JOT
JOURNAL OF
OBJECT TECHNOLOGY

[BMR22] A. Butting, J. Michael, B. Rumpe:
Language Composition via Kind-Typed Symbol Tables.
In: Journal of Object Technology, Volume 21, pp. 4:1-13, AITO - Association Internationale pour les Technologies Objets, Oktober 2022.
www.se-rwth.de/publications/

Language Composition via Kind-Typed Symbol Tables

Arvid Butting*, **Judith Michael***, **and Bernhard Rumpe***
*Software Engineering, RWTH Aachen University

ABSTRACT The modularization of domain-specific modeling languages (DSMLs) fosters individual reuse of DSMLs in different contexts. Within this article, we discuss how it is possible to refer to model elements of other languages when composing different DSMLs. Related approaches usually rely on a DSML-agnostic language infrastructure that tests the compatibility of model elements via types encoded in Strings without any consistency checks. We propose the "strongly kind-typed" symbol table as an extension to the compiler approach to integrate the syntax of the languages using symbol tables that assign a symbol kind to each name definition. Our approach can be integrated into language workbenches that provide a symbol table infrastructure as part of a DSML implementation. The kind-typed symbol tables are integrated into the language workbench MontiCore. Strongly kind-typed symbol tables utilize the type system of the language workbench's host language to ensure type consistency between the language-specific symbol table infrastructures during DSML composition, which ultimately supports DSML engineering in the large.

KEYWORDS Software Language Engineering, Symbol Tables, Language Composition.

1. Introduction

Domain-specific modeling languages (DSMLs) (Combemale et al. 2016) reduce the problem-solution gap (France & Rumpe 2007) as they use terms of specific domains, allow for domain-specific abstractions, and prepare the foundations for domain-specific analyses and tooling. If the UML modeling methods should be applied, UML profiles (Fuentes-Fernández & Vallecillo-Moreno 2004) are another possibility to realize domain-specific needs: a topic studied in particular by Antonio. The fact that these languages are better tailored to domains results in their greater use in practice, e.g., in systems engineering (Gupta et al. 2021), architecture modeling for safety critical automotive software systems (Schlichthaerle et al. 2020), cyber attacks in the automotive domain (Wolschke et al. 2021), the German tax forms (Rumpe et al. 2021), marine ecosystem simulation (Johanson & Hasselbring 2017), or the verification of valid TV program planning according to contracts (Drave et al. 2020), just to mention a few published ones.

JOT reference format:
Arvid Butting, Judith Michael, and Bernhard Rumpe. *Language Composition via Kind-Typed Symbol Tables*. Journal of Object Technology. Vol. 21, No. 1, 2022. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2022.21.1.a5

From a longer-term software engineering perspective, the greater use of DSMLs has to lead to the modularization of DSMLs as well, since this improves their reusability (Şutîi et al. 2018). This also implies, however, that they in turn can be composed (Vallecillo 2010; Meyers et al. 2012). The use of models from different languages increases the need to allow references to model elements from other languages and thus to provide tool support. Existing approaches using symbol tables rely on a DSML-agnostic language infrastructure (Mir Seyed Nazari 2017) that tests the compatibility of model elements via types encoded in Strings without consistency checks.

We tackle the research question *how it is possible to refer to model elements of other languages when composing different DSMLs*. Within this paper, we propose the concept of strongly kind-typed symbol tables as an extension to the compiler approach. They integrate the syntax of the languages using symbol tables that assign a symbol kind to each name definition. This approach can be integrated into language workbenches (Erdweg et al. 2015) that provide a symbol table infrastructure as part of a DSML implementation. As an example, we have integrated them into the language workbench MontiCore (Hölldobler et al. 2021). Strongly kind-typed symbol tables utilize the type system of the language workbench's host language. This ensures type consistency between the language-specific symbol

table infrastructures during DSML composition. We discuss how strongly kind-typed symbol tables can support the four different types of language composition within MontiCore and critically discuss the approach.

Structure. The next section introduces an example for different languages used to model one cyber-physical system. Section 3 introduces the used technology stack and basic principles for handling DSMLs. Section 4 presents the strongly kind-typed symbol table infrastructure and Section 5 describes how it can be used to support language composition. Section 6 discusses our approach, Section 7 compares it to related approaches, and the last section concludes.

2. Example

In modern model-driven software projects, heterogeneous models describe different aspects of the application (Butting & Wortmann 2021). In the domain of cyber-physical systems (CPS), e.g., class diagrams can be used to describe the data types of the CPS, automata could describe its behavior, and a CAD model might describe its physical geometry.

For this example, we want to model an excerpt of the software for a coffee machine. We model its data structure through a class diagram and use an automaton to describe an excerpt of the coffee machine's behavior. To this end, our aim is to ensure that the class diagram model and the automaton model do not contradict each other due to inconsistencies.

One way to achieve consistency between such heterogeneous models is to compose the languages to which the models conform. Such language composition can detect whether symbolic references between models conforming to different languages are valid.

For our example, the automata language represents automata models that have states, which can optionally be initial states, and transitions between pairs of states that are triggered by events. The class diagram language models a simplified form of class diagrams that contain classes with inheritance as well as enums. Classes may have typed class attributes and enums contain enum constants. Furthermore, class diagram models can have named, directed, and cardinalized associations between classes and from classes to enums. Both languages are realized as textual DSLs with the language workbench MontiCore (Hölldobler et al. 2021).

Textual languages usually rely on names as identifiers for language elements. Other languages then may refer to such language elements via their identifying name. For example, the automata language uses names to identify states and transitions. Another language can use the name to refer to the state with that name. This, obviously, requires that the name identifies the state uniquely.

Figure 1 depicts the two textual models of the coffee machine examples. The listing on the left side of the figure presents the automaton `CoffeeCtrl` that contains the four states `idle`, `check`, `operating`, and `maintenance` where the state `idle` is marked as the initial state. The initial state has an initial action block surrounded by curly brackets. Moreover, the automaton model contains 4 transitions between the states. Transitions can have trigger conditions surrounded by square brackets and action blocks surrounded by curly brackets.

In the first line of the listing, the automaton model imports the class `Machine` of the artifact `CoffeeMachine`, which refers to the class diagram model with this name that is depicted on the right side of Figure 1. Through this import statement, all names defined in the class diagram model can be used in the automaton model. For example, the name of the association `led` in l. 10 of the class diagram model can be used in the action blocks of the initial state (l. 4) transitions (ll. 8, 12, and 14).

These forms of links between name usages and name definitions that can not only cross the boundaries of model artifacts of a single language but also cross different languages, can be achieved with a suitable symbol table infrastructure and proper symbol resolution strategies. The remainder of this paper introduces a suitable symbol table infrastructure that enables realizing such links while coupling the language infrastructures only loosely and ensuring type compatibility of the symbol table infrastructures.

The coffee machine example introduced in this section serves as an analytical evaluation that represents a larger class of problems, i.e., transfers to symbol table infrastructures of other languages as well.

3. Preliminaries

We use the language workbench MontiCore (Hölldobler et al. 2021) for the compositional engineering of DSMLs. MontiCore languages use context-free grammars that are defined using an EBNF-like grammar format from which the workbench generates large parts of the infrastructure necessary to handle models of these languages. Figure 2 gives an overview of the generated infrastructures and their usage.

