after it has been defined. The language architect can conveniently introduce new features in the language family's feature model section and add a component that realizes these feature definitions. In doing so, the language architect must verify that the extension is meaningful, select an appropriate location in the feature model-e.g., introducing a new kind of reference type under the data types feature would be inappropriate-and ensure that the parent feature's component offers suitable required extensions.

After defining the language family, we can derive OPC UA variants tailored to our use case. Figure 12 illustrates the resulting composed language component that is generated from the configuration shown in Fig. 12 with the features `OPCUA`, `HasProperty`, `Struct`, and `Method`. The resulting component includes all required extensions of the `OPCUABase` component. Furthermore, it incorporates the required extensions of those components that are selected through the configuration of the language family. In this specific configuration, only the component corresponding to the `Method` feature provides required extensions, and thus, only these are included in the composed component.

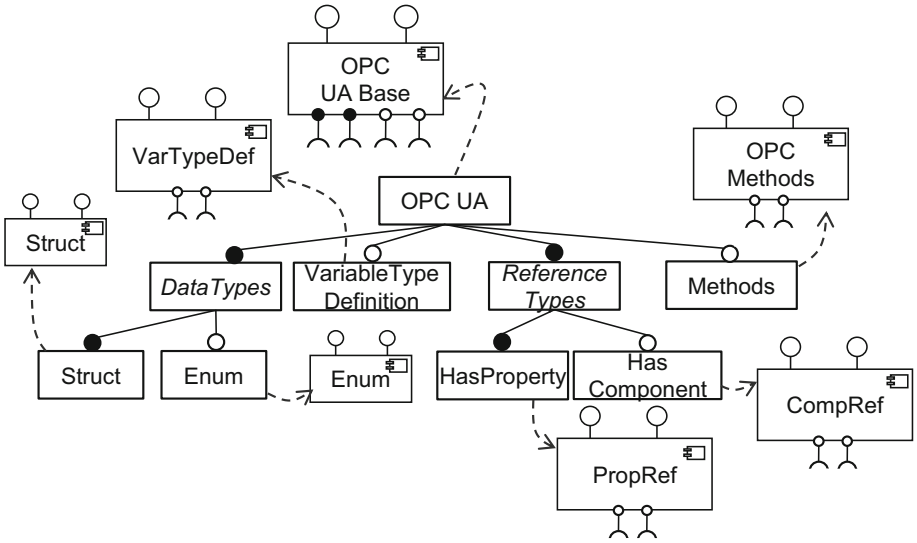

Fig. 9 OPC UA language family is leveraged in the in-depth case study of this paper

Fig. 10 Language component `OPCUABase` that represents the core functionality of the OPC UA language

```
1  dsl component OPCUABase {
2    grammar opcua.OPCUA;
3
4    requires mandatory production CustomDataTypeDef;
5    requires optional production ObjectTypeDefElements;
6    requires optional production OPCUAElement;
7
8    wfrs MetadataCorrect for OPCArtifact {
9      AllMetadataAvailable, EndPointAccessible
10   }
11
12   wfrs ReferentialIntegrity for ObjectTypeDefElements {
13     DataTypeExists, NamesUpperCase, DefaultValueFitsType
14   }
15
16   requires mandatory gen CustomDataTypeGenerator for CustomDataTypeDef {
17     generator OPCUAServerGenerator;
18     producer ICustomDataTypeDefGenerator;
19     product ICustomDataType;
20   }
21
22   requires optional gen ObjectTypeGen for ObjectTypeDefElements {
23     generator OPCUAServerGenerator;
24     producer IObjectTypeElementGenerator;
25     product IObjectTypeElement;
26   }
27
28   requires optional gen OPCElementGen for OCPUAElement {
29     generator OPCUAServerGenerator;
30     producer IOPCUAElementGen;
31     product IOPCUAElement;
32   }
33 }
```

4.2.3 Effort reduction

For measuring the scalability of the approach, we measure the reduction factor as well as the comprehension reduction. The reduction factor indicates how many times more code a language engineer must comprehend in the monolithic setting, where the language is formulated as one big language across all three constituents, versus the modular component-oriented setting that is proposed by this thesis. The reduction factor is measured as follows:

$$\text{ReductionFactor} = \frac{\text{TotalArtifactLoC}}{\text{AvgComponentLoC}}$$