MontiCore reads the grammar of a DSML and generates the infrastructure needed to process models defined according to this language such as a parser, the abstract syntax including the Abstract Syntax Tree (AST) of the model and the symbol table (Hölldobler et al. 2015) representing the abstract syntax (Burgueño et al. 2019) of a model, as well as a visitor infrastructure. The parser is able to read models that conform to the defined grammar and instantiates the AST. The symbol table enables realizing symbolic links between usages and definitions of names. Details about the symbol table are given in Section 4. The generated visitor infrastructure enables traversing the abstract syntax data structure of both, the model AST and the symbol table. This visitor infrastructure together with the abstract syntax can be used by analyses and transformations, e.g., to check Context Conditions (CoCos). Such CoCos are Java classes that realize well-formedness rules of the language. They are needed as context-free grammars do not ensure the well-formedness of models - an aspect necessary for the further use of models within software systems. Each CoCo is implemented against an element of the abstract syntax. A visitor for the language traverses the abstract syntax and systematically executes the CoCo checks.

Each MontiCore grammar defines the abstract and concrete syntax of a language via productions that define nonterminals

```
01 import CoffeeMachine.Machine;
02
03 automaton CoffeeCtrl {
04   initial state idle / {led = GREEN};
05   state check, operating, maintenance;
06
07   idle -> check
08       [buttonPressed] / {led = YELLOW};
09   check -> operating
10       [coffeeCounter <1234] / {makeCoffee()};
11   check -> maintenance
12       [coffeeCounter>=1234] / {led = RED};
13   operating -> idle
14       / {led = GREEN; coffeeCounter++};
15 }
```

```
01 classdiagram CoffeeMachine {
02   class Machine {
03     long coffeeCounter;
04     boolean buttonPressed;
05     makeCoffee();
06   }
07
08   enum LED { GREEN, YELLOW, RED;}
09
10   association led: Machine -> LED [1];
11 }
```

*link from a **usage** of the name "led" to the **definition** of the name in another model, conforming to another language*

Figure 1 References between models of the automata and class diagram languages.

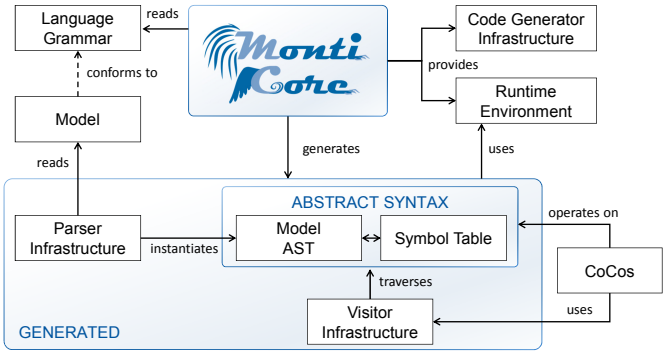

Figure 2 An overview of the generated infrastructures and their usage

(NTs). Grammar bodies include such productions (or grammar rules) consisting of a left-hand and a right-hand side separated by an '=' sign. One NT is on the left-hand side and it is defined by the right-hand side part of the grammar rule. The right-hand side defines the abstract and concrete syntax and can include terminals and nonterminals arranged as alternatives, concatenations, or combinations thereof. Grammar rules are translated into class types in the AST. Additionally, there exist several kinds of special grammar rules, such as interface rules which define an interface NT and are translated into interface types in the AST. The right-hand side of an interface rule can contain (cardinalized) NTs. This results in the presence of these NTs (with matching cardinalities) in the right-hand side of rules which implement the interface NT. This allows preventing undesired forms of implementations, and they can act as extension points of a grammar.

MontiCore distinguishes four different ways of composing languages (Bryant et al. 2015; Erdweg et al. 2012), namely inheritance, extension, embedding, and aggregation (Hölldobler et al. 2021). Except for language aggregation, all forms produce a composed language with an integrated syntax of the related models. The MontiCore generator has to be executed to synthesize an integrated language tooling. *Language inheritance* is indicated by an inheritance relationship between grammars. The inheriting grammar reuses all NTs and terminals of the grammar it inherits from. Moreover, it can extend or override some NTs. *Language extension* is a particular form of inheritance, also called conservative extension. Grammar rules reuse the start NT of the extended language and add alternatives to any NT of the extended language. *Language embedding* allows to embed one or more languages into another one, called host language. This is realized via extension points of the syntax in the host language. The embedded languages realize or implement these extension points. *Language aggregation* is a loose coupling, where models remain in individual artifacts but may refer to elements of other models via names. This form of language composition uses composed symbol tables (see Section 4).

Additionally, MontiCore provides a code generator infrastructure as well as a run-time environment for each language, which is used by the generated artifacts. The template-based code generation infrastructure uses the FreeMarker template engine. Moreover, the transformation into other models or the generation of templates is also possible.

4. Strongly Kind-Typed Symbol Table Infrastructures

MontiCore integrates a strongly kind-typed symbol table infrastructure (STI) and generates an STI as part of every language. Because it is generated, the STI can rely on language-specific types for most of its parts, which reduces the language-agnostic parts of the implementation in MontiCore's runtime-environment. The major advantage of a strongly typed symbol table infrastructure, however, is that large parts of the compatibility of different STIs during language composition are offloaded to the type system of Java.

Symbol table infrastructures serve multiple purposes and different languages have particular concepts, e.g., for the visibility of names or the rules for handling qualified names. Therefore, the STI offers default solutions for such concepts and relies on easy adjustability of these. To this end, all generated parts of the STI are adjustable via the top mechanism (Hölldobler et al. 2021). With this mechanism, each generated Java class or interface can be supplemented with a handwritten Java artifact that extends the generated class and automatically integrates with the remaining generated code.

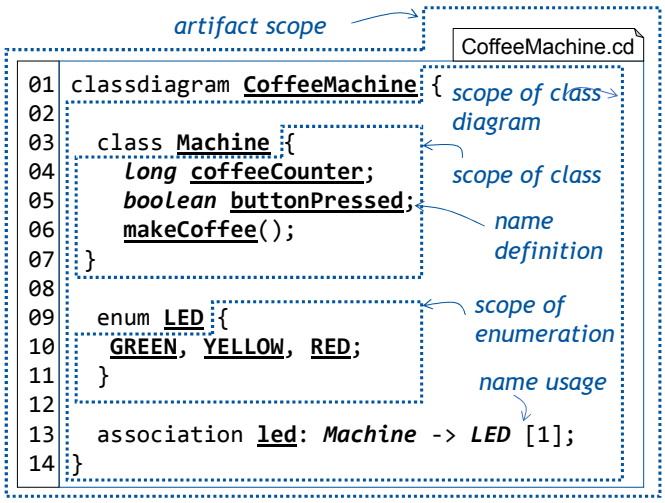

Figure 3 Illustration of name definitions, name usages, and scopes in a model

The section first introduces general notions about the symbol tables in Section 4.1 before explaining the symbols (in Section 4.2) and scopes (in Section 4.3) of the STI in detail. Section 4.4 describes the relations of the STI elements and the AST of a language. Section 4.5 presents an approach for instantiating and Section 4.6 an approach for traversing symbol tables. The process of resolving for symbol definitions from given name usages is described in Section 4.7.

4.1. General Notions of Symbols, Scopes, and Symbol Tables

Before we explain the concepts behind the STI, we make some general definitions for basic elements that appear in the context of a symbol table.

In classical compiler construction (Aho et al. 2007), symbol tables are part of a language's compiler and support engineering the type system of a language. A symbol table is a data structure that stores all name definitions that the parser encounters while parsing a model or program. Together with the name, the symbol table may hold additional information belonging to the name definition. We say that the kind of additional information and the kind of the name that is introduced is tied to a particular named model entity, i.e., a language syntax element. Such named model entities, hence, are language-specific and can be states, classes, methods, etc.

For example, the class diagram model depicted in Figure 3 defines names of different kinds. It defines names of class diagrams (the name `CoffeeMachine`), type definitions (`Machine`, `LED`), class variables (e.g., `coffeeCounter`), methods (`makeCoffee`), enum constants (e.g., `GREEN`), and associations (`led`). While the class diagram model defines the actual names, i.e., the *symbols*, the class diagram language defines the kinds of these names, i.e., the *symbol kinds*[1].

[1] Usually, languages define symbol kinds and models define symbols. Rarely, a language defines symbols (e.g., of built-in types) or a model introduces new symbol kinds (e.g., via stereotypes)

The additional information for each name definition of a symbol depends on the model entity and, hence, can vary. For example, a class variable has a type expression indicating the variable's type while states have a Boolean property indicating whether the state is initial. Besides keeping kind-specific information, distinguishing symbols of different kinds enables language engineers to allow different model elements with the same name. For example, the class diagram `CoffeeMachine` could contain a class with the name `CoffeeMachine`. The language infrastructure can distinguish the names based on their kind. Another example in Java is a class that contains a method with the same name as the class without causing a conflict.

All names that are defined in a model may be used within this model (or other models, but more on this later). For example, the `CoffeeMachine` class diagram uses the name of the classes `Machine` and `Status` in the association defined in l. 13 of Figure 3.

Some languages allow also multiple definitions of the same name with the same kind in a model. To distinguish such names, symbol tables support visibility of names. In Java, for instance, a variable in a method and a variable inside the body of a `for` loop in the method may have the same name without causing a conflict in the compiler.

To realize such visibility concepts, the symbol table of a language can contain *scopes*. A *scope* holds a collection of symbols and impacts their visibility (Mir Seyed Nazari 2017). Each scope can import symbols from and export them to other scopes and has three particular Boolean properties: a scope can be *ordered*, *exporting*, and *shadowing*. In an ordered scope, names in the concrete syntax must be defined before they are allowed to be used. In Java, e.g., this is the case for variable definitions inside of method bodies. Other scopes may be agnostic of such orders, such as, class attributes in Java can be used before they are defined. An exporting scope exports symbols for other scopes while a non-exporting scope keeps symbol definitions local. Examples in Java are the scope of a Java class that exports symbols of (non-private) attributes and methods while the scope of a method does not export local variable symbols. In a shadowing scope, a local symbol of a concrete kind and name "shadows" symbols with this name and kind imported from other scope without causing a conflict. In non-shadowing scopes, this situation causes a conflict. In Java, e.g., the body of a for-loop shadows variables from the scope of the surrounding method body.

In MontiCore, the scopes are arranged as a tree. Hence, scopes are nested and each scope (except the scope at the root of the tree) has exactly one enclosing scope. MontiCore supports two special kinds of scopes: artifacts scopes and the global scope. An artifact scope encloses the scopes of a certain model artifact and the (singleton) global scope is the root of the scope tree. Global scopes usually have artifact scopes as their direct subscopes and, hence, can bridge the gap between multiple individual model artifacts.

The type check of a compiler uses the symbol table to look up whether the type used in a type expression refers to an actual type definition. In this sense, type systems introduce a controlled form of redundancy between a name of a type definition and

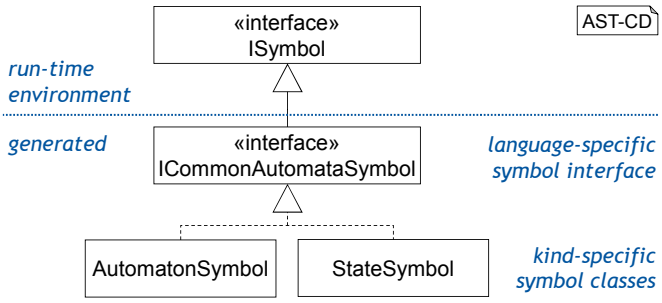

Figure 4 Classes per symbol kind.

names that refer to this definition. We refer to the process of searching a suitable name definition from a given name usage as *symbol resolution* or *resolving* a symbol.

Besides for type checks, the symbol table also support the realization of well-formedness checks that involve multiple elements of a model. For example, a context condition of an automata language can check that the initial state has no incoming transitions.

4.2. Kind-Specific Symbol Classes

As introduced before, the STI foresees that the language infrastructure contains a dedicated class for each symbol kind. This facilitates compatibility checks during language composition and avoids re-creating a type system for symbol kinds on top of the type system in Java.

To properly generate STIs for each language, MontiCore requires knowledge about (named) language elements that define symbol kinds and about the additional information that should be associated with each symbol kind. To specify this, language engineers can influence the generation of the STI via the grammar of the language. To this end, each nonterminal of a MontiCore grammar that has a name on the right-hand side of the corresponding grammar rule can be annotated with the keyword `symbol`. Through this, the nonterminal is marked as to define a new symbol kind. The annotation can be added both to class nonterminals and to interface nonterminals. If an interface nonterminal defines a symbol kind, all implementing nonterminals also define this symbol kind. The property of defining a symbol kind is also inherited if an (interface) nonterminal extends another (interface) nonterminal.

From each nonterminal that defines a new symbol kind, MontiCore generates an individual Java class that represents the symbol kind. The generated symbol class implements a language-specific symbol interface that is generated once per language. This interface extends a language-agnostic symbol interface in the run-time environment of MontiCore.

An example for the relevant classes and interfaces for the symbol kinds of the automata language is depicted in Figure 4. In this language, the `Automaton` and the `State` nonterminal define symbol kinds, as both are identifiable via names. Transitions of this language are not identifiable via names and, hence, do not introduce symbol kinds. The generated STI of the automata language contains two classes `AutomatonSymbol` and `StateSymbol` that represent the two symbol kinds. Both classes implement the interface `ICommonAutomataSymbol` that introduces language-specific method signatures that are common among all symbol kinds of the language. This interface extends the interface `ISymbol` that introduces method signatures common among all symbol kinds of all languages. In the example model of the automata language depicted in Figure 1, there is one symbol of the symbol kind `AutomatonSymbol` with the name `CoffeCtrl` and four symbols of the symbol kind `StateSymbol` with the names `idle`, `check`, `operating`, and `maintenance`.

The MontiCore grammar language contains a special `symbolrule` notation for grammar rules that define additional class attributes and methods for each symbol class, i.e., for each symbol kind. From these grammar rules, the MontiCore generator produces corresponding attributes and methods in the generated symbol classes. For example, the class attribute `coffeeCounter` depicted in Figure 3 creates a symbol of the symbol kind `ClassAttributeSymbol`. In the grammar of the class diagram language, a language engineer can add a symbol rule to indicate that class attributes have a type expression that keeps track of the attribute's type.

4.3. Language-Specific Scope Types

As introduced in Section 4.1, MontiCore distinguishes the scopes that may appear in a model from artifact scopes and the global scope. The global scope is realized as a singleton that is instantiated only once. Artifact scopes are created for every model, for which the symbol table is instantiated. Similar to nonterminals that define a symbol kind, MontiCore supports nonterminals that span a new scope. This enables language engineers to conceive various visibility concepts for symbols and, hence, create different namespaces. A nonterminal that spans a scope is introduced by adding the keyword `scope` to the corresponding grammar rule. The same nonterminal can also both introduce a symbol kind and span a scope. This combination is often used to create hierarchical namespaces. In hierarchical namespaces, symbols are addressed via qualified names that comprise individual name parts separated by dots. Each name part equals the name of a symbol and a consecutive dot addresses the scope spanned by that symbol. A prefix of a qualified name can also address the artifact in which the symbols are located, where the name of the artifact may be qualified with a package in the same way as Java.