Table 2 shows the LoC for the different language components by presenting the LoC for the language component interface, the grammar rules, the WFRs, and the code generators (considering only the freemarker templates). The modular OPC UA language family consists of seven components with an average size of 13 LoC. The internal artifacts amount to 558 LoC across grammar rules, well-formedness rules, and code generators. This corresponds to a reduction

Fig. 11 Textual model of the OPC UA language family

LF

```
package opclpl;

language family OPCUAFamily {

  featurediagram OPCUAFM {
    OPCUA -> DataTypes & VariableTypeDefinition? & ReferenceTypes & Method?;
    DataType -> Enum? & Struct!;
    ReferenceTypes -> HasProperty & HasComponent;
  }

  feature OPCUA {
    component opcua.OPCUABase;
    select wfrs MetadataCorrect;
    select wfrs ReferentialIntegrity;
  }

  abstract feature DataTypes;
  abstract feature ReferenceTypes;

  feature VariableTypeDefinition {
    component  opcua.VariableType;
    production VariableTypeDef   -> OPCUA.OPCUAElement;
    wfrs       VariableIntegrity -> OPCUA.ReferentialIntegrity
    gen        VariableTypeGen   -> OPCUA.OPCElementGen;
  }

  feature Method {
    component  opcua.Method;
    production Method    -> OPCUA.ObjectTypeDefElements;
    wfrs       IOArgumentsCorrect;
    gen        MethodGen -> OPCUA.ObjectTypeGen;
  }

  feature Enum {
    component  opcua.Enum;
    production Enum    -> OPCUA.CustomDataTypeDef;
    gen        EnumGen -> OPCUA.CustomDataTypeGenerator;
  }

  feature Struct {
      component  opcua.Struct;
      production Struct    -> OPCUA.CustomDataTypeDef;
      gen        StructGen -> OPCUA.CustomDataTypeGenerator;
  }

  feature HasProperty {//…}
  feature HasComponent {//…}
}

```

Fig. 12 One configuration of the language family from Fig. 11 and its resulting language component `OPCUAVariant`

FC

```
featureconfig CoreWithVariables for OPCUAFamily {
  OPCUA, Method, ReferenceType, HasProperty, DataTypes, Struct;
}
```

LC

```
language component OPCUAVariant {
  grammar opcua.OPCUAWithMethodsWithStructWithHasPropertyWithMethods;

  requires optional production MethodImplementation;

  requires optional gen MethodImplGen for MethodImplementation {
    generator MethodGenerator;
    producer IMethodImplGenerator;
    product IMethodExecutor;
  }
  //… all extensions of the OPCUABase language component
}
```

Table 2 Size comparison between modular language components and internal definition artifacts for the OPC UA Language family

Feature/Component	Component LoC	Grammar rules	WFRs	Generator LoC
OPCUABase	22	29	65	191
Enum	11	8	0	26
Struct	11	7	0	40
VariableTypeDefinition	13	14	15	22
Method	14	7	13	59
HasProperty	11	1	0	26
HasComponent	11	2	0	33
Average per Component	**13**	**10**	**13**	**56**
Absolute LoC	–	**68**	**93**	**397**
Total	–	–	–	**558**

Bold text is used to denote aggregate data and summary statistics, distinguishing calculated totals and averages from the specific, individual measurements of the modular language components listed in the preceding rows

factor of 42.92x in the amount of LoC that must be comprehended at one time.

5 Discussion

The presented formalization for composable modeling language components that can be composed systematically based on a language family facilitates the engineering of modeling languages by the holistic reuse of existing languages. For the definition of languages, we limit the scope to grammars, well-formedness rules, and code generators. This enables compatibility with language workbenches that support defining these constituents, e.g., Xtext and MontiCore. To what extent this formalism applies to other possible language constituents like metamodels, editors, scoping rules or languages with executable semantics is up to future research. The compatibility of the presented approach with other language workbenches, e.g., Neverlang, that requires the adaption of its role-based architecture, needs to be investigated in the future.

Furthermore, we assume that for opportunistic reuse, code generators must be built as proposed in [19]. Consequently, generators need a defined provided and required interface of which each comprise a producer interface and artifact interface. Concerning composition operators for these language constituents and also for language components, we only consider and define language embedding. The applicability of other operators, e.g., merging or aggregation, for various kinds of syntax and semantic specifications [46] could be investigated.

Language components have a defined interface over the mentioned language constituents. However, for the grammar, only the abstract syntax is referenced in the interface of a component. In the future, referencing the concrete syntax, e.g., to exchange or extend keywords of a language could also be worthwhile. Since the composition of language constituents, as of now, is limited to language embedding, the composition of language components is too. The composition of language components, currently, is purely additive, meaning that we can only add new constituents to a language component, and not remove existing ones. Besides, the composition of code generators prevents our approach from being fully black box since it requires the handwritten implementation of adapters between the generator interfaces. Another concern is that composing language constituents naively based on the bound interfaces may cause semantic inconsistencies that cannot be resolved automatically by our formalism. Furthermore, the composition of two language components relies on a structural composition of the interfaces that does not account for crosscutting features, e.g., error reporting, editor features. The composition may cause semantic inconsistencies or mismatches that cannot be resolved automatically by our formalism and are out of scope currently. Furthermore, our approach assumes that the interface level imposes modularity of the implementation level in the way presented in the definitions of grammars, well-formedness rules, and code generators. Another assumption of the composition is that it checks ambiguities of bound extensions only on the level of extension names, i.e., checks whether two extensions of the same name exist in one component. The check for ambiguities or issues such as the introduction of left recursion in grammars is not automatically detected and must be checked by the language workbench's specific implementation of the method. For the systematic reuse of language components, we provide language families that associate features of a feature tree with language components. The formal description of this closed variability is open to future work. The formalism could be further accompanied by an open variability concept that also enables the extension of language components after their configuration through the language family. This would provide language engineers with an even more specialized language tailored to their requirements.

6 Related work

Overall, our approach builds upon ideas formulated as concern-oriented language development [49, 50]. This approach suggests the creation of languages that are composed of modular components, known as "concerns" each equipped with interfaces that capture their variability, customization, and usage. According to this perspective, concerns consist of interconnected artifacts, adhering to metalanguages that are typified by "perspectives" found in libraries. Modularization concepts for DSLs can be categorized into endogenous and exogenous modularity [51]. The former enables modularity right in the language definition artifacts, e.g., inheritance in MontiCore [33], whereas the latter does require an additional artifact that binds to languages together, e.g., slices or traits in Neverlang [40, 52] or metamodel concerns in Melange [53] that adapt between two metamodel classes of two independent metamodels.

Concerning the formalization of modeling language constituents, composition operators, modularization, and language product lines, only a few related works exist. One approach investigates the formalization of language constituents and language variability with a focus on semantics [54]. In contrast with our approach, however, it follows a more abstract definition of the constituents, without a formal definition of the realization details of the syntax specification, the well-formedness rules, and semantics. Furthermore, they do not consider composition operators or reusable language interfaces. Other approaches solely focus on formalizing the variability of syntax defined in grammars [55, 56] or the modular reuse of metamodels' formal definitions leveraging RDF and graph transformations [57–59] without proposing a solution to the holistic reuse of the three language constituents considered in this paper. Melange [41] offers a formal description of the holistic reuse of metamodel-based syntax and operational semantics of (DSLs).

For the systematic reuse of languages via product lines, there exist formal definitions based on metamodels [60–62]. Besides, Neverlang supports the reverse engineering of a language family [63] based on a dependency graph over slices. Slices do not have an explicit interface and can comprise a grammar-like syntax definition and operational semantics for nodes of the AST. In contrast, our approach focuses on grammar-based textual syntax and translational semantics and considers the language composition operator embedding for language components, including their language constituents and interface along a language family. Another approach uses 'funcons' [64, 65] to establish reusable operational semantics across different syntax specifications. This has influenced the iCoLa [66] framework for incremental language development. On the contrary, we aim to bundle the language constituents' syntax, well-formedness rules, and translational semantics into reusable components via an explicit language component interface.