For example, in the class diagram, the qualified name `CoffeeMachine.Machine.coffeeCounter` addresses the artifact `CoffeeMachine` and within this artifact, the symbol `Machine`. From this symbol, it accesses the spanned scope and in this scope, again, it addresses the symbol `coffeeCounter`.

Contrary to the kind-specific symbol classes, MontiCore does not generate individual scope classes per nonterminal that spans a scope. Instead, MontiCore generates a single scope class per language that can be configured through arguments to reflect scope attributes, such as, being an ordered scope or exporting symbols. However, MontiCore distinguishes scope classes that are used within models from classes for artifact scopes and the global scope. Artifact scopes must contain functionality that deviates from the functionality of scopes within

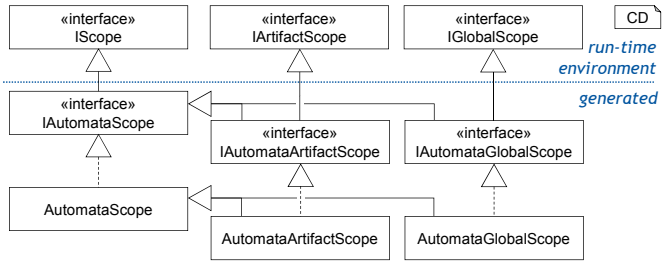

Figure 5 Interfaces and classes MontiCore uses for scopes.

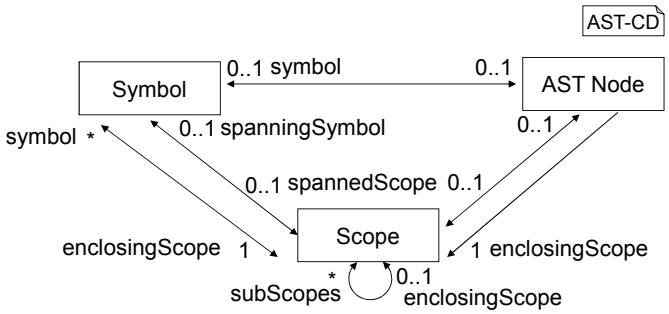

Figure 6 Relationships between scopes, symbols, and AST nodes.

models to be able to address the model artifact during symbol resolution. The global scope also requires custom functionality to realize the resolution algorithm for the connection between different model artifacts and to realize loading the symbol tables from foreign models. However, as artifact scopes and the global scope share large parts of the functionality with the scopes classes used for scopes within models, the artifact and global scope classes are subclasses from the scope classes. This is depicted by example of the automata language at the bottom of Figure 5. The classes `AutomataArtifactScope` and `AutomataGlobalScope` extend the class `AutomataScope`.

As MontiCore supports multiple inheritance on grammars for achieving language composition, the scopes that are defined per language have to reflect this inheritance. However, Java does not support multiple inheritance for classes. Hence, MontiCore realizes most of the scope functionality via default methods in scope interfaces. Each scope class implements a corresponding scope interface. Through this, the inheritance of scope interfaces can follow the language inheritance controlled through the grammar and the default methods in scope interfaces can be reused during language composition. The scope classes, on the other hand, are primarily used for internal purposes, such as, managing the access to scope attributes via getter and setter methods. The functionality in scope classes cannot be reused for language composition.

An overview of the scope interfaces generated for the automata language is depicted in the center row of Figure 5. The interfaces `IAutomataArtifactScope` and `IAutomataGlobalScope` extend the interface `IAutomataScope`. Each scope interface is implemented by the corresponding scope class.

Similar to the *ISymbol* interface, the run-time environment of MontiCore contains language-agnostic interfaces for (artifact/global) scope interfaces.

4.4. Relationship Between Symbols, Scopes, and AST

The abstract syntax data structure of a language includes symbol classes, scope interfaces & classes, and the classes that constitute the AST data structure. These classes are tightly coupled to each other through various relationships as depicted in Figure 6. Each symbol class (depicted left) and each scope (depicted bottom) are associated with an optional AST node (depicted right). If the symbol table has been instantiated from the result of parsing a model, the AST nodes are always present. If, however, the symbol table has been loaded from a symbol file, the AST nodes are not present. In the reverse direction, an AST node has an association to a symbol if the corresponding nonterminal introduces a symbol kind and an association to the scope if the nonterminal spans a scope. Each AST node and each symbol has an enclosing scope. A scope contains symbols. As in MontiCore, scopes are arranged in a tree-shape, all scopes (except the global scope) have an enclosing scope and each scope may have subscopes.

Additionally, a scope may be bidirectionally associated with a symbol that spans the scope. This association supports the qualification of names to make these globally unique, which is required for symbol resolution.

As the STI uses symbol classes strongly typed per symbol kind, scope classes strongly typed per language, and AST node classes strongly typed per nonterminal, the relations between symbols, scopes, and AST nodes are also realized as attributes with the strongest possible types. The relations between scopes and from AST nodes to scopes can be realized with the scope type of the language. A scope contains individual data structures per symbol kind and, hence, also per symbol class. Therefore, these can be strongly typed as well. An AST node may define at most one symbol kind, which enables to indicate the concrete type of the symbol kind's class.

However, a symbol kind may be defined by different nonterminals, e.g., if the symbol-defining nonterminal is an interface nonterminal. Therefore, the relation from an AST node to a symbol cannot be typed with a single concrete symbol class.

4.5. Instantiating Symbol Tables

In the STI, symbol tables are instantiated via *scope genitors* that traverse the AST data structure with the language's visitor infrastructure. To realize a correct nesting of scopes, the scope genitor internally manages a stack of scopes, where the current scope is always on top of the stack.

At the beginning of instantiating the symbol table of a model artifact, a new artifact scope instance is created and added as first element to the scope stack. Afterward, the AST of the model is traversed with the language's traverser. For each nonterminal, the scope genitor implements the `visit` method for the corresponding AST class and sets the current scope as the enclosing scope.

For each nonterminal that defines a symbol kind, the scope genitor further creates a new symbol of the correct symbol kind.

Analogously, the scope genitor creates new scopes on each nonterminal that spans a scope. After a new scope is created, it is put on top of the scope stack and, hence, set as the current scope. At the end of visiting and traversing the nonterminal that spans the scope, the scope is removed from the scope stack and the stack below is set as current scope.

All created symbol and scope objects are linked to each other and to the AST nodes as described in Section 4.4. Beyond this, the attributes of symbol rules and scope rules should be initialized as well. However, this initialization cannot be generated as the information required for the initialization is not specified in the grammar. Instead, language engineers have to customize the symbol table instantiation to initialize such attributes.

The symbol table instantiation can be customized by applying the top mechanism (Hölldobler et al. 2021) to the scope genitor class. Furthermore, all scope genitor classes are generated with hook point methods that have an empty implementation by default and are called at certain phases of the symbol table creation. These hook points include methods that are called before and after each instantiation of a symbol of a specific kind and at the beginning and the end of the entire symbol table creation process.