Furthermore, modularization concepts without formal descriptions exist that bundle languages into reusable components. For metamodel-based languages, there exist the notions of service components [67], fragments [68], modules [69], and concerns [53]. For grammar-based languages, the notion of slices or traits exists [40, 52, 70]. Furthermore, endogenous modularity concepts, based on the reuse via languages composition operators, exist. These can be categorized into language extension, language unification, self-extension, and extension composition [71]. Our presented formalism presents embedding, a composition operator, that can be associated with language extension. To which degree our approach can be used with other language composition operators [46] is up to future work. Examples for endogenous modularity are MontiCore [33], Spoofax [32], Rascal [72], Xtext [28], and MPS [39]. Besides, an extended taxonomy on language reuse [73] differentiates between:

- Sub-languages composition: This refers to creating a language based on joining a subset of languages without changes. In our approach, language components represent a subset of languages that can be composed via their interface, not requiring any changes to the components' constituents, to build a new language.
- Language feature composition: This refers to creating a sub-language based on a language feature selection/composition. Our approach represents language features as features in a feature tree that can be composed by its configuration.
- Syntactic and semantic assets composition: This category refers to the capability to compose both syntax and semantics of a language of two language features to create a new language. This is supported by the composition of two language components that comprise grammar-based syntax and semantics realized by a code generator.
- Semantic assets composition: Given two semantic definitions, this category describes the capability to create a new visitor that chains both semantic definitions. The paper gives the example of chaining two evaluation semantics. This is supported in our approach by providing two composition operators for composing well-formedness rules.
- Action composition: Refers to the possibility of chaining actions, e.g., desugaring, in a defined order in the language workbench. This is currently not supported in our approach.
- Action extension: This refers to overriding existing actions through extension. This is also not supported by our approach since it is purely additive.