4.6. Traversing Symbol Tables

As introduced in Section 3, the traverser infrastructure of a language can traverse the AST data structure. However, in the STI, the visitor infrastructure is also capable of traversing the symbol table data structure of a language. In this, the traverser of a language offers methods to traverse scopes and symbols. By default, the symbol tables are traversed as part of traversing the AST. Hence, `traverse` methods for scopes traverse all symbols that are directly located in the scope and the `traverse` methods of symbols do not traverse anything. This default behavior can be altered by customizing the language traverser. For instance, for serializing symbol tables without traversing the AST, a custom implementation with a novel traversal strategy is provided. In this example, a scope traverses all local symbols and symbols that span a scope also traverse the spanned scope.

4.7. Symbol Resolution

An essential functionality of the symbol table of a language is the symbol resolution, i.e., to search the name definition that corresponds to a given name usage. Symbol resolution searches the scopes and considers all symbol visibility rules. We introduce the main concepts of the symbol resolution, an in-depth introduction of the symbol resolution algorithm would go beyond the scope of this paper.

Resolution of a symbol s with the kind k in the STI begins with searching for a symbol with the name s in the list of symbols of kind k in the current scope. This part of the algorithm is called the *local symbol resolution*. If no symbol was found, the resolution begins with local symbol resolution in the enclosing scope of the current scope. This is proceeded until either a symbol is found or the current scope is an artifact scope. This phase of the resolution is called *bottom-up intra-model symbol resolution*.

In the artifact scope, the symbol name is prepared for *inter-model symbol resolution*, i.e., the resolution in foreign models. As preparation, the name of the model is qualified. Depending on the language, there may be different possible qualifications for a name to look up and the symbol resolution tests all of these "candidates" for qualified names. For example, in Java, potential qualifications may be all import statements of a model that end with a star ('*'). The inter-model resolution continues in the global scope, which attempts to load artifacts that may contain the symbol based on the given qualified name. The symbol tables of all these artifacts are loaded, either from persisted symbol table files or by parsing the models and creating the symbol tables from their ASTs. After all these symbol tables have been loaded, the global scope has new artifact scopes as subscopes. If the same artifacts are to be used for another symbol resolution, they do not have to be loaded again.

Afterward, the symbol resolution continues with the top-down intra-model symbol resolution in all of the considered artifact scopes. This part of the resolution begins with local symbol resolution in each artifact scope and from there, continues resolution in the subscopes iteratively. This procedure can ignore scopes that are spanned by symbols that do not match the expected parts of the qualified name that is resolved for.

All parts of the symbol resolution are realized in different `resolve` methods of the scope interfaces. As the scopes are strongly typed with the language, the resolved symbol kind can be encoded into the method names. This avoids passing the symbol kind to the resolution algorithm as an argument, which would require using reflection (at least to some extend). For example, the interface `IAutomataScope` contains methods for resolving all symbol kind the the automata language is aware of. This includes the methods `resolveState` for resolving symbols of the kind `StateSymbol` and `resolveAutomaton` for resolving `AutomatonSymbols`.

5. Language Composition

Modularization of languages supports the reusability of entire languages or parts of these. In general, the techniques for this and their benefits are known from component-based software engineering (Naur & Randell 1968). Languages can only be modularized in combination with suitable means for language composition (Erdweg et al. 2012). As introduced in Section 3, MontiCore distinguishes four different kinds of language composition. All of these rely on the symbol table to bridge the individual languages. As language extension and language embedding in MontiCore rely on language inheritance, it suffices to consider composing the STIs for language inheritance and for language aggregation.

5.1. Language Inheritance in the STI

During language inheritance, the symbol table infrastructures of the involved languages have to be composed. This is required for a language to resolve in all scopes and for all symbol kinds of the inherited languages. Ideally, all parts of the reused language's STIs are reused without modifying these, without recompiling the code for inherited languages, and without causing code

duplication between the code for the actual language and the code reused from other languages. Because MontiCore supports multiple inheritance on languages, the STIs also must support multiple inheritance.

To achieve this, the scopes of the languages are realized by scope classes and interfaces as introduced in Section 4. The inheritance of scope interfaces, artifact scope interfaces, and global scope interfaces follows the inheritance of the languages. Through this, the resolve algorithm, which is realized as default methods, can be completely reused. Via the top mechanism (Hölldobler et al. 2021), any handwritten adjustments to the resolve algorithm that are performed in the default methods are reused as well. Scope, artifact scope, and global scope classes, however, cannot be reused for language composition. Hence, we recommend carrying out handwritten adjustments to scopes via the interfaces and not via the classes whenever possible.

The symbol classes of inherited languages can be reused without modification. The scopes genitors, which instantiate symbol tables, rely on the visitor infrastructure of the languages. The visitor infrastructures of MontiCore languages are compositional: the individual visitors for a language are modular and the traversal strategy for the composed language delegates to the visitors of individual languages (Hölldobler et al. 2021). Thus, scope genitors are also compositional. For each inherited language, the scope genitor of the language traverses the AST nodes for nonterminals defined in the language and creates the symbol kinds introduced in the language.

A challenge in reusing scope genitors is the creation of the correct scope objects. For instance, in the automata language, the scope genitor creates instances of the class `AutomataScope`. If, e.g., a language extends the automata language to add hierarchical states, the scope genitor of the automata language is reused. In this case, however, the scope genitor of the automata language has to create scope instances of the class `HierarchicalAutomataScope` instead. Obviously, it should be avoided that the automata language itself is aware of the hierarchical automata language that extends it. To create the correct scope objects, e.g., for setting the enclosing scope of AST nodes, scope genitors use a *language mill*. A language mill is a central configuration point for obtaining objects of builder classes for all reconfigurable parts of the language infrastructure. It is realized as singleton class that MontiCore generates for each language. For example, the scope genitor of the automata language instantiates a new scope by calling `Automata.scope()`, which internally uses a class `AutomataScopeBuilder` to create the new scope.

The hierarchical automata language can reconfigure the automata mill to use the class `HierarchicalAutomataScopeBuilder` instead, which is part of the hierarchical automata language and creates new instances of the scope class of the hierarchical automata languages. Through this, the automata language can be configured from external languages without causing dependencies from the reusing (i.e., hierarchical automata) language to the reused (i.e., automata) language.

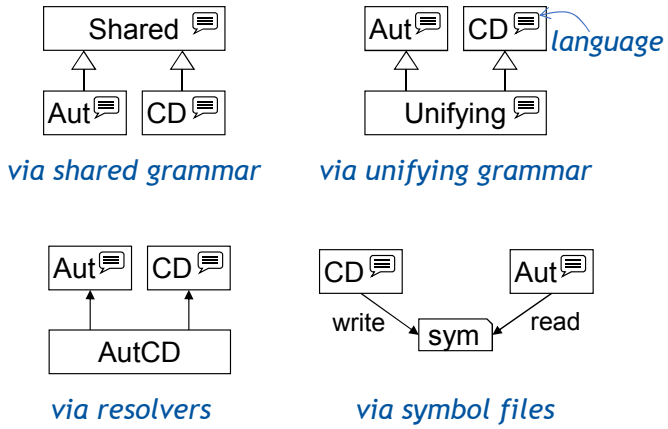

Figure 7 Four different ways to achieve language aggregation between the languages `Aut` and `CD`.

5.2. Symbol Adapters

In all forms of language composition in MontiCore, the situation may occur that the kind of an exchanged symbol that one language provides does not match the kind of the symbol that another language expects. For example, it may be the case that the class diagram language in the example provides class attribute symbols, but the automata language requires symbols used in the guard expressions to be of a different kind, e.g., variable symbols. This can be solved by creating symbol adapters.

A symbol adapter is a class that realizes the adapter pattern (Gamma et al. 1995) for the classes that realize two symbol kinds. A symbol adapter, hence, is a handwritten class that subclasses the symbol class of the target symbol kind and has an attribute of the source symbol kind. Internally, the symbol adapter has to override all necessary methods of the target symbol class and delegate the implementation to methods of the source symbol kind, i.e., the adaptee.

In the example, the symbol adapter would adapt from class attribute symbols to variable symbols. The adapter class, hence, would extend the class `VariableSymbol` and have an attribute `adaptee` of the type `ClassAttributeSymbol`, to which it delegates in the method implementations. This symbol adapter can be created by customizing the symbol resolution for variable symbols. The customization would plug into this resolution, e.g., at dedicated hook point methods. It internally would resolve for class attribute symbols in the scopes of the class diagram language. If this resolution finds a suitable class attribute symbol, it instantiates the adapter class and returns it as the result of the resolution for variable symbols in the automata language.

It would also be possible to describe symbol adapters in terms of models of a "composition meta-language". However, MontiCore currently relies on Java classes to realize adapters to avoid limiting language engineers in the adapter realization that may perform complex calculations. Nevertheless, the implementation of symbol adapters is usually straight-forward.

5.3. Language Aggregation in the STI

Languages in MontiCore can be aggregated in different ways. We distinguish two different forms of language aggregation

that rely on language inheritance. Language aggregation can be achieved between languages that inherit from a common, "shared" language that introduces all symbol kinds that are exchanged between the language. Through this, no symbol adapters are required, as all scopes can resolve for all symbol kinds. An example for language aggregation for two languages `Aut` and `CD` via a shared grammar is depicted in the top left of Figure 7. In the example (cf. Figure 1), the automaton model uses class attributes of the class diagram model. Therefore, the symbol kind for class attributes has to be introduced in the `Shared` language.

Similarly, languages can be aggregated by creating a new grammar that unifies the grammars of the languages that are to be aggregated. An example for this is depicted in the top right of Figure 7. This form of language aggregation is similar to language embedding, but instead of grammar rules that integrate the syntaxes of the languages, the novel grammar only creates a new start rule that uses the start rules of the aggregated languages as alternatives. The scope of the unifying language then is able to resolve for all symbol kinds introduced in any of the aggregated languages. This includes, in particular, also the symbol kind for class attributes.

A further option for creating an aggregation of languages is to make use of symbol resolvers. This option does not rely on language inheritance, but instead reconfigures one or more of the input languages. This is depicted by example in the bottom left of Figure 7. Symbol resolvers are classes that carry out the resolution for a foreign symbol kind and adapt it to a symbol kind that the language is aware of. Symbol resolvers can be added to the global scope of a reused language to reconfigure its resolving algorithm. In the example, the aggregated language has to introduce a resolver that is added to the automata language. This resolver resolves for the class attribute symbols in class diagram models and adds their result to the resolution in the automata language.

Alternatively, languages can be aggregated by exchanging symbol tables via symbol files. For this option, at least one of the aggregated languages has to export their symbol tables to symbol files. Other languages can read such files and import the symbol tables or parts of these. For this option, the symbol files have to contain symbol tables either in a language-agnostic representation or the loading language has to be able to extract the relevant information from the symbol file although it is unaware of the language that has stored the file.

Language aggregation via shared or unifying grammars have the advantage that they produce an integrated scope that is capable of resolving all symbol kinds of the aggregated language. Therefore, the language aggregation results in a completely unified symbol table data structure. However, the aggregation couples the aggregated languages to each other and requires to execute the MontiCore generator. This increases the effort for adding new languages to the language aggregation.

Aggregation through resolvers and symbol files requires less coupling between the languages but also keeps the symbol tables of the individual languages separated from each other.

6. Discussion

While the presented approach for realizing symbol tables is technologically tied to the MontiCore language workbench, its concepts should be transferable to other language workbenches. With means to translate languages defined via grammars into metamodels (Butting, Jansen, et al. 2018) or to create internal DSLs from grammars (Butting, Dalibor, et al. 2018), a broader range of technological spaces can be covered.

With the STI, languages can be composed with loose coupling of the language infrastructures only. It suffices to bridge the scopes of the STIs involved in the language, which can be achieved either by relying on a common inherited language that introduces common symbol kinds or by introducing suitable symbol adapters. However, the STI as presented in this paper would not allow language engineers to completely decouple arbitrary languages. To overcome this limitation, we implemented a mechanism to persist symbol tables and load persisted symbol tables. The loading STI can then be configured to load also symbols of kinds the language is not aware of at all.

We have applied the STI in a great variety of modeling languages, some of which are open-source[2]. These languages include class diagrams, object diagrams, automata, feature diagrams, statecharts, sequence diagrams, use case diagrams, and the MontiArc (Butting et al. 2017) component & connector ADL. The presented approach for generating STIs for these languages was successfully applied and can represent the symbol tables of all these languages. However, most languages required manual adjustment of the STI at some parts. It was necessary to extend the scopes genitor of most languages to correctly instantiate symbol attributes. In some languages, such as the class diagrams, it was also necessary to adjust the symbol resolution.

As explained in Section 4 and Section 5, the STI requires to integrate the scope types of different languages during all forms of language composition that rely on language inheritance. Otherwise, a language cannot resolve symbol kinds it is not aware of. However, for the current realization of the STI it does not always suffice to be aware of the actual symbol kind that is resolved for. For nested scopes, it is also necessary to adapt all foreign symbol kinds that may span an enclosing scope of the scope that contains the symbol that is resolved for. In the example of the class diagram language, a type symbol spans a scope that contains the class attribute symbols. If such a class attribute symbol should be resolved, e.g., in the automata language as part of a guard expression, it is not sufficient that the automata language is aware of the symbol kind for class attribute symbols. Additionally, it must be aware of type symbols to be able to search for class attribute symbols in the scopes these span.

Built-in types can be realized as symbols that are added directly to the global scopes once. Through this, all models of a language are able to resolve these symbols and the symbols do not have do be duplicated, which could cause inconsistencies.

In MontiCore, it is a deliberate decision that all model elements that define symbols must be identifiable through names that modelers indicate. In other words, it is not desired that

[2] Open-source MontiCore languages at Github: https://github.com/MontiCore

model elements that can be identified via derived names or by other means of identification (e.g., source position) define symbols.

With the STI, languages can be aggregated although their infrastructure remain completely separated. The composed language can also add adapters to compose languages that, otherwise, would not be compliant to another. This enables engineering the languages independent of another, which fosters language engieering in the large. This is also a suitable foundation for realizing product lines of modular language components (Butting et al. 2019, 2020).

7. Related Work

Beyond MontiCore, there is a number of language workbenches that can be used to define textual DSLs via grammars, such as Neverlang (Vacchi & Cazzola 2015), Rascal (van der Storm 2011), Spoofax (Kats & Visser 2010), and Xtext (Bettini 2016). Other language workbenches, such as EMF (Steinberg et al. 2008), GEMOC Studio (Degueule et al. 2015), and MetaEdit+ (Tolvanen & Kelly 2009) use metamodels for defining the syntax of merely graphical DSLs. In metamodels, consistency is typically achieved in different ways than by using symbol tables. One could compare different, mainly metamodel-based approaches such as (Leduc et al. 2017), (Garmendia et al. 2019), (Voelter 2013), or (Voelter et al. 2013) more precisely with our approach, however, this would require a large, comprehensive study that tests each approach with examples.