7 Conclusion

In this paper, we have presented a novel formally-grounded definition of our method for systematic component-oriented language reuse. We define the three language constituents, grammars, well-formedness rules, and code generators. Furthermore, we introduce a definition for language components that have a provided and required interface to make their possible points of reuse and extension explicit. Besides, we define the application of language embedding to the language constituents and language components. All formal steps of our method are accompanied by a running example of a statechart language that showcases the feasibility of our concepts. Overall, our approach is defined as independent of a technological space. Consequently, our method is generalizable and realizable with language workbenches that support the defined constituents. In the evaluation, we successfully adapt the method to the language workbenches MontiCore and Xtext and demonstrate its benefits through a case study with a language family for OPC UA. Although currently our method is limited to the composition operator embedding and the three mentioned language constituents, the formal specification of the approach provides a strong foundation to be extended to other language constituents and composition operators in the future. In the future, we plan to work on the formalization of language families and the systematic composition of language components along their feature tree. While the current formalization establishes a foundation for the structural composition of grammars, WFR, and generators, future work will extend this framework to include semantic compatibility checks, such as pre/postcondition validation and automated mediation mechanisms, to detect behavioral mismatches between composed language components. In addition, future research will focus on defining a set of compositional design patterns and interface constraints to preemptively identify mismatches. This could include a compatibility checker that alerts practitioners when generators diverge semantically, alongside a library of adapter wrappers to harmonize non-composable generators. In the future, we will also work on guidelines and patterns on how to make existing languages and their constituents compatible to our framework.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Hölldobler, K., Michael, J., Ringert, J.O., Rumpe, B., Wortmann, A.: Innovations in model-based software and systems engineering. J. Object Technol. **18**, 1–60 (2019)
2. Drath, R., Lüder, A., Peschke, J., Hundt, L.: AutomationML - the glue for seamless automation engineering. In: Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, pp. 616–623 (2008)
3. Jansen, N., Pfeiffer, J., Rumpe, B., Schmalzing, D., Wortmann, A.: The language of SysML v2 under the magnifying glass. J. Object Technol. **21**(3), 1–15 (2022)
4. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**, 1–45 (2009)
5. Wortmann, A., Barais, O., Combemale, B., Wimmer, M.: Modeling languages in Industry 4.0: an extended systematic mapping study. Softw. Syst. Model. **19**, 67–94 (2020)
6. Heineck, T., Gonçalves, E.J.T., Sousa, A., de Oliveira, M.A., Castro, J.: Model-driven development in robotics domain: a systematic literature review. In: 2016 X Brazilian Symposium on Software Components. Architectures and Reuse (SBCARS), vol. 42, pp. 151–160 (2016)
7. Du, J., He, Q., Fan, X.: Automating generation of the assembly line models in aircraft manufacturing simulation. In: 2013 IEEE International Symposium on Assembly and Manufacturing (ISAM), pp. 155–159 (2013)
8. Franca, R.B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., Thomas, D.: The AADL behaviour annex–experiments and roadmap. In: 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), pp. 377–382 (2007)
9. Seidewitz, E., Miyashita, H., Wilson, M., de Koning, H.P., Ujhelyi, Z., Gomes, I., Schreiber, T., Bock, C., Grill, B., Zoltán, K., Marquez, S., Piers, W., Adavani, A., Graf, A.: Sysml v2 pilot implementation. GitHub repository (2023). Accessed 18 May 2024
10. Hölldobler, K., Roth, A., Rumpe, B., Wortmann, A.: Advances in modeling language engineering. In: International Conference on Model and Data Engineering, pp. 3–17 (2017)
11. Combemale, B., Barais, O., Wortmann, A.: Language Engineering with the GEMOC studio. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 189–191 (2017)
12. Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A.: Systematic language extension mechanisms for the MontiArc architecture description language. In: European Conference on Modelling Foundations and Applications (ECMFA'17), pp. 53–70 (2017)
13. Heim, R., Mir Seyed Nazari, P., Rumpe, B., Wortmann, A.: Compositional language engineering using generated, extensible, static type safe visitors. In: Conference on Modelling Foundations and Applications (ECMFA), pp. 67–82 (2016)
14. Butting, A., Eikermann, R., Hölldobler, K., Jansen, N., Rumpe, B., Wortmann, A.: A library of literals, expressions, types, and statements for compositional language design. In: Special Issue Dedicated to Martin Gogolla on his 65th Birthday, Journal of Object Technology **19**, 1–16 (2020). Special Issue dedicated to Martin Gogolla on his 65th Birthday
15. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Systematic composition of independent language features. J. Syst. Softw. **152**, 50–69 (2019)

16. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: 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), pp. 75–82 (2018)
17. Pfeiffer, J., Wortmann, A.: Towards the black-box aggregation of language components. In: International Conference on Model Driven Engineering Languages and Systems Companion, (MODELS-C), pp. 576–585 (2021)
18. Butting, A., Pfeiffer, J., Rumpe, B., Wortmann, A.: A Compositional framework for systematic modeling language reuse. In: International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 35–46 (2020)
19. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Modeling language variability with reusable language components. In: International Conference on Systems and Software Product Line (SPLC'18), pp. 65–75 (2018)
20. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. TOSEM **14**, 331–380 (2005)
21. Hölldobler, K., Rumpe, B., Wortmann, A.: Software language engineering in the large: towards composing and deriving languages. Comput. Lang. Syst. Struct. **54**, 386–405 (2018)
22. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Pearson Education, London (2008)
23. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**, 79–85 (2013)
24. Kühne, T.: Matters of (meta-) modeling. Softw. Syst. Model. **5**, 369–385 (2006)
25. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2009)
26. Vacchi, E., Olivares, D.M., Shaqiri, A., Cazzola, W.: Neverlang 2: a framework for modular language implementation. In: Proceedings of the Companion Publication of the 13th International Conference on Modularity, pp. 29–32 (2014)
27. Wachsmuth, G.H., Konat, G.D., Visser, E.: Language design with the spoofax language workbench. IEEE Softw. **31**, 35–43 (2014)
28. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd, Birmingham (2016)
29. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, Aachen (2017)
30. Souza Amorim, L. E., Visser, E.: Multi-purpose syntax definition with SDF3. In: International Conference on Software Engineering and Formal Methods, pp. 1–23 (2020)
31. Campagne, F.: The MPS Language Workbench: Volume I. Fabien Campagne, New York (2014)
32. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp. 444–463 (2010)
33. Hölldobler, K., Kautz, O., Rumpe, B. MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag (2021)
34. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of the 2015 ACM Sigplan International Conference on Software Language Engineering, pp. 137–148 (2015)
35. van Binsbergen, L.T., Mosses, P.D., Sculthorpe, N.: Executable component-based semantics. J. Log. Algebraic Methods Program. **103**, 184–212 (2019)
36. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**, 31–39 (2008)
37. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon transformation language. In: International Conference on Theory and Practice of Model Transformations, pp. 46–60 (2008)
38. Erdweg, S., Van Der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A. et al.: The state of the art in language workbenches: conclusions from the language workbench challenge. In: Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26–28, 2013. Proceedings 6, pp. 197–217 (2013)
39. Voelter, M.: Language and IDE modularization and composition with MPS. In: International Summer School on Generative and Transformational Techniques in Software Engineering, pp. 383–430 (2011)
40. Vacchi, E., Cazzola, W.: Neverlang: a framework for feature-oriented language development. Comput. Lang. Syst. Struct. **43**, 1–40 (2015)
41. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.: Melange: a meta-language for modular and reusable development of dsls. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE), pp. 25–36 (2015)
42. Thum, T., Kastner, C., Erdweg, S., Siegmund, N.: Abstract features in feature modeling. In: 2011 15th International Software Product Line Conference (SPLC), pp. 191–200 (2011)
43. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Atomata theory, Languages, and Computation, 3rd edn, Pearson International Edition. Addison-Wesley, New York (2007)
44. Robinson, D.F., Foulds, L.R.: Comparison of weighted labelled trees. In: Combinatorial Mathematics VI, pp. 119–126 (1979)
45. Broy, M., Cengarle, M. V., Grönniger, H., Rumpe, B.: In: Lano, K. (eds) Definition of the UML System Model, pp. 63–93. John Wiley & Sons, Munich (2009)
46. Pfeiffer, J., Rumpe, B., Schmalzing, D., Wortmann, A.: Composition operators for modeling languages: a literature review. J. Comput. Lang. **76**, 101226 (2023)
47. Schobbens, P., Heymans, P., Trigaux, J.: Feature diagrams: a survey and a formal semantics. In: 14th IEEE International Conference on Requirements Engineering (RE), pp. 136–145 (2006)
48. Beuche, D., Papajewski, H., Schröder-Preikschat, W.: Variability management with feature models. Sci. Comput. Program. **53**, 333–352 (2004)
49. Kienzle, J., Mussbacher, G., Alam, O., Schöttle, M., Belloir, N., Collet, P., Combemale, B., Deantoni, J., Klein, J., Rumpe, B.: VCU: the three dimensions of reuse. In: Software Reuse: Bridging with Social-Awareness: 15th International Conference (ICSR), pp. 122–137 (2016)
50. Combemale, B., Kienzle, J., Mussbacher, G., Barais, O., Bousse, E., Cazzola, W., Collet, P., Degueule, T., Heinrich, R., Jézéquel, J.-M., et al.: Concern-oriented language development (COLD): fostering reuse in language engineering. Comput. Lang. Syst. Struct. **54**, 139–155 (2018)
51. Méndez-Acuña, D., Galindo, J.A., Degueule, T., Combemale, B., Baudry, B.: Leveraging software product lines engineering in the development of external DSLs: a systematic literature review. Comput. Lang. Syst. Struct. **46**, 206–235 (2016)
52. Cazzola, W., Vacchi, E.: Language components for modular DSLs using traits. Comput. Lang. Syst. Struct. **45**, 16–34 (2016)
53. Leduc, M., Degueule, T., Combemale, B.: Modular language composition for the masses. In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE), pp. 47–59 (2018)
54. Grönniger, H., Rumpe, B.: Modeling language variability. In: Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems - 16th Monterey Workshop, vol. 6662, pp. 17–32 (2010)