The STI presented in this paper is integrated into the current version of the language workbench MontiCore. Previously, MontiCore supported the symbol management infrastructure (SMI) to realize symbol tables for its languages (Mir Seyed Nazari 2017). While the STI and the SMI share large parts of the general notions behind symbol tables, the specific concepts for realizing symbols, scopes, and the symbol resolution in Java differ. The SMI largely relies on language-agnostic realizations for scopes that are contained in the run-time environment of MontiCore. While this reduces the number of generated artifacts, it is more difficult to apply customization to the scopes, as the top mechanism cannot be applied specific to a single language. Our experiences have shown, that language-specific customizations of scopes (e.g., for custom symbol resolution strategies) are used frequently.

Furthermore, language-agnostic scopes cannot have language-specific symbol resolution methods. Instead, the SMI passes the symbol kind to the resolution algorithm as an argument. To realize this in Java, specific classes for each symbol kind are generated and objects of these are passed to the resolve methods as method arguments. The STI avoids the unnecessary complexity of recreating a type system for symbol kinds and passing symbol kind objects between scopes. A side effect of kind-specific `resolve` method names is that scopes of different languages are not automatically compatible. An advantage of language-agnostic scopes is that no adaptation has to be performed for scopes during language composition. However, the disadvantage is that adjusting the resolution per symbol kind is not easily possible.

The language workbench Spoofax (Kats & Visser 2010) supports realizing type systems via the name binding language (Konat et al. 2012). Contrary to the STI and the SMI, the name binding language relies on declarative rules that describe the type system. While in the STI, symbol kinds differentiate between names for different model elements, the name binding language uses different name spaces to differentiate between different model elements. The STI demands that language engineers indicate a symbol kind for each name definition, whereas the name binding language uses a default namespace for all name definitions that are not explicitly added to a different namespace. Both the STI and the name binding language enable using scopes, but the STI manages symbols in separate data structures for each scope while the name binding language uses a joint data structure for the symbols of all scopes.

8. Conclusion

We expect that the need for heterogeneous models to describe different aspects of an application will grow. Some example domains are the modeling of structure and behavior of CPS used within digital twins (Bordeleau et al. 2020; Brockhoff et al. 2021), process models and resources defined in class diagrams for assistive systems which support human behavior (Michael et al. 2020), or the composition of class diagrams and architectural languages for the deployment of IoT systems (Kirchhof et al. 2022). Using models from two languages where one has an inheritance relationship to the other one might be interesting for languages which are used in requirements engineering, where models can be underspecified, and then in system design, where implementation details have to be added. Examples include, e.g., to use only classes, attributes, and relations in class diagrams for analysis and later on also methods and their signatures for generating code, or if we have human-centric information in requirements models that should be used in design modeling languages (Grundy et al. 2021). Moreover, kind-typed symbol tables enable enhanced functionalities for reusable model libraries (Dalibor et al. 2022), e.g., providing model sets of different languages which use symbols of each other and can be reused in combination.

Within this paper, we have shown how it is possible to refer to model elements of other languages when composing different DSMLs. Our approach to realize this are strongly kind-typed symbol tables as an extension to a generative approach. Strongly kind-typed symbol tables utilize the type system of the language workbench's host language to ensure type consistency between the language-specific symbol table infrastructures during DSML composition, which ultimately supports DSML engineering in the large. In future, it would be useful to develop a technique for behavioral semantics of models (Rivera & Vallecillo 2007; Rivera et al. 2009), that provides information about the behavior on symbol level.

Learning from others and being inspired by the work of brilliant colleagues like Antonio is an important pillar of our scientific work. It is a privilege to have him as part of our community.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools*. Pearson Education.

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd.

Bordeleau, F., Combemale, B., Eramo, R., van den Brand, M., & Wimmer, M. (2020). Towards Model-Driven Digital Twin Engineering: Current Opportunities and Future Challenges. In Ö. Babur, J. Denil, & B. Vogel-Heuser (Eds.), *Systems Modelling and Management* (pp. 43–54). Springer. doi: 10.1007/978-3-030-58167-1_4

Brockhoff, T., Heithoff, M., Koren, I., Michael, J., Pfeiffer, J., Rumpe, B., … Wortmann, A. (2021). Process Prediction with Digital Twins. In *Int. conf. on model driven engineering languages and systems companion (models-c)* (p. 182-187). ACM/IEEE. doi: 10.1109/MODELS-C53483.2021.00032

Bryant, B., Jézéquel, J.-M., Lämmel, R., Mernik, M., Schindler, M., Steinmann, F., … Völter, M. (2015). Globalized Domain Specific Language Engineering. In B. Combemale, B. H. Cheng, R. B. France, J.-M. Jézéquel, & B. Rumpe (Eds.), *Globalizing Domain-Specific Languages* (p. 43-69). Springer. doi: 10.1007/978-3-319-26172-0_4

Burgueño, L., Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., … Wimmer, M. (2019). Contents for a Model-Based Software Engineering Body of Knowledge. *Software and Systems Modeling*, *18*(6), 3193–3205. doi: 10.1007/s10270-019-00746-9

Butting, A., Dalibor, M., Leonhardt, G., Rumpe, B., & Wortmann, A. (2018). Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)* (p. 187-199). ACM. doi: 10.1145/3276604.3276621

Butting, A., Eikermann, R., Kautz, O., Rumpe, B., & Wortmann, A. (2019). Systematic Composition of Independent Language Features. *Journal of Systems and Software*, *152*, 50-69. doi: https://doi.org/10.1016/j.jss.2019.02.026

Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., & Wortmann, A. (2017). Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)* (p. 53-70). Springer. doi: 10.1007/978-3-319-61482-3_4

Butting, A., Jansen, N., Rumpe, B., & Wortmann, A. (2018). Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)* (p. 174-186). ACM. doi: 10.1145/3276604.3276605

Butting, A., Pfeiffer, J., Rumpe, B., & Wortmann, A. (2020). A Compositional Framework for Systematic Modeling Language Reuse. In *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (p. 35–46). ACM. doi: 10.1145/3365438.3410934

Butting, A., & Wortmann, A. (2021). Language Engineering for Heterogeneous Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems* (p. 239-253). Springer. doi: 10.1007/978-3-030-62136-0_11

Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J., & Vojtisek, D. (2016). *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series.

Şutîi, A. M., van den Brand, M., & Verhoeff, T. (2018). Exploration of modularity and reusability of domain-specific languages: an expression DSL in MetaMod. *Computer Languages, Systems & Structures*, *51*, 48-70. doi: 10.1016/j.cl.2017.07.004

Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., … Wortmann, A. (2022). Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, *70*. doi: 10.1016/j.cola.2022.101117

Degueule, T., Combemale, B., Blouin, A., Barais, O., & Jézéquel, J.-M. (2015). Melange: A Meta-language for Modular and Reusable Development of DSLs. In *8th International Conference on Software Language Engineering (SLE)*. Pittsburgh, United States.

Drave, I., Henrich, T., Hölldobler, K., Kautz, O., Michael, J., & Rumpe, B. (2020). Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In D. Bork, D. Karagiannis, & H. C. Mayr (Eds.), *Modellierung 2020* (p. 173-188). Gesellschaft für Informatik e.V.