55. Johnstone, A., Scott, E., van den Brand, M.: Modular grammar specification. Sci. Comput. Program. **87**, 23–43 (2014)
56. Andersen, J., Brabrand, C., Christiansen, D.R.: Banana algebra: compositional syntactic language extension. Sci. Comput. Program. **78**, 1845–1870 (2013)
57. de Lara, J., Guerra, E., Kienzle, J.: Facet-oriented Modelling. ACM Trans. Softw. Eng. Methodol. (TOSEM) **30**, 1–59 (2021)
58. Braatz, B., Brandt, C.: A framework for families of domain-specific modelling languages. Softw. Syst. Model. **13**, 109–132 (2014)
59. Rabbi, F., Lamo, Y., Kristensen, L.M.: A model driven engineering approach for heterogeneous model composition. In: Model-Driven Engineering and Software Development - 5th International Conference (MODELSWARD), vol. 880, pp. 198–221 (2017)
60. de Lara, J., Guerra, E.: Language family engineering with product lines of multi-level models. Formal Asp. Comput. **33**, 1173–1208 (2021)
61. de Lara, J., Guerra, E., Bottoni, P.: Modular language product lines: a graph transformation approach. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 334–344 (2022)
62. Lara Jaramillo, J., Guerra Sánchez, E., Bottoni, P. et al.: Modular language product lines: concept, tool and analysis. In: Software and Systems Modeling (SoSyM), vol. 23, pp. 1619–1666 (2024)
63. Vacchi, E., Cazzola, W., Pillay, S., Combemale, B.: Variability support in domain-specific language development. In: Software Language Engineering - 6th International Conference (SLE), vol. 8225, pp. 76–95 (2013)
64. Mosses, P.D.: Software meta-language engineering and CBS. J. Comput. Lang. COLA **50**, 39–48 (2019)
65. Mosses, P. D.: Fundamental constructs in programming languages. In: Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods (ISoLA), vol. 13036, pp. 296–321 (2021)
66. Frölich, D., van Binsbergen, L.T.: iCoLa+: an extensible meta-language with support for exploratory language development. J. Syst. Softw. **211**, 111979 (2024)
67. Berg, H.: Service-oriented design of metamodel components. In: ICSOFT 2012 - Proceedings of the 7th International Conference on Software Paradigm Trends, pp. 70–79 (2012)
68. Zivkovic, S., Karagiannis, D.: Towards metamodelling-in-the-large: interface-based composition for modular metamodel development. In: Enterprise, Business-Process and Information Systems Modeling - 16th International Conference (BPMDS), vol. 214, pp. 413–428 (2015)
69. Méndez-Acuña, D., Galindo, J.A., Combemale, B., Blouin, A., Baudry, B.: Reverse engineering language product lines from existing DSL variants. J. Syst. Softw. **133**, 145–158 (2017)
70. Kühn, T., Cazzola, W., Olivares, D. M.: Choosy and picky: configuration of language product lines. In: Proceedings of the 19th International Conference on Software Product Line (SPLC), pp. 71–80 (2015)
71. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31–April 1, 2012 (2012)
72. Basten, B., van den Bos, J., Hills, M., Klint, P., Lankamp, A., Lisser, B., van der Ploeg, A., van der Storm, T., Vinju, J.J.: Modular language implementation in Rascal - experience report. Sci. Comput. Program. **114**, 7–19 (2015)
73. Bertolotti, F., Cazzola, W., Favalli, L.: On the granularity of linguistic reuse. J. Syst. Softw. **202**, 111704 (2023)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Jérôme Pfeiffer is a research assistant and PhD candidate at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. His research interests are software language engineering, digital twins, and artificial intelligence. For more information, please visit: https://www.isw.uni-stuttgart.de/en/institute/team/Pfeiffer-00005/



Andreas Wortmann is head of the chair "Model-Driven Engineering for Manufacturing Automation" at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. He conducts research on model-driven engineering, software language engineering, and systems engineering with a focus on Industry 4.0 and digital twins. For more information, please visit: www.wortmann.ac.