Erdweg, S., Giarrusso, P. G., & Rendel, T. (2012). Language Composition Untangled. In *12th Workshop on Language Descriptions, Tools, and Applications* (pp. 1–8). doi: 10.1145/2427048.2427055

Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., … van der Woning, J. (2015). Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, *44*, 24-47. (SI on the 6th and 7th Int. Conf. on Software Language Engineering (SLE'13 and SLE'14)) doi: 10.1016/j.cl.2015.08.007

France, R., & Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, 37-54. doi: 10.1109/FOSE.2007.14

Fuentes-Fernández, L., & Vallecillo-Moreno, A. (2004). An Introduction to UML Profiles. *UML and Model Engineering*, *2*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Garmendia, A., Guerra, E., de Lara, J., García-Domínguez, A., & Kolovos, D. (2019). Scaling-Up Domain-Specific Modelling Languages through Modularity Services. *Information and Software Technology*, *115*, 97–118. doi: 10.1016/j.infsof.2019.05.010

Grundy, J., Khalajzadeh, H., McIntosh, J., Kanij, T., & Mueller, I. (2021). HumaniSE: Approaches to Achieve More Human-Centric Software Engineering. In R. Ali, H. Kaindl, & L. A. Maciaszek (Eds.), *Evaluation of Novel Approaches to Software Engineering* (pp. 444–468). Springer. doi:

10.1007/978-3-030-70006-5_18

Gupta, R., Kranz, S., Regnat, N., Rumpe, B., & Wortmann, A. (2021). Towards a Systematic Engineering of Industrial Domain-Specific Languages. In *IEEE/ACM 8th Int. Workshop on Software Engineering Research and Industrial Practice (SE&IP)* (p. 49-56). IEEE. doi: 10.1109/SER-IP52554.2021.00016

Hölldobler, K., Kautz, O., & Rumpe, B. (2021). *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag. Retrieved from http://www.monticore.de/handbook.pdf

Hölldobler, K., Mir Seyed Nazari, P., & Rumpe, B. (2015). Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In *Domain-Specific Modeling Workshop (DSM'15)* (p. 23-30). ACM. doi: 10.1145/2846696.2846700

Johanson, A. N., & Hasselbring, W. (2017). Effectiveness and Efficiency of a Domain-Specific Language for High-Performance Marine Ecosystem Simulation: A Controlled Experiment. *Empirical Software Engineering*, *22*(4), 2206-2236. doi: 10.1007/s10664-016-9483-z

Kats, L. C., & Visser, E. (2010). The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *25th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)* (pp. 444–463). doi: 10.1145/1932682.1869497

Kirchhof, J. C., Rumpe, B., Schmalzing, D., & Wortmann, A. (2022). MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software*, *183*, 1-21. doi: https://doi.org/10.1016/j.jss.2021.111087

Konat, G., Kats, L., Wachsmuth, G., & Visser, E. (2012). Declarative Name Binding and Scope Rules. In *Int. Conf. on Software Language Engineering (SLE'12)* (pp. 311–331). doi: 10.1007/978-3-642-36089-3_18

Leduc, M., Degueule, T., Combemale, B., van der Storm, T., & Barais, O. (2017). Revisiting Visitors for Modular Extension of Executable DSMLs. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (p. 112-122). doi: 10.1109/MODELS.2017.23

Meyers, B., Cicchetti, A., Guerra, E., & de Lara, J. (2012). Composing Textual Modelling Languages in Practice. In *6th Int. Workshop on Multi-Paradigm Modeling (MPM'12)* (pp. 31–36). ACM. doi: 10.1145/2508443.2508449

Michael, J., Rumpe, B., & Varga, S. (2020). Human Behavior, Goals and Model-Driven Software Engineering for Assistive Systems. In *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)* (Vol. 2628, p. 11-18). CEUR Workshop Proceedings.

Mir Seyed Nazari, P. (2017). *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Shaker Verlag.

Naur, P., & Randell, B. (Eds.). (1968). *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.*

Rivera, J. E., Durán, F., & Vallecillo, A. (2009). Formal Specification and Analysis of Domain Specific Models Using Maude. *SIMULATION*, *85*(11-12), 778-792. doi: 10.1177/0037549709341635

Rivera, J. E., & Vallecillo, A. (2007). Adding Behavior to Models. In *11th IEEE Int. Enterprise Distributed Object Computing Conference (EDOC 2007)* (p. 169-169). doi: 10.1109/EDOC.2007.40

Rumpe, B., Michael, J., Kautz, O., Krebs, R., Gandenberger, S., Standt, J., & Weber, U. (2021). *Digitalisierung der Gesetzgebung zur Steigerung der digitalen Souveränität des Staates* (Vol. 19). Nationales E-Government Kompetenzzentrum e. V.

Schlichthaerle, S., Becker, K., & Sperber, S. (2020). A Domain-Specific Language Based Architecture Modeling Approach for Safety Critical Automotive Software Systems. In *Software Engineering Workshops 2020.* CEUR-WS.org.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Tolvanen, J.-P., & Kelly, S. (2009). MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications* (pp. 819–820). doi: 10.1145/1639950.1640031

Vacchi, E., & Cazzola, W. (2015). Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, *43*, 1–40. doi: 10.1016/j.cl.2015.02.001

Vallecillo, A. (2010). On the Combination of Domain Specific Modeling Languages. In T. Kühne, B. Selic, M.-P. Gervais, & F. Terrier (Eds.), *Modelling Foundations and Applications* (pp. 305–320). Springer. doi: 10.1007/978-3-642-13595-8_24

van der Storm, T. (2011). *The Rascal Language Workbench*. CWI. Software Engineering [SEN].

Voelter, M. (2013). Language and IDE Modularization and Composition with MPS. In R. Lämmel, J. Saraiva, & J. Visser (Eds.), *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, 2011. Revised Papers* (pp. 383–430). Springer. doi: 10.1007/978-3-642-35992-7_11

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., ... Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. Retrieved from http://www.dslbook.org

Wolschke, C., Marksteiner, S., Braun, T., & Wolf, M. (2021). An Agnostic Domain Specific Language for Implementing Attacks in an Automotive Use Case. In *16th International Conference on Availability, Reliability and Security (ARES 2021).* ACM. doi: 10.1145/3465481.3470070

About the authors

Arvid Butting received his B. Sc. and M. Sc. degrees in computer science from the RWTH Aachen University, in 2014 and 2016. Currently, he is a research assistant and Ph.D. candidate at the Software Engineering chair at RWTH Aachen University.

His PhD thesis is about the systematic composition of language components in the language workbench MontiCore. His research interests cover software language engineering, software architectures, and model-driven development. You can contact the author at butting@se-rwth.de or visit www.se-rwth.de.

Judith Michael is PostDoc and leads the model-based assistance and information services team at the Software Engineering chair at RWTH Aachen University. Her PhD thesis at Universität Klagenfurt was about cognitive modeling for assistive systems. Her research focus is model-driven software engineering and software architectures, domain-specific languages, and (conceptual) modeling in a variety of domains and applications. Recent work deals with software architectures of assistive and information systems, digital twins and digital shadows in the production domain, privacy-preserving system design, smart assisted living, and human behavior goal modeling. You can contact the author at michael@se-rwth.de or visit www.se-rwth.de.

Bernhard Rumpe is heading the Software Engineering chair at RWTH Aachen University, Germany. Earlier, he had positions at INRIA Rennes, Colorado State University, TU Braunschweig, Vanderbilt University, Nashville, and TU Munich. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model-engineering based on UML/SysML-like notations and domain specific languages. He also helps to apply modeling, e.g. to autonomous cars, human brain simulation, BIM energy management, juristical contract digitalization, production automation, cloud, and many more.
He is author and editor of 34 books including "Agile Modeling with the UML" and "Engineering Modeling Languages: Turning Domain Knowledge into Tools". You can contact the author at rumpe@se-rwth.de or visit www.se-rwth.de.